原文地址:http://drops.wooyun.org/binary/7485

看着文章不错，顺手翻译了下，如有不足，敬请指正，同时希望能和大家一起交流。 From :http://googleprojectzero.blogspot.com/2015/07/one-perfect-bug-exploiting-type_20.html

对于一些攻击者而言，exp能稳定的利用是很重要的。换言之，exp在运行特定flash版本的特定平台的系统上，exp要每次都能导致代码执行。编写这种高质量的exp的途径之一就是利用高质量的bug(也就是主动因素都利用编写稳定利用的exp).此文讨论的就是利用一个这种类型的bug,并讨论它适合编写稳定利用的exp的因素。

0x00 The Bug

Cve-2015-3077是一种发生在Adobe Flash Button和MovieClip的fliters(是一个包含filter对象的索引数组)属性的setters方法的类型混淆问题，使得任意类型的filter都可以用其它类型的filter来混淆。该缺陷在五月份上报给了Adobe,并在五月得到了修复。该bug的根源在于，攻击者可以覆盖初始化filter对象的构造函数。下面的代码能说明这个问题：

```
var filter = new flash.filters.BlurFilter();
object.filters = [filter];
var e = flash.filters.ConvolutionFilter;
flash["filters"] = [];
flash["filters"]["BlurFilter"] = e;
var f = object.filters;
var d = f[0];
```

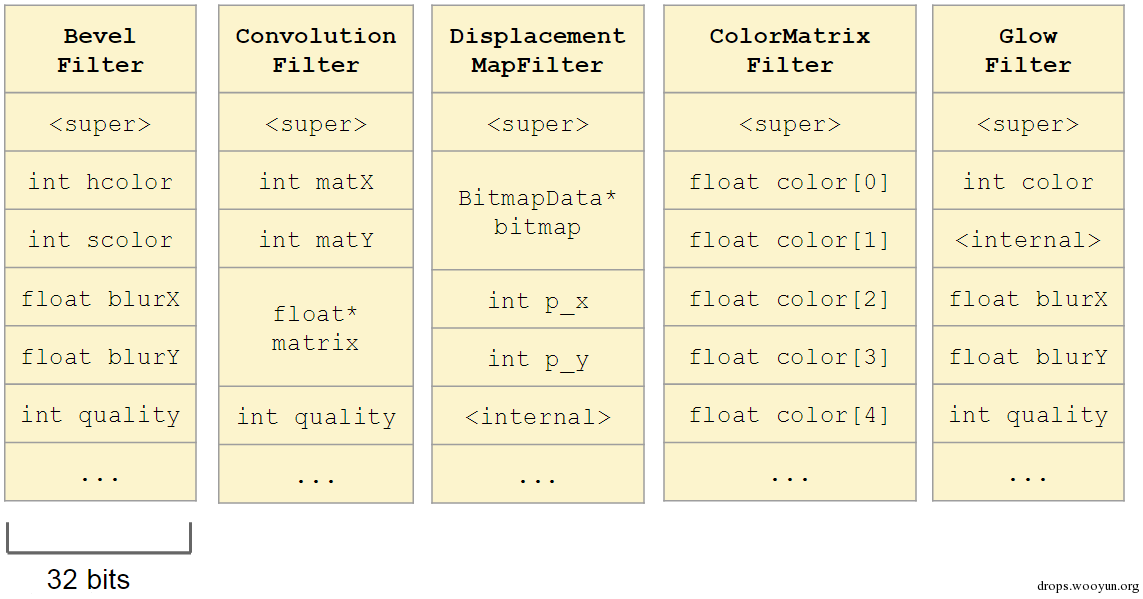
这段代码可能因使用操作符'[]'而让人疑惑，而它在Flash CS编译的时候是需要的。逻辑上等价的代码如下(不保证能编译)：

```
var filter = new flash.filters.BlurFilter();
object.filters = [filter];
flash.filters.BlurFilter = flash.filters.ConvolutionFilter;
var f = object.filters;
var d = f[0];
```

上面这段代码将button对象或者一个MovieClip对象的filters属性设置为了BlurFilters(这当然也代码中object本身是什么类型有关),然后由Flash进行原生存储。BlurFilter的构造函数接着被ConvolutionFilter的构造函数覆盖了。然后filters的getter方法被调用，生成了ActionScript对象存储起初的BlurFilter,然而，构造函数已经被覆盖了，因此调用的是ConvolutionFilter的构造函数。这就导致了原始是ConvolutionFilter对象，却以BlurFilter对象返回。

最终的效果是ConvolutionFilter的域可以被当做BlurFilter来访问(进行读写操作)，其它类型的filter也与此类似。这就为进行利用提供了很多修改的机会。

图1展示了，利用该漏洞能够在64位linux上进行类型混淆的原始对象的内存布局。



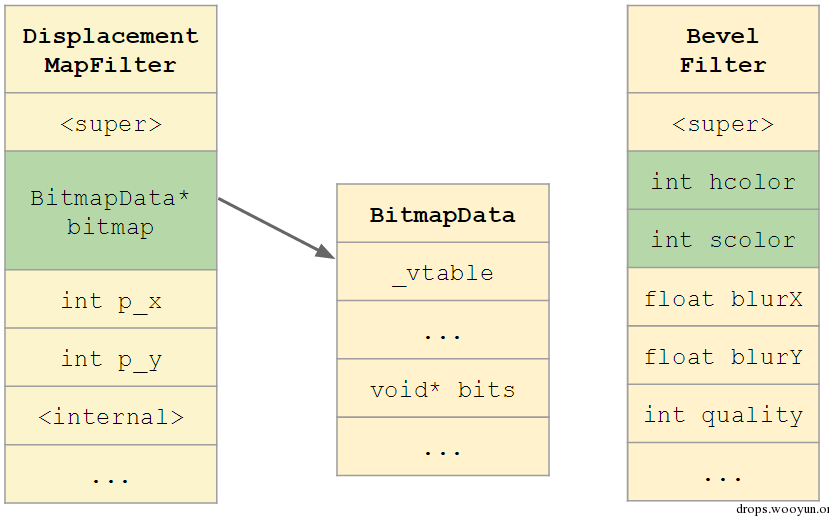
在这两种情形下，指针的占用的空间与多个证书或者浮点占用的空间对齐(如图1中ConvolutionFilter的float* matrix刚好与float blurX和float blurY占用的空间对齐)，这也就意味着，指针能直接进行读写。同样，由于对象的域的顺序和大小都是由类的定义决定的，其位置通常都是可预期的，因此对其进行读写不会失败。这些特性在编写稳定利用的exp时是很关键的。

0x01 The Exploit

由于利用该缺陷可能需要多次触发类型混淆，因此编写了用于类型混淆的功能函数FilterConfuse.confuse.该函数也进行了一些清理操作，如将ActionScript filter的构造函数重新修改为正常的构造函数，这样缺陷函数能够调用多次而不影响ActionScript在该函数外的行为(如果不这样做，很可能就直接崩溃了)。

首先要做的就是确定vtable(虚表)的地址来绕过ASLR技术。理想的方式是对含有虚表的对象使用一个特殊的对象，该对象的一个成员和虚表有交叉，并能被修改，但是所有filter对象的虚表都在同一偏移处。相反，我使用了DisplacementMapFilter的BitmapData对象来确定虚表的地址。

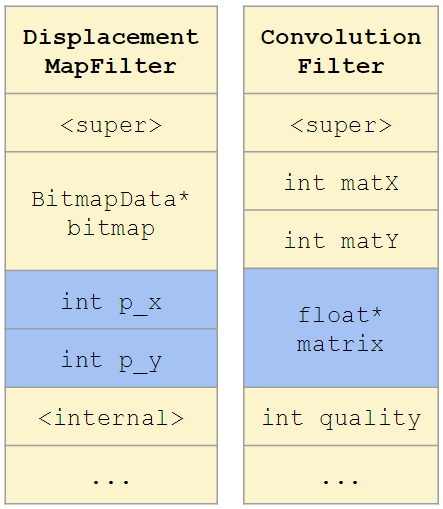
为了确定BitmapData对象在内存中的位置，我用BevelFilter对DisplacementMapFilter进行了类型混淆。结果就是，存放在DisplacementMapFilter中的 BitmapData指针与BevelFilter的颜色属性(shadwoColor,shadowAlpha,highlightColor,hightlightAlpha，同样是从内存分布上而言)是对齐的(占用的空间大小事一样的)。这些属性信息，存放在2个32位整数中(如图2的scolor，hcolor所示)，颜色属性能够访问到每个整数的低24 bit,而alpha属性能够访问到整数的高8 bit.通过读取这些属性并结合位运算，提取BitmapData对象的真实地址是可能的。



drops.wooyun.org

接着，我们需要读出BitmapData对象顶部的虚表。因此，我使用了ConvolutionFilter对象的matrix属性来。该属性的值是一个指向浮点数组的指针，其指向的数组在该属性值设定的时候进行分配。当获取该属性的值的时候，就会返回包含这些浮点数的ActionScript数组。通过将matrix指针指向BitmapData对象，以浮点数组的形式读出对象在内存中的值是可行的(其实就有点类似于C中，`char* buf=(char*)malloc(size),int* buf2=(int*)buf`,这样缓冲区中的数据，可以以char为单位读写，也可以以int为单位进行读写)。

为了设置该指针，我用一个DisplacementMapFilter对象(与上面用的DisplacementMapFilter对象，不是同一个对象实例)混淆了ConvolutionFilter对象，并将mapPoint属性设置为以上BitmapData对象的指针的位置。mapPoint属性是一个含有x,y坐标的点(坐标值为整数，也就是图3的p_x,p_y),并与ConvolutionFilter的matrix是对齐的，这使得设置其值很容易。接着，通过读取ConvolutionFilter对象的matrix 数组(该对象得用DisplacementBitmapFilter进行类型混淆，接着又混淆回ConvolutionFilter对象)，从BitmapData对象读取出虚表地址是可行的。




```
var vtable_low = f.matrix[0];
var vtable_high = f.matrix[1];
```

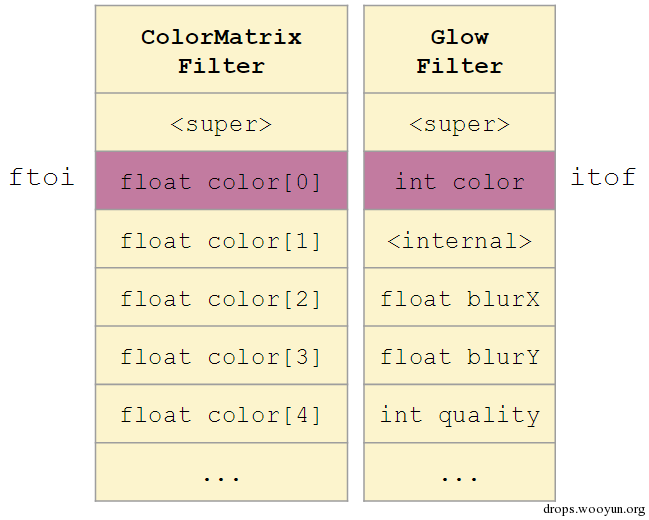
drops.wooyun.org

从这一点上看，由于使用了浮点，代码要能稳定利用就更难了。vtbale_low和vtable_high的值是以浮点的形式从ConvolutionFilter的matrix读出来的(matrix对应的是一个数组类型)。但是不幸的是，不是所有有效的指针都是有效的单精度浮点数(如需要遵守ieee 754标准等等)，这也就意味着读取该值可能会导致NaN(not a number)错误，或者更糟情况下，读取的数值有误。

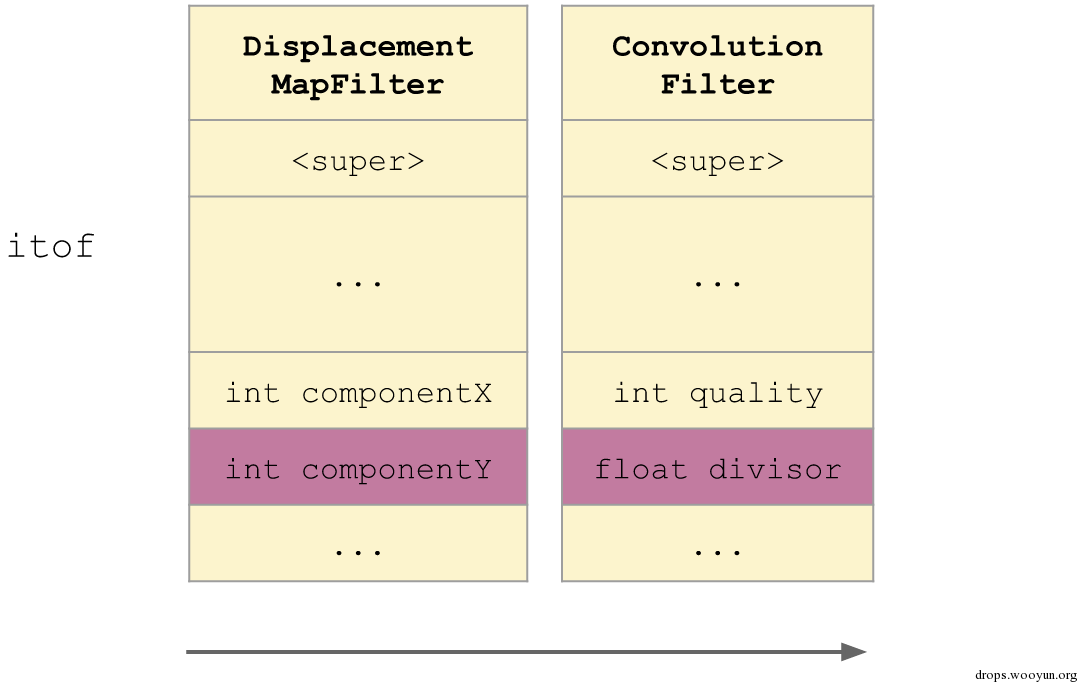
该问题理想的解决方式是通过以整数形式获取的gettter方法来获取vtable_high和vtable_low的值，但是没有这样的getter，因为filter成员由于其功能特性其类型往往是浮点数。

幸运的是，AS2虚拟机在解析浮点数的时候较宽松—它仅仅在ActionScript对其进行操作的时候，才将其内存中的值转换为浮点。原生操作通常并不会导致将其解析为浮点，除非进行了特殊的操作，如使用了它的算数运算。这就意味着，当matrxi数组的浮点数据拷贝到vtable_low和vtable_high时，它会在内存中保持其值，直到被拷贝到ActionScript中并被实际使用，或者以原生代码的形式对其进行了算数运算，即使它不是有效的浮点数。因此，如果变量的值立马类型混淆为不同的类型，并且其值的范围涵盖了32bit能表示的数值，如int，那么可以确定的是在matrix数组对应的内存中其值保持不变。因此为了避免给利用代码带来不稳定性，在ActionScript中操作任何浮点数之前都有必要进行这种类型的类型混淆。

为了达到这一目的，我写了一个转换类—FloatConverter(如图4),它在filters中使用类型混淆来实现整数到浮点和浮点到整数。它使用GlowFilterd的color,alpha属性来混淆ColorMatrixFilter的matrix(内部是一些浮点数组)属性，它能访问到int的不同字节。



虽然这实现了稳定的float到int的转换，但没能实现int 到float的稳定的转换。在ActionScript中`ColorMatrix filter`的color 数组被访问的时候，整个数组都会被拷贝，即使是只访问了其中的第一个元素。当数组被拷贝时，每个元素都被转换为了一个Number(有时包含访问指针，如调用对象的valueof方法)。由于color数组比整个GlowFilter对象占用的空间大，当使用`GlowFilter`混淆的时候它会扩展到堆上(也就是会把堆上的一部分数据也当做GlowFilter的数据)。这也就意味着，转换可以发生在堆的未知的值上，当被转换为Number时，如果他们引用了无效的指针就可能导致崩溃。因此，对于int到float的转换，我实现了一个浮点转换(如图5所示),它在`ConvolutionFilter`和`DisplacementMapFilter`使用了一种不同的混淆—使用的直接的强制类型 转换，不会导致堆上任何未知的值被访问到。



这解决了因访问未知堆值而出现的崩溃，但不幸的是，还有一个涉及浮点的问题关系到利用代码的稳定性。这是由`ConvolutionFilter matrix`的 getter的实现导致的。在ActionScript2中，所有的数字值都是以Number类型存储的，而Number是一个int和一个double指针的联合。原始的`ConvolutionFilter matrix`被存储为一组浮点数组，但是它被复制到了一个ActionScript数组中，因此在ActionScript中能通过matrix 的getter方法来访问它，它的值在这个过程中被转换为了double.接着，当float转换作用在该值上时，他们又被强制转换回了float.

将float强制转换为double，然后又将得到的double转换为float,通常值都能保持不变，除非在一种特殊的情况，单精度浮点值是一个SNaN值。根据单精度浮点标准，有两种类型的NaN(not a number，其实也就是浮点值异常)，QNaN和SNaN。出现QNaN的时候，不会做什么处理，但出现SNaN的时候，在某些情况下会抛出浮点异常。在x86中，将double强制转换为float通常出现的NaN是QNaN(即使double导致的是SNaN)，来避免非预期的异常。

因此一个指针的低bit位恰好是SNaN值,它将会被转换为QNaN值。这意味着其中一个bit位(mantissa的第一个bit,bit 22)将会被置位，而此时本不应该被置位。这个问题在vtable被读的时候可以避免---||||指针的第三个字节（含有能被反转的bit）可以以非对齐的方式读取，来确定其真正的值。因此代码会进行非对齐方式的读取(用Bitmap指针加1后进行第二次读取vtable), 如果浮点恰好是SNaN，则对int值进行校正。.

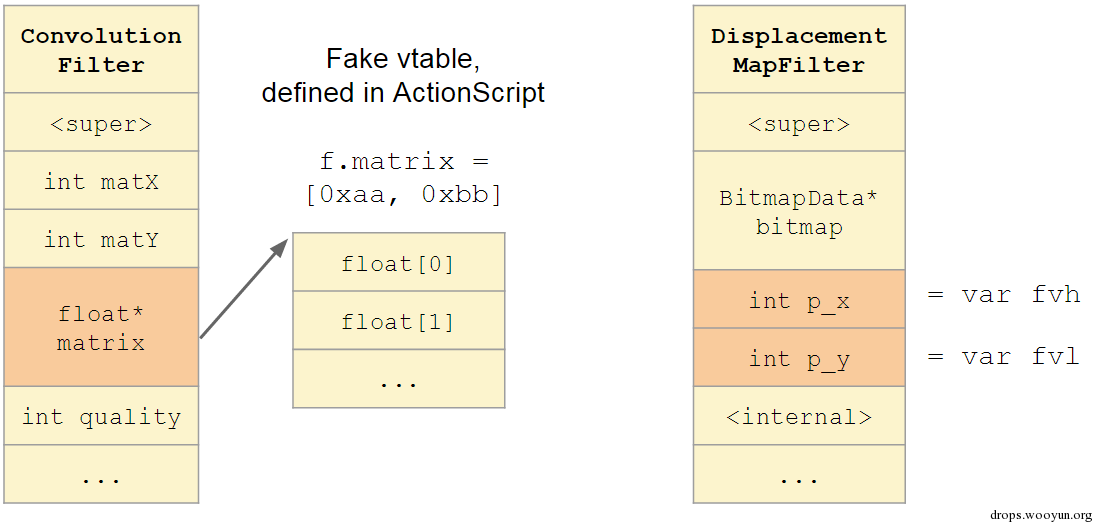
使用上面实现的float转换，vtable的地址可以被转换为一个整数。现在我们需要利用该地址来获取代码执行。一种简单的移动EIP的方式是覆盖一个对象的vtable（或者指针指向的对象的vtable）。这可以通过用`DisplacementFilter BitmapData`指针类型混淆`ConvolutionFilter matrix` 数组来实现。

BitmapData对象包含一系列原生对象。ActionScript对象包含一个指向`BitmapData native object`的指针,而后者又包含指向其他原生对象的指针。其中的一个对象是bits 对象,它包含实际的bitmap bit数据。这个bits 对象包含许多虚方法，这些虚方法通常是任何操作作用于BitmapData对象时首先调用的方法。为了对该特性进行利用，Exp伪造了一个含有指向伪造的bits对象的指针的BitmapData对象，然后调用了一个方法，该方法会导致对伪造的bits对象调用虚方法。

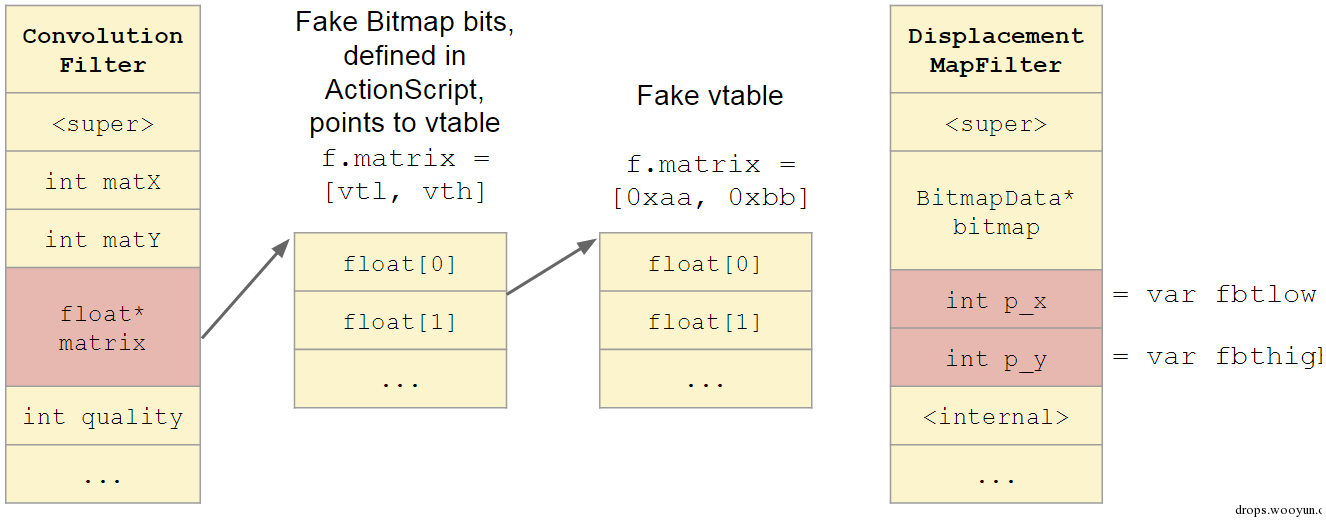
通过使用ConvolutionFilter的属性matrix的setter方法，可以为浮点数据分配任意大小的缓冲区。而缓冲区的位置，可以通过用`DisplacementMapFilter`类型混淆`ConvolutionFilter`,并结合使用`DisplaementMapFilter`的mapPoint属性来确定，操作有点类似于读取vtable的位置。由于分配的数组是不变的，首先就得伪造一个vtable对象，和一个指向vtable的bits对象，最后生成一个伪造

的bitmap指向它。

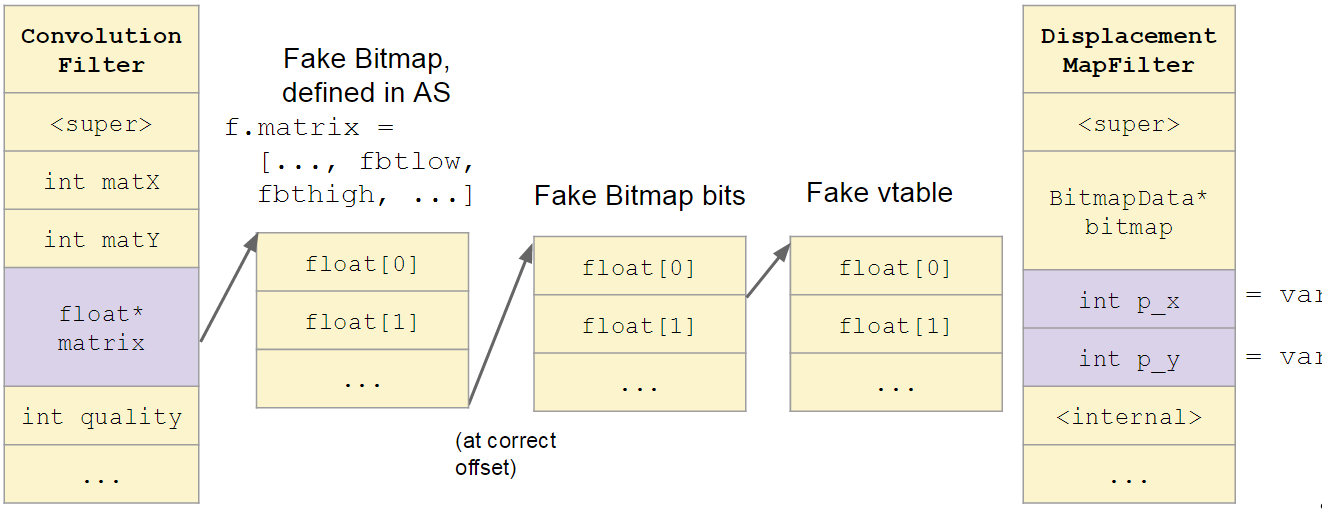
第一步是生成一个伪造的vtable,并使用ConvolutionFilter/DisplacementMapFilter混淆来确定其位置。



接着，以同样的方式伪造一个bitmap bits。



接着，以同样的方式伪造一个bitmap bits。



通过设置DisplacementMapFilter对象的BitmapData对象为指向伪造的bitmap的指针，并使用BevelFilter进行混淆，设置color属性为指针的值(读取原始BitmapData对象的位置逆过程)，可以在

ActionScript中获取到对伪造的bitmap的引用。这个对象接着又利用类型混淆混淆回`DisplacementMapFilter`,`BitmapData`对象可以通过调用mapImage 的getter方法进行访问。那么无论何时对象包含虚方法调用的方法被调用(如setPixel32)，该方法都会调用由伪造的vtale确定的函数地址。

从这一点上看，值得细究伪造的vtable中具体的内容。前面的float转换忽略了SNaN：写float。`ConvolutionFilter.matrix`的setter也转换float到double,然后在写之前又将double转换回float,因此如果一个指针值碰巧是SNaN值，那么写回matrix数组时，bit 23会被置位，即使在原始值中没有置位。这可以利用受限的非对齐写来避免。

在内存中，一个SNaN指针布局如下：

```
00: XX XX YY QQ XX ZZ 00 00
```

其中： xx：可以为从00到0xFF的任意值 YY:bit 5被置位，其它无限制 QQ:除了第7bit，其它所有bit位为1 ZZ:第7bit被置位1，其它bit位无限制

它可以作为32bit的单精度浮点数对其进行写，如下：

```
00: 00 XX XX YY
04: QQ XX ZZ 00
08: 00 00 00 00
```

这就保证了，如果原始的指针值是一个SNaN值，所有非对齐值都不会是SNaN(因为如果原始浮点是一个SNaN值，YY的第5bit会恒为0)。必须使用两个常量指针才能达到这一目的(除非这两个事先都知道是SNaN值)，布局将是下面这样的：

```
00: 00 XX XX YY
04: QQ XX ZZ 00
08: 00 XX XX XX
0C: QQ XX ZZ 00
10: 00 00 00 00
```

位于0x08处的浮点，bit22到bit31没有限制，因此最终可能会是一个SNaN值，并被不正确的写。

因此写任何指针到浮点数组是可能的，不论它是否是SNaN值，但仅仅能被做一次。在原始指针被写后，如果原始指针是SNaN值，那么其它的指针都应该是SNaN值，如果原始值不是SNaN值，那么所有其它的指针都不能是SNaN值。该exp处理这一限制的方式是仅仅写单个指针到`Convolution matrix`数组中。

对于伪造的Bitmap和伪造的Bitmap bits,这是很容易实现的，因为他们仅仅需要包含一个指针。难点在于伪造的vtable只能包含一个指针，这很难用于为函数调用建立参数，并调用函数。

好的解决办法是在首次调用了vtable后移动到内容完全可控的缓冲区中去。好消息是，BitmapData(伪造的bitmap模拟的)类的paletteMap方法创建了一个这样的缓冲区。 该方法有四个参数(`redArray`,`greenArray`,`buleArray`,`alphaArray`),这四个参数都是`ActionScript Numbers`型的数组。当该方法被调用的时候，他们被转换为整数，并被拷贝到了原生的的int型数组中去。接着另一个含有四个指向这个数组(指向输入数组的特定的偏移处)的指针的原生函数被调用。该方法调用虚方法，跳转到伪造的vtable.

作为初始调用的一部分，数组指针存储在x64的r12,r13,r14,r15寄存器中。指针指向了四个可以控制的缓冲区，这点很有用。伪造的vtable中的单个指针实质是如下内容：

```
mov rdi, r13
call [r12]
```

r13指向的缓冲区的内容为"gedit",r12对应的缓冲区的内容为一段Flash库中调用系统方法的gadget(不关心SNaN)。最终的结果就是，当虚函数调用到了伪造的vtable ，gedit被调用。

exp到此为止，虽然没有优雅的退出(当gedit退出的时候，Flash player会崩溃)。优雅的退出的关键在于多次在这四个缓冲区上调用函数。即使这里也处理好了，也不能幸免于Flash Player的销毁(如tab关闭了，或者swf刷新了)。因为调用了filter对象的析构函数会导致崩溃，因为指向`ConvolutionFilter matrix`数组的指针被类型混淆为指向`BitmapData`的对象的指针。这些对象分配在不同的堆上，因此当对一个对象调用删除而其本身实质是另一个对象时，会导致崩溃。这个问题并不能通过通过类型混淆到一个好的状态来完成，因为类型混淆在这个bug中创建了对象的一个副本，因此原始的有问题的对象任然存在，任然需要被释放。同样也不能通过在原始对象上设置参数来完成，因为BitmapData对象，matrix对象的setter会在进行创建之前试图删除已经存在的对象。当exp在运行中并避免这个问题是可能的，只要player保持打开，通过获取到这些对象的引用，这样对象就不会被释放掉。但是当player销毁的时候，这个崩溃还是会出现。当然，exp执行的代码是可以避免崩溃的，可以通过在内存中校正类型混淆对象，把他们置于正确的状态，亦或将其析构函数的指针指向为空。该段代码中并没有实现这个。

0x02 What Makes this Bug Reliable?

虽然类型混淆通常是可以利用的，还有其他的因素使得`cve-2015-3077`能够被很稳定的利用。

当类型混淆被触发，通常包括两种类型，实例化时的有缺陷的对象，漏洞触发后混淆的类型。原始类型的对象成员如何对其混淆的对象成员，对exp的可利用性和利用的稳定性有很大的影响。在这种情形下，原始缺陷类型成员(如指针)以混淆类型成员进行排列，攻击者可以直接进行操作，反之亦然。这是导致exp能可靠执行的最理想的情形。另一个常见情形是原始对象成员的大小超过了混淆对象成员的大小，他们的值由内存越界区域的值决定。这种情形通常也是可以利用的，尽管利用不稳定，因为很难保证对象的堆以预期的方式排列。另一种可能是，对象操作的方式受限，例如，原始对象成员仅仅能设置为一些受限的值，而混淆的对象又是只读的，或者只有一个特定的方法调用使用它。这种情形下，是否能利用，取决于bug本身的特性，也能导致如需要结合其他bug综合利用的信息泄露型bug。这种类型的bug也可能能稳定的利用，但需要那种运气好的bug，碰巧有对的成员又有对的数值。

这个bug的另一方面是，类型混淆发生在导致类型混淆的缺陷函数调用的最后。这很关键，因为这意味着，一个对象可以被混淆，并且不会以攻击者不期望的方式被修改。一些类型混淆的bug不能利用或者不能稳定利用，因为类型混淆后调用的方法是用了混淆的对象，而该值应该是有效的，而后面使用的时候此处又不是有效的。值得注意的是，在以上的利用代码中，其成员发生类型混淆的MovieClip对象，被设置为了0*0纬。这能避免那些对filter对象的特定调用导致不可稳定利用，因为filter没必要作用于没有像素数据的对象。

同样，在这个bug中，攻击者没有进行访问的对象成员也没有被软件使用。这是另一个影响类型混淆型bug能否利用以及稳定利用的因素。有时，对攻击者无用的成员(从攻击者角度看)能导致崩溃，如果攻击者不能修正它的值的话。同样，MovieClip设置为了0*0纬避免了该问题，因为Flash 不能对没有像素的图像使用filter方法。

最后，保持对原始对象和混淆对象的引用是很重要的，因为它阻止了垃圾回收，当垃圾回收作用于类型混淆对象时通常都会导致崩溃。垃圾回收以一定的方式认为对象成员是有效的，如包含有效的指针，如果指针无效将会导致崩溃，攻击者通常也很难对这种情形进行校正，因为垃圾回收可以发生在任意时间，因此任何无效的窗体也是一个问题。完全可靠的解决方式是，当对象有效的时候阻止垃圾回收。

0x03 Conclusion

很多因素，包括原始对象的布局，混淆对象的成员，对象时何时以何种方式使用，以及该对象时候容易遭受垃圾回收都能影响到类型混淆漏洞利用的稳定性。Cve-2015-3077由于结合了以上因素是一个高质量的bug，能够被稳定的利用。利用该代码需要触发该bug 31次：其中8次用户设置和获取利用需要的对象成员，19次到第23次用于float转换，这依赖于发生了多少SNaN值。虽然次数看着有点多，但利用是稳定的，因为每一步都是确定的，并不依赖于那些不一定会发生的行为。

参考链接：

【1】：Issue 254: Adobe Flash: Type Confusion in Button.filters https://code.google.com/p/google-security-research/issues/detail?id=254

【2】MovieClip.filters property http://help.adobe.com/en_US/AS2LCR/Flash_10.0/help.html?content=00001293.html