java代码审计手书(四）

niexinming / 2018-11-30 07:52:00 / 浏览数 4662 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

外部文件访问(Android)

漏洞特征：ANDROID_EXTERNAL_FILE_ACCESS
应用经常往外部存储上写数据（可能是SD卡），这个操作可能会有多个安全问题。首先应用可以可以通过READ_EXTERNAL_STORAGE
获取SD卡上存储的文件。而且如果数据中包含用户的敏感信息的话，那么需要把这些数据加密。
有漏洞的代码：

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

更好的措施：

```
fos = openFileOutput(filename, Context.MODE_PRIVATE);
fos.write(string.getBytes());
```

引用：
Android Official Doc: Security Tips
CERT: DRD00-J: Do not store sensitive information on external storage
Android Official Doc: Using the External Storage
OWASP Mobile Top 10 2014-M2: Insecure Data Storage
CWE-312: Cleartext Storage of Sensitive Information

Broadcast漏洞(Android)

漏洞规则：ANDROID_BROADCAST
所有应用通过申请适当权限就可以监听Broadcast的意图，所以尽量不要通过Broadcast传输敏感数据。
有漏洞的代码：

```
Intent i = new Intent();
i.setAction("com.insecure.action.UserConnected");
i.putExtra("username", user);
i.putExtra("email", email);
i.putExtra("session", newSessionId);

this.sendBroadcast(v1);
```

解决方案（如果有可能的话）：

```
Intent i = new Intent();
i.setAction("com.secure.action.UserConnected");

sendBroadcast(v1);
```

配置（接收者）

```
<manifest ...>

   <!-- Permission declaration -->
   <permission android:name="my.app.PERMISSION" />

   <receiver
       android:name="my.app.BroadcastReceiver"
       android:permission="my.app.PERMISSION"> <!-- Permission enforcement -->
       <intent-filter>
           <action android:name="com.secure.action.UserConnected" />
       </intent-filter>
   </receiver>
```

```
   ...
</manifest>
```

配置（发送者）

```
<manifest>
   <!-- We declare we own the permission to send broadcast to the above receiver -->
   <uses-permission android:name="my.app.PERMISSION"/>

   <!-- With the following configuration, both the sender and the receiver apps need to be signed by the same developer certif
   <permission android:name="my.app.PERMISSION" android:protectionLevel="signature"/>
</manifest>
```

引用：
CERT: DRD03-J. Do not broadcast sensitive information using an implicit intent
Android Official Doc: BroadcastReceiver (Security)
Android Official Doc: Receiver configuration (see android:permission)
[1] StackOverflow: How to set permissions in broadcast sender and receiver in android
CWE-925: Improper Verification of Intent by Broadcast Receiver
CWE-927: Use of Implicit Intent for Sensitive Communication

任意文件写 (Android)

创建文件使用MODE_WORLD_READABLE模式，可以让文件写入环境中的任意位置。一些文件文件被改写的话，可能会发生一些不希望发生的事情。
有漏洞代码：

```
fos = openFileOutput(filename, MODE_WORLD_READABLE);
fos.write(userInfo.getBytes());
```

解决方案（使用MODE_PRIVATE）：

```
fos = openFileOutput(filename, MODE_PRIVATE);
```

解决方案（使用本地SQLite数据库）
使用本地SQLite数据库可能是存储结构数据最好的解决方案了。要确定数据库文件不会被创建到外部存储中。见下面的开发文档引用

引用：
CERT: DRD11-J. Ensure that sensitive data is kept secure
Android Official Doc: Security Tips
Android Official Doc: Context.MODE_PRIVATE
vogella.com: Android SQLite database and content provider - Tutorial
OWASP Mobile Top 10 2014-M2: Insecure Data Storage
CWE-312: Cleartext Storage of Sensitive Information

已激活地理位置的WebView(Android)

漏洞特征：ANDROID_GEOLOCATION
建议去询问用户是否能获取他们的位置信息
漏洞代码：

```
webView.setWebChromeClient(new WebChromeClient() {
   @Override
   public void onGeolocationPermissionsShowPrompt(String origin, GeolocationPermissions.Callback callback) {
       callback.invoke(origin, true, false);
   }
});
```

建议代码：
限制使用地理位置的例子，并且要得到用户的确认

```
webView.setWebChromeClient(new WebChromeClient() {
   @Override
   public void onGeolocationPermissionsShowPrompt(String origin, GeolocationPermissions.Callback callback) {
       callback.invoke(origin, true, false);

       //Ask the user for confirmation
   }
});
```

引用：
CERT: DRD15-J. Consider privacy concerns when using Geolocation API
Wikipedia: W3C Geolocation API
W3C: Geolocation Specification

允许JavaScript脚本运行的webview (Android)

漏洞特征：ANDROID_WEB_VIEW_JAVASCRIPT
WebView如果允许允许JavaScript脚本的话，就意味着它会受到xss的影响。应该检查页面的渲染，以避免潜在的反射型xss，存储型xss，dom型xss。

```
WebView myWebView = (WebView) findViewById(R.id.webView);
WebSettings webSettings = myWebView.getSettings();
webSettings.setJavaScriptEnabled(true);
```

有漏洞的代码：
允许JavaScript运行是一个坏的习惯。这就意味着后端代码需要被审计，以避免xss。xss也会使用dom xss的形式引入到客户端。

```
function updateDescription(newDescription) {
   $("#userDescription").html("<p>"+newDescription+"</p>");
}
```

引用：
Issue: Using setJavaScriptEnabled can introduce XSS vulnerabilities
Android Official Doc: WebView
WASC-8: Cross Site Scripting
OWASP: XSS Prevention Cheat Sheet
OWASP: Top 10 2013-A3: Cross-Site Scripting (XSS)
CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')

带有JavaScript接口的WebView (Android)

漏洞特征:ANDROID_WEB_VIEW_JAVASCRIPT_INTERFACE
使用JavaScript接口可能会将WebView暴露给有危害的api。如果在WebView中触发xss的话，恶意的JavaScript代码会钓鱼一些敏感的类。
有漏洞代码：

```
WebView myWebView = (WebView) findViewById(R.id.webView);

myWebView.addJavascriptInterface(new FileWriteUtil(this), "fileWriteUtil");

WebSettings webSettings = myWebView.getSettings();
webSettings.setJavaScriptEnabled(true);

[...]
class FileWriteUtil {
   Context mContext;

   FileOpenUtil(Context c) {
       mContext = c;
   }

   public void writeToFile(String data, String filename, String tag) {
       [...]
   }
}
```

引用：
Android Official Doc: WebView.addJavascriptInterface()
CWE-749: Exposed Dangerous Method or Function

没有用secure标志的cookie

漏洞特征：INSECURE_COOKIE
一个新的cookie的创建应该设置Secure标志。Secure标志命令浏览器确保cookie不会通过不安全的链路发送(http://)
有漏洞的代码：

```
Cookie cookie = new Cookie("userName",userName);
response.addCookie(cookie);
```

解决方案（特殊的设置):

```
Cookie cookie = new Cookie("userName",userName);
cookie.setSecure(true); // Secure flag
cookie.setHttpOnly(true);
```

解决方案（Servlet 3.0 配置）

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee" version="3.0">
[...]
<session-config>
<cookie-config>
 <http-only>true</http-only>
 <secure>true</secure>
</cookie-config>
</session-config>
</web-app>
```

引用：
CWE-614: Sensitive Cookie in HTTPS Session Without 'Secure' Attribute
CWE-315: Cleartext Storage of Sensitive Information in a Cookie
CWE-311: Missing Encryption of Sensitive Data
OWASP: Secure Flag
Rapid7: Missing Secure Flag From SSL Cookie

没有用HttpOnly标志的cookie

漏洞特征：HTTPONLY_COOKIE
一个新的cookie的创建应该设置Secure标志。Secure标志命令浏览器确保cookie不会被恶意脚本读取。当用户是"跨站脚本攻击"的目标的时候，攻击者会获得用户的sessio
id，从而能够接管用户的账户。
有漏洞的代码：

```
Cookie cookie = new Cookie("email",userName);
response.addCookie(cookie);
```

解决方案（特殊的设置):

```
Cookie cookie = new Cookie("email",userName);
cookie.setSecure(true);
cookie.setHttpOnly(true); //HttpOnly flag
```

解决方案（Servlet 3.0 配置）

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee" version="3.0">
[...]
<session-config>
<cookie-config>
 <http-only>true</http-only>
 <secure>true</secure>
</cookie-config>
</session-config>
</web-app>
```

引用：
Coding Horror blog: Protecting Your Cookies: HttpOnly
OWASP: HttpOnly
Rapid7: Missing HttpOnly Flag From Cookie

使用反序列化对象

漏洞特征：OBJECT_DESERIALIZATION
反序列化不受信任的数据可能会导致远程命令执行，如果有可用的执行链，那么就会触发恶意操作。库的开发者在逐渐提高防御策略，以避免潜在的恶意利用。但是还是有一
反序列化是一个敏感的操作，因为历史上曾经有很多比较有名的漏洞都是出自它。web应用是很脆弱的，因为很快java虚拟机里面将会爆发出一波新的漏洞。
有漏洞的代码：

```
public UserData deserializeObject(InputStream receivedFile) throws IOException, ClassNotFoundException {

   try (ObjectInputStream in = new ObjectInputStream(receivedFile)) {
       return (UserData) in.readObject();
   }
}
```

解决方案：
避免反序列从远程用户输入的数据
引用：
CWE-502: Deserialization of Untrusted Data
Deserialization of untrusted data
Serialization and Deserialization
A tool for generating payloads that exploit unsafe Java object deserialization
[1] Example of Denial of Service using the class java.util.HashSet
[2] OpenJDK: Deserialization issue in ObjectInputStream.readSerialData() (CVE-2015-2590)
[3] Rapid7: Sun Java Calendar Deserialization Privilege Escalation (CVE-2008-5353)

不安全的Jackson发序列化配置

漏洞特征：JACKSON_UNSAFE_DESERIALIZATION
如果Jackson databind库被用来反序列不受信任的数据的话，就会导致远程命令执行。如果有可用的执行链，那么就会触发恶意操作。
解决方案：
当通过JsonTypeInfo.Id.NAME使用多态性时，应该明确定义想要的类型和子类型。并且不要调用ObjectMapper.enableDefaultTyping（readValue包含Object 或 Serializable 或 Comparable 或 已知的反序列化类型）
有漏洞的代码：

```
public class Example {
   static class ABean {
       public int id;
       public Object obj;
   }

   static class AnotherBean {
       @JsonTypeInfo(use = JsonTypeInfo.Id.CLASS) // or JsonTypeInfo.Id.MINIMAL_CLASS
       public Object obj;
   }

   public void example(String json) throws JsonMappingException {
        ObjectMapper mapper = new ObjectMapper();
        mapper.enableDefaultTyping();
        mapper.readValue(json, ABean.class);
   }

   public void exampleTwo(String json) throws JsonMappingException {
        ObjectMapper mapper = new ObjectMapper();
        mapper.readValue(json, AnotherBean.class);
   }

}
```

引用：
Jackson Deserializer security vulnerability
Java Unmarshaller Security - Turning your data into code execution

在反序列化漏洞中被利用的类

漏洞特征：DESERIALIZATION_GADGET
反序列化利用链是一些可以被攻击者利用的类，这些类通常存在于远程api中。这些类也会被添自定义行为，目的是用readObject方法去反序列化(Serializable)或者调用来自序列化对象中的方法(InvocationHandler).
这个检查工具主要用于研究人员。真实的场景是反序列化会被用于远程操作。为了减少恶意代码的利用，必须要强制移除利用链中所使用的类。
引用：
CWE-502: Deserialization of Untrusted Data
Deserialization of untrusted data
Serialization and Deserialization
A tool for generating payloads that exploit unsafe Java object deserialization
[1] Example of Denial of Service using the class java.util.HashSet
[2] OpenJDK: Deserialization issue in ObjectInputStream.readSerialData() (CVE-2015-2590)
[3] Rapid7: Sun Java Calendar Deserialization Privilege Escalation (CVE-2008-5353)

违反信任边界

漏洞特征：TRUST_BOUNDARY_VIOLATION
信任边界被认为是通过程序画的一根线。在线的一边，数据是不可信的。在线的另一边，数据是被可信任的。身份效验的目的是为了数据能够安全的通过信任边界-从不信任
有漏洞的代码：

```
public void doSomething(HttpServletRequest req, String activateProperty) {
   //..

   req.getSession().setAttribute(activateProperty,"true");

}

public void loginEvent(HttpServletRequest req, String userSubmitted) {
   //..

   req.getSession().setAttribute("user",userSubmitted);
}
```

解决方案：
解决方案是在设置新的session属性前要添加验证。如果有可能，最好数据是来自安全的地方而不是用户提供的输入数据
引用：
[1] CWE-501: Trust Boundary Violation
OWASP : Trust Boundary Violation

恶意的XSLT

漏洞特征：JSP_XSLT
XSLT(可扩展样式表转换语言)是一种用于将XML 文档转换为其他XML 文档的语言。
xslt的样式表中可能会携带恶意的行为。所以，如果一个攻击者控制了源样式表的内容，那么它可能会触发远程代码执行
有漏洞的代码：

```
<x:transform xml="${xmlData}" xslt="${xsltControlledByUser}" />
```

解决方案：
解决方案确保源样式表来自安全的源，并且保证不会有类似于路径穿透的漏洞。
引用
[1] Wikipedia: XSLT (Extensible Stylesheet Language Transformations)
Offensive XSLT by Nicolas Gregoire
[2] From XSLT code execution to Meterpreter shells by Nicolas Gregoire
XSLT Hacking Encyclopedia by Nicolas Gregoire
Acunetix.com : The hidden dangers of XSLTProcessor - Remote XSL injection
w3.org XSL Transformations (XSLT) Version 1.0 : w3c specification
[3] WASC: Path Traversal
[4] OWASP: Path Traversal

恶意的XSLT

漏洞特征：MALICIOUS_XSLT
XSLT(可扩展样式表转换语言)是一种用于将XML 文档转换为其他XML 文档的语言。
xslt的样式表中可能会携带恶意的行为。所以，如果一个攻击者控制了源样式表的内容，那么它可能会触发远程代码执行
有漏洞的代码：

```
Source xslt = new StreamSource(new FileInputStream(inputUserFile)); //Dangerous source to validate

Transformer transformer = TransformerFactory.newInstance().newTransformer(xslt);

Source text = new StreamSource(new FileInputStream("/data_2_process.xml"));
transformer.transform(text, new StreamResult(...));
```

解决方案：
解决方案确保源样式表来自安全的源，并且保证不会有类似于路径穿透的漏洞。
引用
[1] Wikipedia: XSLT (Extensible Stylesheet Language Transformations)
Offensive XSLT by Nicolas Gregoire
[2] From XSLT code execution to Meterpreter shells by Nicolas Gregoire
XSLT Hacking Encyclopedia by Nicolas Gregoire
Acunetix.com : The hidden dangers of XSLTProcessor - Remote XSL injection
w3.org XSL Transformations (XSLT) Version 1.0 : w3c specification
[3] WASC: Path Traversal
[4] OWASP: Path Traversal

潜藏在Scala Play中的信息泄露

漏洞特征：SCALA_SENSITIVE_DATA_EXPOSURE
应用总是无意识的泄露一些配置信息，比如内部结构或者通过各种应用问题侵犯隐私。
基于各种有效的输入数据页面会返回不同的返回数据，尤其当机密数据被当成结果被web应用展示出来的时候，就会导致信息的泄露。
敏感数据包括（不仅仅是列出来的这些）：api密钥，密码，产品版本，环境配置。
有漏洞的代码：

```
def doGet(value:String) = Action {
 val configElement = configuration.underlying.getString(value)

 Ok("Hello "+ configElement +" !")
}
```

应用配置的关键部分不应该被输出到返回数据报文中，并且用户也不能操作那些被用于代码的关键配置。
引用：
OWASP: Top 10 2013-A6-Sensitive Data Exposure
[1] OWASP: Top 10 2007-Information Leakage and Improper Error Handling
[2] WASC-13: Information Leakage
CWE-200: Information Exposure

Scala Play服务器端请求伪造(SSRF)

漏洞特征：SCALA_PLAY_SSRF
当服务器端发送一个请求，这个请求的目标地址是用户输入指定的，且这个请求没有被严格的效验时，就会发生服务器端请求伪造漏洞。这个漏洞允许攻击者用你的web服务
有漏洞代码：

```
def doGet(value:String) = Action {
   WS.url(value).get().map { response =>
       Ok(response.body)
   }
}
```

解决方案/对策

- 不要让用户控制请求的目的地址
- 接受一个目的地址的key，使用这个key去查找合法的目的地址
- urls地址白名单（如果可能的话）
- 用白名单校验url地址开头的部分

引用：
CWE-918: Server-Side Request Forgery (SSRF)
Understanding Server-Side Request Forgery

URLConnection中的服务器端请求伪造(SSRF) 和任意文件访问

漏洞特征：SCALA_PLAY_SSRF
当服务器端发送一个请求，这个请求的目标地址是用户输入指定的，且这个请求没有被严格的效验时，就会发生服务器端请求伪造漏洞。这个漏洞允许攻击者用你的web服务
URLConnection能够使用file://协议获取其他的协议去访问本地的文件系统和其他的服务
有漏洞代码：

```
new URL(String url).openConnection()
new URL(String url).openStream()
new URL(String url).getContent()
```

解决方案/对策

- 不要让用户控制请求的目的地址
- 接受一个目的地址的key，使用这个key去查找合法的目的地址
- urls地址白名单（如果可能的话）
- 用白名单校验url地址开头的部分

引用：
CWE-918: Server-Side Request Forgery (SSRF)
Understanding Server-Side Request Forgery
CWE-73: External Control of File Name or Path
Abusing jar:// downloads

在Scala Twirl模板引擎里面潜在的xss

漏洞规则：SCALA_XSS_TWIRL
可能会有潜在的xss漏洞。这可能会在客户端执行未期望的JavaScript。（见引用）
有漏洞的代码：

```
@(value: Html)

@value
```

解决方案：

```
@(value: String)

@value
```

抵御xss最好的方式是像上面在输出中编码特殊的字符。有4种环境类型要考虑：HTML, JavaScript, CSS (styles), 和URLs.请遵守OWASP XSS Prevention备忘录中定义的xss保护规则，里面会介绍一些重要的防御细节。
引用：
WASC-8: Cross Site Scripting
OWASP: XSS Prevention Cheat Sheet
OWASP: Top 10 2013-A3: Cross-Site Scripting (XSS)
CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
OWASP Java Encoder

在Scala MVC API引擎里面潜在的xss

漏洞规则： SCALA_XSS_MVC_API
可能会有潜在的xss漏洞。这可能会在客户端执行未期望的JavaScript。（见引用）
有漏洞的代码：

```
def doGet(value:String) = Action {
   Ok("Hello " + value + " !").as("text/html")
 }
```

解决方案：

```
def doGet(value:String) = Action {
   Ok("Hello " + Encode.forHtml(value) + " !")
 }
```

抵御xss最好的方式是像上面在输出中编码特殊的字符。有4种环境类型要考虑：HTML, JavaScript, CSS (styles), 和URLs.请遵守OWASP XSS Prevention备忘录中定义的xss保护规则，里面会介绍一些重要的防御细节。
引用：
WASC-8: Cross Site Scripting
OWASP: XSS Prevention Cheat Sheet
OWASP: Top 10 2013-A3: Cross-Site Scripting (XSS)
CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
OWASP Java Encoder

在Velocity中潜在的模板注入

漏洞特征：TEMPLATE_INJECTION_VELOCITY
Velocity模板引擎非常强大。你可以在模板中使用条件判断，循环，外部函数调用等逻辑代码。它里面也没有一个沙箱去限制操作。一个恶意的用户如果可以控制模板，那么
有漏洞的代码：

```
[...]

Velocity.evaluate(context, swOut, "test", userInput);
```

解决方案：
避免让终端用户操作Velocity中的模板。如果你需要让你的用户去操作模板，那么最好限制模板引擎的能力，就像Handlebars 或 Moustache 一样（见引用）
引用：
PortSwigger: Server-Side Template Injection
Handlebars.java

在Freemarker中潜在的模板注入

漏洞特征：TEMPLATE_INJECTION_FREEMARKER
Freemarker模板引擎非常强大。你可以在模板中使用条件判断，循环，外部函数调用等逻辑代码。它里面也没有一个沙箱去限制操作。一个恶意的用户如果可以控制模板，
有漏洞的代码：

```
Template template = cfg.getTemplate(inputTemplate);
[...]
template.process(data, swOut);
```

解决方案：
避免让终端用户操作Freemarker中的模板。如果你需要让你的用户去操作模板，那么最好限制模板引擎的能力，就像Handlebars 或 Moustache 一样（见引用）
引用：
PortSwigger: Server-Side Template Injection
Handlebars.java

过度宽松的cors策略

漏洞规则：PERMISSIVE_CORS
在html5之前，web浏览器强制使用同源策略，目的是保证JavaScript能够访问web页面的内容，JavaScript和web页面的起源必须来自于同一个域下。如果没有同源策略，
有漏洞的代码：

```
response.addHeader("Access-Control-Allow-Origin", "*");
```

解决方案：
避免在Access-Control-Allow-Origin这个头中使用*，这表示运行在其他域下的任何JavaScript都可以访问这个域下的应用数据
引用：
W3C Cross-Origin Resource Sharing
Enable Cross-Origin Resource Sharing

匿名的LDAP绑定

漏洞特征：LDAP_ANONYMOUS
没有做合适的访问控制，攻击者可以滥用ldap配置，让ldap服务器执行一段包含用户控制的代码。所有依赖ctx的ldap查询都可以以不需要用户认证和访问控制的方式去执行
有漏洞代码：

```
...
env.put(Context.SECURITY_AUTHENTICATION, "none");
DirContext ctx = new InitialDirContext(env);
...
```

解决方案：
考虑ldap中其他的用户认证模式并且确保有合适的访问控制

引用：
Ldap Authentication Mechanisms

ldap 入口投毒

漏洞特征： LDAP_ENTRY_POISONING
JNDI api支持在ldap目录上绑定序列化对象。如果提供确定的属性，反序列化对象将会被用于应用数据的查询（详细信息见Black Hat USA 2016
白皮书）。反序列化对象是一个有风险的操作，他可能会导致远程代码执行。
如果攻击者获得ldap基本查询的入口点，那么这个漏洞就可能会被利用。通过添加一个属性给已存在的ldap入口或者通过配置应用，就可以恶意的使用ldap服务器了。
有漏洞的代码：

```
DirContext ctx = new InitialDirContext();
//[...]

ctx.search(query, filter,
       new SearchControls(scope, countLimit, timeLimit, attributes,
           true, //Enable object deserialization if bound in directory
           deref));
```

解决方案：

```
DirContext ctx = new InitialDirContext();
//[...]

ctx.search(query, filter,
       new SearchControls(scope, countLimit, timeLimit, attributes,
           false, //Disable
           deref));
```

引用：
Black Hat USA 2016: A Journey From JNDI/LDAP Manipulation to Remote Code Execution Dream Land (slides & video) by Alvaro Muñoz and

Oleksandr Mirosh

HP Enterprise: Introducing JNDI Injection and LDAP Entry Poisoning by Alvaro Muñoz

TrendMicro: How The Pawn Storm Zero-Day Evaded Java's Click-to-Play Protection by Jack Tang

使用持久性的cookie

漏洞特征：COOKIE_PERSISTENT
将敏感数据存储在持久性的cookie中会危害到数据的保密性和账户的安全性
解释：
如果隐私信息被存储在持久性的cookie中，攻击者就会利用这个巨大的时间窗口来窃取数据，尤其持久性cookie会在用户的电脑中保存非常长的一段时间。持久性cookie一
持久性cookie会被经常使用，目的是为了在用户和网站互动时能够分析用户的行为。依靠持久性cookie去追踪数据，这可能已经侵犯了用户的隐私
有漏洞的代码：下面的代码可以让cookie保存一年

```
[...]
Cookie cookie = new Cookie("email", email);
cookie.setMaxAge(60*60*24*365);
[...]
```

解决方案:

- 在有必要的时候使用持久性cookie，并且要限制最大过期时间
- 不要在敏感上使用持久性cookie

引用：
Class Cookie setMaxAge documentation
CWE-539: Information Exposure Through Persistent Cookies

url重写方法

漏洞规则：URL_REWRITING
该方法的实现包括确定是否需要在URL中编码session ID的逻辑。
url重写已经是非常严重的安全问题了，因为session ID 出现在url中，这就很容易被第三方获取到。在url中的session ID会以很多种的方式被暴露。

- 日志
- 浏览器历史
- 复制粘贴到邮件中或者文章中
- http的Referrer头中

有漏洞的代码：

```
out.println("Click <a href=" +
               res.encodeURL(HttpUtils.getRequestURL(req).toString()) +
               ">here</a>");
```

解决方案：
避免使用这些方法，如果您要编码URL字符串或表单参数，请不要将URL重写方法与URLEncoder类混淆。
引用：
OWASP Top 10 2010-A3-Broken Authentication and Session Management

不安全的SMTP SSL链接

漏洞特征：INSECURE_SMTP_SSL
当进行ssl连接时，服务器会禁用身份验证。一些启用ssl连接的邮件库默认情况下不会验证服务器的证书。这就等于信任所有的证书。当试图去连接服务器的时候，应用会很
有漏洞的代码：

```
...
Email email = new SimpleEmail();
email.setHostName("smtp.servermail.com");
email.setSmtpPort(465);
email.setAuthenticator(new DefaultAuthenticator(username, password));
email.setSSLOnConnect(true);
email.setFrom("user@gmail.com");
email.setSubject("TestMail");
email.setMsg("This is a test mail ... :-)");
email.addTo("foo@bar.com");
email.send();
```

```
...
```

解决方案：
请添加验证服务器证书的模块

```
email.setSSLCheckServerIdentity(true);
```

引用：
CWE-297: Improper Validation of Certificate with Host Mismatch

AWS查询注入

漏洞特征：AWS_QUERY_INJECTION
如果SimpleDB数据库查询字符串中包含用户输入的话就会让攻击者查看未授权的记录。
下面这个例子就是动态的创建查询字符串并且执行SimpleDB的select()查询，这个查询中允许用户指定productCategory。攻击者可以修改查询，绕过customerID的身份验
有漏洞的代码：

```
...
String customerID = getAuthenticatedCustomerID(customerName, customerCredentials);
String productCategory = request.getParameter("productCategory");
...
AmazonSimpleDBClient sdbc = new AmazonSimpleDBClient(appAWSCredentials);
String query = "select * from invoices where productCategory = '"
           + productCategory + "' and customerID = '"
           + customerID + "' order by '"
           + sortColumn + "' asc";
SelectResult sdbResult = sdbc.select(new SelectRequest(query));
```

解决方案：
这个问题类似于sql注入，在进入SimpleDB数据库查询语句的之前要过滤用户的输入
引用：
CWE-943: Improper Neutralization of Special Elements in Data Query Logic

JavaBeans属性注入

漏洞特征：BEAN_PROPERTY_INJECTION
攻击者可以设置任意bean的属性，这样会降低系统的完整性。Bean的population函数允许设置bean的属性或者嵌套属性。
攻击者会影响这个函数从而去访问特殊的bean属性，比如class。类加载器允许他去操控系统属性并且会有潜在的执行任意代码的可能性。
有漏洞的代码：

```
MyBean bean = ...;
HashMap map = new HashMap();
Enumeration names = request.getParameterNames();
while (names.hasMoreElements()) {
   String name = (String) names.nextElement();
   map.put(name, request.getParameterValues(name));
}
BeanUtils.populate(bean, map);
```

解决方案：
避免使用用户能够控制的数据去设置Bean属性的名称

引用：
CWE-15: External Control of System or Configuration Setting

Struts敏感文件暴露

漏洞特征：STRUTS_FILE_DISCLOSURE
用户通过输入去访问服务器端的任意路径，这样会允许攻击者下载服务器端的任意文件（包含应用的类文件或者jar文件），或者直接查看在保护目录下的文件。
攻击者可能会伪造请求去寻找服务器中敏感的文件。例如，请求"http://example.com/?returnURL=WEB-INF/applicationContext.xml"，服务器就会展示出applicationC
有漏洞的代码：

```
...
String returnURL = request.getParameter("returnURL");
Return new ActionForward(returnURL);
...
```

解决方案：
避免把用户输入的数据放入路径查询字符串之中。

引用：
CWE-552: Files or Directories Accessible to External Parties

Spring敏感文件暴露

漏洞特征：SPRING_FILE_DISCLOSURE
用户通过输入去访问服务器端的任意路径，这样会允许攻击者下载服务器端的任意文件（包含应用的类文件或者jar文件），或者直接查看在保护目录下的文件。
攻击者可能会伪造请求去寻找服务器中敏感的文件。例如，请求"http://example.com/?returnURL=WEB-INF/applicationContext.xml"，服务器就会展示出applicationC
有漏洞的代码：

```
...
String returnURL = request.getParameter("returnURL");
return new ModelAndView(returnURL);
...
```

解决方案：
避免把用户输入的数据放入路径查询字符串之中。

引用：
CWE-552: Files or Directories Accessible to External Parties

RequestDispatcher敏感文件暴露

漏洞特征：REQUESTDISPATCHER_FILE_DISCLOSURE
用户通过输入去访问服务器端的任意路径，这样会允许攻击者下载服务器端的任意文件（包含应用的类文件或者jar文件），或者直接查看在保护目录下的文件。
攻击者可能会伪造请求去寻找服务器中敏感的文件。例如，请求http://example.com/?jspFile=../applicationContext.xml%3F"，服务器就会展示出applicationContext.xm
有漏洞的代码：

```
...
String jspFile = request.getParameter("jspFile");
request.getRequestDispatcher("/WEB-INF/jsps/" + jspFile + ".jsp").include(request, response);
...
```

解决方案：
避免把用户输入的数据放入路径查询字符串之中。

引用：
CWE-552: Files or Directories Accessible to External Parties

格式化字符串操作

漏洞特征：FORMAT_STRING_MANIPULATION
如果用户输入能够控制格式化字符串参数的话，那么攻击者这个漏洞让应用抛出异常或者泄露信息。
攻击者可能会改变格式化字符串的参数，比如可以让应用抛出错误。如果错误没有被捕获，那么应用就会崩溃。
此外，如果敏感信息保留在内存中的话，那么攻击者就会改变格式化字符串去泄露敏感数据。
下面这个示例代码是让用户指定一个浮点数来展示余额，实际上，用户输入任何东西都会让应用抛出异常从而导致显示失败。甚至，更有害的例子是，如果攻击者输入"2f %3$s %4$.2"，那么格式化字符串就会变成"The customer: %s %s has the balance %4$.2f %3$s %4$.2"。这就会导致在输出结果中显示敏感的账户ID。
有漏洞代码：

```
Formatter formatter = new Formatter(Locale.US);
String format = "The customer: %s %s has the balance %4$." + userInput + "f";
formatter.format(format, firstName, lastName, accountNo, balance);
```

解决方案：
避免让用户输入控制格式化字符串参数
引用:
CWE-134: Use of Externally-Controlled Format String

http参数被污染

漏洞特征：HTTP_PARAMETER_POLLUTION
将未验证的用户输入直接拼接到url中，这会让攻击者操控请求参数的值。攻击者可能会操控已存在参数的值，注入新的参数或者利用非变量字典中的参数。http参数污染(HPP)
攻击包含将已编码的查询字符串分隔符注入其他现有参数。如果应用没有过滤用户输入，那么恶意的用户就可以构造特殊的输入攻击服务器端或者客户端程序。
在下面的例子中，程序员可能没有考虑到攻击者会给参数lang输入en&user_id=1，这可能会让他的用户id发生改变。
有漏洞代码：

```
String lang = request.getParameter("lang");
GetMethod get = new GetMethod("http://www.host.com");
get.setQueryString("lang=" + lang + "&user_id=" + user_id);
get.execute();
```

解决方案：
在使用http参数之前过滤用户输入数据
引用：
CAPEC-460: HTTP Parameter Pollution (HPP)

通过报错泄露敏感信息

漏洞特征：INFORMATION_EXPOSURE_THROUGH_AN_ERROR_MESSAGE
在用户看来敏感信息是非常有价值的（比如密码），或者它可能会对其他平台有用，更多的情况下，会引发非常致命的攻击。如果攻击失败，攻击者就会参考服务器提供的错
有漏洞的代码：

```
try {
 out = httpResponse.getOutputStream()
} catch (Exception e) {
 e.printStackTrace(out);
}
```

引用:
CWE-209: Information Exposure Through an Error Message

SMTP 头部注入

漏洞特征：SMTP_HEADER_INJECTION
简单邮件传输协议 (SMTP) 是基于纯文本协议来投递邮件的。就像http，头部字段被new line
所分割。如果用户输入被放置到邮件的头部，那么应用应该删除或者替换掉new line字符串(CR / LF)。你应该使用安全的封装，比如 Apache Common Email
和Simple Java Mail ，这些库会过滤掉那些会导致头部注入的特殊字符。
有漏洞代码：

```
Message message = new MimeMessage(session);
message.setFrom(new InternetAddress("noreply@your-organisation.com"));
message.setRecipients(Message.RecipientType.TO, new InternetAddress[] {new InternetAddress("target@gmail.com")});
message.setSubject(usernameDisplay + " has sent you notification"); //Injectable API
message.setText("Visit your ACME Corp profile for more info.");
Transport.send(message);
```

解决方案：
使用Apache Common Email 或Simple Java Mail

引用：
OWASP SMTP Injection)
CWE-93: Improper Neutralization of CRLF Sequences ('CRLF Injection')
Commons Email: User Guide
Simple Java Mail Website
StackExchange InfoSec: What threats come from CRLF in email generation?

上一篇：XCTF BCTF 2018 W... 下一篇：windows内核系列五: 从wi...

1. 1 条回复


如风 2018-11-30 17:10:30

文章不错，从1看到4了。感谢分享！

0 回复Ta

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