Java 安 全 编 程 指 南

	版 本	姓 名	时 间	备 注
创建者	V1.0	吴 友 胜	2016/03/01	创 建
修 改				

目　　录

1. SQL 注入

1.1 漏洞原理

动态应用需对数据库进行查询、更新、删除等操作来实现业务逻辑数据的持久化存储，若应用没有对输入进行正确处理时，恶意输入将作为参数注入到应用对数据库的SQL操作中，导致SQL注入攻击。SQL注入漏洞可用来从数据库获取、修改、导出记录、读写文件、执行命令，甚至获取数据库管理及系统的权限。

1.2 漏洞场景

常见的场景包括：查询商品和个人信息、添加个人订单信息、更新商品信息等，漏洞常见产生原因：

■ SQL 语句使用拼接，没有验证，不使用 SQL 转义

```
String sql = "select * from content where id=" + request.getParameter("id");
ResultSet rs = stmt.executeQuery(sql);
```

■ SQL 语句使用拼接，不使用预编译

```
String sql = "select * from product order by " + request.getParameter("type");
ResultSet rs = stmt.executeQuery(sql);
```

■ 一般情况下错误使用预编译

```
String sql = "select * from product order by " + request.getParameter("type");
PreparedStatement ps = con.prepareStatement(sql);
```

■ Hibernate 中错误使用预编译

```
Query unsafeHQLQuery = session.createQuery("from Inventory where
productID='"+userSuppliedParameter+"'");
contacts = (List)session.createSQLQuery("select * from Contacts where
id="+id).addEntity(Contact.class).list();
```

1.3 漏洞识别

A. 检查输入变量的SQL操作是否有符合性验证；

B. 检查输入变量的SQL操作是否采用了预编译。

1.4 漏洞防御

A. 对输入变量进行符合性验证；

■ 对输入变量进行整形验证

```
int id;
try{
    id = Integer.parseInt(request.getParameter("id"));
} catch(Exception e) {
    id = 0;
}

//检查是否符合业务逻辑
if (id > 100000) {
    return null;
}
```

■ 对输入变量进行格式验证

```
String name = request.getParameter("name");
if (name = = null) {
    return null;
}

//检查长度
```

```
if (name.length() < minLength || name.length() > maxLength) {
    return null;
}

//检查用户名符合规则
String patternName = "^[A-Za-z0-9_-]{4,}$";
String patternMobile = "^\\d{11,11}$";
String patternEmail = "^[A-Za-z0-9._%-]+@[A-Za-z0-9.-]+\\.[a-zA-Z]{2,4}$";
String patternIPAddress = "^(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\\.){3}(?:25
[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$";
String patternIPAddress = "^(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\\.){3}(?:25
[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$";
String patternUrl = "^(^http://([\\w-]+\\.)+[\\w-]+(/[\\w-./?%&=]*)?$";

//检查业务匹配规则
Pattern pattern = Pattern.compile(patternName);
Matcher matcher = pattern.matcher(name);
boolean matchFound = matcher.find();
if (matchFound) {
    return matcher.group(0);
} else {
    return null;
}
```

B. 使用预编译执行SQL操作；

```
int id;
try{
    id = Integer.parseInt(request.getParameter("id"));
} catch(Exception e) {
    id = 0;
}
//检查是否符合业务逻辑
if (id > 100000) {
    return null;
}
//检查是否符合格式
String name=request.getParameter("name");
String patternName = "^[A-Za-z0-9_-]{4,}$";
Pattern pattern = Pattern.compile(patternName);
Matcher matcher = pattern.matcher(name);
boolean matchFound = matcher.find();
if (!matchFound) {
    name=”DEFAULT_VALUE”;
}
使用预编译方法
String sql="select * from content where id=? and name=?";
ps.setInt(1,id);
ps.setString(2,name);
PreparedStatement ps = con.prepareStatement(sql);

order by 也要使用预编译
String sql="select * from content order by ?";
PreparedStatement ps = con.prepareStatement(sql);
ps.setString(1,request.getParameter("type"));

Hibernate 中使用预编译
Query safeHQLQuery = session.createQuery("from Invent where productID=:productid");
safeHQLQuery.setParameter("productid", userSuppliedParameter);

使用安全 API 进行 SQL 转义（不推荐）
import org.owasp.esapi.ESAPI;
import org.owasp.esapi.codecs.*;
type=request.getParameter( "type" );
Codec MYSQL_CODEC = new MySQLCodec(0);
String sql="select * from content where type= "+ESAPI.encoder().encodeForSQL( MYSQL
_CODEC,type)
ResultSet rs=stmt.executeQuery(sql);
```

2. XSS 漏洞

2.1 漏洞原理

由于应用程序没有对输入变量实施正确的转义并输出到页面中，导致攻击者可以提交包含Javascirpt代码的输入，在程序输出时Javascript代码在浏览器中被解析执行。 攻击者可以通过执行Javascript代码窃取用户信息、劫持用户的会话，控制用户的浏览器、挂马等一系列恶意攻击。漏洞分类如下：

A. 反射型：应用将用户输入直接回显给用户，不对用户的输入进行存储。

B. 存储型：应用将用户输入在数据库、文件等系统中持久化存储，并将存储数据显示在用户页面中。

C. Dom 型：DOM型指输入数据不发送到服务器端，服务端应用不处理输入，由客户端浏览器执行。

2.2 漏洞场景

XSS漏洞的产生分为需要服务端应用的交互处理和无需服务端应用的交互处理，交互处理的包括：

◆ ServerToHtml：服务端应用将用户的输入显示在HTML中。

```
<DIV STYLE="<%=style%>">
<p>Orgin html:<%=cid%></p>
<input type="text" name="test_input" value="<%=cid%>">
<a href="http://XX.example.com/<%=cid%>">href</a>
```

◆ ServerToJs：服务端应用将用户的输入显示在Javascript代码或函数调用中。

```
<script>var x='<%=cid%>';</script>
<a href="javascript:test('<%=cid2%>')">Javascript Href</a>
```

◆ ServerToJsToJs指应用将用户输入JS，函数又将用户输入通过dom写到HTML或JS。

```
<a href="javascript:test('<%=cid2%>')">Javascript Href</a>
function test(argA)
{
    document.write(argA);
}
```

◆ Dom类型：客户端代码(javascript)直接对用户的输入进行处理，无需服务端交互处理。

```
g_a=window.location.hash.split("#")[1];
function test_html_encode(a){
    alert("input arg a : "+a);
    document.write("<p>Nothing: "+a+"</p>");
}
function test_html_attribute_encode(a){
    document.write("<p><font size='"+a+"'>testAttr</font></p>");
}

function test_jsstring(a){
    document.write("<script>var t=\""+a+"\";<\/script>");
}
function test_js_function_protocol(a){
    document.write("<a href=\"javascript:test("+a+")\">test</a><br>");
}
(function(){
    var originalUrl = window.location.href,
    toUrl = originalUrl .indexOf('#')!=-1 && originalUrl.split('#')[1];
    if(toUrl) {
        window.location.href = toUrl;
    }
})();
```

2.3 漏洞识别

A. 检查用户输入变量是否进行了类型及输入符合性验证；

B. 检查所有的输出前（所有数据来源含数据库、用户输入）是否使用了正确转义（依据场景）操作。

2.4 漏洞防御

A. 对输入变量进行符合性验证，验证是否符合应用逻辑；

B. 对输出变量【服务器端】使用正确的转义操作；

```
<% String cid=request.getParameter("cid"); %>
<p>Reform HtmlEncode: <%=Reform.HtmlEncode(cid)%></p>
<p>Apache StringEscape: <%=StringEscapeUtils.escapeHtml3(cid)%></p>
```

◆ 输出HTML元素

```
<% String cid=request.getParameter("cid"); %>
double_string_StringEscape_input:
<input type="text" name="test" value="<%=StringEscapeUtils.escapeHtml3(cid)%>"><br>
single_string_StringEscape_html3_input:
<input type="text" name="test" value='<%=StringEscapeUtils.escapeHtml3(cid)%>'><br>

//StringEscape 不会处理单引号'的转义
single_string_StringEscape_html4_input:
<input type="text" name="test" value='<%=StringEscapeUtils.escapeHtml3(cid)%>'><br>
*double_string_Reform_HtmlEncode_input:
<input type="text" name="test" value="<%=Reform.HtmlEncode(cid)%>"><br>
*double_string_Reform_HtmlAttributeEncode_input:
<input type="text" name="test" value="<%=Reform.HtmlAttributeEncode(cid)%>"><br>
```

◆ 输出URL地址

```
<% String cid=request.getParameter("cid"); %>URLEncoder_href:
<a href="http://www.example.com/<%=URLEncoder.encode(cid,"gb2312")%>">href</a>
```

◆ 输出JS正文

```
<% String cid=request.getParameter("cid"); %>
<script>
    var y=<%=Reform.JsString(cid)%>;
    var y='<%=StringEscapeUtils.escapeEcmaScript(cid)%>';
    alert(name);
</script>
```

C. 对输出变量【客户端】使用正确的转义操作。

```
JSstring_href:
<a href="javascript:test(<%=JsString(cid2)%>)">Javascript Href</a><br>
function test(argA)
{
    //然后使用客户端转义
    document.write(HtmlEncode(argA));
}
```

3. 身份认证及会话管理

3.1 漏洞原理

需用户交互的应用涉及用户的身份认证、会话管理，在用户注册时可能采用了不安全的用户注册方式，错误的注册流程，导致用户身份可以伪造，不安全的会话管理将导致用户会话被劫持。漏洞产生原因：

A. 用户名长度及格式受限、导致用户名可被遍历或者被猜测；

B. 不安全的用户激活方式，可能导致用户身份伪造；

C. 不安全的验证码使用，导致验证码可以重复利用错误的伪随机代码使用，导致随机码可以被预知；

D. 会话ID不够随机可被预知或破解，角色转变后的会话ID未变更，导致用户身份被模拟或劫持；

E. 查询敏感信息(用户名）的应用接口无限制，导致用户暴力获取大量敏感信息。

3.2 漏洞场景

A. 用户注册：用户使用邮箱注册并通过链接进行账户激活；

B. 用户登录：用户使用邮箱登录应用；

C. 验 证 码：用户注册、用户登录、敏感操作；

D. 权限管理：订单查询、个人信息查询。

3.3 漏洞识别

见3.1。

3.4 漏洞防御

A. 用户名空间要足够大，应尽量避免使用手机号作为用户名；

B. 使用正确的随机函数，验证码在每次使用后都要进行更新；

```
//使用正确的随机函数
public static final String RandomStrSpace = "ABCDEFGJKLMNPRSTUVWXYZ0123456789;#%^&*
()@$abcdefghtijklmnopqrst";
public static final String RANDOM_ALGORITHM = "SHA1PRNG";
public static final int STR_LENGTH = 15;
public static SecureRandom secureRandom = null;
try {
    secureRandom = SecureRandom.getInstance(RANDOM_ALGORITHM);
} catch (NoSuchAlgorithmException e) {
    e.printStackTrace();
}
char [] buf = new char[STR_LENGTH];
for (int idx = 0; idx < buf.length; ++idx) {
    buf[idx] = RandomStrSpace.charAt(secureRandom.nextInt(RandomStrSpace.length
()));
}
String randomString = new String(buf);
```

C. 用户激活、认证成功后后、角色变化后应及时更新会话ID；

```
//获取登录信息判断业务逻辑
String username = request.getParameter("username");
String password = request.getParameter("password");
String capature = request.getParameter("kaptcha");
if (validate_user(username, password, capature)) {
    //执行验证逻辑
}

//如果会话非空，则表示已经启用了会话，注销会话重新生成
HttpSession sessOrig = request.getSession(false);
if (sessOrig != null) {
    sessOrig.invalidate();
    HttpSession sessLogin = request.getSession(true);
    sessLogin.setAttribute("LOGIN", true);
}
```

D. 查询敏感信息的接口应进行限制：如依据IP的频率限制等。

```
//验证码每次使用后都需要更新
String capature = request.getParameter("kaptcha");
String session_capature = (String)(session.getAttribute("capature");)
session.setAttribure("capature")=null;
if (capature.equals(session_capature)){
    //执行业务逻辑
}
```

4. 不安全的加密和存储

4.1 漏洞原理

应用在存储密码时以明文存储或未加Salt直接用密码进行哈希，导致数据泄漏后可被直接利用。在使用加密、哈希算法时没有采用正确的算法，导致数据可被破解或可逆。漏洞产生原因：

A. 用户密码以明文或可逆密文如AES存储；

B. 没有使用、使用固定或可猜测Salt对密码进行哈希如MD5存储，导致密码可逆；

C. 使用不安全的keyed hashes如SHA1(plain||secret)做签名，导致被篡改数据获取了正确签名；

D. 加密算法使用错误-流加密使用固定密匙，块加密使用固定IV，导致密文被破解或生成指定明文；

4.2 漏洞场景

A. 密码会话：存储用户密码，加密用户会话信息；

B. 伪随机码：随机数的生成方法；

C. 数字签名：使用密匙和哈希算法对数据进行签名。

4.3 漏洞识别

见4.1。

4.4 漏洞防御

A. 安全存储密码：用伪随机算法SecureRandom生成定制Salt，对hash结果进行多次递归处理；

```
//使用安全的随机函数
SecureRandom secureRandom = SecureRandom.getInstance("SHA1PRNG");
byte[] bytes = new byte[512];
secureRandom.nextBytes(bytes);
//对 Hash 结果进行多次递归处理
public static byte[] getHash(int iterationNb, String password, byte[] salt) throws
NoSuchAlgorithmException {
    MessageDigest digest = MessageDigest.getInstance("SHA-1");
    digest.reset();
    digest.update(salt);
    byte[] input = digest.digest(password.getBytes());
    for (int i = 0; i < iterationNb; i++) {
        digest.reset();
        input = digest.digest(input);
    }
    return input;
}
```

B. 尽量不要使用流加密算法，块加密算法不使用固定IV，使用hmac做签名。

```
public static String calculateRFC2104HMAC(String data, String key) throws java.secu
rity.SignatureException {
    String result;
    try {
        SecretKeySpec signingKey = new SecretKeySpec(key.getBytes(), HMAC_SHA1_ALGO
RITHM);
        Mac mac = Mac.getInstance(HMAC_SHA1_ALGORITHM);
        mac.init(signingKey);
        byte[] rawHmac = mac.doFinal(data.getBytes());
        result = Encoding.EncodeBase64(rawHmac);
    } catch (Exception e) {
        throw new SignatureException("Fail to generate HMAC : " + e.getMessage());
    }
    return result;
    }
}
```

5. 不安全的对象直接引用

5.1 漏洞原理

直接将明显的操作对象（数据库字段/文件名）暴露给客户端，给其提示信息，同时应用缺乏必要的权限验证，导致攻击者可以通过猜解、遍历的方式获取大量信息。漏洞常见原因：

A. 对特定的访问对象缺乏权限验证；

示例：addr_id字段为数据库字段，通过遍历此字段来获取其他用户的地址，同时访问没有进行限制

http://www.example.com/myaddress.aspx?addr_id=10000&cid=1&op=bindselected

B. 文件下载中采用文件直接路径。

示例：down.jsp?f=/docs/000003.xls，攻击者通过猜测文件名获取敏感信息或下载系统文件

down.jsp?f=/docs/00000{3,4,5}.xls

down.php?f=/etc/passwd

5.2 漏洞场景

A. 订单查询：用户订单信息的查询；

B. 文件下载：用户下载文件。

5.3 漏洞识别

A. 检查所有操作的访问权限是否进行了正确验证；

B. 是否采用了直接对象的引用方式。

5.4 漏洞防御

A. 检查所有操作的访问权限是否进行了正确验证；

B. 最好不要采用直接引用的方式，应将直接对象引用转化为间接对象引用；

例如：地址字段编号 10000 -> de3v5 ，10001 -> mo9y3。

C. 验证请求对象类型，验证输入是否符合业务逻辑，实施安全的文件访问。

```
String FileName = new String(request.getParameter("filename"));
String FileExt = fileName.substring(fileName.lastIndexOf(".")+1)
//依据逻辑对用户输入的文件名（正则），文件类型（允许的文件类型列表）
if (validate_fileext(FileName, FileExt)) {
    File fileReq = new File(aFileName);
    //安全的获取输入的文件名或路径
    String fileReqName fileReq.getName();
    String uploadPath = this.getServletContext().getRealPath("/")+"/img/";
    //安全的构造需要读取的文件
    File fileLoad = new File(uploadPath, fileReqName);
}
```

6. CSRF 漏洞

6.1 漏洞原理

CSRF跨站伪造请求漏洞，当敏感操作接口未进行身份的二次确认(验证码、一次性token、重复输入密码)时，攻击者通过直接构造恶意页面或引用特定html元素，此页面或元素的访问将导致一次敏感操作的产生，攻击者诱导使用户直接或间接访问此页面或元素，用户以登录身份、角色执行了特定的请求并进行了敏感操作以达到攻击者的目的。CSRF漏洞可产生如转账、自动发布、修改、添加信息等敏感操作，甚至用来添加用户、修改密码等操作。漏洞常见产生原因：

A．敏感操作接口操作未进行身份的二次验证，导致CSRF攻击；

B．只允许post提交，攻击者通过JS构造表单自动执行能越过此限制；

C．检查Header或Referer，特定软件的漏洞可以伪造Header或Referer，越过此限制；

D．检查Cookie中的特定值，Cookie只要属于该网站即会发送，所以检查没有任何用途。

6.2 漏洞场景

A．个人信息页面操作：更新个人信息功能；

B．转账、订单操作：生成订单、修改订单、转账；

C．内容发布：发帖、发留言；

D．用户操作：管理员界面添加、删除、修改用户信息。

6.3 漏洞识别

检查敏感操作是否使用了验证码、一次性token,重复输入密码的方式进行用户身份的验证。错误的防范方式：只允许 post 提交；检查 Header 或 Referer；检查 Cookie 中的特定值，检查没有任何用途。

6.4 漏洞防御

原则： 应至少实现用户会话级别的token，比如用户登录成功后，生成随机字符并存储在用户会话中，在用户每次敏感操作时都应通过http参数的方式携带此值，以确保用户操作未被劫持。

A．一般操作要自动添加CSRF token，Token可以采用会话Token；

◆ 用户登录后生成token

```
//获取用户的输入验证业务逻辑
String user = request.getParameter("user");
String password = request.getParameter("password");
String capature = request.getParameter("kaptcha");
if(validate_user(user, password, capature)) {
    sessLogin = request.getSession(true);
    sessLogin.setAttribute("LOGIN", true);
    //生成随机字符的 token 并存入到用户会话中
    sessLogin.setAttribute("csrf_token",getRandomString(20));
}
```

◆ 页面(模板）渲染时输出token

```
//输出在连接中
<a href=logout.jsp?token=<%sessLogin.getAttribute("token")%>">logout</a>
//输出在 form 表单中
<form method=POST>
发信息:
    <input name="message" type=text>
    <input type="hidden" name="token" value=<%=sessLogin.getAttribute("token")%/>
    <input type=submit value="post">
</form>
```

◆ 逻辑检查中验证token是否正确

```
if (sessLogin.getAttribute("token").equals(request.getParameter("token"))) {
    //处理应用逻辑
}
```

B．重要操作要添加验证码，如修改个人信息；

C．非常重要操作要求用户输入密码进行重复验证，如修改邮箱、转账、下单。

7. 未验证的转发及重定向

7.1 漏洞原理

部分应用会依据用户的输入来进行页面跳转，当应用程序没有对用户的跳转地址进行验证时，攻击者可利用此功能进行钓鱼攻击或其他攻击。漏洞常见产生原因：

A. 客户端没有对输入进行验证，直接进行跳转；

```
String query = request.getQueryString();
if (query.contains("url")) {
    String url = request.getParameter("url");
    response.sendRedirect(url);
}
```

B. 服务器端没有进行跳转地址验证，直接进行跳转。

```
(function(){
    var originalUrl = window.location.href,
    toUrl = originalUrl .indexOf('#')!=-1 && originalUrl.split('#')[1];
    if(toUrl) {
        window.location.href = toUrl;
    }
})();
```

7.2 漏洞场景

A. 客户端使用(JS)从用户端获取输入，并将输入做跳转地址，进行对象访问；

B. 服务端应用程序从用户端获取输入，并将输入做跳转地址，进行登录跳转、对象访问。

7.3 漏洞识别

检查所有使用跳转功能的代码是否对跳转地址进行了验证。

7.4 漏洞防御

对跳转地址进行符合性验证。

```
String query = request.getQueryString();
if (query.contains("url")) {
    String url = request.getParameter("url");
    if (matches("^((http|https)://)?[^.]*\\.example\\.com($|/)"))
        response.sendRedirect(url);
}
```

8. 文件上传

8.1 漏洞原理

由于没有对用户上传的文件进行正确的判断，包括由于错误处理文件名导致的文件读写操作，以及错误的文件类型导致存储了攻击者恶意上传的WEB服务可解析的后门文件、浏览器可执行的HTML文件、FLASH文件，从而导致代码执行或其越权操作。漏洞产生原因：应用没有对上传的文件名进行验证,攻击者可以直接上传以.jsp、.html、swf文件，并获取文件名后执行，或者可以通过写操作覆盖已有的文件。

漏洞场景：文件上传漏洞主要出现在允许用户上传头像、图片、文件等类型的场景中。

```
//上传文件的目录
String uploadPath = this.getServletContext().getRealPath("/")+"/img/";
DiskFileItemFactory factory = new DiskFileItemFactory();
```

```
ServletFileUpload upload = new ServletFileUpload(factory);
List<FileItem> items = upload.parseRequest(request);
Iterator<FileItem> i = items.iterator();
while (i.hasNext()) {
    FileItem fi = (FileItem) i.next();
    fileName = fi.getName();
    //直接使用用户输入的文件名
    File savedFile = new File(uploadPath, fileName);
    fi.write(savedFile);
    break;
}
response.setContentType("text/html;charset=GBK");
System.out.println("download file name: "+ fileName);
response.getWriter().print("上传: <a href='download?name="+fileName+"'>下载</a>");
```

8.2 漏洞识别

A. 是否对用户上传的文件进行了类型限制；

B. 是否对用户上传的文件是否对文件名进行了随机处理；

C. 是否将用户上传文件的目录与Web服务器分离或将上传目录设置为不可执行应用代码。

8.3 漏洞防御

A. 使用白名单方式对上传的文件名后缀进行检查；

B. 上传文件要存贮在资源服务器，尽量避免存放于Web相同的服务器，尤其是服务端的可执行目录下；

C. 文件名要随机，不使用上传的文件名。

```
String uploadPath = this.getServletContext().getRealPath("/")+"/img/";
ArrayList<String> extList = new ArrayList<String>();
extList.add("jpg");
extList.add("gif");
boolean allow = false;
DiskFileItemFactory factory = new DiskFileItemFactory();
ServletFileUpload upload = new ServletFileUpload(factory);
List<FileItem> items = upload.parseRequest(request);
Iterator<FileItem> i = items.iterator();
while (i.hasNext()) {
    FileItem fi = (FileItem) i.next();
    fileName = fi.getName();
    if (fileName == null) {
        return false;
    }
    //获取文件的扩展名，并进白名单的验证
    String fileExt=fileName.substring(fileName.lastIndexOf(".")+1);
    For (String tmp :extList) {
        If (tmp.equals(fileExt)) {
            allow = true;
            break;
        }
    }
    if (!allow) {
        return false;
    }

    //生成一个随机文件名
    SecureRandom secureRandom = SecureRandom.getInstance("SHA1PRNG");
    Long randFilename=secureRandom.nextLong();
    String downFileName=randfileName+"."+fileExt;
    //存储文件
    File savedFile = new File(uploadPath, downfileName);
    fi.write(savedFile);
    return true;
 }
```

9. XXE 注入

9.1 漏洞原理

用户的输入为恶意的xml【场景】，未经正确配置的server端xml解析器直接将xml输入解析执行，由于在解析过程中会对外部实体进行引用导致代码执行、文件读取或拒绝服务。漏洞产生原因：

server端xml解析没有进行正确配置，导致xml输入中的外部实体引用被解析、执行。

```
import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.InputStream;
import org.xml.sax.helpers.DefaultHandler;
import javax.xml.parsers.ParserConfigurationException;
import org.xml.sax.InputSource;
import org.xml.sax.SAXException;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
import org.xml.sax.XMLReader;
import org.xml.sax.EntityResolver;

public class testXXE {
    private static void receiveXMLStream(InputStream inStream, DefaultHandler defHa
ndler) throws ParserConfigurationException, SAXException, IOException {
        SAXParserFactory spf = SAXParserFactory.newInstance();
        SAXParser saxParser = spf.newSAXParser();
        XMLReader reader = saxParser.getXMLReader();

        //reader.setErrorHandler(new ErrorHandler());
        InputSource is = new InputSource(inStream);
        reader.parse(is);
    }

    public static void main(String[] args) {
    String evilxmlString = "<?xml version=\"1.0\"?>"+ "<!DOCTYPE foo SYSTEM \"file:
/dev/tty\">]>";
    InputStream is = new ByteArrayInputStream(xmlString.getBytes());
    try {
        receiveXMLStream(is, null);
    } catch (ParserConfigurationException e) {
        e.printStackTrace();
    } catch (SAXException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace(); // MalformedURLException
    }
        System.out.println("done!");
    }
}
```

9.2 漏洞识别

是否对server端xml解析器进行了正确配置。

9.3 漏洞防御

正确配置server端xml解析器，不支持外部实体引用或通过XMLReader、DOM解析。

```
import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.InputStream;
import org.xml.sax.helpers.DefaultHandler;
import javax.xml.parsers.ParserConfigurationException;
import org.xml.sax.InputSource;
import org.xml.sax.SAXException;
```

```
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
import org.xml.sax.XMLReader;
import org.xml.sax.EntityResolver;

class CustomResolver implements EntityResolver {
    public InputSource resolveEntity(String publicId, String systemId) throws SAXEx
ception, IOException {
        //允许引用的外部文件
        String entityPath = "allowxxefilefile";
        If (systemId.equals(entityPath)) {
            System.out.println("Resolving entity: " + publicId + " " + systemId);
            return new InputSource(entityPath);
        } else {
            return new InputSource();
        }
    }
}

public class testxxe {
    private static void receiveXMLStream(InputStream inStream, DefaultHandler
defHandler) throws ParserConfigurationException, SAXException, IOException {
    SAXParserFactory spf = SAXParserFactory.newInstance();
    SAXParser saxParser = spf.newSAXParser();
    XMLReader reader = saxParser.getXMLReader();
    //指定自定义的实体解析器
    reader.setEntityResolver(new CustomResolver());
    InputSource is = new InputSource(inStream);
    reader.parse(is);
 }

 public static void main(String[] args) {
    String evilxmlString = "<?xml version=\"1.0\"?>"+"<!DOCTYPE foo SYSTEM \"file:/
//c:/boot.ini\">]>";
    InputStream is = new ByteArrayInputStream(xmlString.getBytes());
    try {
        receiveXMLStream(is, null);
    } catch (ParserConfigurationException e) {
        e.printStackTrace();
    } catch (SAXException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
    System.out.println("done!");
}
```