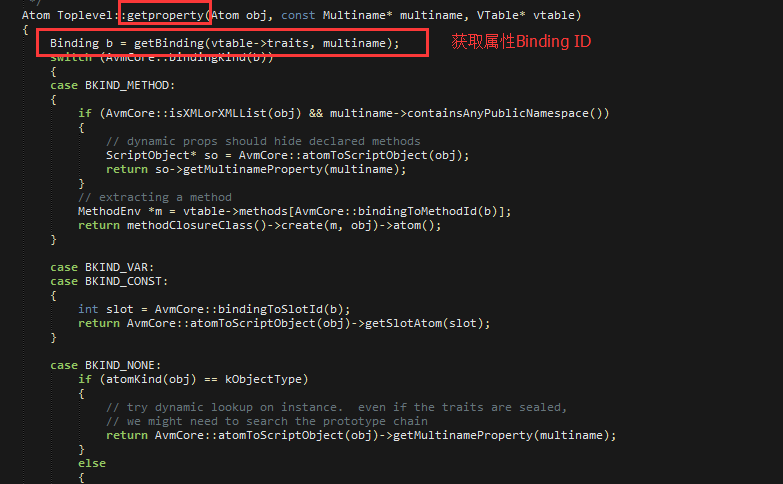
**CVE-2015-7645漏洞利用分析**

**0x00 调试环境**

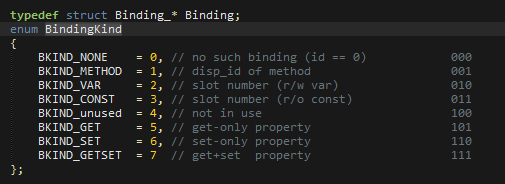
|  |  |
| --- | --- |
| OS: | win7 x86 |
| IE: | ie10 |
| Flash: | flashplayer18\_0r0\_209\_win\_debug |

**0x01 漏洞成因**

as3脚本获取对象属性(比如obj.var，obj.foo())在avm层面是通过getproperty函数获取的：

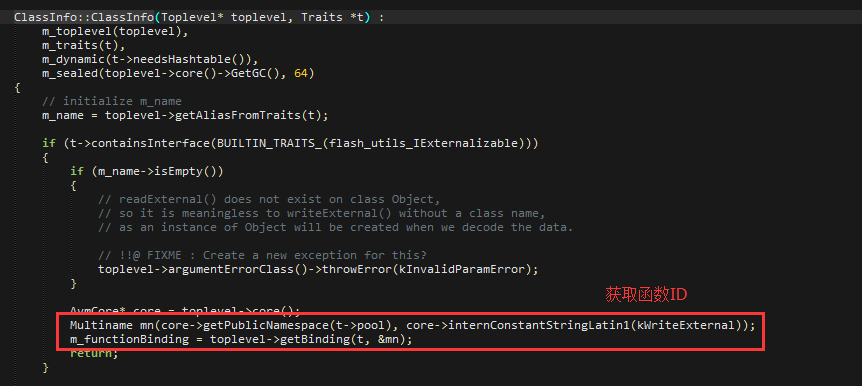


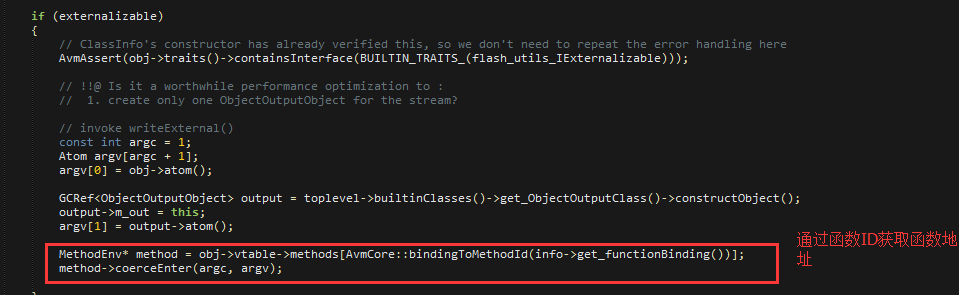
getBinding()根据属性名字返回bind ID，再根据ID取出不同变量或者函数的地址。这个ID由两部分组成：0~2bit是这个属性的类型：



其余位用来获取这个属性的值。以BKIND\_METHOD和BKIND\_VAR为例，avm取得bind ID后先将这个ID和0x7进行与运算取出属性类型，如果是BKIND\_VAR（假如剩余位的值是n）则该属性的值存放在该对象偏移0x4\*n的位置，如果是BKIND\_METHOD该函数地址存放在该对象偏移0x4\*n+3C的位置。

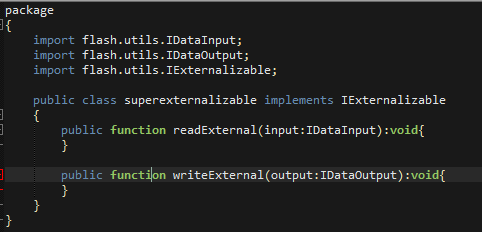
该漏洞是IExternalizable接口的writeExternal函数的一个类型混淆漏洞。avm在获取writeExternal函数地址时没有对返回的Bind ID进行校验，因此可以通过申明一个writeExternal变量(BKIND\_VAR)来混淆writeExternal函数的调用。



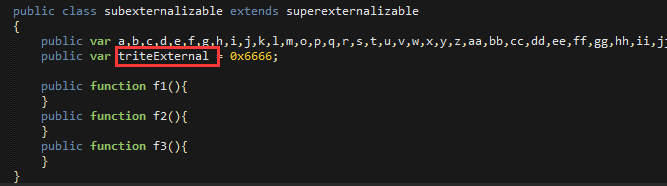


POC如下：

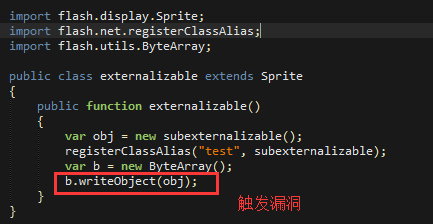
首先声明一个superexternalizable类继承IExternalizable，并实现readExternal，writeExternal函数。



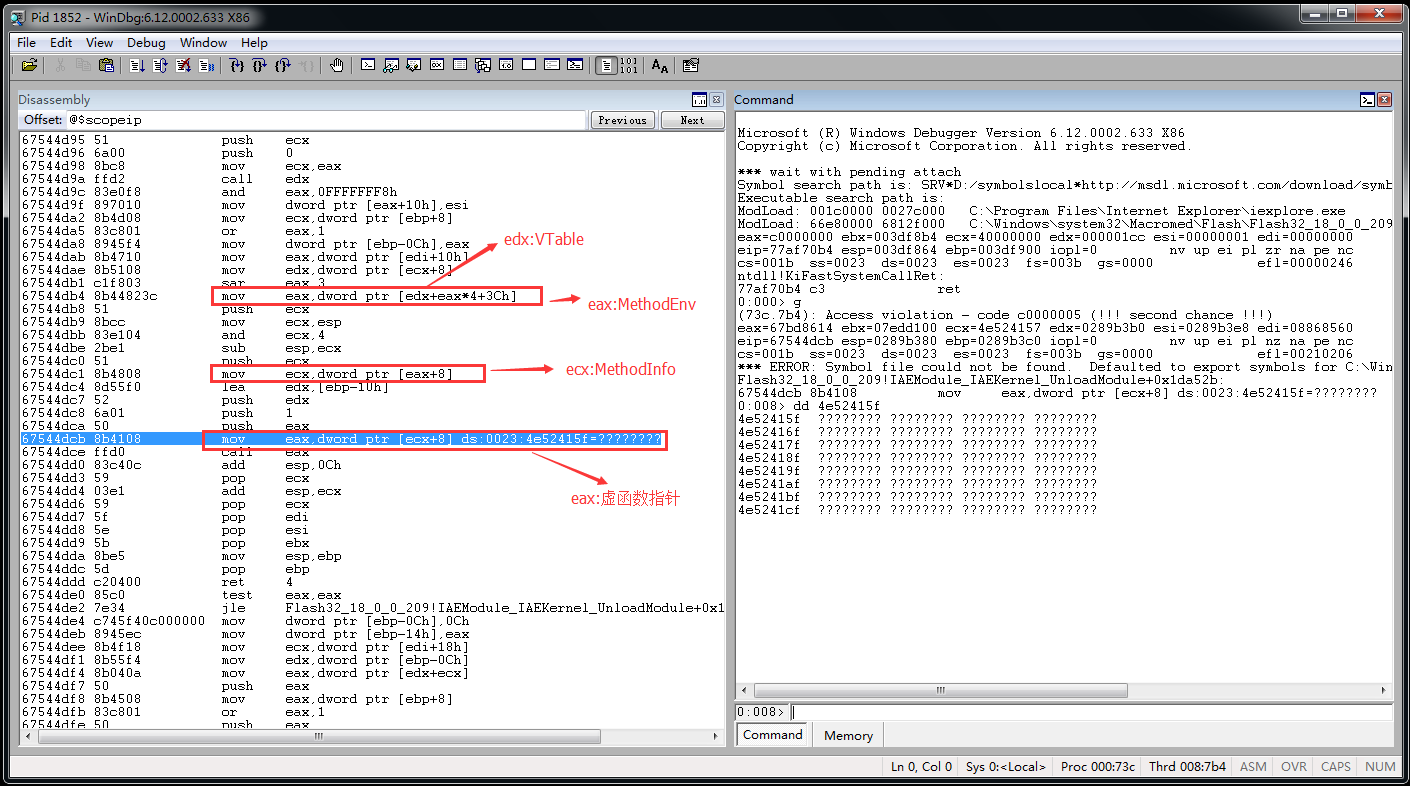
再声明一个subexternalizable类继承superexternalizable，并声明变量triteExternal(因为编译器不允许声明writeExternal函数同名的变量，这里先triteExternal后面通过修改swf来实现)：



最后在主函数通过writeObject()触发漏洞：



编译后浏览器加载运行，发现IE crash:

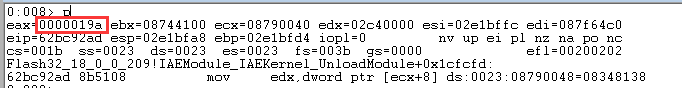


这里是一个VTable的调用，最后发现函数指针指向一片未知内存区域，最终call失败。

可以看到67544db1处的指令是

sar eax, 3

这和之前分析获取对象的函数地址逻辑是一致的，在此处前一个指令下断，观察eax的值：



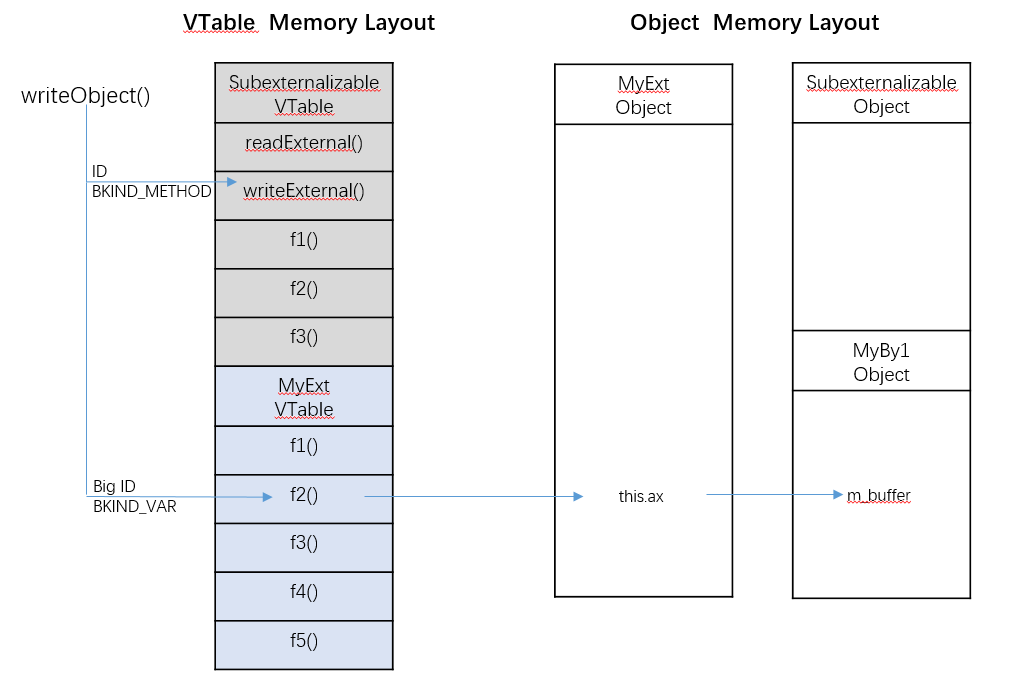
eax=19a，转成二进制：

mov edx, dword ptr [edx+eax\*4+3Ch]

这里的eax可以被控制，从而通过构造eax实现subexternalizable的VTable的越界。

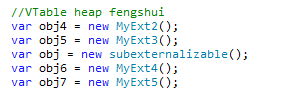
**0x02 漏洞利用**

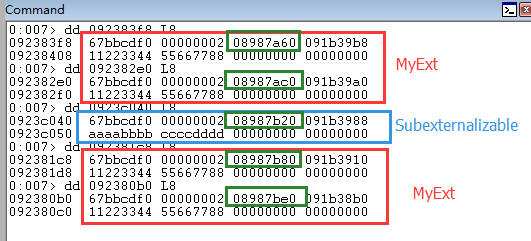
考虑在subexternalizable的VTable相邻处排布MyExt类的VTable（此处MyExt类为攻击者可控），构造索引很大的writeExternal变量实现subexternalizable的VTable越界访问到MyExt类的VTable，此时调用函数为MyExt的this.f2()但是this却是subexternalizable类，因此如果MyExt类内存空间大于subexternalizable类，就可以通过this.f2()实现subexternalizable对象的越界读写。同时再通过heapspray将subexternalizable对象和MyBy1对象(MyBy1继承ByteArray)相邻排布，这样就可以通过MyExt的this.f2()和MyExt的变量修改MyBy1的内存了：

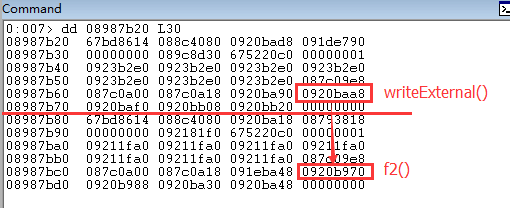


1. **VTable 布局**

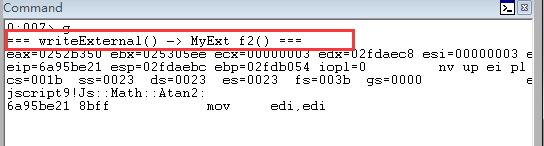
Subexternalizable一共5个虚函数（readExternal, writeExternal, f1 ,f2, f3,），因此MyExt也需要5个虚函数才能和Subexternalizable VTable占用空间一样（MyExt已有f1,f2,f3），这里构造MyExt的子类MyExt2，MyExt3，MyExt4，MyExt5并且都申明了函数f4,f5这样MyExt2，MyExt3，MyExt4，MyExt5的VTable和Subexternalizable VTable占用空间一样，通过下面函数进行堆排列：





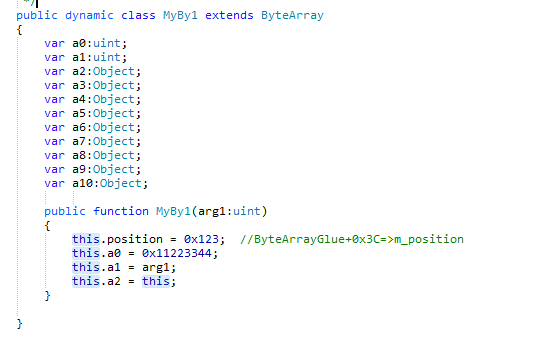


可以看到Subexternalizable VTable和MyExt VTable相邻，通过混淆漏洞导致Subexternalizable VTable越界最终调用了MyExt f2():

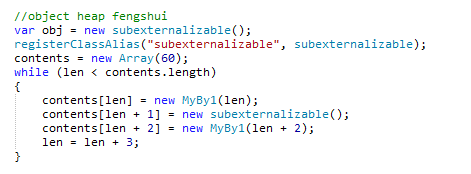


1. **Object 布局**

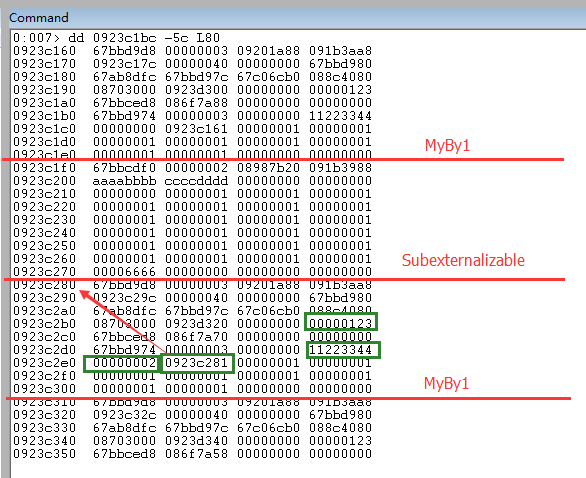
声明了MyBy1类继承ByteArray：



通过heap spray排列内存

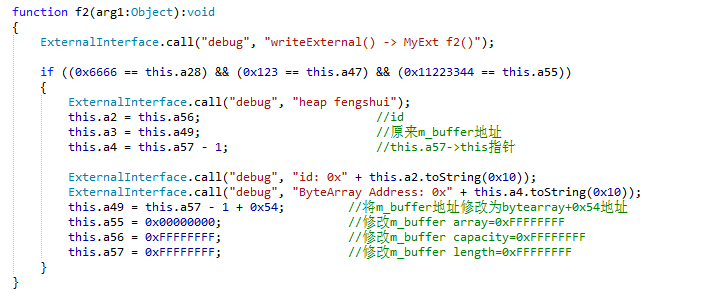


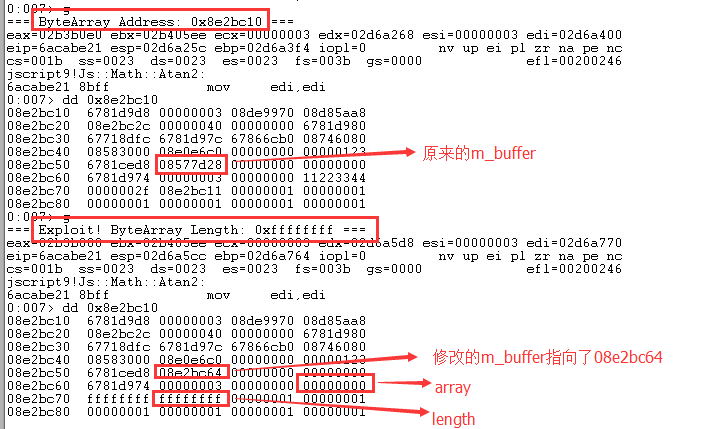
heapspray完内存情况:



1. **修改MyBy1长度**

ByteArray在0x44（flashplayer 18）处保存了m\_buffer, m\_buffer在0x8处保存了array，在0xc处保存了capacity，在0x10处保存了length，这里只需要将m\_buffer指向可控MyBy1内存空间，并修改长度即可：





这里就获得了一个其实地址为0x00000000长度为0xFFFFFFFF的ByteArray。后面的利用就很简单了：

1. 构造任意地址读写函数
2. 通过this指针bypass ASLR，获取VirtualProtect函数地址
3. 将shellcode和ROP存放在ByteArray中
4. 构造函数，并将VTable替换为VirtualProtect Bypass DEP
5. 执行Shellcode