

# 红队java代码审计生命周期

KBAT (/u/34923) / 2022-12-20 10:11:00 / 发表于广东 / 浏览数 2029

## 红队java代码审计生命周期

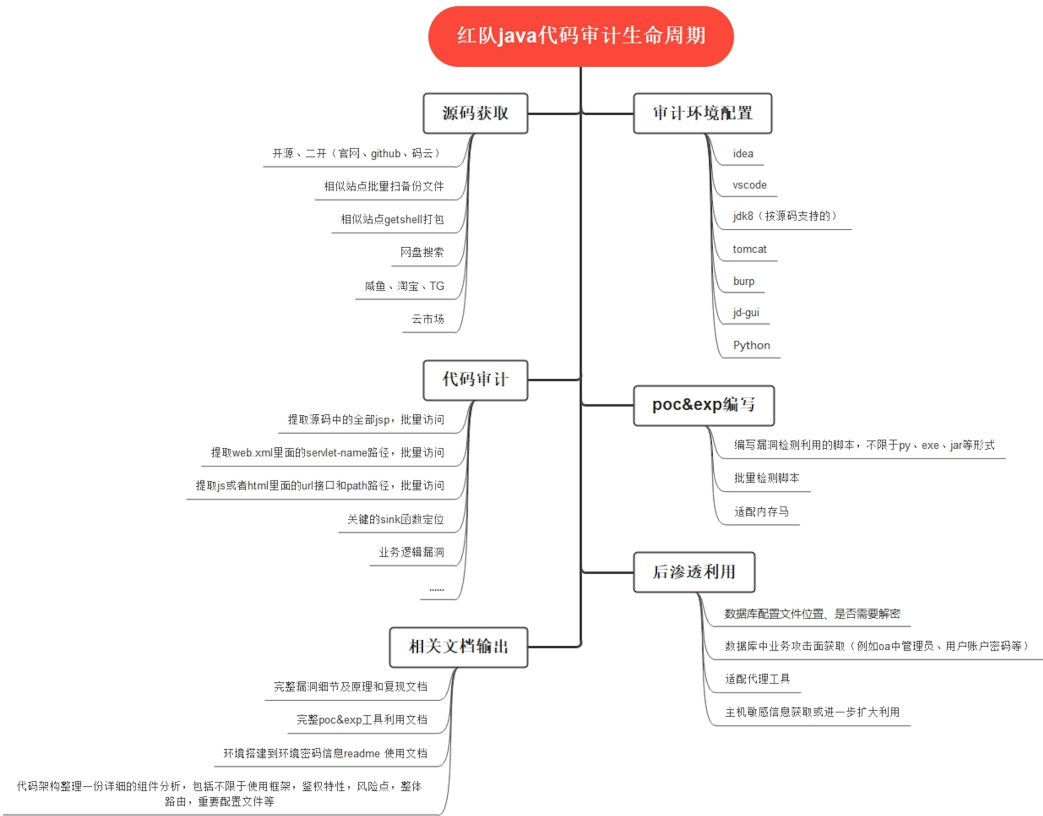
@深信服-华南天玄攻防战队-KBAT

### 前言

红队java代码审计生命周期中常见的一些漏洞学习总结以及一些审计思路。

## 红队java代码审计生命周期

源码获取->审计环境配置->代码审计->poc&exp编写->后渗透利用->相关文档输出



(https://xzfile.aliyuncs.com/media/upload/picture/20221219151302-933fe594-7f6c-1.png)

### 源码获取

- 开源、二开 (官网、github、码云)
- 相似站点批量扫备份文件
- 相似站点getshell打包
- 网盘搜索
- 咸鱼、淘宝、TG
- 云市场
- .....

### 审计环境

- idea
- vscode
- jdk8 (按源码支持的)

- tomcat
- burp
- jd-gui
- .....

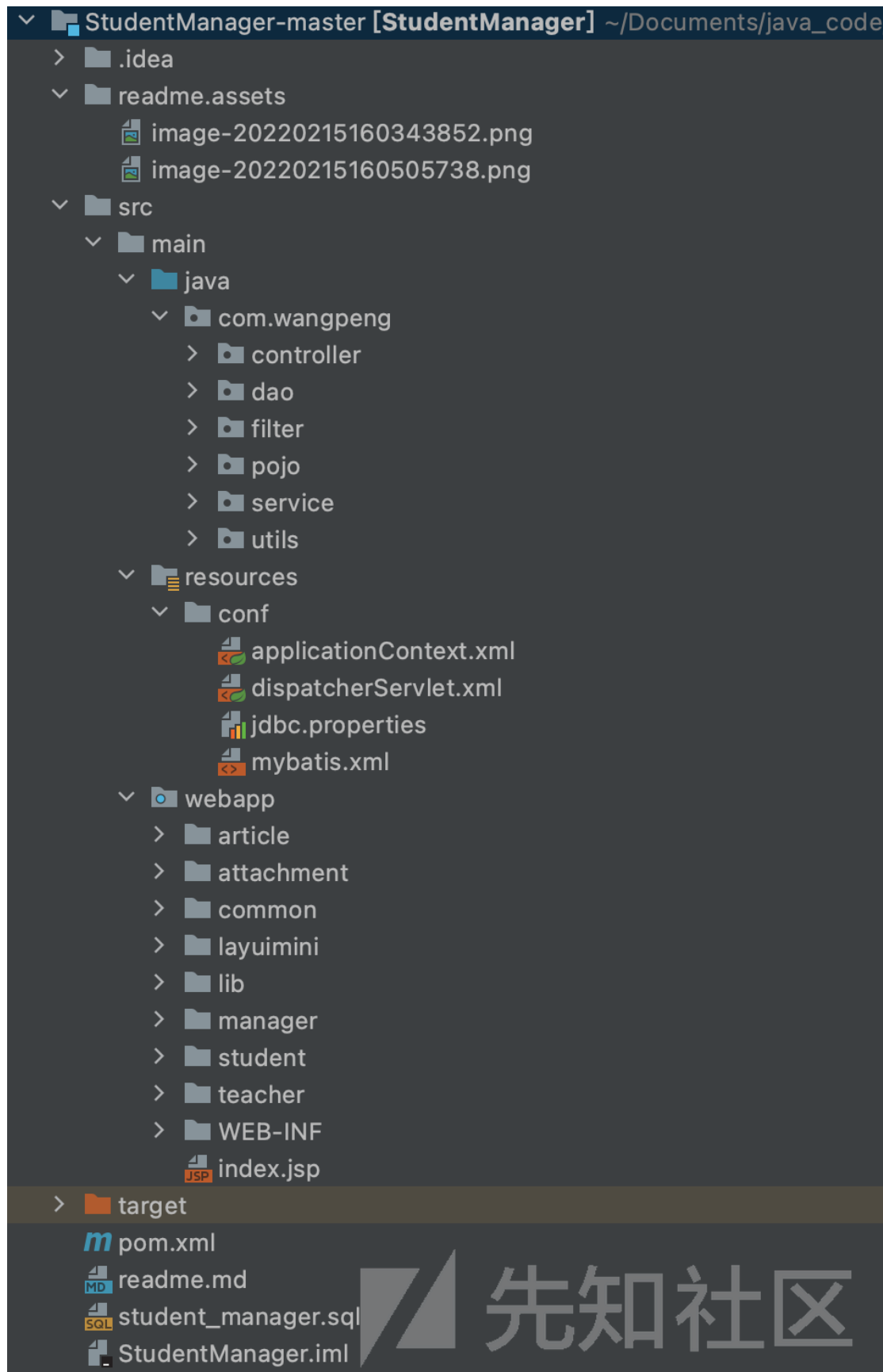
## 代码审计

### 快速代码审计

- 提取源码中的全部jsp, find /domain/ -name "\*.jsp",提取出来后用路径直接去用burp批量跑目录（可换请求的方法，GET、POST），把返回200(根据网站情况)的路径提取出来，表示可以直接未认证访问；然后在跟进这些jsp代码，通过查找相关的sink函数，在去定位source是否可控，如果可控那么就形成一条污染链，在看是否需要绕过sanitizer。
- 通过提取web.xml里面的servlet-name路径，然后拼接路径直接去用burp批量跑目录（可换请求的方法，GET、POST），把返回200(根据网站情况)的路径提取出来，表示可以直接未认证访问，然后根据对应的servlet-name的servlet-class根据相关联的class代码（用idea可直接将.class逆向出来，比较完整；有些是引用jar包形式，可通过jd-gui进行逆向或者idea逆向出来源码审计）。同样的通过查找相关的sink函数，在去定位source是否可控，如果可控那么就形成一条污染链，在看是否需要绕过sanitizer。
- 通过批量提取js或者html里面的url接口和path路径。同样方法类似如上。
- 关键的sink函数定位。

## SpringBoot项目结构

一个简单的springboot项目结构如下



(<https://xzfile.aliyuncs.com/media/upload/picture/20221219151405-b939bb26-7f6c-1.png>)

其中Java代码全部放在/src/main/java/目录下，资源文件在/src/main/resources/下

### 代码结构

common/: 存放通用类，如工具类和通用返回结果

config/: 存放配置文件

controller/: 存放控制器，接收从前端传来的参数，对访问控制进行转发、各类基本参数校验或者不复用的业务简单处理等。

dao/: 数据访问层，与数据库进行交互，负责数据库操作，在Mybaits框架中存放自定义的Mapper接口

entity/: 存放实体类

interceptor/: 拦截器

service/: 存放服务类，负责业务模块逻辑处理。Service层中有两种类，一是Service，用来声明接口；二是ServiceImpl，作为实现类实现接口中的方法。

utils/: 存放工具类

NewBeeMallApplication.java: Spring Boot启动类

dto/: 存放数据传输对象（Data Transfer Object），如请求参数和返回结果

vo/: 视图对象（View Object）用于封装客户端请求的数据，防止部分数据泄漏，保证数据安全

constant/: 存放常量

filter/: 存放过滤器

component/: 存放组件

## 资源目录结构

在src/main/resources下存放资源文件

mapper/: 存放Mybatis的mapper.xml文件

static/: 静态资源文件目录（Javascript、CSS、图片等），在这个目录中的所有文件可以被直接访问

templates/: 存放模版文件

application.properties: Spring Boot默认配置文件

META-INF/: 相当于一个信息包，目录中的文件和目录获得Java 2平台的认可与解释，用来配置应用程序、扩展程序、类加载器和服务

i18n/: 国际化文件的简称，来源是英文单词internationalization的首末字符i和n，18为中间的字符数

## 其他结构

⚠ Spring Boot无需配置 web.xml，但在其他Java项目中，web.xml是一个非常重要的文件，用来配置Servlet、Filter、Listener等。

pom.xml: maven的配置文件，记录项目信息、依赖信息、构建配置等

如果使用gradle进行自动化构建，则会存在build.gradle文件



Java审计难上手的一大因素是Java一般都是大中型系统，架构相比于PHP开发的小系统会复杂很多，大型系统开发过程中难免出现不规范的编码习惯，再加上函数调用错综复杂，审计代码时光弄明白程序逻辑，理解程序员的编码习惯就要花费大量精力了。

首先弄明白请求流程的处理，知道用户请求内容会经过哪些代码才能理解程序处理逻辑，可以对我们后续的审计提供非常大的帮助。

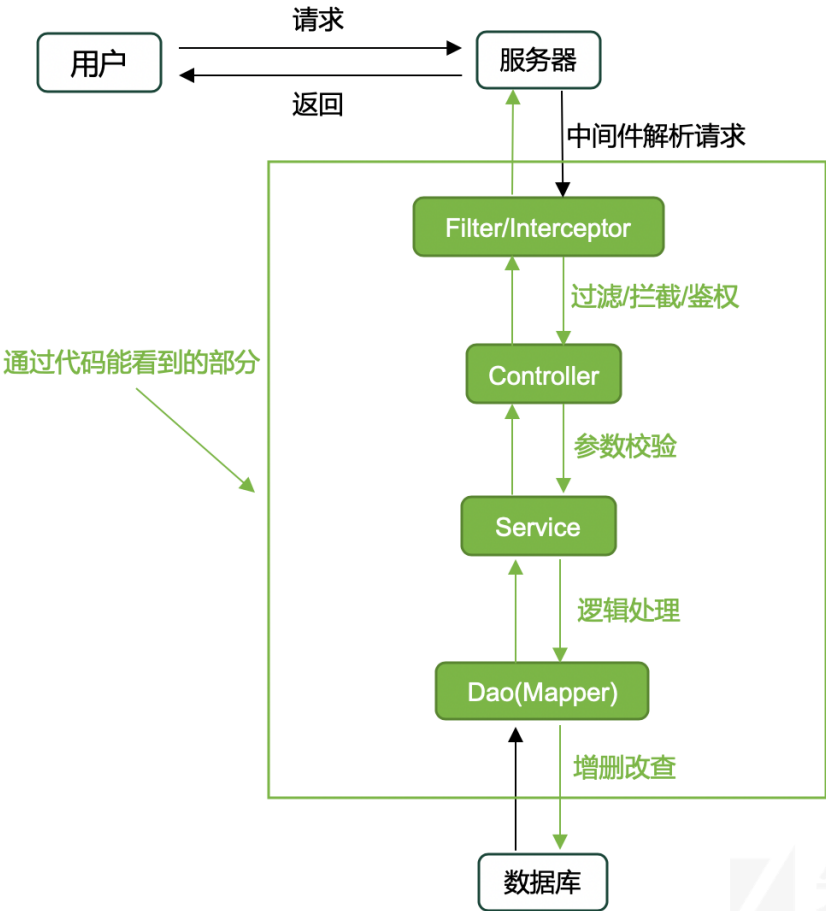
用户的请求发送给服务器之后，中间件（案例项目使用的是Tomcat）会将请求解析发送给Spring的DispatcherServlet，DispatcherServlet的作用是分配请求，详细的过程我们暂时不深入。只需要知道中间件解析请求之后请求会经过Filter和Interceptor。Filter（过滤器）和Interceptor（拦截器）做的事很相似，但他们的触发时机不同，且Interceptor只在Spring中生效，它们可以用来对请求进行过滤字符、拦截、鉴权、日志记录等功能，简单说就是可以在参数进入应用前对其处理，做到全局的处理。

请求经过Filter和Interceptor之后会被DispatcherServlet分配到对应路径的Controller（控制器），文件名为ExampleController，Controller负责简单的逻辑处理和参数校验功能，之后调用Service。

Service主要负责业务模块逻辑处理。Service层中有两种类，一是接口类，文件名为ExampleService，用来声明接口；二是接口实现类，文件名为ExampleServiceImpl，作为实现类实现接口中的方法。实现的代码都在ExampleServiceImpl中。当Service涉及到数据库操作时就会调用Dao。

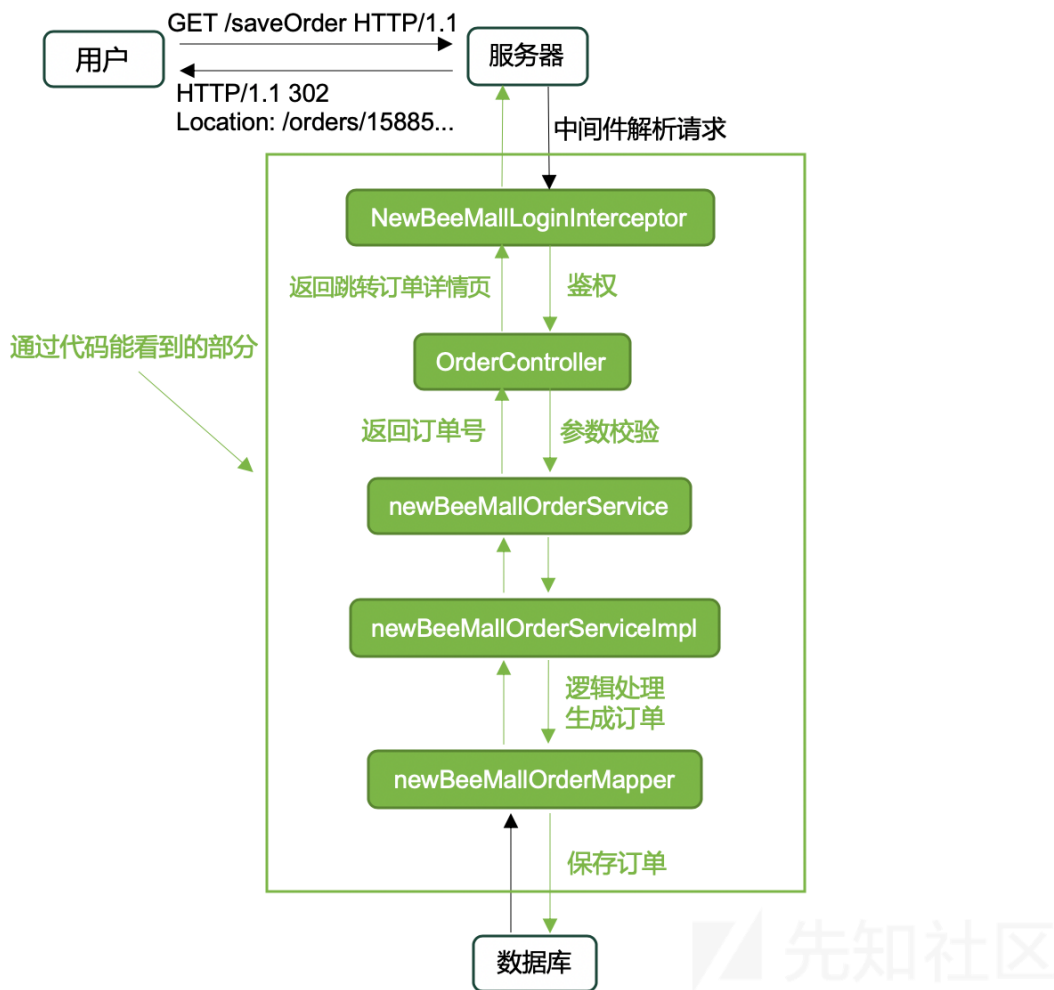
Dao主要负责数据库的操作，由于使用Mybatis作为ORM框架，只做中间传递的作用，所有SQL语句都是写在配置文件中的，配置文件名为ExampleMapper.xml，存放在src/main/resources/mapper中。

从用户请求到服务器处理的主要过程如下图所示（省略了DispatcherServlet）：



(<https://xzfile.aliyuncs.com/media/upload/picture/20221219151456-d75a70be-7f6c-1.png>)

为了更好地理解，以「保存订单」功能为例，主要的请求流程如下图，不了解Spring请求传递的同学可以在代码中跟一遍请求流程，会加深请求传递的印象。



(<https://xzfile.aliyuncs.com/media/upload/picture/20221219151549-f6f28a60-7f6c-1.png>)

## java项目分层

- 视图层(View 视图)
- 控制层 (Controller、Action控制层)
- 服务层(Service)
- 业务逻辑层BO(business object)
- 实体层(entity 实体对象、VO(value)object)值对象、模型层(bean)
- #### Servlet
- Servlet是在Java Web容器上运行的小程序
- Servlet3.0之前的版本都需要在web.xml中配置
- Spring MVC框架就是基于Servlet技术实现的

## sql注入漏洞

### 成因

本质是将用户的输入当做代码执行，程序将用户的输入拼接到了sql语句中，改变原来sql语句的语义造成攻击。

### 常见的一些例子

DAO: 存在拼接的SQL语句

```
String sql="select * from user where id="+id
```

Hibernate框架

```
session.createQuery("from Book where title like '%" + userInput + "%' and published = true")
```

Mybatis框架

```
Select * from news where title like '${title}%'
Select * from news where id in (${id}),
Select * from news where title ='java' order by ${time} asc
```

## 审计方法

对于sql注入来讲，只要是与数据库存在交互的地方，应用程序对用户的输入没有进行有效的过滤，都有可能存在SQL注入漏洞。在实际环境中，**中间件漏洞的sql注入漏洞可能更多**：

- Mybatis框架中的like、in和order by语句。
- Hibernate框架中的createQuery()函数

快速定位相关sql语句上下文，查看是否有显式过滤机制。

## 修复

- 参数化查询，使用java.sql.PreparedStatement来对数据库发起参数化查询。

```
stmt=connnection.prepareStatement(sqlString);
    stmt.setString(1,userName);
    stmt.setString(2,itemName);
    rs=stmt.executeQuery();
```

- 使用预编译能够预防绝大多数SQL注入，**java.sql.PreparedStatement代替java.sql.Statement**,但对于order by后的不能用预编译进行处理，只能手动过滤。

```
String sqlString = "select * from db_user where username=? and password=?";
    PreparedStatement stmt = connection.prepareStatement(sqlString);
    stmt.setString(1, username);
    stmt.setString(2, pwd);
    ResultSet rs = stmt.executeQuery();
```

- Mybatis的SQL配置中，采用#变量名称

### XSS漏洞

#### 成因

网站与后端交互的输入输出没有做好过滤，导致攻击者可以插入恶意js语句进行攻击。根据后端代码不同，大致可以分为反射型、存储型、DOM型

举例:

```
@RequestMapping("/xss")
public ModelAndView xss(HttpServletRequest request,HttpServletResponse
response) throws ServletException,IOException{
    String name = request.getParameter("name");
    ModelAndView mav = new ModelAndView("mmc");
    mav.getModel().put("uname", name);
    return mav;
}
```

这里接收了用户输入的参数name，然后又直接输出到了页面，整个过程没有任何过滤，

#### 存储型

根据已知的用户ID查询该用户的数据并显示在JSP页面上。如果存入的数据存在未经过滤的恶意js代码。就会造成xss攻击。

```
<% ...
    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery("select * from users where id =" + id);
    String address = null;
    if (rs != null) {
        rs.next();
        address = rs.getString("address");
    }
%>
```

#### 审计方法

全局搜索用户的输入与输出，查找是否存在过滤。

#### 修复



- 配置全局过滤器web.xml

```
<filter>
    <filter-name>XssSafe</filter-name>
    <filter-class>XssFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>XssSafe</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

- 使用commons.lang包，主要提供了字符串查找、替换、分割、去空白、去掉非法字符等等操作。有几个函数可以用来过滤。

- StringEscapeUtils.escapeHtml(string)，使用HTML实体，转义字符串中的字符。
- StringEscapeUtils.escapeJavaScript(string)，使用JavaScript字符串规则转义字符串中的字符。

### XXE漏洞

#### 成因

XXE就是XML外部实体注入。当允许引用外部实体时，通过构造恶意内容，就可能导致任意文件读取、系统命令执行、内网端口探测、攻击内网网站等危害。

#### 审计方法

XML解析一般在导入配置、数据传输接口等场景会用到，xml解析器是否禁用外部实体。

全局搜索如下函数:

```
javax.xml.parsers.DocumentBuilder
javax.xml.stream.XMLStreamReader
org.jdom.input.SAXBuilder
org.jdom2.input.SAXBuilder
javax.xml.parsers.SAXParser
org.dom4j.io.SAXReader
org.xml.sax.XMLReader
javax.xml.transform.sax.SAXSource
javax.xml.transform.TransformerFactory
javax.xml.transform.sax.SAXTransformerFactory
javax.xml.validation.SchemaFactory
javax.xml.bind.Unmarshaller
javax.xml.xpath.XPathEx
```

#### 修复

- 使用白名单检验，例如上面的代码增加正则匹配

```
if (!Pattern.matches("[_a-zA-Z0-9]+", user.getUserId()))
if (!Pattern.matches("[_a-zA-Z0-9]+", user.getDescription()))
```

- 使用安全的XML库，使用dom4j来构建XML,dom4j会对文本数据域进行xml编码。

### SSRF漏洞

#### 成因

代码中提供了从其他服务器应用获取数据的功能但没有对目标地址做过滤与限制。

java的SSRF利用方式比较局限:

- 利用file协议任意文件读取。

- 利用http协议端口探测

#### 支持的一些协议:

```
file ftp mailto http https jar netdoc
```

举例:

```
String url = request.getParameter("url");
String htmlContent;
try {
    URL u = new URL(url);
    //URL对象用openconnection()获得openConnection类对象。
    URLConnection urlConnection = u.openConnection();
    HttpURLConnection httpUrl = (HttpURLConnection) urlConnection;
    BufferedReader base = new BufferedReader(new InputStreamReader(httpUrl.getInputStream(), "UTF-8"));
    //用inputStream获取字节流然后使用InputStreamReader转化为字符流。
    StringBuffer html = new StringBuffer();
    while ((htmlContent = base.readLine()) != null) {
        html.append(htmlContent);
    }
}
```

## 漏洞代码四种情况

- Request

```
Request.Get(url).execute()
```

- openStream

```
URL u;
int length;
byte[] bytes = new byte[1024];
u = new URL(url);
inputStream = u.openStream();
```

- HttpClient

```
String url = "http://127.0.0.1";
CloseableHttpClient client = HttpClients.createDefault();
HttpGet httpGet = new HttpGet(url);
HttpResponse httpResponse;
try {
    // 该行代码发起网络请求
    httpResponse = client.execute(httpGet);
}
```

- URLConnection和HttpURLConnection

```
URLConnection urlConnection = url.openConnection();
HttpURLConnection urlConnection = url.openConnection();
```

## #### 审计方法

只要是能够对外发起网络请求的地方，就有可能会出现SSRF漏洞。重点查找以下函数。

```
HttpClient.execute
HttpClient.executeMethod
HttpURLConnection.connect
HttpURLConnection.getInputStream
URL.openStream
```

- new URL(): 构造一个url对象
- openConnection(): 创建一个实例URLConnection.
- getInputStream(): 获取URL的字节流
- #### 修复
- 取URL的Host
- 取Host的IP
- 判断是否是内网IP，是内网IP直接return，不再往下执行
- 请求URL
- 如果有跳转，取出跳转URL，执行第1步

- 当判断完成最后会去请求URL

### 任意文件操作类漏洞

#### 成因

常见的一些java文件操作类的漏洞:任意文件的读取、下载、删除、修改，这类漏洞的成因基本相同，都是因为程序没有对文件和目录的权限进行严格控制，或者说程序没有验证请求的资源文件是否合法导致的。

举例:

#### 任意文件读取

```
@GET
@Path("/images/{image}")
@Produces("images/*")
public Response getImage(@javax.ws.rs.PathParam("image") String image) {
    File file = new File("resources/images/", image); //Weak point

    if (!file.exists()) {
        return Response.status(Status.NOT_FOUND).build();
    }

    return Response.ok().entity(new FileInputStream(file)).build();
}
```

```
def getWordList(value:String) = Action {
  if (!Files.exists(Paths.get("public/lists/" + value))) {
    NotFound("File not found")
  } else {
    val result = Source.fromFile("public/lists/" + value).getLines().mkString // Weak point
    Ok(result)
  }
}
```

#### 任意文件写入

```
file file = new File(getExternalFilesDir(TARGET_TYPE), filename);
fos = new FileOutputStream(file);
fos.write(confidentialData.getBytes());
fos.flush();
```

#### 审计方法

全局搜索关键字或者方法

- FileInputStream
- getPath
- getAbsolutePath
- ServletFileUpload

排查程序的安全策略配置文件，查找permission Java.io.FilePermission，[查看IO方案是否只对程序的绝对路径赋予读写权限。](#)

## 修复方法

- 配置全局安全策略
- 使用File.getCanonicalPath()方法，该方法会对所有别名、快捷方式以及符号链接进行一致地解析。特殊的文件名，例如“..”会被移除。  
### 命令执行漏洞  
#### 成因  
服务端没有针对执行命令的函数进行过滤，导致攻击者可以提交恶意构造语句。java中常见如：Runtime.exec() Process  
ProcessBuilder.start  
#### 举例  
Java中的命令执行离不开调用反射的机制，在实际的场景往往离不开反序列化的利用。

```
import java.io.*;
public class DirList {
    public static void main(String[] args) {
        String dir = System.getProperty("dir");
        Process process = null;
        InputStream istream = null;
        try {
            process = Runtime.getRuntime().exec("cmd.exe /c dir" + dir);
            int result = process.waitFor();
            if (result != 0) {
                System.out.println("process error: " + result);
            }
            istream = (result == 0) ? process.getInputStream() : process.getErrorStream();
            byte[] buffer = new byte[512];
            while (istream.read(buffer) != -1) {
                System.out.print(new String(buffer, "gb2312"));
            }
        } catch (IOException e1) {
            e1.printStackTrace();
        } catch (InterruptedException e2) {
            e2.printStackTrace();
        } finally {
            if (istream != null) {
                try {
                    istream.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if (process != null) {
                process.destroy();
            }
        }
    }
}
```

上面的代码利用Runtime.exec()方法调用dir命令。

攻击者可以利用&符号执行多条命令，例如

```
java -Ddir="..\ & whoami
```

#### #### 审计方法

RCE出现的原因和场景很多，以后慢慢学习~

服务端直接存在可执行函数（exec()等），且对传入的参数过滤不严格导致 RCE 漏洞  
 服务端不直接存在可执行函数（exec()等），且对传入的参数过滤不严格导致 RCE 漏洞  
 由表达式注入导致的RCE漏洞，常见的如：OGNL、SpEL、MVEL、EL、FeI、JST+EL等  
 由java后端模板引擎注入导致的 RCE 漏洞，常见的如：Freemarker、Velocity、Thymeleaf等  
 由java一些脚本语言引起的 RCE 漏洞，常见的如：Groovy、JavascriptEngine等  
 由第三方开源组件引起的 RCE 漏洞，常见的如：Fastjson、Shiro、Xstream、Struts2、weblogic等

审计的时候可以重点寻找：

- Runtime.exec()
- Process
- ProcessBuilder.start()
- #### 修复
- 正则表达式匹配用户输入

```
if (!Pattern.matches("[0-9A-Za-z@.]+", dir)) {
```

#### ### 反序列化漏洞

#### #### 成因

当输入的反序列化的数据可被用户控制，那么攻击者即可通过构造恶意输入，让反序列化产生非预期的对象，在此过程中执行构造的任意代码。

#### #### 审计方法

反序列化操作常常出现在**导入模版文件、网络通信、数据传输、日志格式化存储或者数据库存储**等业务功能处,在代码审计时可重点关注一些反序列化操作函数并判断输入是否可控。

- `ObjectInputStream.readObject`
- `ObjectInputStream.readUnshared`
- `XMLDecoder.readObject`
- `XStream.fromXML`
- 第三方jar包:`ObjectMapper.readValue`,jackson中的`JSON.parseObject`,fastjson中的api

#### #### 修复

- 升级服务端所依赖的可能被利用的jar包，包括JDK。
- 在执行反序列前对InputStream对象进行检查过滤

#### ### 中间件漏洞

#### #### 成因

**中间件**是提供系统软件和应用软件之间连接的软件，它将应用程序运行环境与操作系统隔离，从而实现应用程序开发者不必为更多系统问题忧虑，而直接关注该应用程序在解决问题上的能力。容器就是中间件的一种。

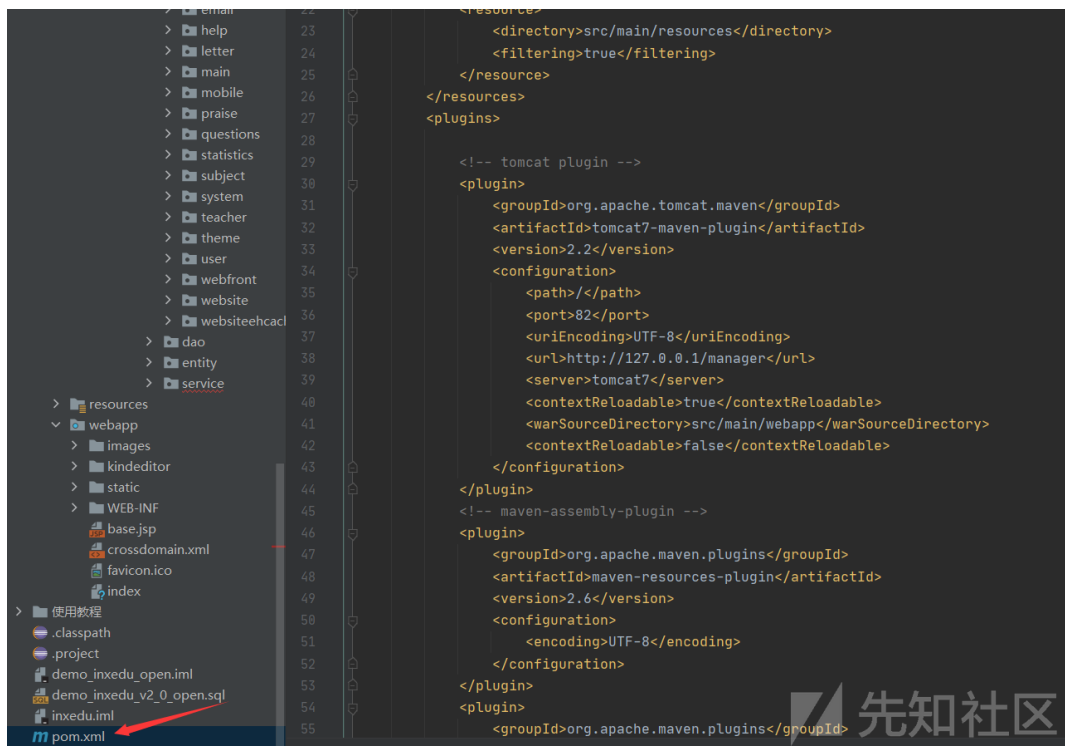
java常见的中间件:



(<https://xzfile.aliyuncs.com/media/upload/picture/20221219151643-16f24ecc-7f6d-1.png>)

#### #### 审计方法

直接打开pom.xml文件查看其使用的中间件及其版本，然后到漏洞库里找漏洞即可。



(<https://xzfile.aliyuncs.com/media/upload/picture/20221219151853-64d72946-7f6d-1.png>)

## 修复

及时更新项目使用的java中间件的版本。

## 业务逻辑漏洞

### 成因

常见的业务逻辑漏洞主要是越权，分为平行越权和垂直越权。在javaweb的各个功能点中都可能存在越权漏洞。主要原因还是因为程序没有对当前用户的权限进行严格控制，或者是后台没有判断当前用户id。

### 审计方法

在每个request.getParameter("userid");之后查看是否有检验当前用户与要进行增删改查的用户。

### 修复

获取当前登陆用户并校验该用户是否具有当前操作权限，并校验请求操作数据是否属于当前登陆用户，当前登陆用户标识不能从用户可控的请求参数中获取。

## 其他漏洞

### ldap注入

就像sql，所有进入到ldap查询的语句都必须要保证安全。不幸的是，ldap没有像sql那样的预编译接口。所以，现在的主要防御方式是，在参数进入ldap查询之前对其进行严格的检验。

有漏洞的代码：

```
NamingEnumeration<SearchResult> answers = context.search("dc=People,dc=example,dc=com",
    "(uid=" + username + ")", ctrls);
```

### jndi注入

Java Naming and Directory Interface，Java命名和目录接口，通过调用JNDI的API应用程序可以定位资源和其他程序对象，现在JNDI能访问的服务有：JDBC、LDAP、RMI、DNS、NIS、CORBA。

JNDI注入的关键是在用户进行远程方法调用时返回的stub是一个reference类型的对象，且用户本地CLASSPATH不存在该类字节码，导致用户需要加载reference类的字节码，直接返回恶意类字节码命令执行

JNDI中有绑定和查找的方法：

- bind：将第一个参数绑定到第二个参数的对象上面
- lookup：通过提供的名称查找对象

目标代码中调用了InitialContext.lookup(URI)，且URI为用户可控；  
攻击者控制URI参数为恶意的RMI服务地址，如：rmi://hacker\_rmi\_server//name；  
攻击者RMI服务器向目标返回一个Reference对象，Reference对象中指定某个精心构造的Factory类；  
目标在进行lookup()操作时，会动态加载并实例化Factory类，接着调用factory.getObjectInstance()获取外部远程对象实例；  
攻击者可以在Factory类文件的构造方法、静态代码块、getObjectInstance()方法等处写入恶意代码，达到RCE的效果；

## rmi反序列化

RMI的主要由三部分组成

- 1.RMI Registry 注册表：服务实例将被注册表注册到特定的名称中（可以理解为电话簿）
- 2.RMI Server 服务端
- 3.RMI Client 客户端：客户端通过查询注册表来获取对应名称的对象引用，以及该对象实现的接口

### 关键类和函数

ObjectOutputStream类的writeObject(Object obj)方法,将对象序列化成字符串数据  
ObjectInputStream类的readObject(Object obj)方法，将字符串数据反序列化成对象  
当然这些函数是rmi里面类会调用了，如果简单使用rmi还用不到这2个函数

### rmi协议的格式

rmi://host:port/name

rmi	协议头	
host	对方ip或域名	(不填默认本地)
port	对方服务的端口号	(不填默认1099)
name	对方服务的类名	

## 所需要的API

java.rmi	客户端所需要的类、接口和异常
java.rmi.server	服务器端所需要的类、接口和异常
java.rmi.registry	注册表的创建查找。命名远程对象的类、接口和异常

## 表达式注入

Spring使用动态值构建。应该严格检验源数据，以避免未过滤的数据进入到危险函数中。

### 关键sink函数

```
SpelExpressionParser  
getValue
```

### 有漏洞的代码

```
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>  
<spring:eval expression="${param.lang}" var="lang" />  
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>  
<spring:eval expression="'${param.lang}'=='fr'" var="languageIsFrench" />
```

### 解决方案：

```
<c:set var="lang" value="${param.lang}"/>  
<c:set var="languageIsFrench" value="${param.lang == 'fr'}"/>
```

一个自动化搜索java敏感函数的脚本:Cryin/JavaID: java source code static code analysis and danger function identify prog  
(github.com) (<https://github.com/Cryin/JavaID>)

```
XXE:
    "SAXReader",
    "DocumentBuilder",
    "XMLStreamReader",
    "SAXBuilder",
    "SAXParser",
    "XMLReader",
    "SAXSource",
    "TransformerFactory",
    "SAXTransformerFactory",
    "SchemaFactory",
    "Unmarshaller",
    "XPathExpression"

JavaObjectDeserialization:
    "readObject",
    "readUnshared",
    "Yaml.load",
    "fromXML",
    "ObjectMapper.readValue",
    "JSON.parseObject"

SSRF:
    "HttpClient",
    "Socket",
    "URL",
    "ImageIO",
    "URLConnection",
    "OkHttpClient"
    "SimpleDriverDataSource.getConnection"
    "DriverManager.getConnection"

FILE:
    "MultipartFile",
    "createNewFile",
    "FileInputStream"

SPelInjection:
    "SpelExpressionParser",
    "getValue"

Autobinding:
    "@SessionAttributes",
    "@ModelAttribute"

URL-Redirect:
    "sendRedirect",
    "forward",
    "setHeader"

EXEC:
    "getRuntime.exec",
    "ProcessBuilder.start",
    "GroovyShell.evaluate"
```

## poc&exp编写

- 编写漏洞检测利用的脚本，不限于py、exe、jar等形式
- 批量检测脚本
- 适配内存马

## 后渗透利用

- 数据库配置文件位置、是否需要解密
- 数据库中业务攻击面获取（例如oa中管理员、用户账户密码等）
- 适配代理工具
- 主机敏感信息获取或进一步扩大利用

## 相关文档输出

- 完整漏洞细节及原理和复现文档
- 完整poc&exp工具利用文档
- 环境搭建到环境密码信息readme 使用文档
- 代码架构整理一份详细的组件分析，包括但不限于使用框架，鉴权特性，风险点，整体路由，重要配置文件等



参考链接

https://www.javasec.org/ (https://www.javasec.org/)  
https://xz.aliyun.com/t/3372 (https://xz.aliyun.com/t/3372)  
https://xz.aliyun.com/t/3460 (https://xz.aliyun.com/t/3460)  
https://xz.aliyun.com/t/3416 (https://xz.aliyun.com/t/3416)  
https://xz.aliyun.com/t/3358 (https://xz.aliyun.com/t/3358)

关注 | 1      点击收藏 | 1

上一篇： CVE-2020-14645 We... (/t/11964)

0 条回复

动动手指，沙发就是你的了！






登录 (https://account.aliyun.com/login/login.htm?oauth\_callback=https%3A%2F%2Fxz.aliyun.com%2Ft%2F11966&from\_type=xianzhi) 后跟帖

先知社区

现在登录 (https://account.aliyun.com/l

社区小黑板 (/notice)

年度贡献榜      月度贡献榜

 LeeH (/u/52868)	3
 Youngmith (/u/56181)	3
 o*区 (/u/64530)	2
 RoboTerh (/u/56486)	2
 000**** (/u/40035)	1

目录

- 前言
- 红队java代码审计生命周期
- 源码获取
- 审计环境
- 代码审计
  - 快速代码审计
  - SpringBoot项目结构
  - 请求传递流程
  - sql注入漏洞

[业务逻辑漏洞](#)  
[其他漏洞](#)  
[poc&exp编写](#)  
[后渗透利用](#)  
[相关文档输出](#)  
[参考链接](#)