简介

SpringMVC的一个目录穿越漏洞，或者说目录遍历漏洞。SpringMVC是Java开发界最常用的框架之一，当静态资源存放在Windows系统上时，攻击可以通过构造特殊URL导致目录遍历漏洞

影响版本

- Spring Framework 5.0 to 5.0.4.
- Spring Framework 4.3 to 4.3.14
- 已不支持的旧版本仍然受影响

漏洞利用条件

- Server运行于Windows系统上
- 要使用file协议打开资源文件目录

环境搭建&漏洞复现

```
git clone https://github.com/spring-projects/spring-mvc-showcase.git
```

修改spring的版本

```
<properties>
    <java-version>1.8</java-version>
    <org.springframework-version>5.0.0.RELEASE</org.springframework-version>
    <org.aspectj-version>1.8.1</org.aspectj-version>
</properties>
```

修改配置文件：org\springframework\samples\mvc\config\WebMvcConfig.java
这里的作用主要是配置资源映射，将url的/resources/映射到file协议的resources下

```
@Override
public void addResourceHandlers(ResourceHandlerRegistry registry) {

 registry.addResourceHandler("/resources/**").addResourceLocations("file:./src/main/resources/","/resources/");
}
```

使用mvn命令启动：mvn jetty:run

访问url:

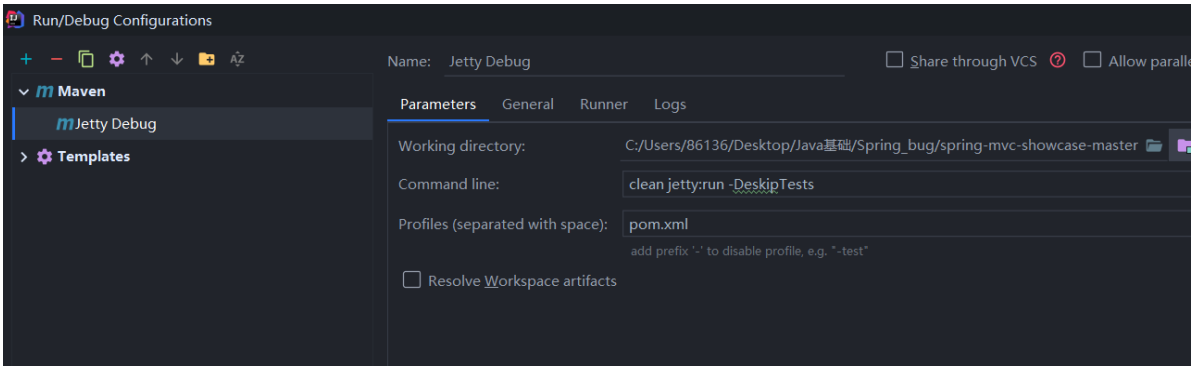
```
http://localhost:8080/spring-mvc-showcase/resources/%255c%255c..%255c/..%255c/..%255c/..%255c/..%255c/..%255c/..%255c/..%255c/windows/win.ini
```

漏洞分析

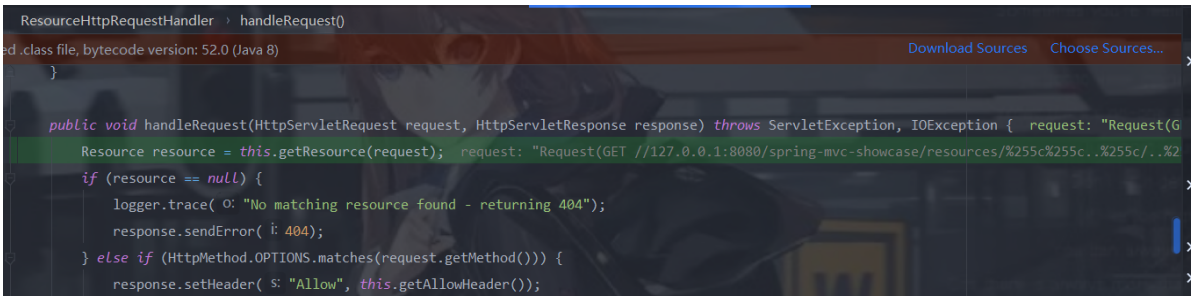
添加依赖(根据自己的报错来)

```
        <!--
https://mvnrepository.com/artifact/javax.xml.bind/jaxb-api
-->
        <dependency>
            <groupId>javax.xml.bind</groupId>
            <artifactId>jaxb-api</artifactId>
            <version>2.3.1</version>
        </dependency>
```

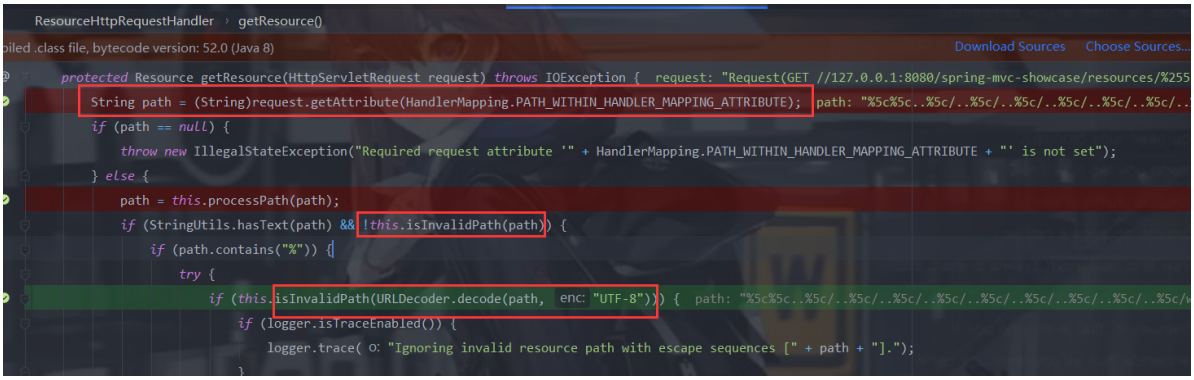
设置Jetty Debug



当外部要访问静态资源时，会调用`org.springframework.web.servlet.resource.ResourceHttpRequestHandler:handleRequest`来处理



经过getAttribute方法后会进行一次url解码

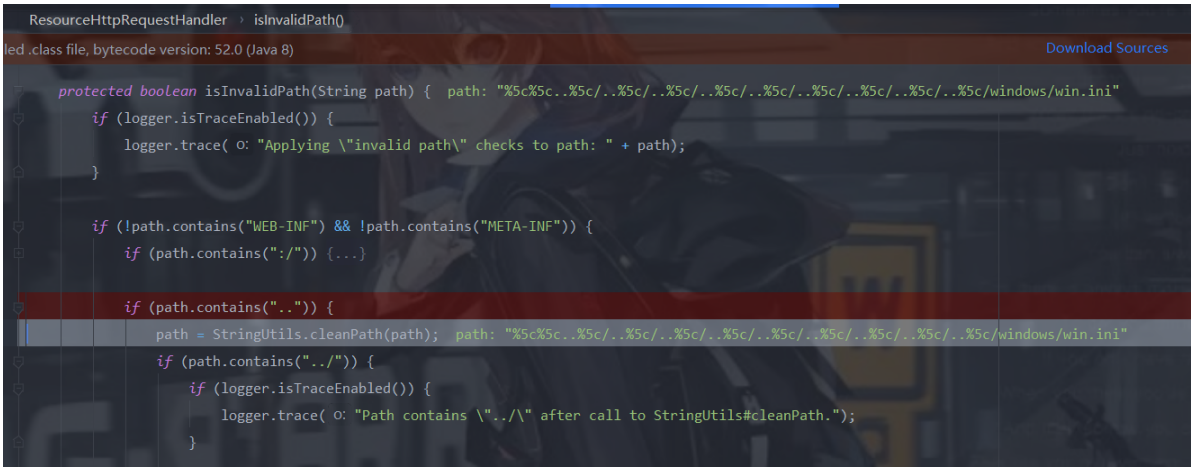


此时path为`%5c%5c..%5c/..%5c/..%5c/..%5c/..%5c/..%5c/..%5c/..%5c/..%5c/windows/win.ini`

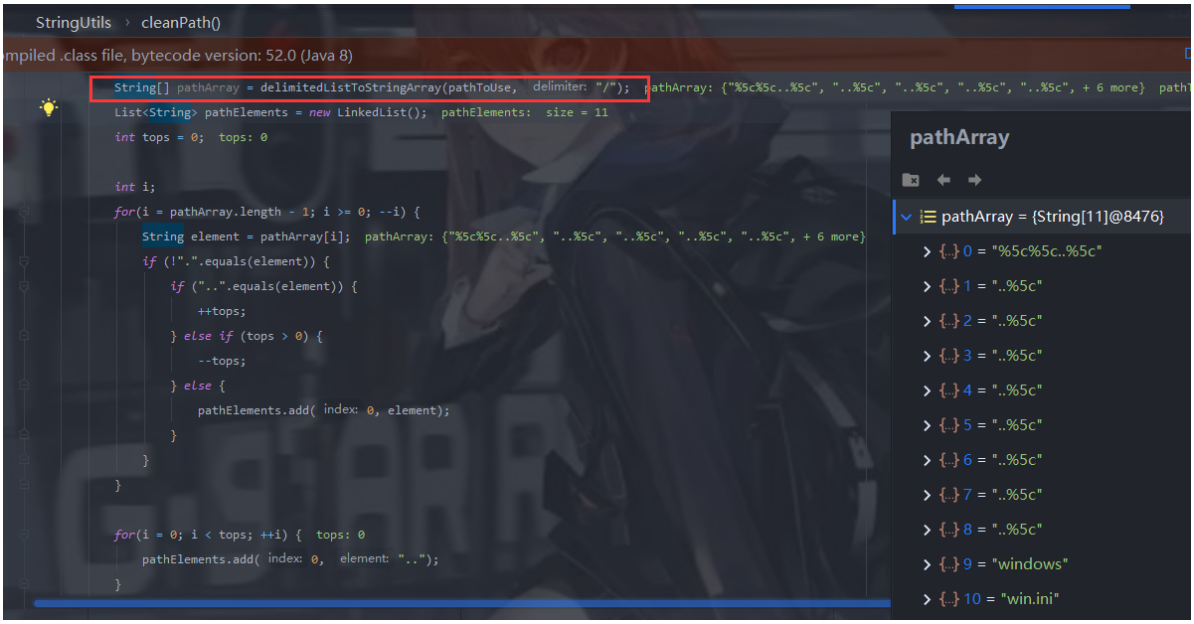
后续有processPath和isInvalidPath方法对path进行处理和校验，其中`processPath`方法作用是处理斜杠的问题，比如多个斜杠或者丢失斜杠的问题，和这里无关系.程序经过它是path也没有变化。isInvalidPath是需要重点关注的。这里依次将path和解码后的path传入isInvalidPath方法

跟进去看下:

首先禁止了WEB-INF和META-INF等关键路径，然后处理是url（带有:/）的情况，最后是处理带有`..`的情况，而`..`正是目录遍历漏洞需要用到的



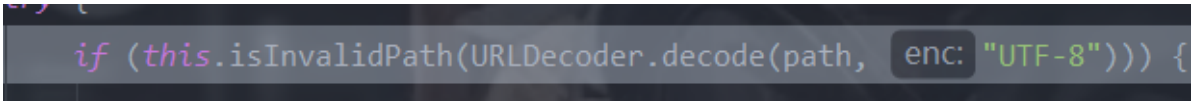
跟进cleanPath,首先将url解码后的path分割成数组,然后经过两个for循环。这个cleanPath的主要作用就是将/foo/bar/../ 这样的路径转换为 /foo/ (可能就是为了满足windows特性吧,想起了一个apache的cve)



由于返回的path中没有包含"../",返回false

```
if (path.contains("..")) {
                path = StringUtils.cleanPath(path);
                if (path.contains("../")) {
                    if (logger.isTraceEnabled()) {
                        logger.trace("Path contains
\"../\" after call to StringUtils#cleanPath.");
                    }

                    return true;
                }
            }

            return false;
```

这里其实并没有发生什么,关键点在后面那次isInvalidPath



进入cleanPath,这里将 \替换为了/,然后进行一些替换

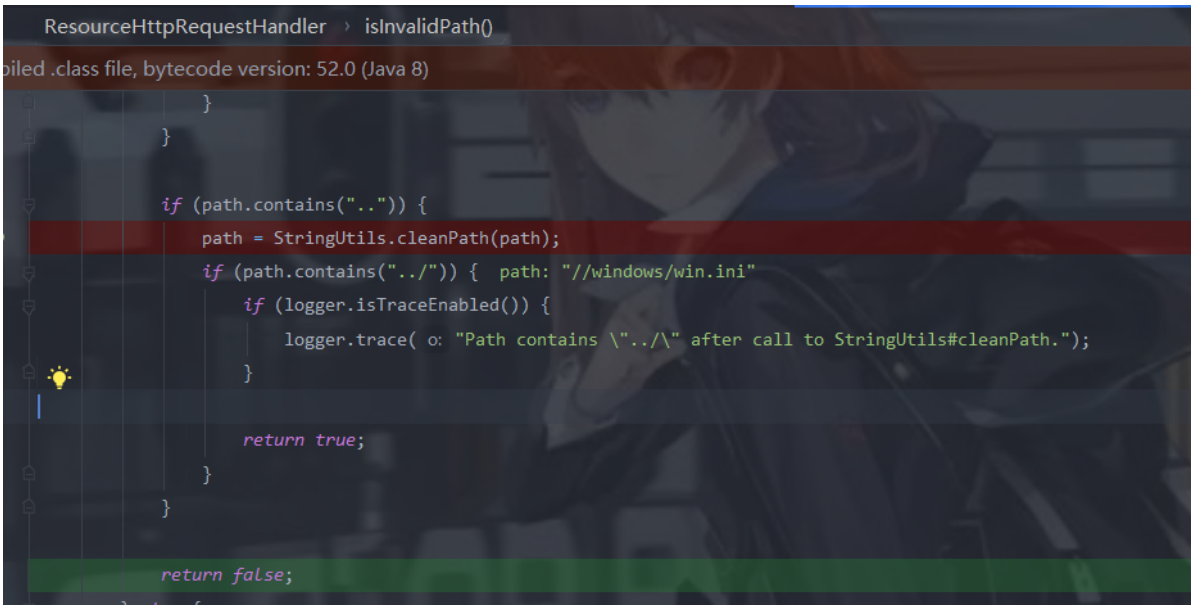
然后这里有点问题:

> `cleanPath`的问题在于`String[] pathArray = delimitedListToStringArray(pathToUse, "/");`这个是允许空元素存在的，也就是说`cleanPath`会把`//`当成一个目录，而操作系统是不会把`//`当成一个目录的。

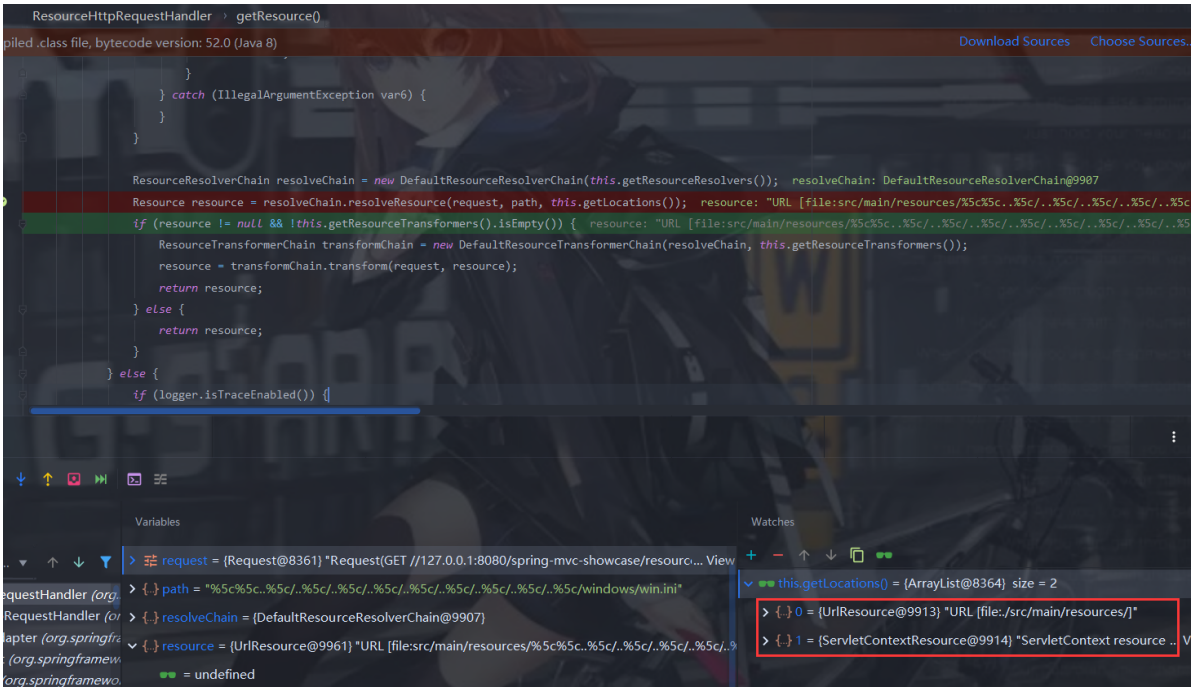
Input	cleanPath	Filesystem
/foo/../	/	/
/foo/../../	/../	/../
/foo//../	/foo/	/
/foo///../../	/foo/	/../
/foo////../../../	/foo/	/../../

Seebug

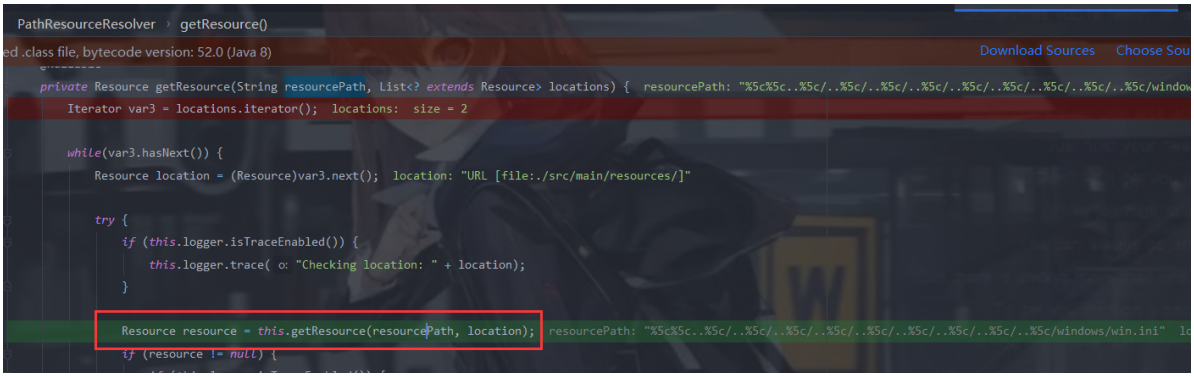
最后返回的path中仍然没有../,返回false



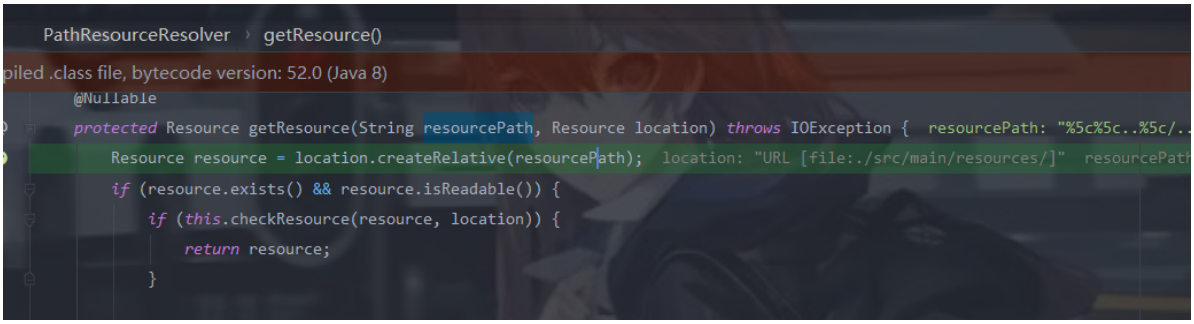
当我们通过两次检测之后,这里有一个getResourceTransformers()方法,getLocations方法包含配置文件中配置的路径,跟进resoslveResource



进入getResource



进入createRelative()

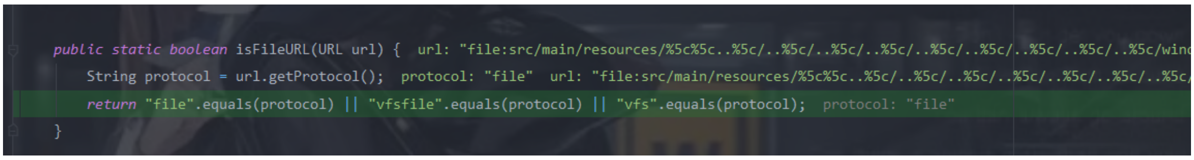


经过`createRelative`方法拼接后得到

`file:./src/main/resources/%5c%5c..%5c/..%5c/..%5c/..%5c/..%5c/..%5c/..%5c/..%5c/windows/win.ini`

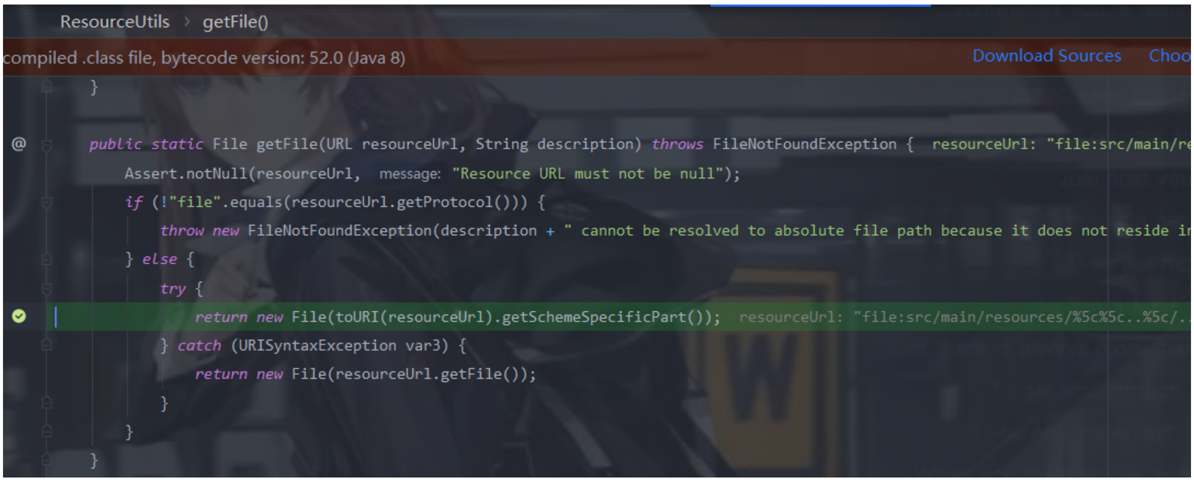
然后调用resource.exists()方法判断文件是否存在

这里会调用`isFileURL`对`url`进行判断，是否以`file://`协议来读取文件，这也是为什么配置静态目录的时候要使用`file://`协议。

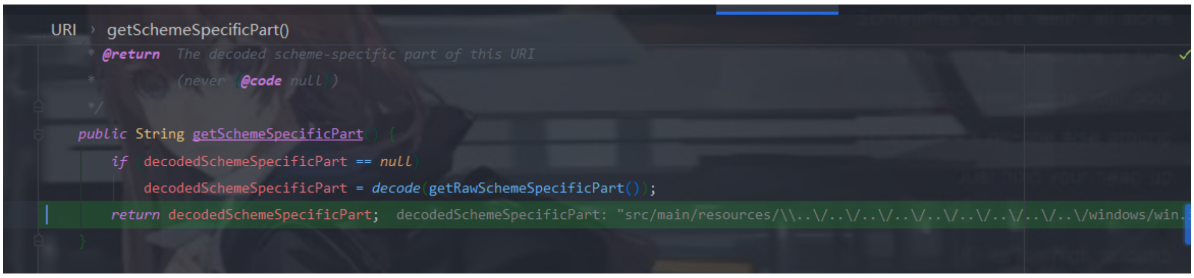


然后调用this.getFile()来获取这个文件对象,来到ResourceUtils#getFile

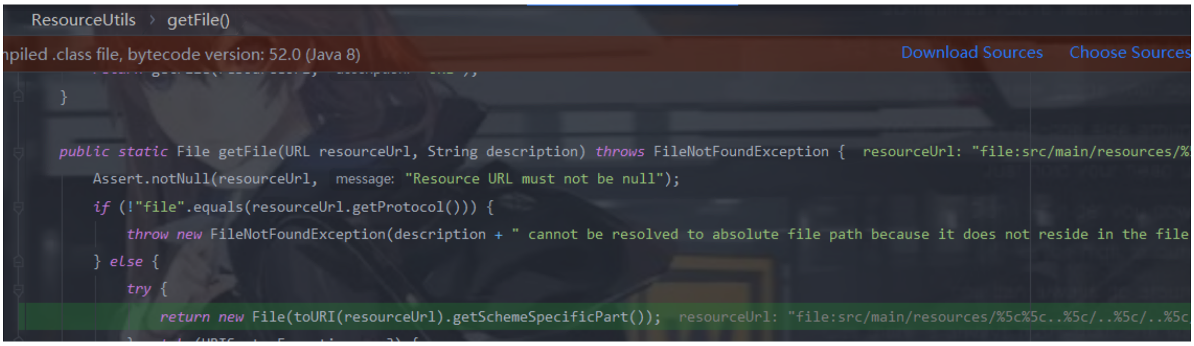
首先对是否为file://协议又判断了一遍,之后进行了一步最重要的操作`new File(toURI(resourceUrl).getSchemeSpecificPart())`



将`resourceUrl`转换为URL对象，最后调用`URI`类的`getSchemeSpecificPart()`获取到文件路径，而在`getSchemeSpecificPart()`里面是有一次`decode`操作的，也就是在这里把`%5c`解码成了`\`



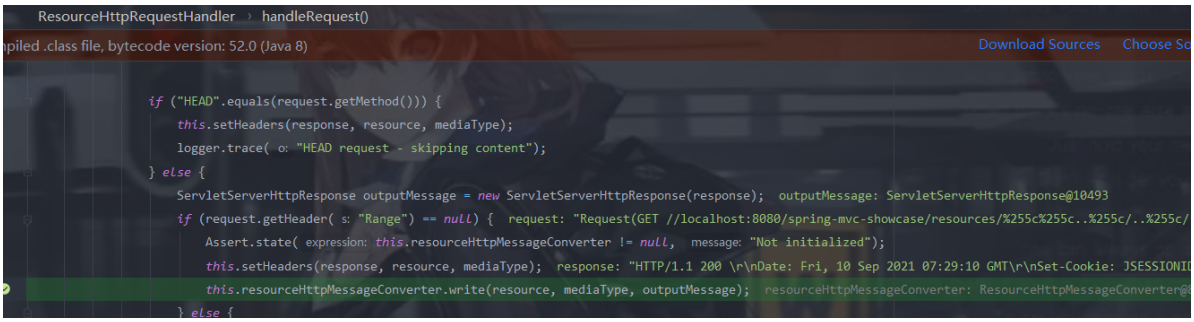
这里进行返回



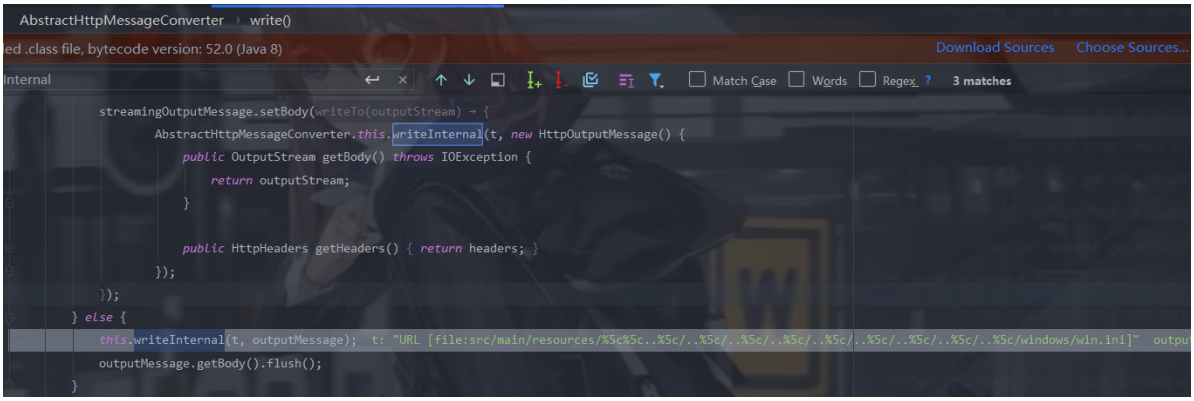
最后返回到`exists()`，最终返回`true`，即文件存在。之后调用`isReadable()`方法检测该文件是否可读的时候，同样会调用这个`getFile`，最终返回`true`，即文件可读。

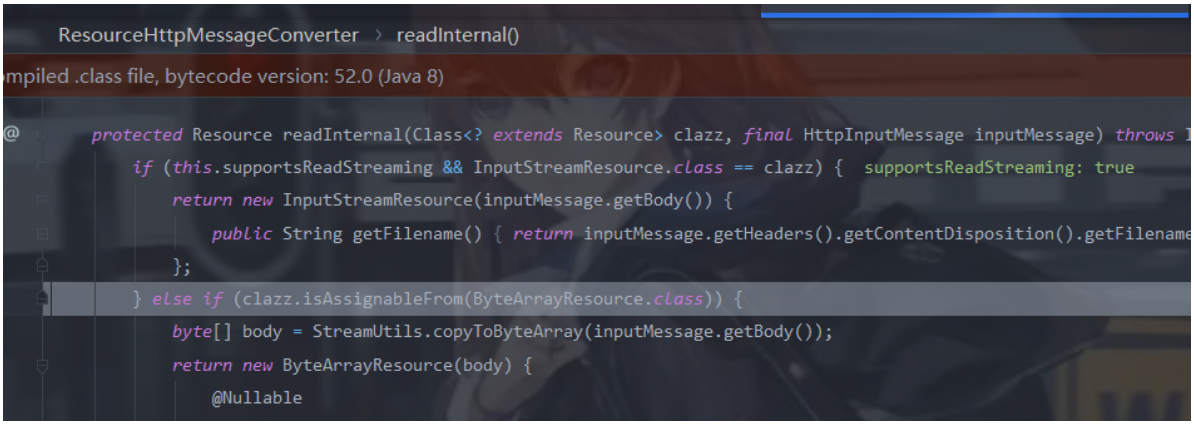
Resource的检测都通过后,来到ResourceHttpRequestHandler#handleRequest方法准备response的内容

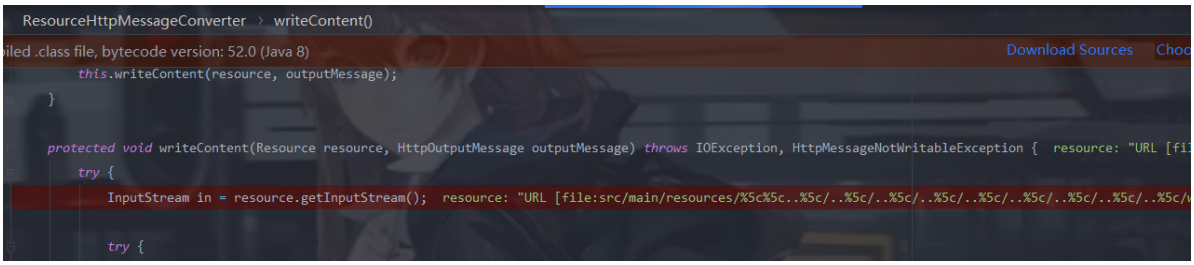
跟进write()



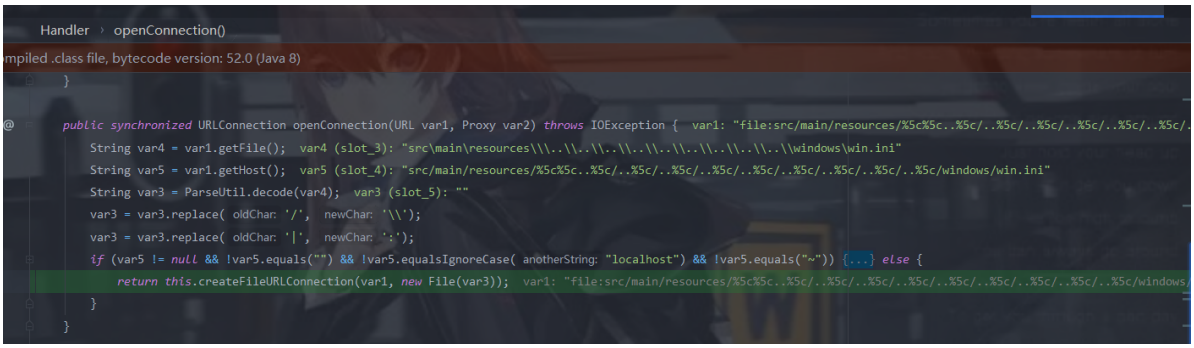
然后依次进入

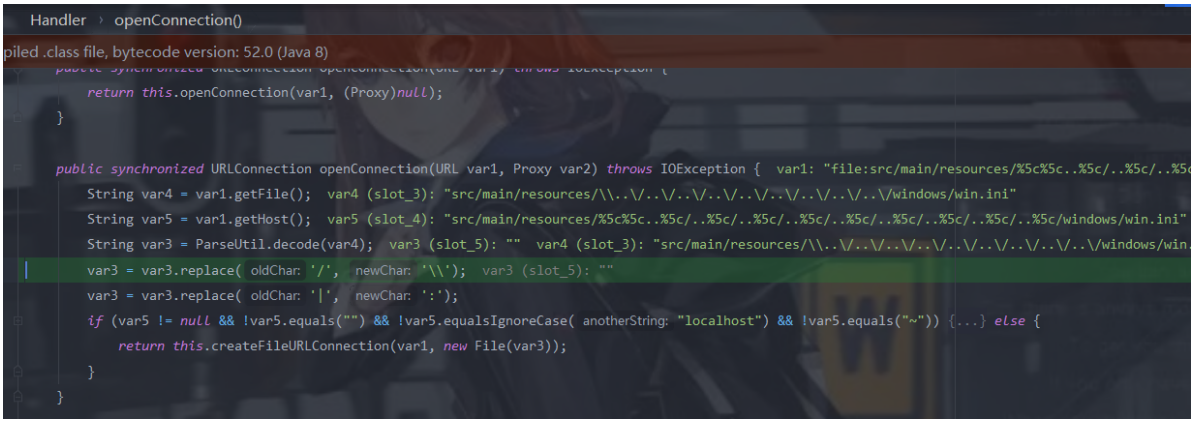






然后在url.openConnection阶段会进行解码,将%5c解码成/。然后返回文件的`InputStream`对象，最终读取内容返回给用户。





总结

感觉流程挺复杂的,这里大概总结一下,一共分三步

1. 在isInvalidPath方法判断路径是否合法(经过isInvalidPath后不存在../),但是isInvalidPath有点问题,可以被绕过
2. 利用createRelative方法拼接,然后判断文件是否存在并可读(判断时会进行url解码)
3. 构造response数据,在url.openConnection会进行url解码,然后成功读取我们的文件并返回

注意事项

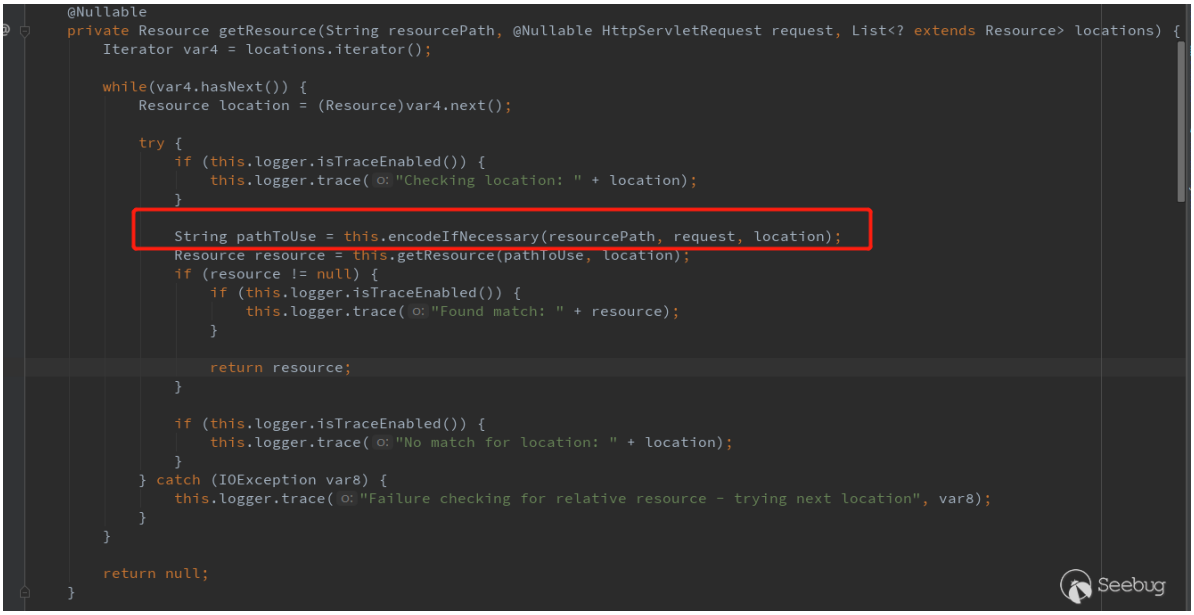
1、在Spring Framework 大于5.0.1的版本，双编码payload在tomcat、jetty下触发不了,

但是单编码payload可以在jetty下触发，tomcat下失败(因为在默认情况下Tomcat遇到包含%2f(/)、%5c()的URL直接http 400)

总结:

```
Spring Framework <=5.0.1

tomcat、jetty:(双编码)

/resources/%255c%255c..%255c/..%255c/..%255c/..%255c/..%255c/..%255c/..%255c/..%255c/..%255c/windows/win.ini

Spring Framework >5.0.1

jetty:(单编码)

/resources/%5c%5c..%5c/..%5c/..%5c/..%5c/..%5c/..%5c/..%5c/..%5c/..%5c/windows/win.ini
```

至于为什么双编码不行,仅为在getResource函数中进行了一次url编码



首先我们传入经过两次编码的url,isInvalidPath前进行解码一次,然后getResource函数又编码一次

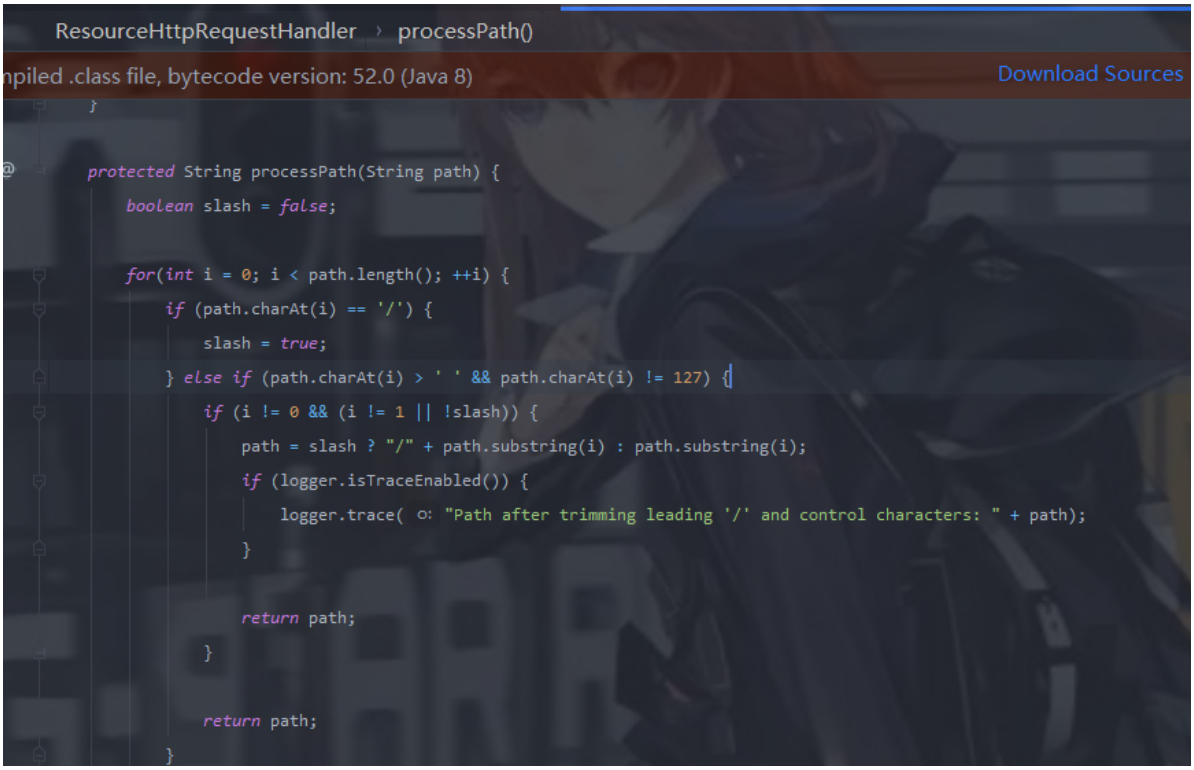
当我们进入第二阶段的时候,我们传递的url是经过两次url编码的,但是它只进行一次url解码,所以无法找到文件

> 1. 在isInvalidPath方法判断路径是否合法(经过isInvalidPath后不存在../),但是isInvalidPath有点问题,可以被绕过
> 2. 利用createRelative方法拼接,然后判断文件是否存在并可读(判断时会进行url解码)
> 3. 构造response数据,在url.openConnection会进行url解码,然后成功读取我们的文件并返回

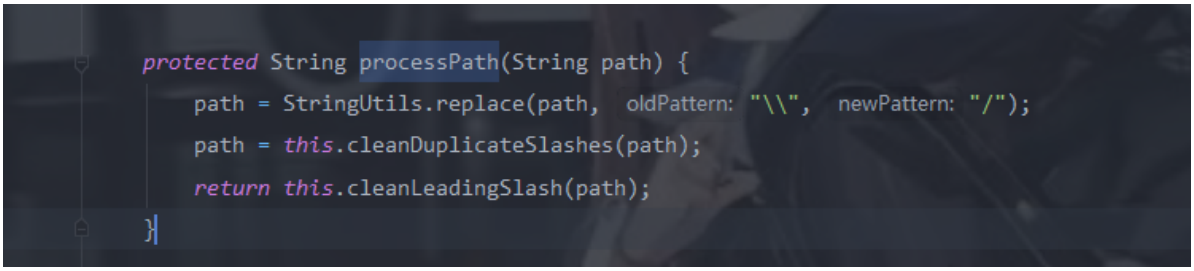
漏洞修复

这个漏洞产生的主要原因就是因为isInvalidPath中的cleanPath 会把 //当成一个目录,所以修复时重写了这里的processPath方法

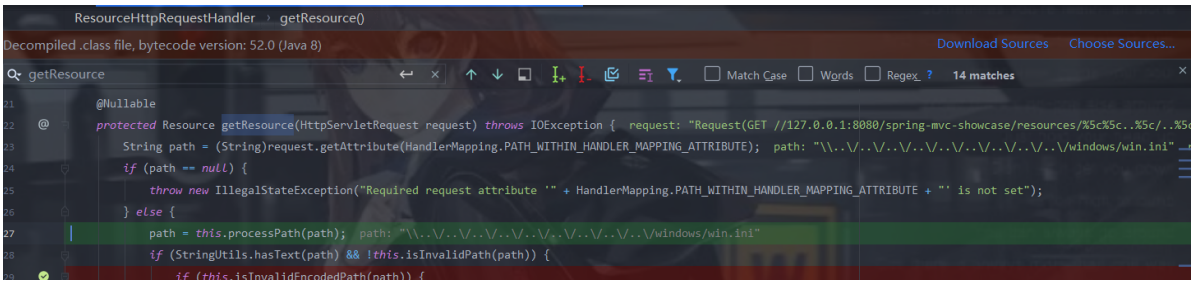
修复前:



修复后:



版本大于5.0.1之后就只能用单编码了,这里进来金国getAttribute解码一次,然后传入paocessPath



```
protected String processPath(String path) {
    path = StringUtils.replace(path, "\\", "/");  // 替换反斜线为斜线，path此时
为//..//..//..//..//..//..//..//..//windows/win.ini
    path = this.cleanDuplicateSlashes(path);      // 去掉多余斜线，path此时
为/../../../../../../../../../windows/win.ini
    return this.cleanLeadingSlash(path);
}
```

这样就不存在将//当成目录了

参考

https://xushao.ltd/post/cve-2018-1271-fen-xi/#%E7%AE%80%E4%BB%8B

https://paper.seebug.org/665/#_1

https://blog.spoock.com/2018/05/30/cve-2018-1271/