**CVE-2015-5119漏洞利用分析**

**0x00 调试环境**

|  |  |
| --- | --- |
| OS: | win7 x86 |
| IE: | ie10 |
| Flash: | flashplayer13\_0r0\_182\_win\_debug |

**0x01 漏洞成因**

2015年7月，Hacking Team被黑，多达400G数据被泄露，其中就包括flash 0day CVE-2015-5119的漏洞利用程序。其漏洞利用手法精妙，包括微量的heap spray、精准的定位、无ROP bypass DEP CFG等，均值得学习。

CVE-2015-5119是一个存在于AS3 ByteArray类的UAF漏洞，首先创建一个ByteArray对象：

var ba:ByteArray = new ByteArray();

ba.length = 8;

ba[1] = 1;

可以通过索引访问ba[]的元素，也可以将numeric byte写入ba[]，当然也可以将一个对象写入ByteArray，比如：

var obj = new MyClass();

ba[0] = obj;

AS3会隐式调用MyClass.valueOf()方法将MyClass对象转换成numeric值赋给ba[]。MyClass.valueOf()可以通过用户代码简单重定义：

class MyClass

{

prototype.valueOf = function()

{

ba.length = 88; // reallocate ba[] storage

return 0; // return byte for ba[offset]

}

}

现在来看下native code中隐式调用valueOf()是什么样的：

push esi

mov eax, [esp+8] // the offset value from "ba[offset] = obj"

push eax

add ecx, 0x18 // ecx = this = "ba" object pointer

call ByteArray.getStorage() // gets ba[offset] storage pointer and

mov esi, eax // saves it in esi

mov ecx, [esp+0xC] // "obj" pointer

push ecx

call AvmCore.toInteger() // call MyClass.valueOf()

add esp,4

mov [esi], al // writes returned byte into array

pop esi

ret 8

可以看到通过call ByteArray.getStorage()获取到ba[offset]的指针赋给了局部变量esi，此后通过call AvmCore.toInteger()获取valueOf()的返回值赋给esi指向的地址。

高级语言是这样的：

void ByteArray.setObjInternal(int offset, obj)

{

byte\* dest = this.getStorage(offset);

\*dest = toInteger(obj);

}

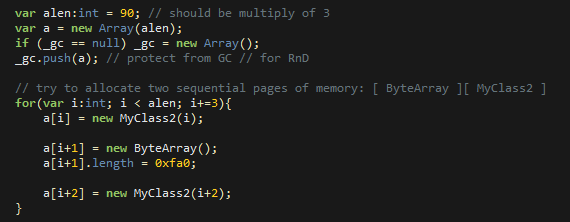
ba[offset]的指针保存在局部变量中， AS3的valueOf()被native code调用后会将返回值赋给局部变量。由于valueOf()可以在用户代码中重定义，如果通过valueOf()修改ByteArray的长度，那么局部变量保存的指针指向的地址将被释放，该指针成为了一个悬挂指针，从而存在UAF的内存破坏漏洞。

**0x02 漏洞利用**

UAF漏洞利用的关键 在于“占位”，形成悬挂指针后，需要申请与悬挂指针指向的已被释放的内存空间相同大小的内存进行“占位”，从而可以通过操作悬挂指针来破坏占位内存数据，最终达到利用效果。而占位的关键则在于HeapSpray。

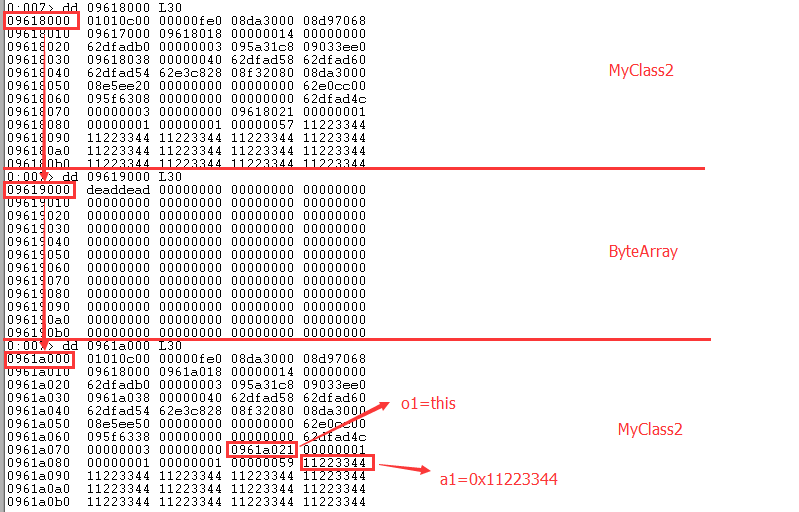
**1) HeapSpray**

Exploit程序通过如下代码进行HeapSpray：

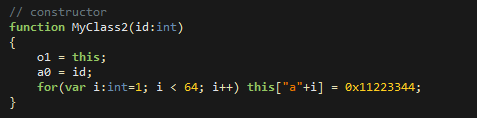


因为HeapSpray需要申请/释放大量内存，为了防止垃圾回收导致的内存布局的不可预测，exploit程序使用了全局静态成员变量\_gc来保存申请的内存。

常规的Flash HeapSpray一般是通过Vector或者ByteArray来实现，这里是通过自定义的MyClass2类的来实现，MyClass2继承MyClass1，MyClass1则继承了ByteArray。注意到MyClass2包含a0～a899 900个uint变量，继承的MyClass1则包含o1~o4 4个Object类型成员，因此MyClass2一共占用： [ MyClass2自身的结构 0x78] +[4个Object:4\*4]+[900个uint:900\*4] = 0xE98，最后对齐到0x1000字节的内存块。同时申请的ByteArray长度为0xfa0，也对齐到0x1000，从而和MyClass2内存对齐。HeapSpray完后内存的局部布局如下图：



注意到MyClass2的构造函数：



o1用来存放MyClass2实例对象地址的Atom值，a0~a63用来定位能够被利用的MyClass2。

**2) 空洞**

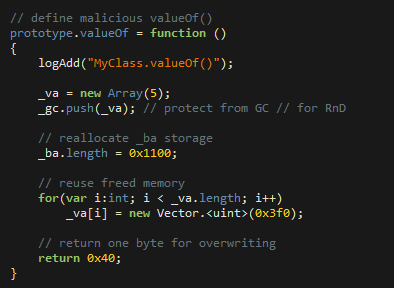
HeapSpray完后内存布局为：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| MyClass2 | ByteArray | MyClass2 | MyClass2 | ByteArray | MyClass2 |

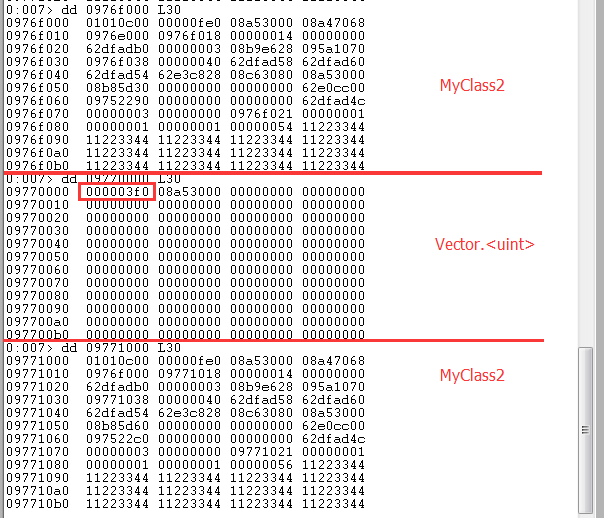
通过漏洞成因分析可知，触发漏洞后我们可以控制一个指向ByteArray内存地址的悬挂指针，因此需要释放ByteArray现在的内存空间，形成空洞，并用相同内存大小的Vector.<uint>来占位。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| MyClass2 | Vector.<uint> | MyClass2 | MyClass2 | Vector.<uint> | MyClass2 |

exploit程序在MyClass.valueOf()方法中进行的空洞及占位操作：

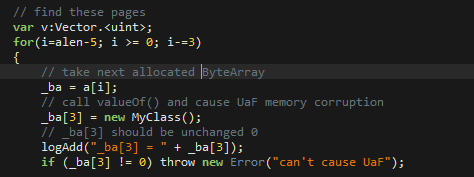


首先同HeapSpray一样将\_va用\_gc保存防止GC，接着通过修改\_ba.length来释放现在的\_ba内存空间，最后申请了5个Vector.<uint>(0x3f0)占位空洞（8+0x3f0×4=0xFC8对齐到0x1000）。

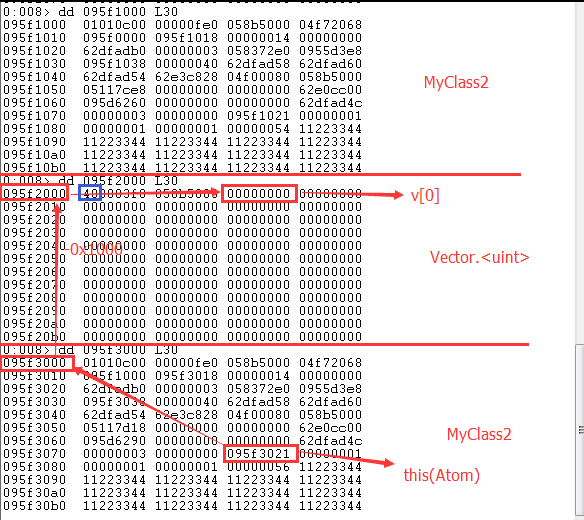


**3) 触发和定位**

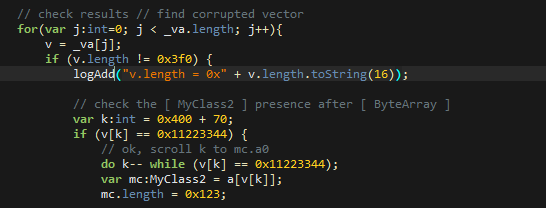
触发漏洞的代码如下：



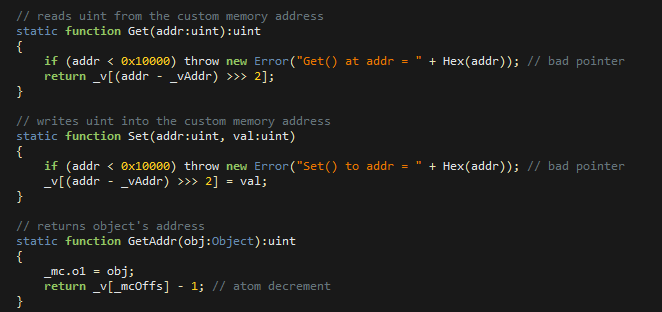
考虑到低地址空间内存申请的碎片化导致的不连续性，exploit程序从高地址空间开始尝试触发漏洞，\_ba = a[i]是将之前申请的ByteArray赋值给\_ba，\_ba[3] = new MyClass()触发漏洞，导致MyClass.valueOf()的返回值0x40并没有赋值给新申请内存空间的\_ba[3]而是赋值给了悬挂指针指向地址的第四个字节，也就是占位Vector.<uint>(0x3f0)的第四个字节，从而将Vector.<uint>(0x3f0)的长度篡改为0x400003f0：



获得了一个长度为0x400003f0的Vector.<uint>后，接下来需要定位该Vector.<uint>，泄露v[0]的VA，这里通过MyClass2对象写入的0x11223344即可完成定位：



首先对占位的\_va[j]的长度判断，不是0x3f0的即为被篡改的\_va[j]。var k:int = 0x400 + 70; k为v的索引，0x400 + 70可以保证v[k]访问到MyClass2空间的0x11223344某一个值，再通过do k-- while (v[k] == 0x11223344);精确定位k=a0（a0存放的是HeapSpray MyClass2的索引）,同时注意到v[k-4]存放的是o1也就是MyClass2的this指针的Atom值，由于Vector.<uint>与MyClass2相邻排布，那么（v[k-4]-1）& 0xfffff000 – 0x1000 + 8即为v[0]的VA。有了0x400003f0长度的v和泄露VA的v[0]，就可以构造任意地址读写的函数了：



**4) 绕过DEP**

有了任意地址读写的权限后，一般的利用手法是：

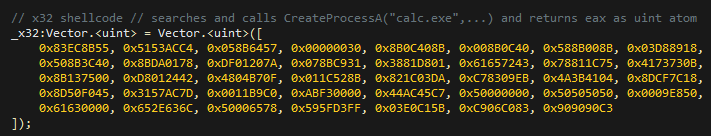
任意地址读写Vector=> 保存shellcode => 修改虚表指针指向shellcode=> 调用虚函数执行shellcode

在开启DEP后，shellcode所在的内存由于没有执行权限导致shellcode无法执行成功，需要借助VirtualProtect修改shellcode所在的内存属性为可执行，绕过DEP的一般流程是：

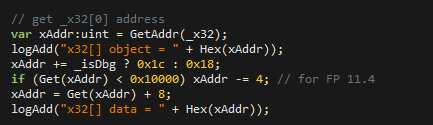
任意地址读写Vector=> 读取flash虚表指针=> flash基址 => flash导入表 => VirtualProtect地址 => 构造ROP

最后需要通过stack\_pivot(xchg eax,esp;retn)将构造的ROP保存到栈中执行。

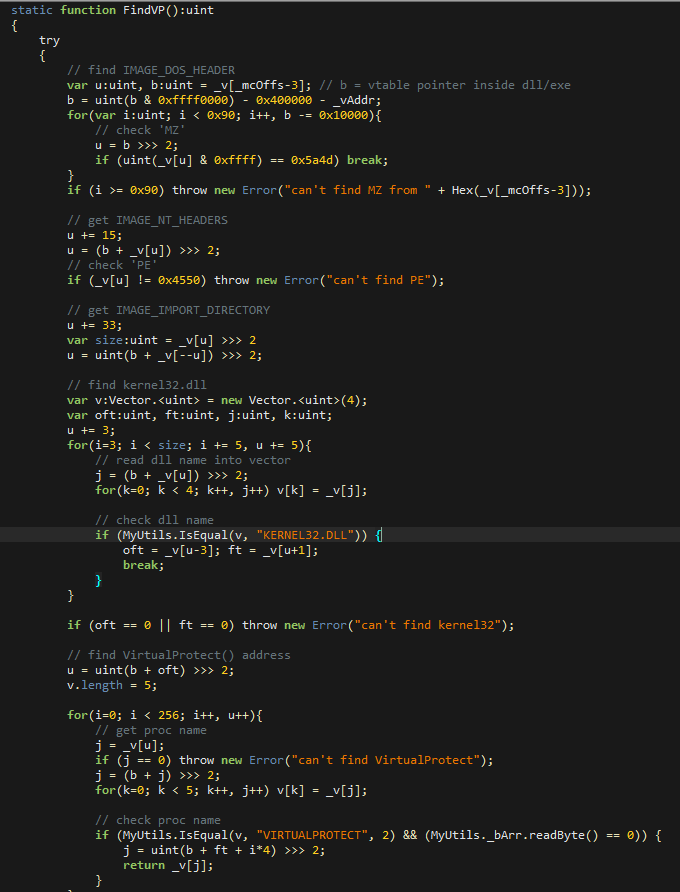
Exploit程序首先将shellcode保存至\_x32中：



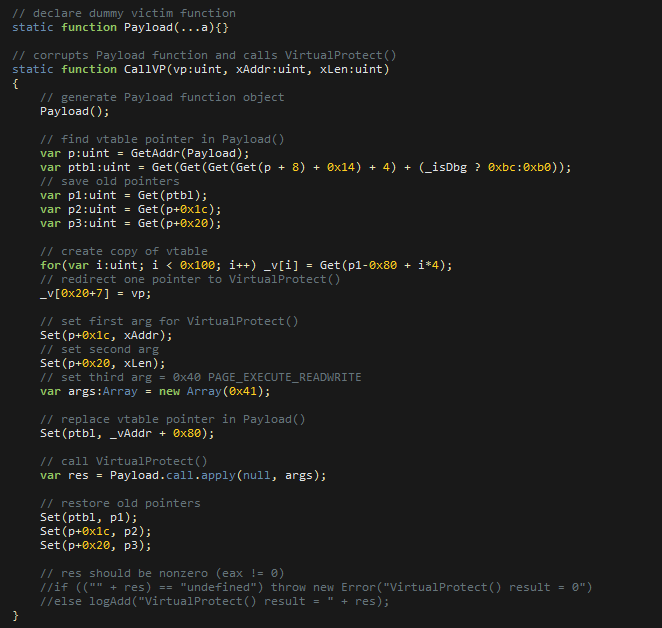
再通过3)中的函数读取\_x32的VA：



接着VirtualProtect的VA则是通过泄露的flash虚表指针（MyClass2的this指针向前偏移3个索引处）暴力搜索PE头获得：



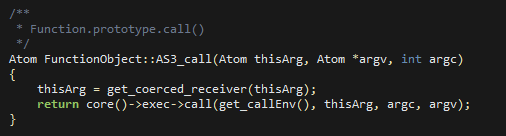
获得VirtualProtect的VA后，一般做法是构造ROP，在通过Sound.toString()等一些容易定位的函数劫持EIP，调用VirtualProtect开启执行权限执行shellcode，但是Hacking Team的exploit程序却写的相当巧妙，省去了复杂的ROP：



CallVP()函数的代码不如一般ROP利用代码的简单直观，通过注释大概思路是先取得Payload 函数对象的vtable替换某一虚函数为VirtualProtect API地址，最终通过Payload.call.apply来调用VirtualProtect API。

具体原理：

Payload是FunctionObject类型，调用Payload.call的时候，native code最终会调用FunctionObject::AS3\_call：



其内部会调用core()->exec()->call(get\_callEnv(), thisArg, argc, argv);也就是说Payload.call会跳转到到call(get\_callEnv(), thisArg, argc, argv)并最终跳转到Payload对象JIT出来的代码，由于call里的参数个数和VirtualProtect参数个数一样，所以可以通过修改call函数的压栈参数实现VirtualProtect的调用。再来看下FunctionObject::AS3\_call的汇编代码：

|  |
| --- |
| int AS3\_call(void \*this, int thisArg, int \*argv, int argc)  push ebx  push esi  mov esi, ecx  mov ecx, [esp+8+thisArg]  mov eax, [esi]  mov edx, [eax+8Ch]  push edi  push ecx  mov ecx, esi  call edx  mov ebx, eax  mov eax, [esi+8]  mov ecx, [eax+14h]  mov edx, [ecx+4]  mov eax, [esi]  mov edi, [edx+0B0h]  mov edx, [eax+90h]  lea ecx, [esp+0Ch+thisArg]  push ecx  mov ecx, esi  call edx  mov ecx, [esp+0Ch+argv]  mov eax, [eax]  mov edx, [edi]  mov edx, [edx+1Ch]  push ecx //forge to lpflOldProtect <= argv  mov ecx, [esp+10h+argc]  push ecx //forge to flNewProtect <= argc  push ebx //forge to dwSize <= p+0x20  push eax //forge to lpAddress <= p+0x1C  mov ecx, edi  call edx // forge to VirtualProtect VA  pop edi  pop esi  pop ebx  retn 0Ch |

该段代码中有3次 call分别对应了core()->exec()->call()，观察最后一个call edx，此时edx中应该保存的就是call()的函数地址，跟踪edx，如上标蓝片段，即可知道Payload对象vtable：

var ptbl:uint = Get(Get(Get(Get(p + 8) + 0x14) + 4) + (\_isDbg ? 0xbc:0xb0));

的由来了，同时call()函数位于该虚函数表偏移0x1C处。

获得了虚表的VA后，exploit程序备份了虚表VA前后0x400 bytes的数据，然后修改[vtable+0x1c]= VirtualProtect，并将压栈参数修改为VirtualProtect需要的参数，所以最后在AS3层调用VirtualProtect就变成了：

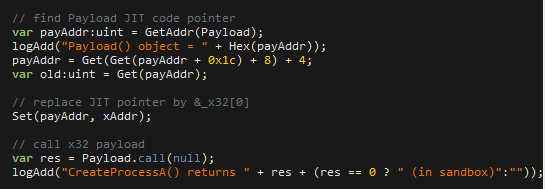
|  |
| --- |
| // set first arg for VirtualProtect()  Set(p+0x1c, xAddr);  // set second arg  Set(p+0x20, xLen);  // set third arg = 0x40 PAGE\_EXECUTE\_READWRITE  var args:Array = new Array(0x41);  // replace vtable pointer in Payload()  Set(ptbl, \_vAddr + 0x80);  // call VirtualProtect()  var res = Payload.call.apply(null, args); |

观察HackTeam用的代码中并没有直接call，而是call.apply，其实是仅仅为了书写方便而已，写成Payload.call(null,null,…null)，共0×41个null也是可用的。之所以是0×41而不是0×40，是因为第一个null为调用者而不是函数参数。call(object, param1, param2…)和apply(object, new Array(param1,param2…))是等价的。args按说完全是参数数组了，长度还要设置为0×41是因为apply的调用者为call，而object指针不会从apply向call传递。在call看来传过来参数包含了object指针和参数，所以数组的第一个参数就作为了object指针，也就是这种传递过程会吃掉一个参数。

调用VirtualProtect后再恢复修改过的虚函数表和Payload内存就可以执行shellcode了。

**5) 执行shellcode**

修改了shellcode内存的执行属性后，下面就需要劫持EIP并将EIP指向shellcode，其实劫持EIP的方法在4)中已经阐述，即修改Payload函数对象vtable，并通过Payload.call劫持EIP，但exploit程序却另辟蹊径，通过如下代码来执行shellcode：



这里比较难以理解的是：payAddr = Get(Get(payAddr + 0x1c) + 8) + 4;

Payload对象+0x1C存储了MethodEnv指针，MethodEnv的+8指向MethodInfo，MethodInfo+4就是\_implGPR，存储JIT后的代码地址。因此payAddr存放的就是Payload对象JIT后的地址，从而实现了EIP劫持。虽然触发条件仍然是Payload.call，但上次是替换了ExecMgr的虚函数call，而这次是直接替换了JIT后的代码地址。JIT代码调用不在CFG监测的范围，而ExecMgr的虚函数表可能是过于频繁使用，出于性能考虑，也没有CFG的守护。

其实用JIT的方法调用shellcode的好处还在于，shellcode可以可以返回需要的结果让AS3获取以指导下一步操作，比如EAX存储了CreateProcessA的执行结果，返回前构造一段类似下面的代码，让EAX转化为atom：

04DDE32D SHL EAX,3

04DDE330 ADD EAX,6

04DDE333 LEAVE

04DDE334 RETN

res = Payload.call(null)后，AS3可以根据进程创建情况判断沙箱的限制，以决定是否进一步部署内核漏洞利用代码进行权限提升。

最终exploit效果：

