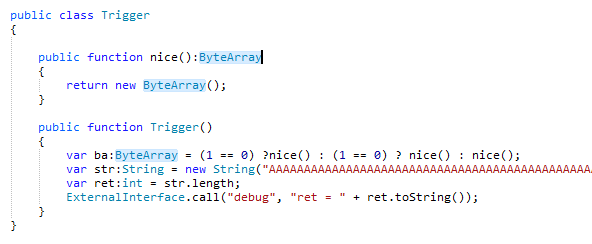
**CVE-2011-0609漏洞利用分析**

**0x00 调试环境**

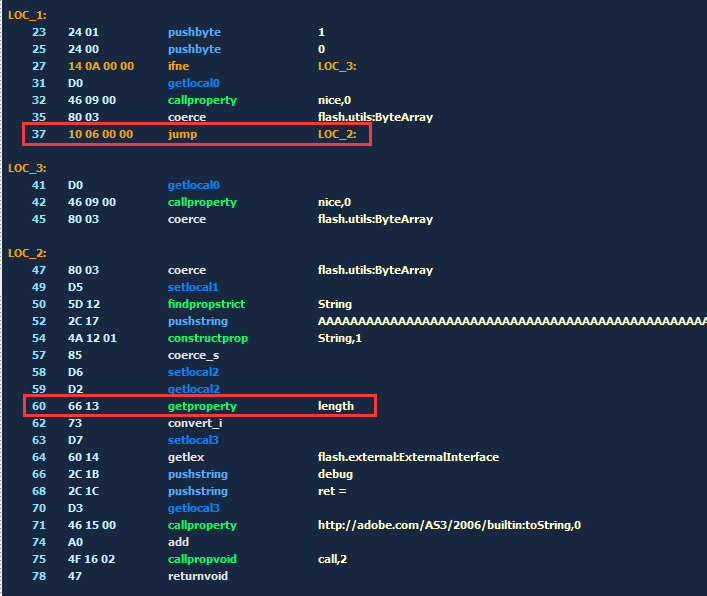
|  |  |
| --- | --- |
| OS: | win7 x86 |
| Flash: | flashplayer10\_1r85\_3\_win\_sa\_debug |

**0x01 漏洞成因**

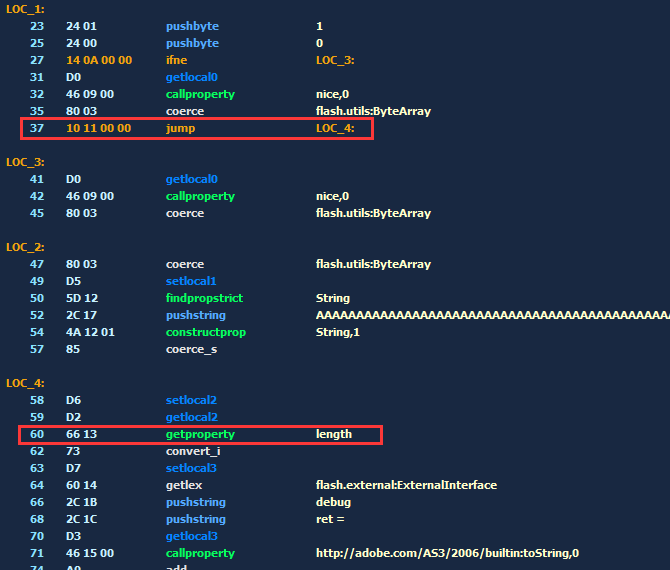
该漏洞是avm2处理as3三目运算符时，verify流程和execute流程的逻辑错误导致类型混淆。Poc代码如下：



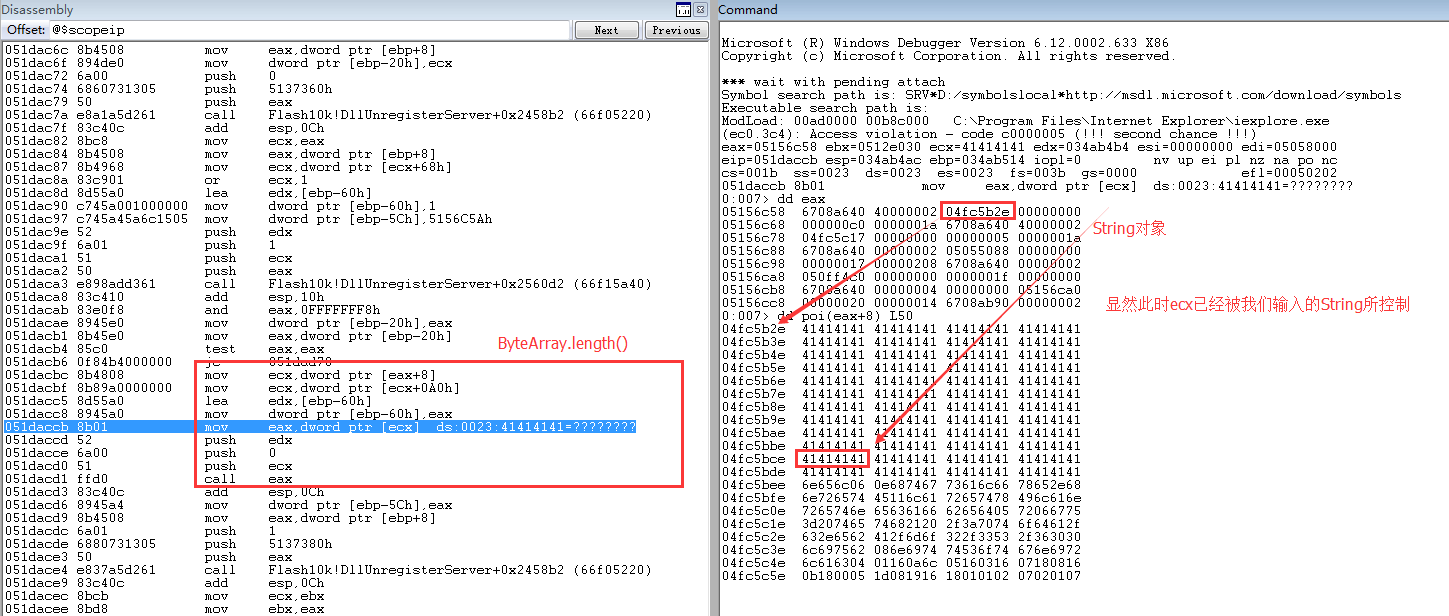
编译后，观察Trigger()对应的ABC code：



注意到获得length属性时此时栈内被压入的是String对象，所以这里调用了String.length()，这里对37的jump指令进行修改，使他跳转到58处：



此时verify流程会从37jump到LOC\_4(58)（但是execute流程还是会按照原来的路径），运行修改后的flash文件，ie crash现场：



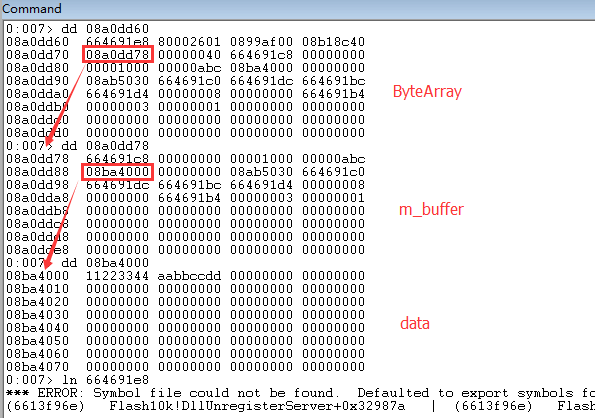
事实上红框内为ByteArray取Length的虚函数调用，但是this指针却指向了String对象，所以这里存在一处类型混淆，其原因是对于被篡改的flash，AVM2verify流程通过栈上数据认为length属性是ByteArray对象的，setjit时生成了ByteArray取Length的虚函数调用的jit code（红框处）。实际上execute流程中此时栈上的对象是String，从而错误的使用了ByteArray取Length的虚函数调用读取了String的内存空间，从而形成类型混淆。

**0x02 漏洞利用**

漏洞利用一般要做三件事：1）找到存放shellcode的地址 2）绕过ASLR和DEP 3）劫持EIP

**1）泄露地址，绕过ASLR和DEP**

Shellcode一般可以存放在ByteArray对象中，Flash10k的ByteArray对象在内存中的布局如下：

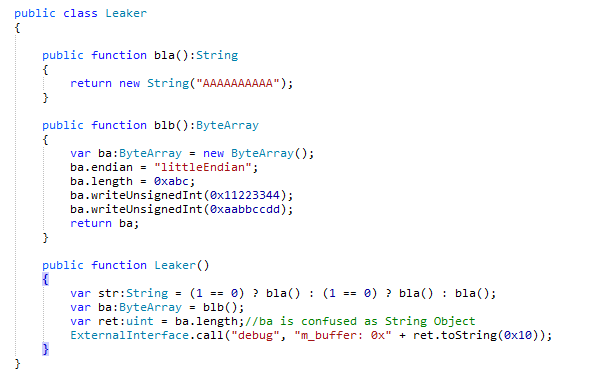


ByteArray对象偏移0x0处为其vtable，在flash module内存地址空间，偏移0x10为m\_buffer对象；

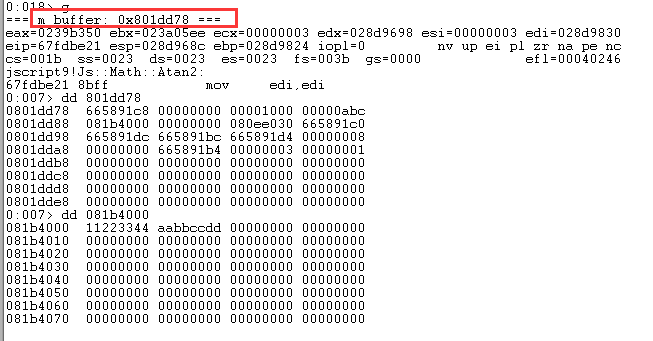
m\_buffer对象偏移0x8为capacity，偏移0xC为length，偏移0x10为其data数据指针。

因此如果能读取ByteArray对象(或者m\_buffer)偏移0x0 4Bytes和m\_buffer偏移0x10 4Bytes的数据就分别能获得flash module基址和ByteArray的data地址。

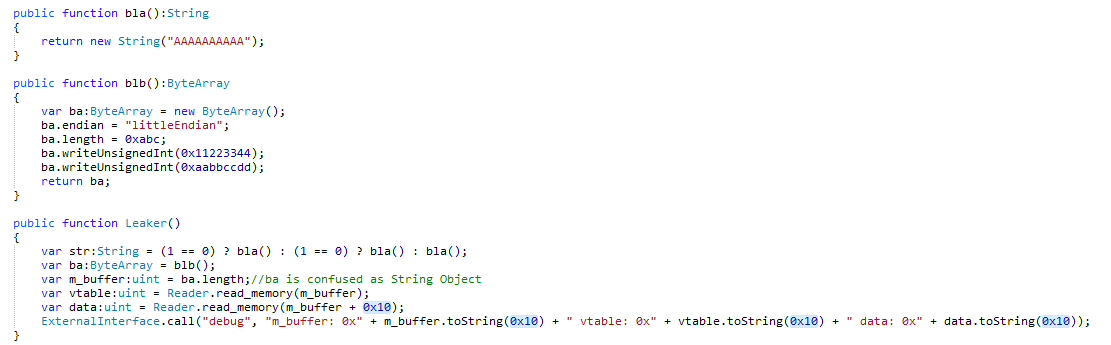
根据前面的分析可以知道String对象的0x10处存放了String的长度，可以通过String.length读取到，这个偏移正好和ByteArray对象m\_buffer指针偏移一致，因此可以利用这个类型混淆漏洞把ByteArray混淆成String，利用String的length方法读取ByteArray偏移0x10的m\_buffer，poc如下：

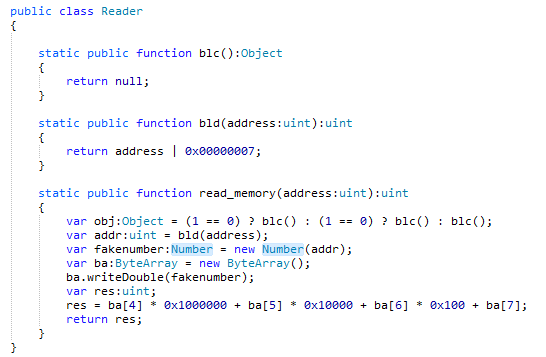


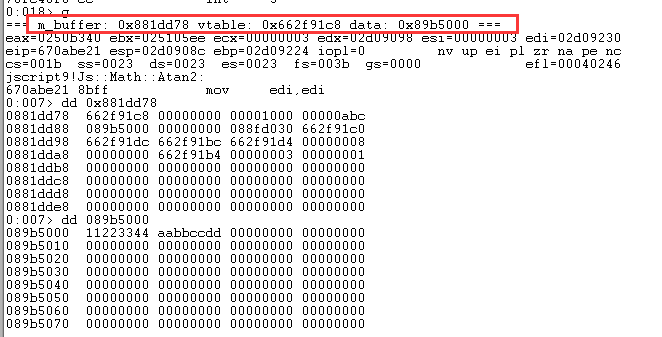
修改ABCcode的jump地址，运行：



显然ByteArray的m\_buffer地址已经被泄露。但是这还不够，真正对我们有用的是poi(m\_buffer)和poi(m\_buffer+0x10)的数据，因此考虑构造任意地址读取方法，这里可以考虑使用as3的Number对象混淆来实现，因为poi(Number)= Number.value。poc:



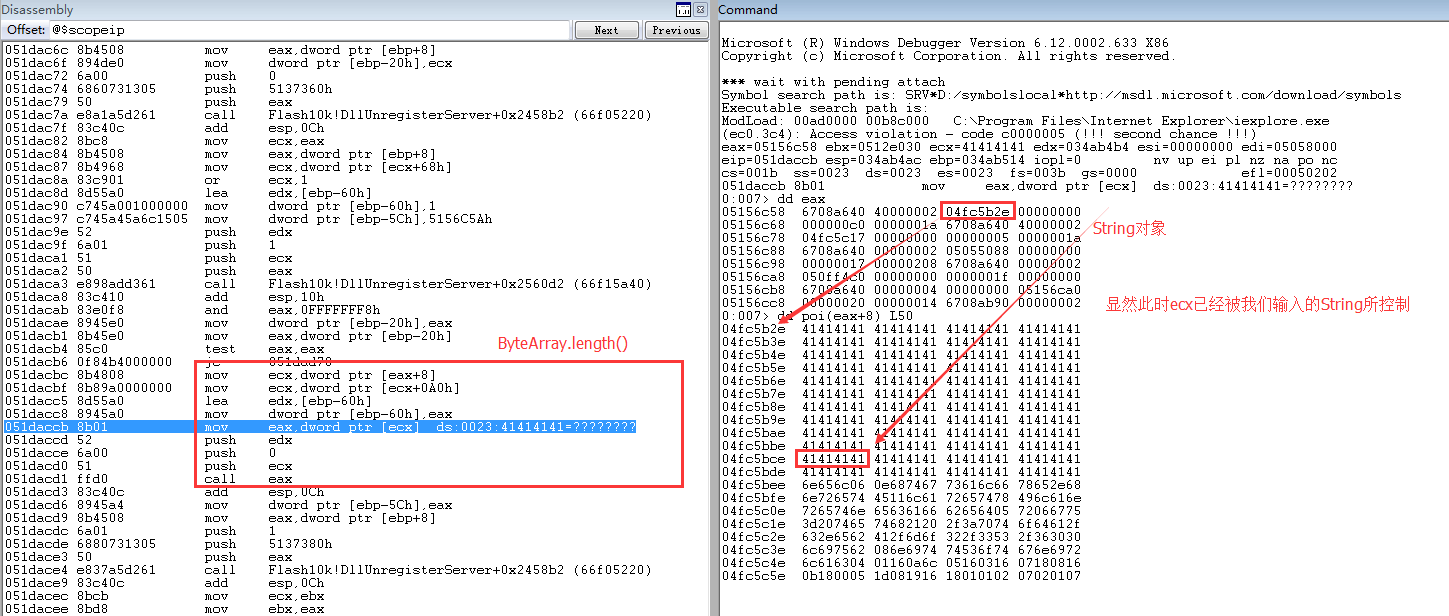




最终通过漏洞成功泄露了vtable和data的地址（shellcode即可存放在data内）。这样就Bypass了ALSR，至于Bypass DEP只要通过vtable的地址向上变量PE头后构建ROP即可，不再详述。

**2）劫持EIP**

获得了可执行权限的shellcode地址后，接下来只需要劫持EIP并将EIP指向shellcode即可。观察最初crash的漏洞现场：



可以发现漏洞现场往下不远处有一个call eax，而eax来自poi(ecx)，而ecx是String数据的第0xA0~0xA3的内容，因此可以把前面获取的m\_buffer放入String第0xA0开始的字符，最终call的是m\_buffer指向的保存shellcode的地址，从而成功执行shellcode。