java代码审计手书(二）

niexinming / 2018-11-24 08:07:00 / 浏览数 4066 技术文章 技术文章 顶(0) 踩(0)

翻译自：http://find-sec-bugs.github.io/bugs.htm
翻译：聂心明

xml解析导致xxe漏洞

漏洞特征：XXE_XMLSTREAMREADER
当xml解析程序收到不信任的输入且如果xml解析程序支持外部实体解析的时候，那么造成xml实体解析攻击（xxe）
危害1：探测本地文件内容（xxe：xml 外部实体）

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ENTITY xxe SYSTEM "file:///etc/passwd" > ]>
<foo>&xxe;</foo>
```

危害2：拒绝服务攻击（xee：xml 外部实体膨胀）

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
<!ENTITY lol "lol">
<!ELEMENT lolz (#PCDATA)>
<!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
<!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;">
<!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
[...]
<!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">
]>
<lolz>&lol9;</lolz>
```

解决方法：
为了避免解析xml带来的攻击，你应该按照下面的示例代码修改你的代码
有漏洞的代码：

```
public void parseXML(InputStream input) throws XMLStreamException {

   XMLInputFactory factory = XMLInputFactory.newFactory();
   XMLStreamReader reader = factory.createXMLStreamReader(input);
   [...]
}
```

禁用外部实体的解决方案：

```
public void parseXML(InputStream input) throws XMLStreamException {

   XMLInputFactory factory = XMLInputFactory.newFactory();
   factory.setProperty(XMLInputFactory.IS_SUPPORTING_EXTERNAL_ENTITIES, false);
   XMLStreamReader reader = factory.createXMLStreamReader(input);
   [...]
}
```

禁用DTD的方案：

```
public void parseXML(InputStream input) throws XMLStreamException {

   XMLInputFactory factory = XMLInputFactory.newFactory();
   factory.setProperty(XMLInputFactory.SUPPORT_DTD, false);
   XMLStreamReader reader = factory.createXMLStreamReader(input);
   [...]
}
```

引用：
CWE-611: Improper Restriction of XML External Entity Reference ('XXE')
CERT: IDS10-J. Prevent XML external entity attacks
OWASP.org: XML External Entity (XXE) Processing
WS-Attacks.org: XML Entity Expansion

WS-Attacks.org: XML External Entity DOS
WS-Attacks.org: XML Entity Reference Attack
Identifying Xml eXternal Entity vulnerability (XXE)
JEP 185: Restrict Fetching of External XML Resources

xml解析导致xxe漏洞(XPathExpression)

漏洞特征：XXE_XPATH
当xml解析程序收到不信任的输入且如果xml解析程序支持外部实体解析的时候，那么造成xml实体解析攻击（xxe）
危害1：探测本地文件内容（xxe：xml 外部实体）

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ENTITY xxe SYSTEM "file:///etc/passwd" > ]>
<foo>&xxe;</foo>
```

危害2：拒绝服务攻击（xee：xml 外部实体膨胀）

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
<!ENTITY lol "lol">
<!ELEMENT lolz (#PCDATA)>
<!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
<!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;">
<!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
[...]
<!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">
]>
<lolz>&lol9;</lolz>
```

解决方法：
为了避免解析xml带来的攻击，你应该按照下面的示例代码修改你的代码
有漏洞的代码：

```
DocumentBuilder builder = df.newDocumentBuilder();

XPathFactory xPathFactory = XPathFactory.newInstance();
XPath xpath = xPathFactory.newXPath();
XPathExpression xPathExpr = xpath.compile("/somepath/text()");

xPathExpr.evaluate(new InputSource(inputStream));
```

下面的两个片段展示了可能的解决方案。你可以设置其中一个，或者两个都设置

使用"Secure processing" 模式的解决方案

```
DocumentBuilderFactory df = DocumentBuilderFactory.newInstance();
df.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, true);
DocumentBuilder builder = df.newDocumentBuilder();

[...]

xPathExpr.evaluate( builder.parse(inputStream) );
```

禁用DTD的解决方案：
通过禁用DTD，大多数的xxe攻击都可以被避免

```
DocumentBuilderFactory df = DocumentBuilderFactory.newInstance();
spf.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);
DocumentBuilder builder = df.newDocumentBuilder();

[...]

xPathExpr.evaluate( builder.parse(inputStream) );
```

引用：
CWE-611: Improper Restriction of XML External Entity Reference ('XXE')
CERT: IDS10-J. Prevent XML external entity attacks
OWASP.org: XML External Entity (XXE) Processing
WS-Attacks.org: XML Entity Expansion

WS-Attacks.org: XML External Entity DOS
WS-Attacks.org: XML Entity Reference Attack
Identifying Xml eXternal Entity vulnerability (XXE)
XML External Entity (XXE) Prevention Cheat Sheet_Prevention_Cheat_Sheet#XPathExpression)

xml解析导致xxe漏洞(SAXParser)

漏洞特征：XXE_SAXPARSER
当xml解析程序收到不信任的输入且如果xml解析程序支持外部实体解析的时候，那么造成xml实体解析攻击（xxe）
危害1：探测本地文件内容（xxe：xml 外部实体）

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ENTITY xxe SYSTEM "file:///etc/passwd" > ]>
<foo>&xxe;</foo>
```

危害2：拒绝服务攻击（xee：xml 外部实体膨胀）

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
<!ENTITY lol "lol">
<!ELEMENT lolz (#PCDATA)>
<!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
<!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;">
<!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
[...]
<!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">
]>
<lolz>&lol9;</lolz>
```

解决方法：
为了避免解析xml带来的攻击，你应该按照下面的示例代码修改你的代码
有漏洞的代码：

```
SAXParser parser = SAXParserFactory.newInstance().newSAXParser();

parser.parse(inputStream, customHandler);
```

下面的两个片段展示了可能的解决方案。你可以使用其中一个，或者两个都使用

使用"Secure processing" 模式的解决方案:
这个设置能保护你能避免拒绝服务攻击和ssrf漏洞

```
SAXParserFactory spf = SAXParserFactory.newInstance();
spf.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, true);
SAXParser parser = spf.newSAXParser();

parser.parse(inputStream, customHandler);
```

禁用DTD的解决方案:
通过禁用DTD，大多数的xxe攻击都可以被避免

```
SAXParserFactory spf = SAXParserFactory.newInstance();
spf.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);
SAXParser parser = spf.newSAXParser();

parser.parse(inputStream, customHandler);
```

引用：
CWE-611: Improper Restriction of XML External Entity Reference ('XXE')
CERT: IDS10-J. Prevent XML external entity attacks
OWASP.org: XML External Entity (XXE) Processing
WS-Attacks.org: XML Entity Expansion
WS-Attacks.org: XML External Entity DOS
WS-Attacks.org: XML Entity Reference Attack
Identifying Xml eXternal Entity vulnerability (XXE)
Xerces complete features list

xml解析导致xxe漏洞(XMLReader)

漏洞特征：XXE_XMLREADER
当xml解析程序收到不信任的输入且如果xml解析程序支持外部实体解析的时候，那么造成xml实体解析攻击（xxe）
危害1：探测本地文件内容（xxe：xml 外部实体）

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ENTITY xxe SYSTEM "file:///etc/passwd" > ]>
<foo>&xxe;</foo>
```

危害2：拒绝服务攻击（xee：xml 外部实体膨胀）

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
<!ENTITY lol "lol">
<!ELEMENT lolz (#PCDATA)>
<!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
<!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;">
<!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
[...]
<!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">
]>
<lolz>&lol9;</lolz>
```

解决方法：
为了避免解析xml带来的攻击，你应该按照下面的示例代码修改你的代码
有漏洞的代码：

```
XMLReader reader = XMLReaderFactory.createXMLReader();
reader.setContentHandler(customHandler);
reader.parse(new InputSource(inputStream));
```

下面的两个片段展示了可能的解决方案。你可以使用其中一个，或者两个都使用

使用"Secure processing" 模式的解决方案:
这个设置能保护你能避免拒绝服务攻击和ssrf漏洞

```
XMLReader reader = XMLReaderFactory.createXMLReader();
reader.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, true);
reader.setContentHandler(customHandler);

reader.parse(new InputSource(inputStream));
```

禁用DTD的解决方案:
通过禁用DTD，大多数的xxe攻击都可以被避免

```
XMLReader reader = XMLReaderFactory.createXMLReader();
reader.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);
reader.setContentHandler(customHandler);

reader.parse(new InputSource(inputStream));
```

引用：
CWE-611: Improper Restriction of XML External Entity Reference ('XXE')
CERT: IDS10-J. Prevent XML external entity attacks
OWASP.org: XML External Entity (XXE) Processing
WS-Attacks.org: XML Entity Expansion
WS-Attacks.org: XML External Entity DOS
WS-Attacks.org: XML Entity Reference Attack
Identifying Xml eXternal Entity vulnerability (XXE)
Xerces complete features list

xml解析导致xxe漏洞(DocumentBuilder)

漏洞特征：XXE_DOCUMENT
当xml解析程序收到不信任的输入且如果xml解析程序支持外部实体解析的时候，那么造成xml实体解析攻击（xxe）
危害1：探测本地文件内容（xxe：xml 外部实体）

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ENTITY xxe SYSTEM "file:///etc/passwd" > ]>
```

```
<foo>&xxe;</foo>
```

危害2：拒绝服务攻击（xee：xml 外部实体膨胀）

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
<!ENTITY lol "lol">
<!ELEMENT lolz (#PCDATA)>
<!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
<!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;">
<!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
[...]
<!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">
]>
<lolz>&lol9;</lolz>
```

解决方法：
为了避免解析xml带来的攻击，你应该按照下面的示例代码修改你的代码
有漏洞的代码：

```
DocumentBuilder db = DocumentBuilderFactory.newInstance().newDocumentBuilder();

Document doc = db.parse(input);
```

下面的两个片段展示了可能的解决方案。你可以使用其中一个，或者两个都使用

使用"Secure processing" 模式的解决方案:
这个设置能保护你能避免拒绝服务攻击和ssrf漏洞

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
dbf.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, true);
DocumentBuilder db = dbf.newDocumentBuilder();

Document doc = db.parse(input);
```

禁用DTD的解决方案:
通过禁用DTD，大多数的xxe攻击都可以被避免

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
dbf.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);
DocumentBuilder db = dbf.newDocumentBuilder();

Document doc = db.parse(input);
```

引用：
CWE-611: Improper Restriction of XML External Entity Reference ('XXE')
CERT: IDS10-J. Prevent XML external entity attacks
OWASP.org: XML External Entity (XXE) Processing
WS-Attacks.org: XML Entity Expansion
WS-Attacks.org: XML External Entity DOS
WS-Attacks.org: XML Entity Reference Attack
Identifying Xml eXternal Entity vulnerability (XXE)
Xerces complete features list

xml解析导致xxe漏洞(TransformerFactory)

漏洞特征：XXE_DTD_TRANSFORM_FACTORY
当xml解析程序收到不信任的输入且如果xml解析程序支持外部实体解析的时候，那么造成xml实体解析攻击（xxe）
危害1：探测本地文件内容（xxe：xml 外部实体）

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ENTITY xxe SYSTEM "file:///etc/passwd" > ]>
<foo>&xxe;</foo>
```

危害2：拒绝服务攻击（xee：xml 外部实体膨胀）

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
<!ENTITY lol "lol">
<!ELEMENT lolz (#PCDATA)>
```

```
<!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
<!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;">
<!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
[...]
<!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">
]>
<lolz>&lol9;</lolz>
```

解决方法：
为了避免解析xml带来的攻击，你应该按照下面的示例代码修改你的代码
有漏洞的代码：

```
Transformer transformer = TransformerFactory.newInstance().newTransformer();
transformer.transform(input, result);
```

下面的两个片段展示了可能的解决方案。你可以使用其中一个，或者两个都使用

使用"Secure processing" 模式的解决方案:
这个设置能保护你能避免拒绝服务攻击和ssrf漏洞

```
TransformerFactory factory = TransformerFactory.newInstance();
factory.setAttribute(XMLConstants.ACCESS_EXTERNAL_DTD, "all");
factory.setAttribute(XMLConstants.ACCESS_EXTERNAL_STYLESHEET, "all");

Transformer transformer = factory.newTransformer();
transformer.setOutputProperty(OutputKeys.INDENT, "yes");

transformer.transform(input, result);
```

禁用DTD的解决方案:
通过禁用DTD，大多数的xxe攻击都可以被避免

```
TransformerFactory factory = TransformerFactory.newInstance();
factory.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, true);

Transformer transformer = factory.newTransformer();
transformer.setOutputProperty(OutputKeys.INDENT, "yes");

transformer.transform(input, result);
```

引用：
CWE-611: Improper Restriction of XML External Entity Reference ('XXE')
CERT: IDS10-J. Prevent XML external entity attacks
OWASP.org: XML External Entity (XXE) Processing
WS-Attacks.org: XML Entity Expansion
WS-Attacks.org: XML External Entity DOS
WS-Attacks.org: XML Entity Reference Attack
Identifying Xml eXternal Entity vulnerability (XXE)

XSLT解析导致xxe漏洞(TransformerFactory)

漏洞特征：XXE_XSLT_TRANSFORM_FACTORY
当xml解析程序收到不信任的输入且如果xml解析程序支持外部实体解析的时候，那么造成xml实体解析攻击（xxe）
危害1：探测本地文件内容（xxe：xml 外部实体）

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
      <xsl:value-of select="document('/etc/passwd')">
  </xsl:value-of></xsl:template>
</xsl:stylesheet>
```

解决方法：
为了避免解析xml带来的攻击，你应该按照下面的示例代码修改你的代码
有漏洞的代码：

```
Transformer transformer = TransformerFactory.newInstance().newTransformer();
transformer.transform(input, result);
```

下面的两个片段展示了可能的解决方案。你可以使用其中一个，或者两个都使用

使用"Secure processing" 模式的解决方案:
这个设置能保护你能避免ssrf漏洞但是不能避免拒绝服务攻击

```
TransformerFactory factory = TransformerFactory.newInstance();
factory.setAttribute(XMLConstants.ACCESS_EXTERNAL_DTD, "all");
factory.setAttribute(XMLConstants.ACCESS_EXTERNAL_STYLESHEET, "all");

Transformer transformer = factory.newTransformer();
transformer.setOutputProperty(OutputKeys.INDENT, "yes");

transformer.transform(input, result);
```

禁用DTD的解决方案:
这个设置能保护你能避免ssrf漏洞但是不能避免拒绝服务攻击

```
TransformerFactory factory = TransformerFactory.newInstance();
factory.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, true);

Transformer transformer = factory.newTransformer();
transformer.setOutputProperty(OutputKeys.INDENT, "yes");

transformer.transform(input, result);
```

引用：
CWE-611: Improper Restriction of XML External Entity Reference ('XXE')
CERT: IDS10-J. Prevent XML external entity attacks
OWASP.org: XML External Entity (XXE) Processing
WS-Attacks.org: XML Entity Expansion
WS-Attacks.org: XML External Entity DOS
WS-Attacks.org: XML Entity Reference Attack
Identifying Xml eXternal Entity vulnerability (XXE)

潜在的XPath注入

漏洞特征： XPATH_INJECTION
XPath注入的危险程度就像sql注入一样。如果XPath查询包含不信任的用户输入，那么数据库就会被完全暴露。这样就可以让攻击者访问未授权的数据或者在目标xml数据库

引用：
WASC-39: XPath Injection
OWASP: Top 10 2013-A1-Injection
CWE-643: Improper Neutralization of Data within XPath Expressions ('XPath Injection')
CERT: IDS09-J. Prevent XPath Injection (archive)
Black Hat Europe 2012: Hacking XPath 2.0
Balisage: XQuery Injection

发现Struts 1 服务器端

漏洞特征： STRUTS1_ENDPOINT
这个类是Struts 1 的Action
曾清一个请求被路由到一个控制器中，Form对象将会被自动的实例化为http参数的对象。这些参数应该被严格检查，以保证它们是安全的。

发现Struts 2 服务器端

漏洞特征：STRUTS2_ENDPOINT
在Struts 2中，服务器端是简单的Java对象 (POJOs)，这就意味着没有接口/类 需要被实现/拓展

当一个请求被路由到它的控制器的时候（像这些被选择的类），http提供的参数会被自动的映射到类中的setters中。所以，所有类中的setters都应该被看成来自不被信任源

发现Spring 服务器端

漏洞特征： SPRING_ENDPOINT
这个类是一个Spring的控制器。所有方法的注解都在RequestMapping（还有一些简化注解在GetMapping, PostMapping, PutMapping, DeleteMapping, 和 PatchMapping），这些方法都能被远程访问到。这些类应该被严格的分析，以保证暴露给远程的方法是安全的，不会被攻击者轻易攻击。

Spring关闭 CSRF保护

漏洞特征： SPRING_CSRF_PROTECTION_DISABLED
对于标准的web应用程序来讲，关闭Spring的CSRF保护显然是不安全的。
禁用此保护的有效使用场景是服务器暴露一个可以改变状态的接口，这个接口仅可以被非浏览器操控。

不安全的配置

```
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

   @Override
   protected void configure(HttpSecurity http) throws Exception {
       http.csrf().disable();
   }
}
```

引用：
Spring Security Official Documentation: When to use CSRF protection
OWASP: Cross-Site Request Forgery
OWASP: CSRF Prevention Cheat Sheet
CWE-352: Cross-Site Request Forgery (CSRF)

Spring 中不受CSRF限制的RequestMapping

漏洞特征： SPRING_CSRF_UNRESTRICTED_REQUEST_MAPPING
通过默认的映射所有的HTTP请求方法都会被RequestMapping注解。可是，http请求中的GET, HEAD, TRACE,
和OPTIONS（可能会导致tokens被泄露）方法不会默认开启csrf保护。所以，被RequestMapping注解的可以改变状态的方法和 POST, PUT, DELETE, 或者PATCH这些http请求方法都会受到csrf攻击。

有漏洞的代码：

```
@Controller
public class UnsafeController {

   @RequestMapping("/path")
   public void writeData() {
       // State-changing operations performed within this method.
   }
}
```

解决方案（Spring Framework 4.3和更新的版本）

```
@Controller
public class SafeController {

   /**
    * For methods without side-effects use @GetMapping.
    */
   @GetMapping("/path")
   public String readData() {
       // No state-changing operations performed within this method.
       return "";
   }

   /**
    * For state-changing methods use either @PostMapping, @PutMapping, @DeleteMapping, or @PatchMapping.
    */
   @PostMapping("/path")
   public void writeData() {
       // State-changing operations performed within this method.
   }
}
```

解决方案（在Spring Framework 4.3之前的版本）

```
@Controller
public class SafeController {

   /**
    * For methods without side-effects use either
    * RequestMethod.GET, RequestMethod.HEAD, RequestMethod.TRACE, or RequestMethod.OPTIONS.
```

```
     */
    @RequestMapping(value = "/path", method = RequestMethod.GET)
    public String readData() {
        // No state-changing operations performed within this method.
        return "";
    }

    /**
     * For state-changing methods use either
     * RequestMethod.POST, RequestMethod.PUT, RequestMethod.DELETE, or RequestMethod.PATCH.
     */
    @RequestMapping(value = "/path", method = RequestMethod.POST)
    public void writeData() {
        // State-changing operations performed within this method.
    }
}
```

引用：
Spring Security Official Documentation: Use proper HTTP verbs (CSRF protection))
OWASP: Cross-Site Request Forgery
OWASP: CSRF Prevention Cheat Sheet
CWE-352: Cross-Site Request Forgery (CSRF)

潜在的注入（custom）

漏洞特征： CUSTOM_INJECTION
扫描工具所识别的函数存在注射问题。应验证输入并争取转义。

有漏洞的代码：

```
SqlUtil.execQuery("select * from UserEntity t where id = " + parameterInput);
```

wiki在线有很详细的教程关于如何配置custom

引用：
WASC-19: SQL Injection
OWASP: Top 10 2013-A1-Injection
OWASP: SQL Injection Prevention Cheat Sheet
OWASP: Query Parameterization Cheat Sheet
CAPEC-66: SQL Injection
CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

潜在的sql注入

漏洞特征：SQL_INJECTION
输入进sql查询的数据应该通过严格的检查。在预编译中绑定参数可以更容易的缓解sql注入带来的危害。或者，每一个参数应该被正确的转义。
有漏洞的代码：

```
createQuery("select * from User where id = '"+inputId+"'");
```

解决方案：

```
import org.owasp.esapi.Encoder;

createQuery("select * from User where id = '"+Encoder.encodeForSQL(inputId)+"'");
```

引用（sql注入）
WASC-19: SQL Injection
CAPEC-66: SQL Injection
CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
OWASP: Top 10 2013-A1-Injection
OWASP: SQL Injection Prevention Cheat Sheet
OWASP: Query Parameterization Cheat Sheet

在Turbine中潜在的sql注入

漏洞特征：SQL_INJECTION_TURBINE
输入进sql查询的数据应该通过严格的检查。在预编译中绑定参数可以更容易的缓解sql注入带来的危害。或者，每一个参数应该被正确的转义。
Turbine API 提供DSL在java代码中构建查询

有漏洞的代码：

```
List<Record> BasePeer.executeQuery( "select * from Customer where id=" + inputId );
```

解决方案（使用Criteria DSL）：

```
Criteria c = new Criteria();
c.add( CustomerPeer.ID, inputId );

List<Customer> customers = CustomerPeer.doSelect( c );
```

解决方案（使用特殊方法）：

```
Customer customer = CustomerPeer.retrieveByPK( new NumberKey( inputId ) );
```

解决方法（使用OWASP提供的编码方法）

```
import org.owasp.esapi.Encoder;

BasePeer.executeQuery("select * from Customer where id = '"+Encoder.encodeForSQL(inputId)+"'");
```

引用(Turbine)：
Turbine Documentation: Criteria Howto
引用（sql注入）
WASC-19: SQL Injection
CAPEC-66: SQL Injection
CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
OWASP: Top 10 2013-A1-Injection
OWASP: SQL Injection Prevention Cheat Sheet
OWASP: Query Parameterization Cheat Sheet

潜在的SQL/HQL注入(Hibernate)

漏洞特征：SQL_INJECTION_HIBERNATE
输入进sql查询的数据应该通过严格的检查。在预编译中绑定参数可以更容易的缓解sql注入带来的危害。或者，可以使用Hibernate的Criteria。
有漏洞的代码：

```
Session session = sessionFactory.openSession();
Query q = session.createQuery("select t from UserEntity t where id = " + input);
q.execute();
```

解决方案：

```
Session session = sessionFactory.openSession();
Query q = session.createQuery("select t from UserEntity t where id = :userId");
q.setString("userId",input);
q.execute();
```

动态查询参数法解决方案（Hibernate Criteria）

```
Session session = sessionFactory.openSession();
Query q = session.createCriteria(UserEntity.class)
   .add( Restrictions.like("id", input) )
   .list();
q.execute();
```

引用(Hibernate)
Hibernate Documentation: Query Criteria
Hibernate Javadoc: Query Object
HQL for pentesters: Guideline to test if the suspected code is exploitable.
引用（sql注入）
WASC-19: SQL Injection
CAPEC-66: SQL Injection
CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
OWASP: Top 10 2013-A1-Injection
OWASP: SQL Injection Prevention Cheat Sheet
OWASP: Query Parameterization Cheat Sheet

潜在的sql/JDOQL注入(JDO)

漏洞特征：SQL_INJECTION_JDO
输入进sql查询的数据应该通过严格的检查。在预编译中绑定参数可以更容易的缓解sql注入带来的危害。
有漏洞的代码：

```
PersistenceManager pm = getPM();

Query q = pm.newQuery("select * from Users where name = " + input);
q.execute();
```

解决方案：

```
PersistenceManager pm = getPM();

Query q = pm.newQuery("select * from Users where name = nameParam");
q.declareParameters("String nameParam");
q.execute(input);
```

引用(JDO)：
JDO: Object Retrieval
引用（sql注入）
WASC-19: SQL Injection
CAPEC-66: SQL Injection
CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
OWASP: Top 10 2013-A1-Injection
OWASP: SQL Injection Prevention Cheat Sheet
OWASP: Query Parameterization Cheat Sheet

潜在的sql/JPQL注入(JPA)

漏洞特征： SQL_INJECTION_JPA
输入进sql查询的数据应该通过严格的检查。在预编译中绑定参数可以更容易的缓解sql注入带来的危害。
有漏洞的代码：

```
EntityManager pm = getEM();

TypedQuery<UserEntity> q = em.createQuery(
   String.format("select * from Users where name = %s", username),
   UserEntity.class);

UserEntity res = q.getSingleResult();
```

解决方案：

```
TypedQuery<UserEntity> q = em.createQuery(
   "select * from Users where name = usernameParam",UserEntity.class)
   .setParameter("usernameParam", username);

UserEntity res = q.getSingleResult();
```

引用 (JPA)
The Java EE 6 Tutorial: Creating Queries Using the Java Persistence Query Language
引用（sql注入）
WASC-19: SQL Injection
CAPEC-66: SQL Injection
CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
OWASP: Top 10 2013-A1-Injection
OWASP: SQL Injection Prevention Cheat Sheet
OWASP: Query Parameterization Cheat Sheet

潜在的JDBC注入(Spring JDBC)

漏洞特征：SQL_INJECTION_SPRING_JDBC
输入进sql查询的数据应该通过严格的检查。在预编译中绑定参数可以更容易的缓解sql注入带来的危害。或者，每一个参数应该被正确的转义。
有漏洞的代码：

```
JdbcTemplate jdbc = new JdbcTemplate();
int count = jdbc.queryForObject("select count(*) from Users where name = '"+paramName+"'", Integer.class);
```

解决方案：

```
JdbcTemplate jdbc = new JdbcTemplate();
int count = jdbc.queryForObject("select count(*) from Users where name = ?", Integer.class, paramName);
```

引用 (Spring JDBC)
Spring Official Documentation: Data access with JDBC
引用 (sql注入)
WASC-19: SQL Injection
CAPEC-66: SQL Injection
CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
OWASP: Top 10 2013-A1-Injection
OWASP: SQL Injection Prevention Cheat Sheet
OWASP: Query Parameterization Cheat Sheet

潜在的JDBC注入

漏洞特征：SQL_INJECTION_JDBC
输入进sql查询的数据应该通过严格的检查。在预编译中绑定参数可以更容易的缓解sql注入带来的危害。
有漏洞的代码：

```
Connection conn = [...];
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("update COFFEES set SALES = "+nbSales+" where COF_NAME = '"+coffeeName+"'");
```

解决方案：

```
Connection conn = [...];
conn.prepareStatement("update COFFEES set SALES = ? where COF_NAME = ?");
updateSales.setInt(1, nbSales);
updateSales.setString(2, coffeeName);
```

引用 (JDBC)
Oracle Documentation: The Java Tutorials > Prepared Statements
引用 (sql注入)
WASC-19: SQL Injection
CAPEC-66: SQL Injection
CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
OWASP: Top 10 2013-A1-Injection
OWASP: SQL Injection Prevention Cheat Sheet
OWASP: Query Parameterization Cheat Sheet

潜在的Scala Slick注入

漏洞特征：SCALA_SQL_INJECTION_SLICK
输入进sql查询的数据应该通过严格的检查。在预编译中绑定参数可以更容易的缓解sql注入带来的危害。
有漏洞的代码：

```
db.run {
 sql"select * from people where name = '#$value'".as[Person]
}
```

解决方案：

```
db.run {
 sql"select * from people where name = $value".as[Person]
}
```

引用 (sql注入)
WASC-19: SQL Injection
CAPEC-66: SQL Injection
CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
OWASP: Top 10 2013-A1-Injection
OWASP: SQL Injection Prevention Cheat Sheet
OWASP: Query Parameterization Cheat Sheet

潜在的Scala Anorm注入

漏洞特征：SCALA_SQL_INJECTION_ANORM
输入进sql查询的数据应该通过严格的检查。在预编译中绑定参数可以更容易的缓解sql注入带来的危害。
有漏洞的代码：

```
val peopleParser = Macro.parser[Person]("id", "name", "age")

DB.withConnection { implicit c =>
 val people: List[Person] = SQL("select * from people where name = '" + value + "'").as(peopleParser.*)
}
```

解决方案：

```
val peopleParser = Macro.parser[Person]("id", "name", "age")

DB.withConnection { implicit c =>
 val people: List[Person] = SQL"select * from people where name = $value".as(peopleParser.*)
}
```

引用（sql注入）
WASC-19: SQL Injection
CAPEC-66: SQL Injection
CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
OWASP: Top 10 2013-A1-Injection
OWASP: SQL Injection Prevention Cheat Sheet
OWASP: Query Parameterization Cheat Sheet

潜在的安卓sql注入

漏洞特征：SQL_INJECTION_ANDROID
输入进sql查询的数据应该通过严格的检查。在预编译中绑定参数可以更容易的缓解sql注入带来的危害。
有漏洞的代码：

```
String query = "SELECT * FROM  messages WHERE uid= '"+userInput+"'" ;
Cursor cursor = this.getReadableDatabase().rawQuery(query,null);
```

解决方案：

```
String query = "SELECT * FROM  messages WHERE uid= ?" ;
Cursor cursor = this.getReadableDatabase().rawQuery(query,new String[] {userInput});
```

引用 (Android SQLite)
InformIT.com: Practical Advice for Building Secure Android Databases in SQLite
Packtpub.com: Knowing the SQL-injection attacks and securing our Android applications from them

Android Database Support (Enterprise Android: Programming Android Database Applications for the Enterprise)

Safe example of Insert, Select, Update and Delete queryies provided by Suragch

引用（sql注入）
WASC-19: SQL Injection
CAPEC-66: SQL Injection
CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
OWASP: Top 10 2013-A1-Injection
OWASP: SQL Injection Prevention Cheat Sheet
OWASP: Query Parameterization Cheat Sheet

潜在的LDAP注入

漏洞特征：LDAP_INJECTION
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ladp查询之前对其进行严
有漏洞的代码：

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

引用：
WASC-29: LDAP Injection
OWASP: Top 10 2013-A1-Injection
CWE-90: Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection')
LDAP Injection Guide: Learn How to Detect LDAP Injections and Improve LDAP Security

上一篇：SUCTF招新赛PWN_萌新友好版WP 下一篇：VIRTUALBOX 3D加速：让...
1. 0 条回复

- 动动手指，沙发就是你的了！

登录 后跟帖

先知社区

现在登录

热门节点

技术文章

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板