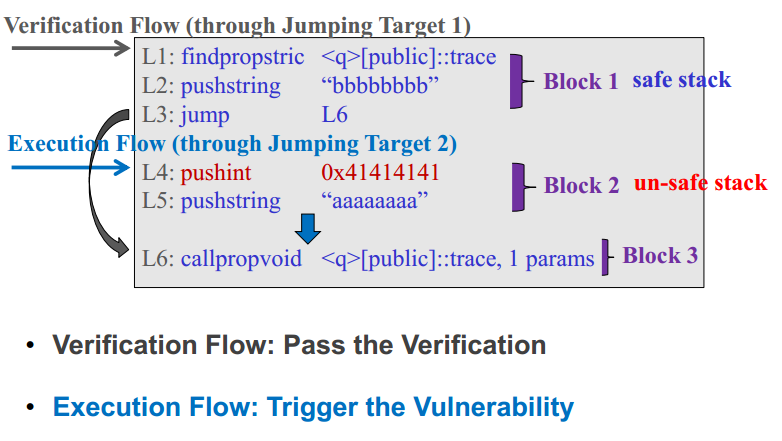
**CVE-2010-3654漏洞利用分析**

**0x00 调试环境**

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

**0x00 背景知识**

AS3 code被编译成为SWF文件（Bytecode），SWF由多个Tag组成，其中DoABC tag里包含的ABC Code由AVM2加载执行，AVM2会根据ABC Code的类型决定是否要进行JIT（一般init()函数不会被JIT）。JIT过程中包括三个流程：1. Verification Process；2. Generation Process；3. Execution Process。其中Verification Process用于检查Bytecode是否安全，通过Verification Process后Generation Process会将Bytecode(ABC Code)编译成JIT code，最后通过Execution Process执行JIT code。由于Bytecode function会被切割成多个block，并通过jmp跳转，所以Verification Process通过验证的分支可能和Execution Process的分支不同，最终导致通过Verification Process但是Execution Process中触发漏洞。



SWF文件的DoABC包含了一个数据结构：常量池（Constant Pool），常量池是ABC文件保存各种常量的地方，而在ABC文件的其他部分，需要使用常量的地方都采用一个uint型的索引指向常量池中的某个元素。常量池包括基本数据类型常量和符号常量，其中符号常量包括Namespace常量和Multiname常量，Multiname常量由指向Namespace的指针和指向字符串常量的Name指针组成，表示一个符号，比如：

类的符号

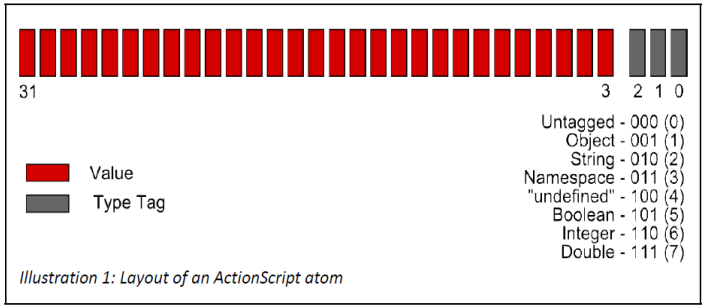
com.orandea.utils:ArrayUtils 表示在Package:com.orandea.utils中的ArrayUtils类

类的方法或者属性

com.orandea.utils.ArrayUtils:wrapFunc 表示类com.orandea.utils.ArrayUtils的wrapFunc方法。

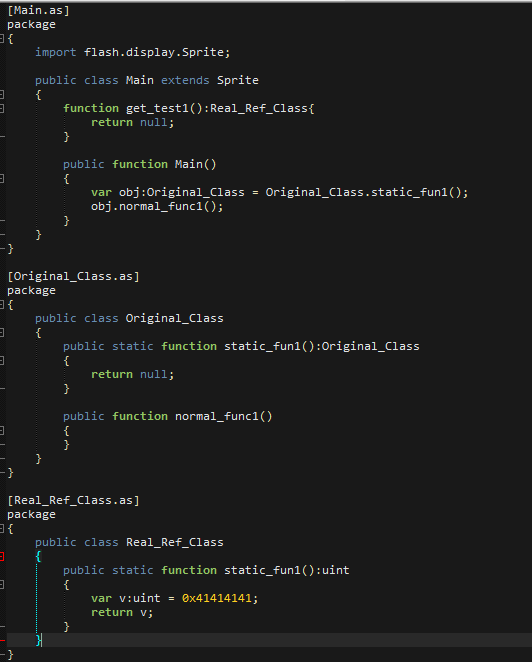
所以可以通过修改这些常量索引实现代码混淆的功能。

AVM为了实现变量的即时类型绑定，将所有变量赋值给一个Atom类型对象，该对象最后三位记录了变量的类型，其余位为变量的值，通过&0xFFFF FFF8即可取出该值：

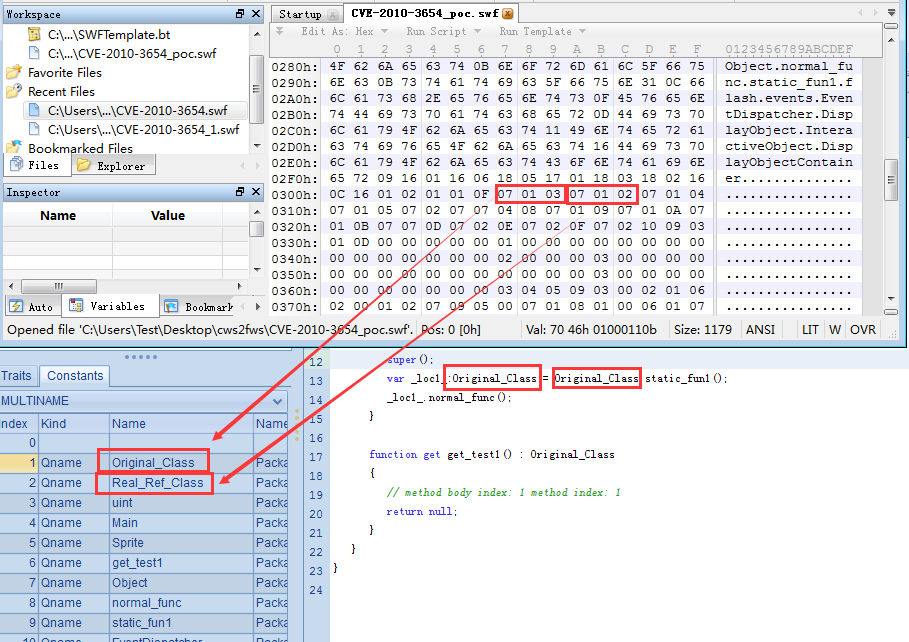


**0x01 漏洞成因**

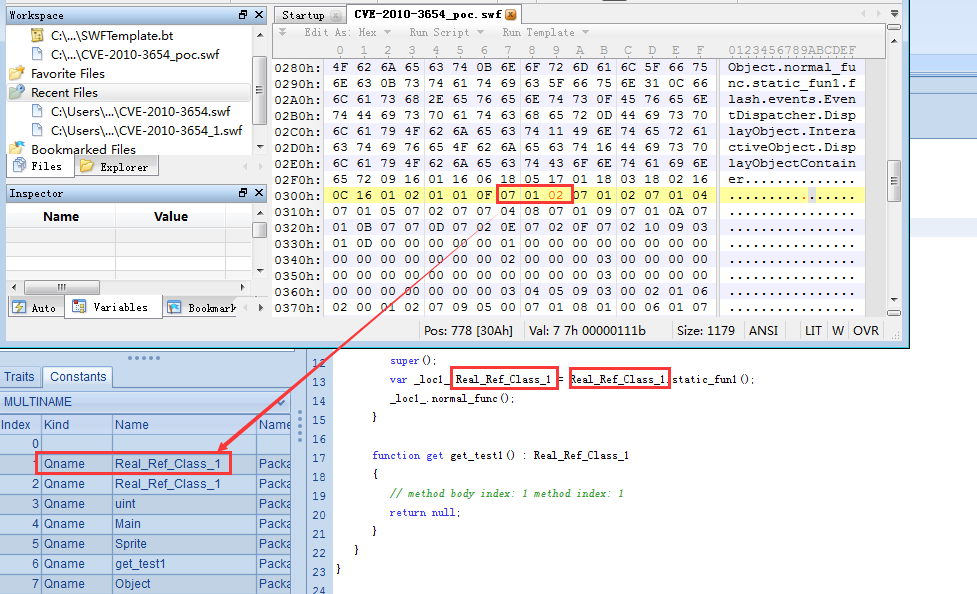
该漏洞是由于avm2对函数返回的类型没有进行校验而触发的类型混淆漏洞，通过该漏洞可以泄露ByteArray对象的VA，并通过Atom类型混淆，实现内存读的功能。触发crash的poc由以下三个类组成：



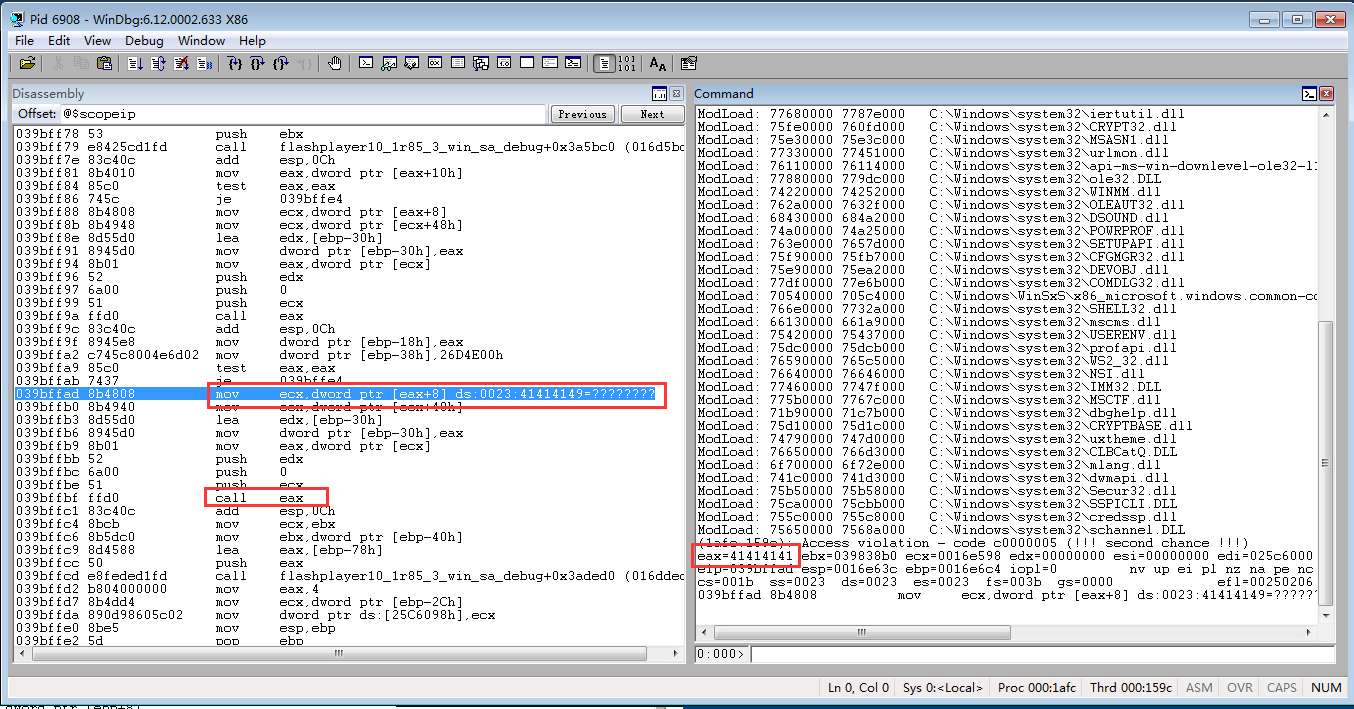
Main.as中通过Origianl\_Class.static\_fun1()返回一个Origianl\_Class类型对象，并调用Origianl\_Class的normal\_func1（）函数。这里通过修改ABC Code的Multiname序号，实现Origianl\_Class和Real\_Ref\_Class类型的混淆：



将Origianl\_Class对应的070103修改为Real\_Ref\_Class对应的070102：



运行修改过SWF，flash Crash:



这里的eax即为Real\_Ref\_Class的static\_func1的返回值，被混淆成了一个对象的调用，从而触发漏洞，通过在crash点不远处有一个call eax，从而可以通过Real\_Ref\_Class的static\_func1的返回值控制EIP。

**0x02 漏洞利用**

现在我们已经可以通过Real\_Ref\_Class成员函数的返回值来控制EIP了，现在需要考虑的就是如果构造一个ByteArray存放shellcode并泄露该ByteArray的VA。

对poc的代码做如下修改：

1. 将Main.as中的

var obj:Original\_Class = Original\_Class.static\_func1();

修改为

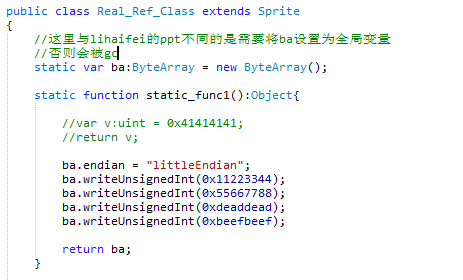
var p\_Atom\_ByteArray:uint = Original\_Class.static\_func1();

并打印该值：

trace("p\_Atom\_ByteArray = 0x" + p\_Atom\_ByteArray.toString(0x10));

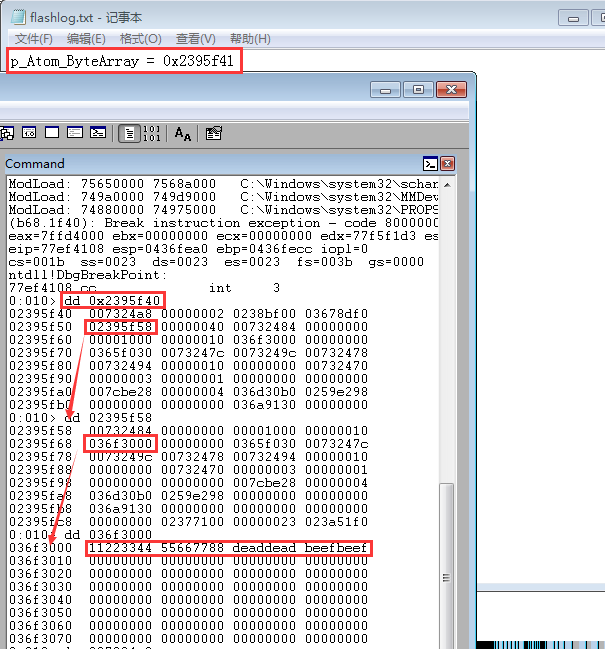
Original\_Class.static\_func1()函数返回值由Original\_Class.static\_func1()修改为uint

2. 将Real\_Ref\_Class的static\_func1做如下修改：



这里static\_func1返回的是BytaArray对象。

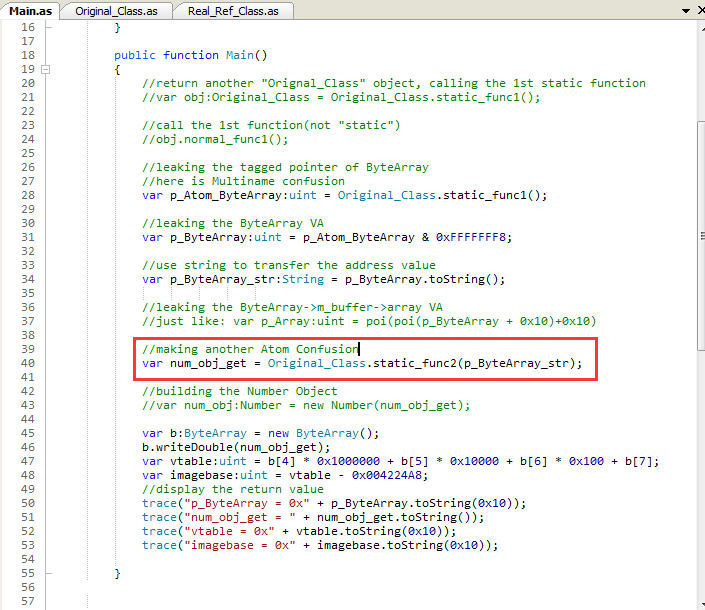
再次编译，修改ABC code Original\_Class的Multiname序号，查看p\_Atom\_ByteArray的值：

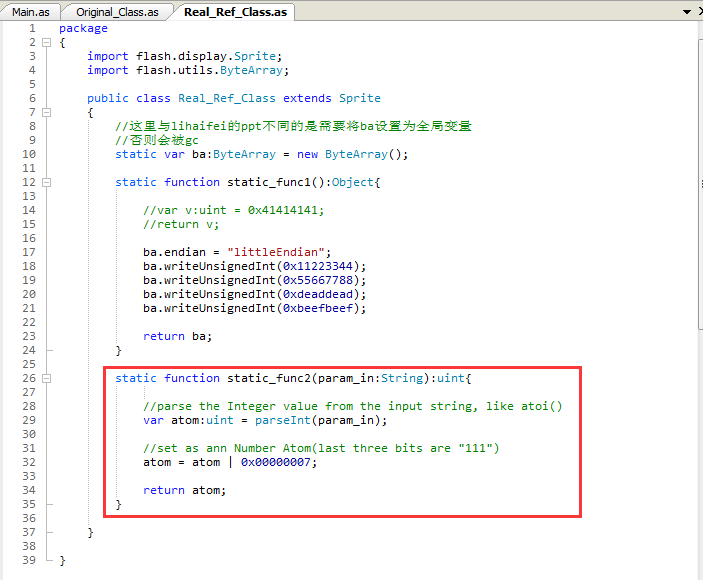


可以发现p\_Atom\_ByteArray就是泄露的ByteArray的地址(因为是Atom，Object最后三位是001)。

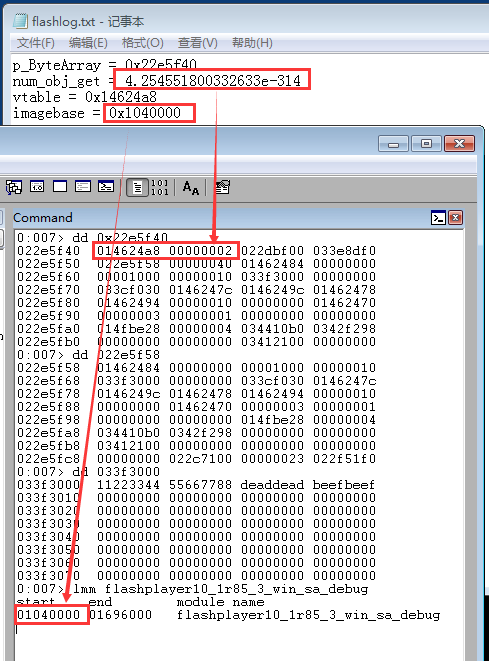
这里我们就得到了用来存放Shellcode的ByteArray的对象的VA 0x2395f40。我们知道ByteArray存放数据的array的地址在poi((0x2395f40+0x10)+0x10)处，下面就需要实现内存的读功能，读取该array的VA以及可以用来bypass ALSR和DEP的flash的基址。

我们知道ByteArray的对象的VA 0x2395f40前四个字节存放了ByteArray的vtable，如果能够读取该vtable的值(0x007324a8)即可找到flash的基址。这里通过Atom里的Double类型实现内存的读取功能（因为AVM在处理Double类型时，其值3～31位是该Double数据的地址，Double数据符合IEEE-754标准，内存），构造方法如下：





这里将泄露的ByteArray的VA转为String，通过Atom类型混淆成Double类型来读取该ByteArray前8Byte的内存（这里并没有通过暴力搜索flash的PE头来获取基址，而是通过减去固定偏移计算，所以不同版本的偏移会不同）：



这里我们就获得了flash的基址，后面ROP需要的gadget都可以由flash获得。

读取ByteArray存放数据的array的地址(poi((ByteArray+0x10)+0x10))的方法与读取ByteArray vtable类型，只需要进行两次Atom混淆即可获得，不再描述。

这样就获得了存放shellcode的ByteArray的array的VA，再由泄露的flash基址搜索rop chain最终可以bypass ASLR和ROP。