**CVE-2015-0311漏洞利用分析**

**0x00 调试环境**

|  |  |
| --- | --- |
| OS: | win7 x86 |
| IE: | ie10 |
| Flash: | Flash32\_14\_0\_0\_145.ocx |

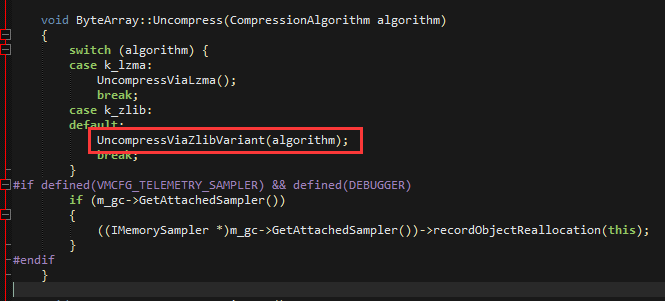
**0x01 漏洞原理**

1. **漏洞概述**

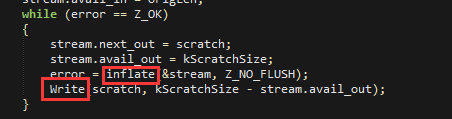
ActionScript3提供对ByteArray数据的压缩方法compress和解压方法uncompress。当调用uncompress方法时，avm会分配一块新的内存存放解压后的数据，并将ByteArray.m\_buffer.array指向这块新的内存，如果uncompress失败则会释放这块新的内存并将ByteArray->m\_buffer->array指向原来的内存。

存在漏洞的flash版本对uncompress失败情况的处理存在UAF漏洞：将ByteArray指向原来的内存后并没有通知其他指向这个ByteArray的指针释放内存，比如ApplicationDomain.currentDomain.domainMemory，这就产生了指向已释放内存的一个悬挂指针。

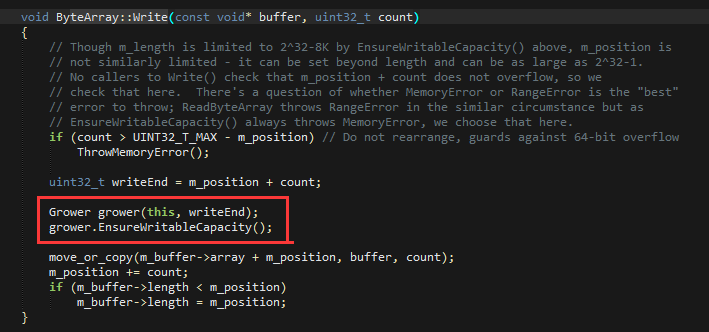
1. **漏洞分析**
2. 解压ByteArray时，avm调用ByteArray::Uncompress()成员函数，该成员函数会根据不同的压缩算法进行解压，这里以zlib为例，调用ByteArray::UncompressViaZlibVariant()：



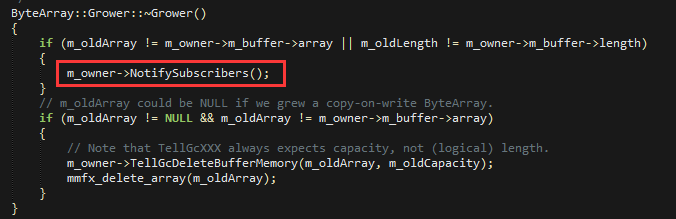
1. ByteArray::UncompressViaZlibVariant()调用zlib库的inflate()方法解压数据，并调用ByteArray::Write()：



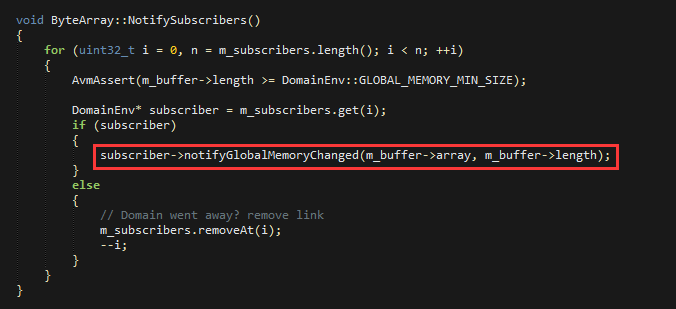
1. ByteArray::Write()调用Grower类实现内存动态分配：



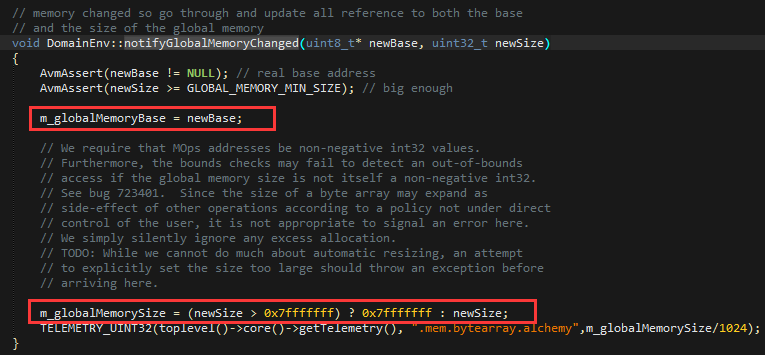
1. Grower类析构函数检测ba的m\_buffer是否有变化，有变化则通过NotifySubscribers()通知ba的domainMemory引用：



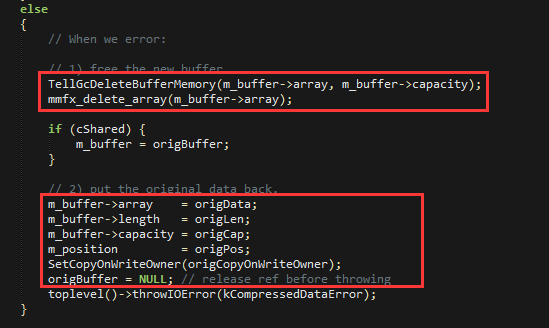
1. NotifySubscribers()检查ba是否存在domainMemory引用，存在则调用notifyGlobalMemoryChanged()通知domainMemory修改m\_buffer：



1. notifyGlobalMemoryChanged修改m\_globalMemoryBase和m\_globalMemorySize，将domainMemory指向ba现在的内存：



(1)~(6)就是avm的ByteArray::Uncompress()的逻辑，然而在(2)中如果ByteArray::UncompressViaZlibVariant()在循环中调用inflate解压数据失败，则会释放Grown类动态分配的内存，并将ba指向原来的压缩数据的内存地址。



这里发现，虽然释放了解压的内存并将ba.m\_buffer指向原来的内存，但是并没有通知domainMemory，从而形成悬挂指针。

1. **漏洞调试**

poc如下:

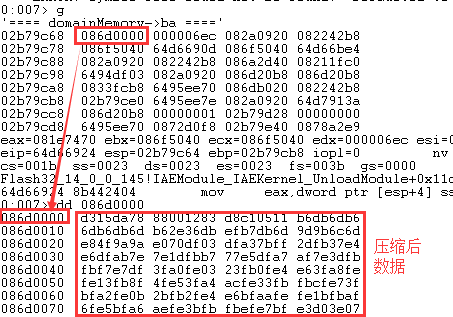


1. 先申请了0x1000Byte大小的ba，压缩，并将domainMemory指向这个ba；
2. 然后从0x200索引处开始破坏压缩数据，再进行解压，此时ba和domainMemory都指向了新分配的内存；
3. 由于压缩数据被破坏，导致解压失败，新分配的内存空间被释放，ba指向原来的内存，而domainMemory仍然指向新分配的内存，形成悬挂指针
4. 再次申请了大小同为0x1000Byte的Vector.<uint>占位，从而通过domainMemory修改这个Vector属性长度。

具体调试情况：

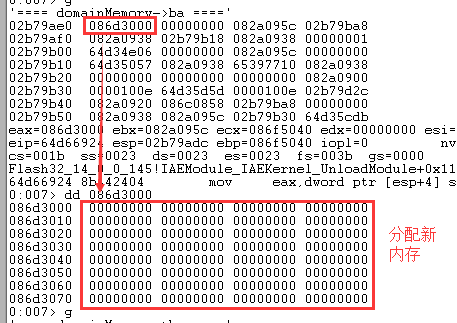
1. ApplicationDomain.currentDomain.domainMemory = ba;

此时domainMemory指向ba的array内存地址，此时数据已经被压缩



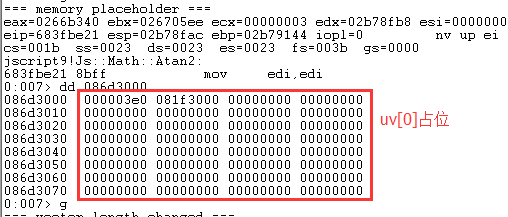
1. ba.uncompress();

domainMemory和ba都指向一块新的内存：



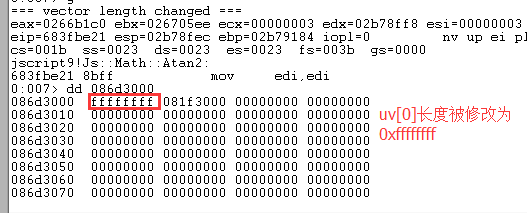
1. uv[0] = new Vector.<uint>(0x3e0);

解压失败，0x86d3000内存被释放，uv占位：



1. casi32(0, 0x3e0, 0xffffffff);

利用悬挂指针domainMemory修改uv[0]长度：



**0x02 漏洞利用**

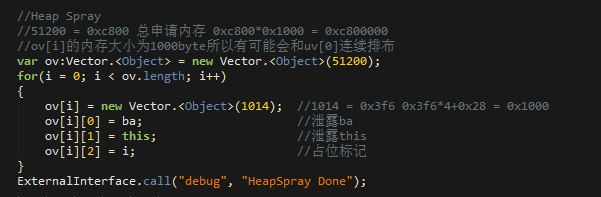
在获得一个长度为0xFFFFFFFF的Vector.<uint>后，要实现漏洞利用win7环境下一般考虑三件事情：

1. Shellcode存放
2. 绕过ASLR、DEP
3. 劫持EIP

其中(3)可以通过覆盖类对象虚表指针来实现，而(1)(2)则需要利用Heap Spray技术。

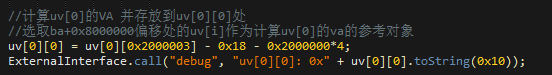
1. **Heap Spray**

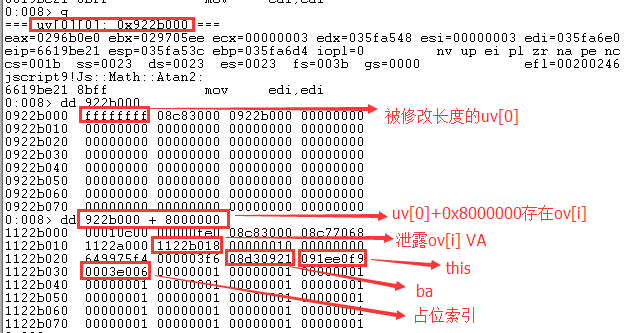
这里Heap Spray通过一个Vector.<Object>实现：



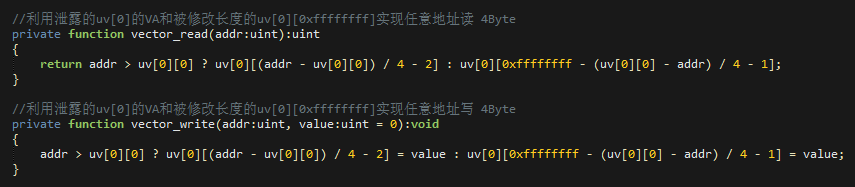
每个ov[i]的大小为0x1000Byte，并且在ov[i][0]，ov[i][1]，ov[i][2]分布排布了ba，this指针和占位的索引。由于ov[i]大小与之前被修改长度的uv[0]一样，由于flash custom heap的连续分配，在uv[0]+n\*0x1000稳定存在ov[i]，这样就可以利用uv[0]读取ov[i]的数据了。经过调试这里选取的n=0x8000。

这里利用uv[0]+0x8000000处的ov[i]来计算uv[0]的VA并保存至uv[0][0]处：

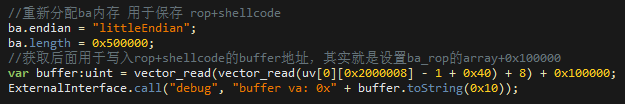




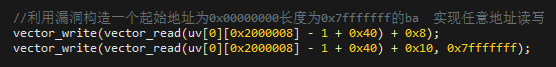
获得了uv[0]的VA就可以利用uv[0]的索引实现指定地址的读写：



这里通过修改ba长度来重新申请一个ba内存地址用来后面保存rop和shellcode，并保存该buffer的VA:



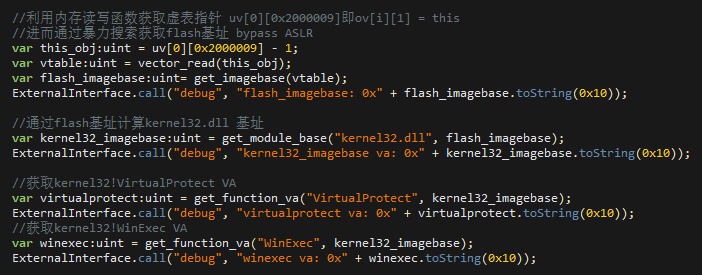
接着利用漏洞修改ba的长度，构造0x00000000~0x7fffffff大小ring3层任意地址读写的函数：



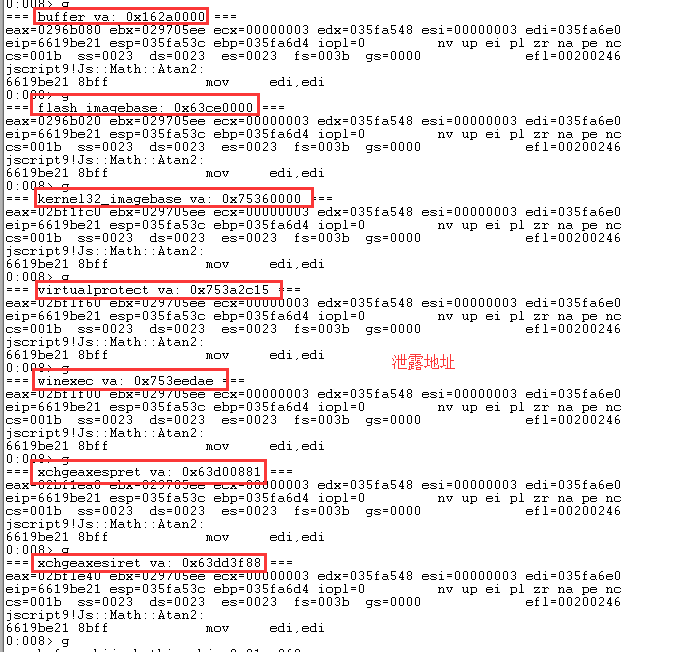


1. **绕过ASLR、DEP**

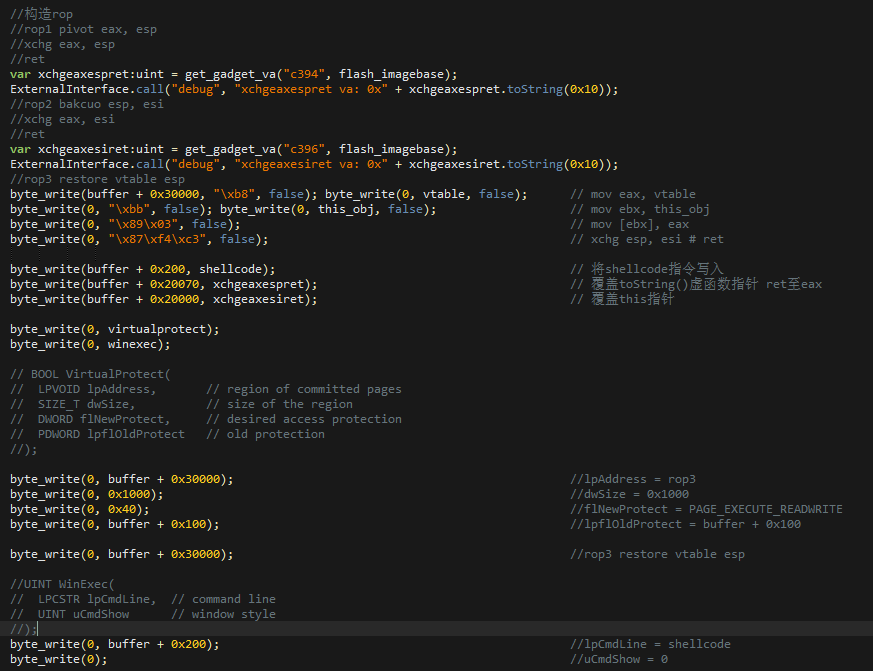
ASLR绕过主要通过Heap Spray中存放在ov[i][1]中的this指针来实现：

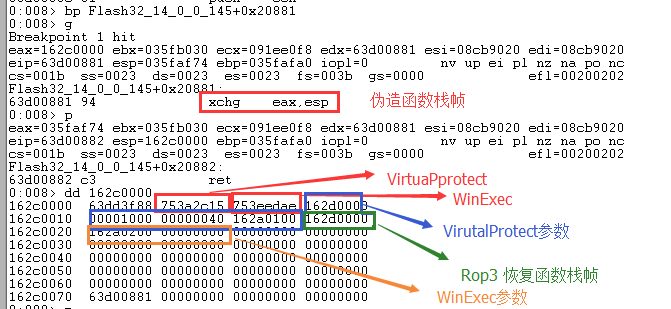


这里就分别获得了VirtualProtect Winexec函数的VA。



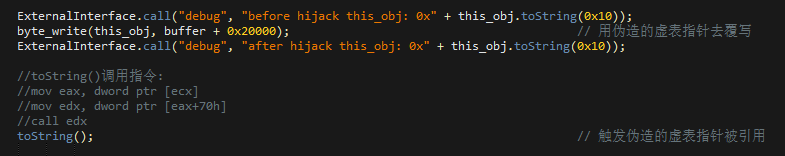
DEP绕过通过ROP实现，这里ROP使用了pivot来将栈转移到堆中：

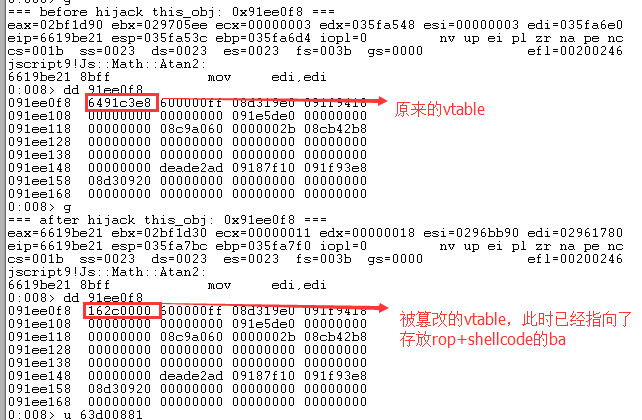




1. **EIP劫持**

EIP劫持利用覆盖对象虚表的方式实现，最终通过toString()调用伪造的虚函数劫持EIP：





Ps:Object.toSting()在内存中的情况：

