CVE-2022-32532_Apache_Shiro_RegExPatternMatcher_认证绕过漏洞

前言

结合自身经历，在使用正则表达式去匹配流量特征时，由于正则表达式中元字符“.”不匹配换行符（\n、\r）导致一直提取不上所需的流量。而如今，之前踩过的坑却出现在了 Apache Shiro 框架之中.....

漏洞描述

6 月 29 日，Apache 官方披露 Apache Shiro 权限绕过漏洞(CVE-2022-32532)，当 Apache Shiro 中使用 RegexRequestMatcher 进行权限配置，且正则表达式中携带“.”时，未经授权的远程攻击者可通过构造恶意数据包绕过身份认证。

相关介绍

Apache Shiro 是一个功能强大且易于使用的 Java 安全框架，它可以执行身份验证、授权、加密和会话管理，可以用于保护任何应用程序——从命令行应用程序、移动应用程序到最大的 web 和企业应用程序。

利用范围

Apache Shiro < 1.9.1

漏洞分析

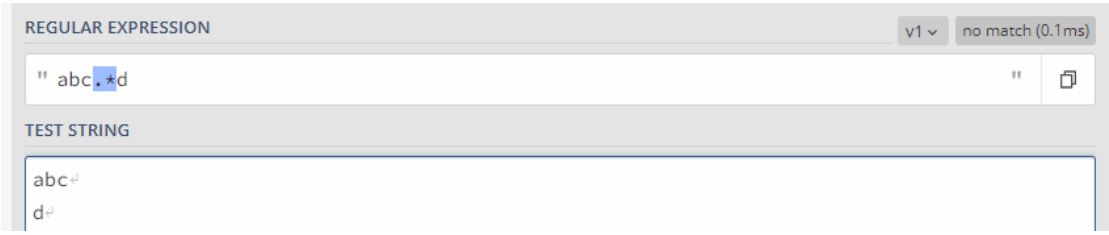
环境搭建

这里使用此漏洞提交者 4ra1n 师傅的漏洞 demo 进行复现分析。

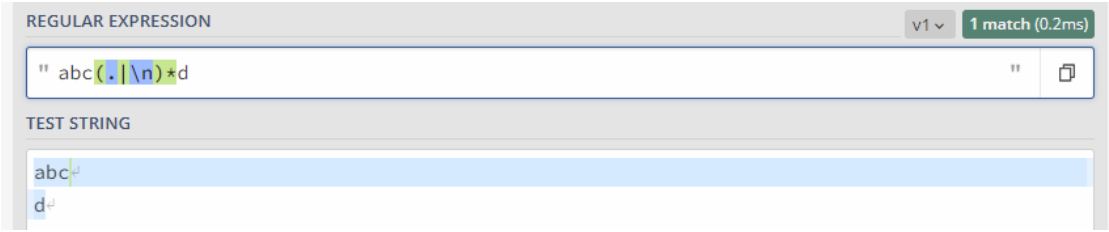
漏洞原理

在漏洞分析之前，先了解一下相关漏洞产生原理。

在正则表达式中元字符`.`是匹配除换行符（`\n`、`\r`）之外的任何单个字符。



要匹配包括 `\n` 在内的任何字符，需使用像`(.|\n)`的模式。



结合 4ra1n 师傅的分析，在 java 中的正则默认情况下`.`也同样不会包含`\n`、`\r`字符，因此在一些场景中，使用正则`.`的规则就有可能被绕过。

```
    public static void main(String[] args) {
        String regex = "a.*b";
        Pattern pattern = Pattern.compile(regex);
        boolean flag1 = pattern.matcher( input: "aaabbb").matches();
// true
        System.out.println(flag1);
        boolean flag2 = pattern.matcher( input: "aa\nbb").matches();
// false
        System.out.println(flag2);
    }
}
```

```
Main
"C:\Program Files\Java\jdk1.8.0_121\bin\java.exe" ...
true
false
```

新增 `Pattern.DOTALL` 模式后，正则表达式`.`就可以匹配任何字符包括换行符。

```
    public static void main(String[] args) {
        String regex = "a.*b";
        Pattern pattern = Pattern.compile(regex,Pattern.DOTALL);
        boolean flag1 = pattern.matcher( input: "aaabbb").matches();
// true
        System.out.println(flag1);
        boolean flag2 = pattern.matcher( input: "aa\nbb").matches();
// false
        System.out.println(flag2);
    }
}
```

```
Main
"C:\Program Files\Java\jdk1.8.0_121\bin\java.exe" ...
true
true
```

源码分析

结合 `shiro` 源码和漏洞场景分析，在 `shiro-core-1.9.0.jar` 中存在一个 `RegExPatternMatcher` 类

`org.apache.shiro.util.RegExPatternMatcher.class`

```
package org.apache.shiro.util;

import ...

public class RegExPatternMatcher implements PatternMatcher {
    public RegExPatternMatcher() {
    }

    public boolean matches(String pattern, String source) {
        if (pattern == null) {
            throw new IllegalArgumentException("pattern argument cannot be null.");
        } else {
            Pattern p = Pattern.compile(pattern);
            Matcher m = p.matcher(source);
            return m.matches();
        }
    }
```

`RegexRequestMatcher` 和 `AntPathMatcher` 类似，提供请求路径匹配功能及拦截器参数解析的功能。而一般情况下 `RegexRequestMatcher` 类是不会在项目中出现，需要用户自己配置。

分析 `RegexRequestMatcher` 用于匹配的代码

```
package org.apache.shiro.util;

import ...

public class RegExPatternMatcher implements PatternMatcher {
    public RegExPatternMatcher() {
    }

    public boolean matches(String pattern, String source) {
        if (pattern == null) {
            throw new IllegalArgumentException("pattern argument cannot be null.");
        } else {
            Pattern p = Pattern.compile(pattern);
            Matcher m = p.matcher(source);
            return m.matches();
        }
    }
}
```

可以看到，这里正如上面漏洞原理分析的一样，`pattern` 存在带`.`的正则表达式匹配，若 `source` 中存在`\r` 或`\n` 字符时，将判断错误。

而在 4ra1n 师傅构造的漏洞场景中

```java
public class MyFilter extends AccessControlFilter {

    public MyFilter(){
        super();
        this.pathMatcher = new RegExPatternMatcher();
    }

    @Override
    protected boolean isAccessAllowed(ServletRequest request, ServletResponse response, Object mappedValue) {
        String token = ((HttpServletRequest)request).getHeader( s: "Token");
        // todo: check permission ...
        return token != null && token.equals("4ra1n");
    }

    @Override
    protected boolean onAccessDenied(ServletRequest request, ServletResponse response) {
        System.out.println("deny -> "+((HttpServletRequest)request).getRequestURI());
        try {
            response.getWriter().println("access denied");
        } catch (IOException e) {
            e.printStackTrace();
        }
        return false;
    }
```

自定义了`Filter`，增加权限认证场景：

1、判断请求头中的`Token`是否匹配

2、如果不存在`Token`或者`Token`头错误则认为认证失败

并且配置了自定义的`AccessControlFilter`实现类，并将`PatternMatcher`配置为`RegExPatternMatcher`

同时自定义`MyShiroFilterFactoryBean`

```java
public class MyShiroFilterFactoryBean extends ShiroFilterFactoryBean {

    public MyShiroFilterFactoryBean() { super(); }

    @Override
    protected AbstractShiroFilter createInstance() {
        SecurityManager securityManager = this.getSecurityManager();
        FilterChainManager manager = new DefaultFilterChainManager();
        manager.addFilter( s: "myFilter",new MyFilter());
        // my filter
        manager.addToChain( s: "/permit/.*", s1: "myFilter");
        // todo: add other filters

        PathMatchingFilterChainResolver chainResolver = new PathMatchingFilterChainResolver();
        chainResolver.setFilterChainManager(manager);
        // set RegExPatternMatcher
        chainResolver.setPathMatcher(new RegExPatternMatcher());
        return new SpringShiroFilter((WebSecurityManager) securityManager, chainResolver);
    }

    static class SpringShiroFilter extends AbstractShiroFilter {
        protected SpringShiroFilter(WebSecurityManager webSecurityManager, FilterChainResolver resolver) {
            this.setSecurityManager(webSecurityManager);
            this.setFilterChainResolver(resolver);
        }
    }
}
```

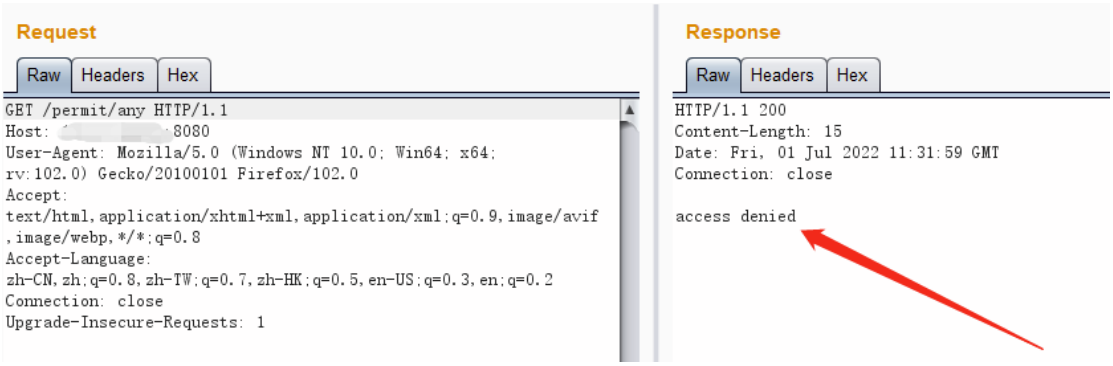
继承自`ShiroFilterFactoryBean`类；设置匹配规则为`/permit/.*`字符串，表示需要拦截`/permit/`下所有的路径

```java
package com.example.shirodemo;

import ...

@RestController
public class DemoController {
    @RequestMapping(path = "/permit/{value}")
    public String permit(@PathVariable String value) {
        System.out.println("success!");
        return "success";
    }

    // Another Bypass
    // @RequestMapping(path = "/permit/*")
    public String permit() {
        System.out.println("success!");
        return "success";
    }
}
```

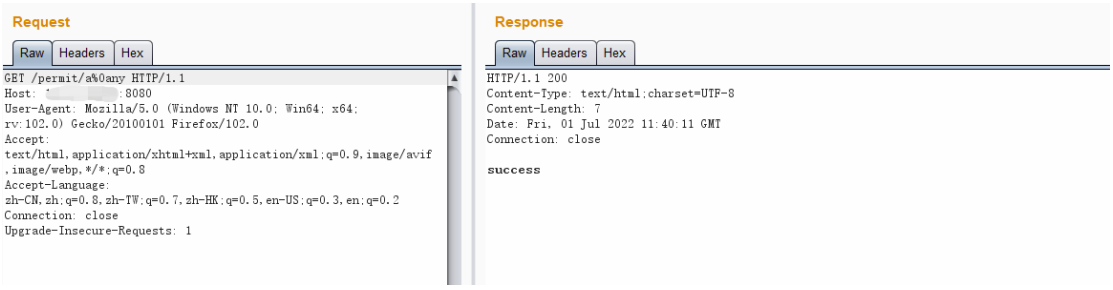
在自定义`controller`中，配置了`/permit/{value}`这样从路径取参数的路由和`/permit/*`这样的通配路由。

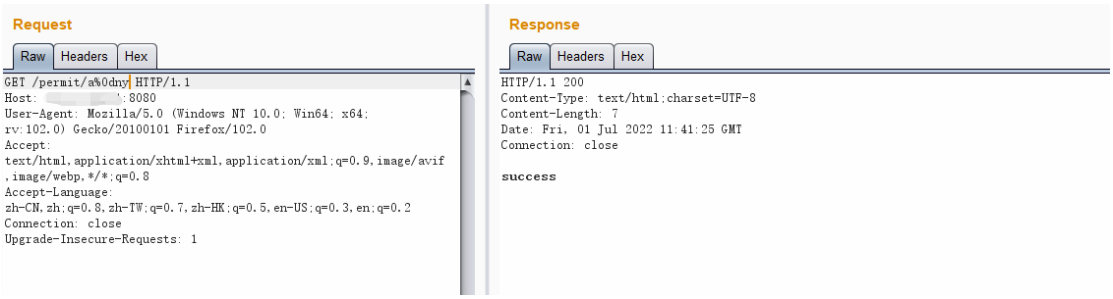
漏洞复现

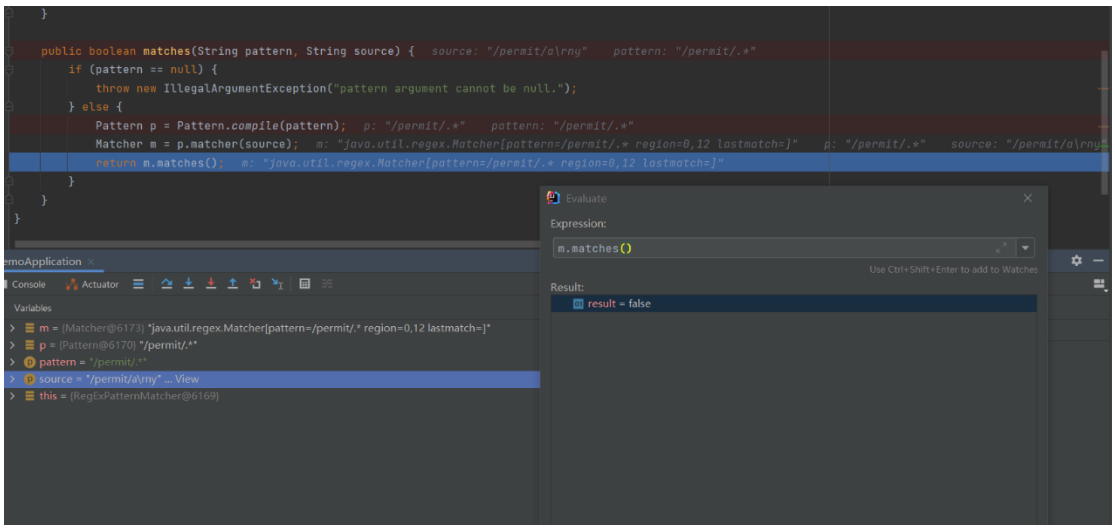
当访问`/permit/any`，返回拒绝访问



当访问`/permit/a%0any`或`/permit/a%0dny`时，返回 success，即认证成功







所以在如上满足使用了`RegExPatternMatcher`类，设置带有`.`的正则表达式等条件的场景下，利用该漏洞可实现权限认证绕过。

修复建议

参考漏洞影响范围进行排查，目前官方已发布修复补丁

https://github.com/apache/shiro/commit/6bcb92e06fa588b9c7790dd01bc02135d58d3f5b