java代码审计手书(三）

niexinming / 2018-11-28 07:43:00 / 浏览数 4006 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

在使用脚本引擎的时潜在的代码注入

漏洞特征：SCRIPT_ENGINE_INJECTION
请严格的评估动态代码。应该仔细分析代码的结构。恶意代码的执行会导致数据的泄露或者执行任意系统指令。
如果你想动态的运行代码，那么请找一个沙箱（见引用）
有害的代码：

```
public void runCustomTrigger(String script) {
   ScriptEngineManager factory = new ScriptEngineManager();
   ScriptEngine engine = factory.getEngineByName("JavaScript");

   engine.eval(script); //Bad things can happen here.
}
```

解决方案：
使用"Cloudbees Rhino Sandbox"库就能安全的评估Javascript代码

```
public void runCustomTrigger(String script) {
   SandboxContextFactory contextFactory = new SandboxContextFactory();
   Context context = contextFactory.makeContext();
   contextFactory.enterContext(context);
   try {
       ScriptableObject prototype = context.initStandardObjects();
       prototype.setParentScope(null);
       Scriptable scope = context.newObject(prototype);
       scope.setPrototype(prototype);

       context.evaluateString(scope,script, null, -1, null);
   } finally {
       context.exit();
   }
}
```

引用：
Cloudbees Rhino Sandbox: Utility to create sandbox with Rhino (block access to all classes)
CodeUtopia.net: Sandboxing Rhino in Java
Remote Code Execution .. by design：里面有一些恶意代码的例子。这些例子能测试沙箱的规则
CWE-94: Improper Control of Generation of Code ('Code Injection')
CWE-95: Improper Neutralization of Directives in Dynamically Evaluated Code ('Eval Injection')

使用Spring表达式时潜在的代码注入(SpEL表达式注入）

漏洞规则：SPEL_INJECTION
Spring表达式被用来构建动态的值。源数据应该被严格的检验，以避免未过滤的时候进入到表达式的执行器中
有漏洞的代码：

```
public void parseExpressionInterface(Person personObj,String property) {

       ExpressionParser parser = new SpelExpressionParser();

       //Unsafe if the input is control by the user..
       Expression exp = parser.parseExpression(property+" == 'Albert'");

       StandardEvaluationContext testContext = new StandardEvaluationContext(personObj);
       boolean result = exp.getValue(testContext, Boolean.class);
[...]
```

引用：
CWE-94: Improper Control of Generation of Code ('Code Injection')
CWE-95: Improper Neutralization of Directives in Dynamically Evaluated Code ('Eval Injection')

Spring Expression Language (SpEL) - Official Documentation
Minded Security: Expression Language Injection
Remote Code Execution .. by design: 里面有一些恶意代码的例子。这些例子能测试沙箱的规则.
Spring Data-Commons: (CVE-2018-1273)
Spring OAuth2: CVE-2018-1260

使用表达式语言时潜在的代码注入（EL）

漏洞特征：EL_INJECTION

表达式语言被用来构建动态的值。源数据应该被严格的检验，以避免未过滤的时候进入到表达式的执行器中
有漏洞代码：

```
public void evaluateExpression(String expression) {
   FacesContext context = FacesContext.getCurrentInstance();
   ExpressionFactory expressionFactory = context.getApplication().getExpressionFactory();
   ELContext elContext = context.getELContext();
   ValueExpression vex = expressionFactory.createValueExpression(elContext, expression, String.class);
   return (String) vex.getValue(elContext);
}
```

引用：
Minded Security: Abusing EL for executing OS commands
The Java EE 6 Tutorial: Expression Language
CWE-94: Improper Control of Generation of Code ('Code Injection')
CWE-95: Improper Neutralization of Directives in Dynamically Evaluated Code ('Eval Injection')
Minded Security: Expression Language Injection
Dan Amodio's blog: Remote Code with Expression Language Injection
Remote Code Execution .. by design: 里面有一些恶意代码的例子。这些例子能测试沙箱的规则.

潜在于Seam logging call中的代码注入

漏洞特征：SEAM_LOG_INJECTION
Seam Logging API支持表达式语言的解析，目的是引出bean的property到日志消息中去。源数据会利用表达式执行未期望的代码。
在这个代码片段里面，表达式语言被用来构建动态的值。源数据应该被严格的检验，以避免未过滤的时候进入到表达式的执行器中
有漏洞的代码：

```
public void logUser(User user) {
   log.info("Current logged in user : " + user.getUsername());
   //...
}
```

解决方案：

```
public void logUser(User user) {
   log.info("Current logged in user : #0", user.getUsername());
   //...
}
```

引用：
JBSEAM-5130: Issue documenting the risk
JBoss Seam: Logging (Official documentation)
The Java EE 6 Tutorial: Expression Language
CWE-94: Improper Control of Generation of Code ('Code Injection')
CWE-95: Improper Neutralization of Directives in Dynamically Evaluated Code ('Eval Injection')

使用OGNL表达式时潜在的代码注入

漏洞规则： OGNL_INJECTION
表达式语言被用来构建动态的值。源数据应该被严格的检验，以避免未过滤的时候进入到表达式的执行器中
有漏洞代码：

```
public void getUserProperty(String property) {
 [...]
 //The first argument is the dynamic expression.
 return ognlUtil.getValue("user."+property, ctx, root, String.class);
}
```

解决方案
一般，解析OGNL表达式的函数不应该接收用户的输入。它旨在被用于静态配置和jsp。

引用：
HP Enterprise: Struts 2 OGNL Expression Injections by Alvaro MuÃ±oz
Gotham Digital Science: An Analysis Of CVE-2017-5638
Apache Struts2: Vulnerability S2-016
Apache Struts 2 Documentation: OGNL

潜在的http返回报文被分割

漏洞特征：HTTP_RESPONSE_SPLITTING
当http请求包含未期望的CR 和
LF字符的时候，服务器可能会把返回的报文流解析成两个HTTP返回报文（而不是一个）。攻击者可以控制第二个报文并且发动诸如xss攻击或者缓存投毒攻击。按照OWASP
EE应用服务器所修复，但还是要严格检验输入。如果你关注这个漏洞，你应该测试你算关心的那个平台，看看这个平台是否允许CR或者
LF被注入到返回报文的头部中。这个漏洞常常被报告为低危，如果你使用有漏洞的平台，请仔细检查低危告警。
有漏洞代码：

```
String author = request.getParameter(AUTHOR_PARAMETER);
// ...
Cookie cookie = new Cookie("author", author);
response.addCookie(cookie);
```

引用:
OWASP: HTTP Response Splitting
CWE-113: Improper Neutralization of CRLF Sequences in HTTP Headers ('HTTP Response Splitting')
CWE-93: Improper Neutralization of CRLF Sequences ('CRLF Injection')

在日志中潜在的CRLF注入

漏洞规则：CRLF_INJECTION_LOGS
当未被信任的输入数据进入到日志中，并且没有正确的做过滤。那么攻击者就可以伪造日志数据或者包含恶意内容。插入恶意的实体通常被用于歪曲统计，分散管理员注意力
有漏洞的代码：

```
String val = request.getParameter("user");
String metadata = request.getParameter("metadata");
[...]
if(authenticated) {
   log.info("User " + val + " (" + metadata + ") was authenticated successfully");
}
else {
   log.info("User " + val + " (" + metadata + ") was not authenticated");
}
```

恶意用户可能会发送这样的metadata数据："Firefox) was authenticated successfully\r\n[INFO] User bbb (Internet Explorer".

解决方案：
你要手工过滤每一个参数

```
log.info("User " + val.replaceAll("[\r\n]","") + " (" + userAgent.replaceAll("[\r\n]","") + ") was not authenticated");
```

你要配置日志服务器，目的是替换掉所有消息中的\r\n。 OWASP Security Logging已经在Logback 和 Log4j.实现了这个功能。
引用：
CWE-117: Improper Output Neutralization for Logs
CWE-93: Improper Neutralization of CRLF Sequences ('CRLF Injection')
CWE-93: Improper Neutralization of CRLF Sequences ('CRLF Injection')
OWASP Security Logging

潜在的外部控制配置

漏洞特征：EXTERNAL_CONFIG_CONTROL
允许外部控制系统设置会导致系统的中断或者导致应用行为异常，和潜在的恶意行为。攻击者通过提供不存在的catalog名称可能会导致错误，或者链接到未授权的数据库服
有漏洞的代码：

```
conn.setCatalog(request.getParameter("catalog"));
```

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

坏的十六进制数据

漏洞特征： BAD_HEXA_CONVERSION
当把十六进制字节数组转换为人类可读的字符串的时候，如果数组是被一个字节一个字节读取的话，可能会导致转换错误。下面这个例子是一个很明显的使用Integer.toHexString() 做转换的例子，它可能会被字节码中的零字节所截断

```
MessageDigest md = MessageDigest.getInstance("SHA-256");
byte[] resultBytes = md.digest(password.getBytes("UTF-8"));

StringBuilder stringBuilder = new StringBuilder();
for(byte b :resultBytes) {
   stringBuilder.append( Integer.toHexString( b & 0xFF ) );
}

return stringBuilder.toString();
```

这个错误削弱了hash的计算值，因为它引入了更多的碰撞。比如，用上面的函数计算"0x0679" 和 "0x6709"都会输出679

在下面的解决方案中，使用String.format()替换toHexString()。

```
stringBuilder.append( String.format( "%02X", b ) );
```

引用：
CWE-704: Incorrect Type Conversion or Cast

Hazelcast对称加密

漏洞规则： HAZELCAST_SYMMETRIC_ENCRYPTION
配置Hazelcast让网络通信使用对称加密（可能是DES或者其他的）
密码本身不能提供完整性和身份验证。使用非对称加密会更好一些

引用：
WASC-04: Insufficient Transport Layer Protection
Hazelcast Documentation: Encryption
CWE-326: Inadequate Encryption Strength

不安全的空密码

漏洞特征： NULL_CIPHER
空密码很少被使用在生产环境中。它通过返回与明文相同的密文来实现Cipher接口。在极少的环境中，比如测试环境，才可能会出现空密码
有漏洞的代码：

```
Cipher doNothingCihper = new NullCipher();
[...]
//The ciphertext produced will be identical to the plaintext.
byte[] cipherText = c.doFinal(plainText);
```

解决方案
避免使用空密码，意外的使用会导致严重的安全风险。
引用：
CWE-327: Use of a Broken or Risky Cryptographic Algorithm

未加密的socket
漏洞特征：UNENCRYPTED_SOCKET
如果网络通信不加密的话，那么传输的数据就会被攻击者拦截并读取里面的内容。
有漏洞的代码：
明文socket（透明传输）

```
Socket soc = new Socket("www.google.com",80);
```

解决方案：
ssl socket（加密传输）

```
Socket soc = SSLSocketFactory.getDefault().createSocket("www.google.com", 443);
```

使用sslsocket，你需要确保你使用的SSLSocketFactory能验证所提供的证书是否有效，这样你就不会遭受中间人攻击。请阅读owasp中关于传输层协议的那一章，以了解更

引用：
OWASP: Top 10 2010-A9-Insufficient Transport Layer Protection
OWASP: Top 10 2013-A6-Sensitive Data Exposure

OWASP: Transport Layer Protection Cheat Sheet
WASC-04: Insufficient Transport Layer Protection
CWE-319: Cleartext Transmission of Sensitive Information

未加密的服务器socket

漏洞特征：UNENCRYPTED_SERVER_SOCKET

如果网络通信不加密的话，那么传输的数据就会被攻击者拦截并读取里面的内容。
有漏洞的代码：
明文socket（透明传输）

```
ServerSocket soc = new ServerSocket(1234);
```

解决方案：
ssl socket（加密传输）

```
ServerSocket soc = SSLServerSocketFactory.getDefault().createServerSocket(1234);
```

使用sslsocket，你需要确保你使用的SSLSocketFactory能验证所提供的证书是否有效，这样你就不会遭受中间人攻击。请阅读owasp中关于传输层协议的那一章，以了解更

引用：
OWASP: Top 10 2010-A9-Insufficient Transport Layer Protection
OWASP: Top 10 2013-A6-Sensitive Data Exposure
OWASP: Transport Layer Protection Cheat Sheet
WASC-04: Insufficient Transport Layer Protection
CWE-319: Cleartext Transmission of Sensitive Information

DES是不安全的

漏洞特征：DES_USAGE

DES被认为是现代加密系统中比较强壮的加密方式，当前，NIST建议使用AES block ciphers来替代DES
有漏洞的代码：

```
Cipher c = Cipher.getInstance("DES/ECB/PKCS5Padding");
c.init(Cipher.ENCRYPT_MODE, k, iv);
byte[] cipherText = c.doFinal(plainText);
```

解决方案示例代码：

```
Cipher c = Cipher.getInstance("AES/GCM/NoPadding");
c.init(Cipher.ENCRYPT_MODE, k, iv);
byte[] cipherText = c.doFinal(plainText);
```

引用：
NIST Withdraws Outdated Data Encryption Standard
CWE-326: Inadequate Encryption Strength

DESede是不安全的

漏洞特征:TDES_USAGE

三次DES（也被称为3DES 或者 DESede）被认为是现代加密系统中比较强壮的加密方式，当前，NIST建议使用AES block ciphers来替代DES
有漏洞的代码：

```
Cipher c = Cipher.getInstance("DESede/ECB/PKCS5Padding");
c.init(Cipher.ENCRYPT_MODE, k, iv);
byte[] cipherText = c.doFinal(plainText);
```

解决方案示例代码：

```
Cipher c = Cipher.getInstance("AES/GCM/NoPadding");
c.init(Cipher.ENCRYPT_MODE, k, iv);
byte[] cipherText = c.doFinal(plainText);
```

引用：
NIST Withdraws Outdated Data Encryption Standard
CWE-326: Inadequate Encryption Strength

不用padding的RSA是不安全的

漏洞特征：RSA_NO_PADDING

软件使用RSA加密算法但是没有使用非对称加密填充(OAEP), 这种加密可能会是比较脆弱的
有漏洞的代码：

```
Cipher.getInstance("RSA/NONE/NoPadding")
```

解决方案：
应该用下面的代码来替换

```
Cipher.getInstance("RSA/ECB/OAEPWithMD5AndMGF1Padding")
```

引用:
CWE-780: Use of RSA Algorithm without OAEP
Root Labs: Why RSA encryption padding is critical

硬编码密码

漏洞特征：HARD_CODE_PASSWORD
密码不应该留在源码里面，在企业里面源码会被广泛的分享，有些部分甚至会被开源出来，为了更安全的管理，密码和密钥应该被单独的存储在配置文件中，或者keystores
Coded Key pattern)
有漏洞的代码

```
private String SECRET_PASSWORD = "letMeIn!";

Properties props = new Properties();
props.put(Context.SECURITY_CREDENTIALS, "p@ssw0rd");
```

引用：
CWE-259: Use of Hard-coded Password

硬编码密钥

漏洞特征： HARD_CODE_KEY
加密密钥不应该留在源码里面，在企业里面源码会被广泛的分享，有些部分甚至会被开源出来，为了更安全的管理，密码和密钥应该被单独的存储在配置文件中，或者keyst
Coded Password pattern)
有漏洞的代码

```
byte[] key = {1, 2, 3, 4, 5, 6, 7, 8};
SecretKeySpec spec = new SecretKeySpec(key, "AES");
Cipher aes = Cipher.getInstance("AES");
aes.init(Cipher.ENCRYPT_MODE, spec);
return aesCipher.doFinal(secretData);
```

引用：
CWE-321: Use of Hard-coded Cryptographic Key

不安全的hash比较

漏洞特征：UNSAFE_HASH_EQUALS
攻击者可能会通过密钥的比较时间来发现密钥的hash值，当Arrays.equals() 或者
String.equals()被调用的时候，如果有一些字节被匹配到的话，它们会推出的更早一些
有漏洞的代码：

```
String actualHash = ...

if(userInput.equals(actualHash)) {
   ...
}
```

解决方案：

```
String actualHash = ...

if(MessageDigest.isEqual(userInput.getBytes(),actualHash.getBytes())) {
   ...
}
```

引用：
CWE-203: Information Exposure Through DiscrepancyKey

来自Struts Form的输入没有被验证

漏洞特征： STRUTS_FORM_VALIDATION
来自Form的输入应该被简单的验证一下，预防性的验证能够抵御更进一步的攻击。
validate这个函数引入了验证的实现

```
public class RegistrationForm extends ValidatorForm {

   private String name;
   private String email;

   [...]

   public ActionErrors validate(ActionMapping mapping, HttpServletRequest request) {
       //Validation code for name and email parameters passed in via the HttpRequest goes here
   }
}
```

引用：
CWE-20: Improper Input Validation
CWE-106: Struts: Plug-in Framework not in Use

XSSRequestWrapper的xss防护是脆弱的

漏洞特征：XSS_REQUEST_WRAPPER
在各种公开的博客里面，博主通过实现HttpServletRequestWrapper调用XSSRequestWrapper
这个过滤函数的脆弱点在于以下的几个方面：

- 它仅仅覆盖参数，而没有覆盖到http头或者侧信道输入 。
- 简单替换的方式很容易会被绕过（见下面的例子）
- 黑名单的方式太脆弱（不如用白名单的方式来验证好的输入）

绕过示例：

```
<scrivbscript:pt>alert(1)</scrivbscript:pt>
```

上面的输入会被转换为：`<script>alert(1)</script>` 。移除了vbscript:"之后就变成了"`<script>.*</script>`"

为了更强的保护，请在view (template, jsp, ...) 中选择自动编码字符串的解决方案，解决方案里面的规则被定义在OWASP XSS Prevention 备忘录中。
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')

Blowfish 使用过短的密钥

漏洞特征：BLOWFISH_KEY_SIZE
Blowfish的密钥支持32 bits 到 448 bits的长度。如果密钥太短，会导致加密内容被黑客暴力破解。如果使用Blowfish的话，密钥至少应该选择128 bits 。
如果算法被改变，那么应该AES分组密码

有漏洞的代码：

```
KeyGenerator keyGen = KeyGenerator.getInstance("Blowfish");
keyGen.init(64);
```

解决方案：

```
KeyGenerator keyGen = KeyGenerator.getInstance("Blowfish");
keyGen.init(128);
```

引用：
Blowfish (cipher))
CWE-326: Inadequate Encryption Strength

RSA使用了过短密钥

漏洞特征: RSA_KEY_SIZE
NIST建议RSA算法应该使用2048bits的密钥或者更长的密钥
“电子签名验证 | RSA: 1024 ≤ len(n) < 2048 | 传统使用”
“电子签名验证 | RSA: len(n) ≥ 2048 | 可接受”

- NIST: 数据传输建议使用的加密方式和密钥长度 p.7

漏洞代码：

```
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
keyGen.initialize(512);
```

解决方案：
密钥的生成至少应该像下面这样使用2048位

```
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
keyGen.initialize(2048);
```

引用：
NIST: Latest publication on key management
NIST: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths p.7
RSA Laboratories: 3.1.5 How large a key should be used in the RSA cryptosystem?
Wikipedia: Asymmetric algorithm key lengths
CWE-326: Inadequate Encryption Strength
Keylength.com (BlueKrypt): Aggregate key length recommendations.

未验证的重定向

漏洞特征：UNVALIDATED_REDIRECT
未验证重定向漏洞是因为应用跳转到用户输入的指定目标url，这个输入的参数没有被充分的验证。这个漏洞可能会被用来钓鱼

假设的场景：

1. 用户被欺骗点了恶意链接：http://website.com/login?redirect=http://evil.vvebsite.com/fake/login
2. 用户被重定向到了一个虚假的登录页面，这样页面看起来就像真的一样(http://evil.vvebsite.com/fake/login)
3. 用户输入了他的凭据
4. 恶意网站偷走了用户的凭据，并且跳转回了原来的网站

这个攻击貌似是合理的，因为大多数用户在被重定向之后不会再次检查url。而且跳转到授权页面也是很普遍的现象。
漏洞代码：

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
   [...]
   resp.sendRedirect(req.getParameter("redirectUrl"));
   [...]
}
```

解决方案/对策

- 不要从用户的输入中接受重定向的目的url
- 接受一个目的地址的key，这个key可以查询到一个合法的目的地址。
- 仅接受相对路径
- urls白名单（如果可行的话）
- 验证url开始的部分是否在白名单里面

引用：
WASC-38: URL Redirector Abuse
OWASP: Top 10 2013-A10: Unvalidated Redirects and Forwards
OWASP: Unvalidated Redirects and Forwards Cheat Sheet
CWE-601: URL Redirection to Untrusted Site ('Open Redirect')

未验证的重定向(Play Framework)

漏洞特征：PLAY_UNVALIDATED_REDIRECT
未验证重定向漏洞是因为应用跳转到用户输入的指定目标url，这个输入的参数没有被充分的验证。这个漏洞可能会被用来钓鱼

假设的场景：

1. 用户被欺骗点了恶意链接：http://website.com/login?redirect=http://evil.vvebsite.com/fake/login
2. 用户被重定向到了一个虚假的登录页面，这样页面看起来就像真的一样(http://evil.vvebsite.com/fake/login)
3. 用户输入了他的凭据
4. 恶意网站偷走了用户的凭据，并且跳转回了原来的网站

这个攻击貌似是合理的，因为大多数用户在被重定向之后不会再次检查url。而且跳转到授权页面也是很普遍的现象。
漏洞代码：

```
def login(redirectUrl:String) = Action {
   [...]
   Redirect(url)
}
```

解决方案/对策

- 不要从用户的输入中接受重定向的目的url
- 接受一个目的地址的key，这个key可以查询到一个合法的目的地址。
- 仅接受相对路径
- urls白名单（如果可行的话）
- 验证url开始的部分是否在白名单里面

引用：
WASC-38: URL Redirector Abuse
OWASP: Top 10 2013-A10: Unvalidated Redirects and Forwards
OWASP: Unvalidated Redirects and Forwards Cheat Sheet
CWE-601: URL Redirection to Untrusted Site ('Open Redirect')

Spring中未验证的重定向

漏洞特征：SPRING_UNVALIDATED_REDIRECT
未验证重定向漏洞是因为应用跳转到用户输入的指定目标url，这个输入的参数没有被充分的验证。这个漏洞可能会被用来钓鱼

假设的场景：

1. 用户被欺骗点了恶意链接：http://website.com/login?redirect=http://evil.vvebsite.com/fake/login
2. 用户被重定向到了一个虚假的登录页面，这样页面看起来就像真的一样(http://evil.vvebsite.com/fake/login)
3. 用户输入了他的凭据
4. 恶意网站偷走了用户的凭据，并且跳转回了原来的网站

这个攻击貌似是合理的，因为大多数用户在被重定向之后不会再次检查url。而且跳转到授权页面也是很普遍的现象。
漏洞代码：

```
@RequestMapping("/redirect")
public String redirect(@RequestParam("url") String url) {
   [...]
   return "redirect:" + url;
}
```

解决方案/对策

- 不要从用户的输入中接受重定向的目的url
- 接受一个目的地址的key，这个key可以查询到一个合法的目的地址。
- 仅接受相对路径
- urls白名单（如果可行的话）
- 验证url开始的部分是否在白名单里面

引用：
WASC-38: URL Redirector Abuse
OWASP: Top 10 2013-A10: Unvalidated Redirects and Forwards
OWASP: Unvalidated Redirects and Forwards Cheat Sheet
CWE-601: URL Redirection to Untrusted Site ('Open Redirect')

jsp动态包含

漏洞特征：JSP_INCLUDE
jsp允许动态包含文件。这可能允许攻击者控制jsp的文件包含。如果出现这样的漏洞的话，攻击者就会包含一个他能控制到的文件。通过直接包含文件，攻击者就能执行任意
有漏洞的代码：

```
<jsp:include page="${param.secret_param}" />
```

解决方案：

```
<c:if test="${param.secret_param == 'page1'}">
   <jsp:include page="page1.jsp" />
</c:if>
```

引用：

InfosecInstitute: File Inclusion Attacks
WASC-05: Remote File Inclusion

Spring 表达式中的动态变量

漏洞特征：JSP_SPRING_EVAL
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。
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'}"/>
```

引用：
CWE-94: Improper Control of Generation of Code ('Code Injection')
CWE-95: Improper Neutralization of Directives in Dynamically Evaluated Code ('Eval Injection')

xml字符转义被禁用

漏洞特征：JSP_JSTL_OUT
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：

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<c:out value="${param.test_param}" escapeXml="false"/>
```

解决方案：

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<c:out value="${param.test_param}"/>
```

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')
JSTL Javadoc: Out tag

jsp中潜在的xss

漏洞特征：XSS_JSP_PRINT
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：

```
<%
String taintedInput = (String) request.getAttribute("input");
%>
[...]
<%= taintedInput %>
```

解决方案：

```
<%
String taintedInput = (String) request.getAttribute("input");
%>
[...]
<%= Encode.forHtml(taintedInput) %>
```

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，里面会介绍一些防御的细节。
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

Servlet中潜在的xss

漏洞特征：XSS_SERVLET
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：

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
   String input1 = req.getParameter("input1");
   [...]
   resp.getWriter().write(input1);
}
```

解决方案：

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
   String input1 = req.getParameter("input1");
   [...]
   resp.getWriter().write(Encode.forHtml(input1));
}
```

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，里面会介绍一些防御的细节。
注意Servlet中的xss规则看着都很类似，但是要用不同的规则寻找'XSS：Servlet反射型xss'和'xss:在Servlet错误页面中反射型xss'

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

XMLDecoder的使用

漏洞规则：XML_DECODER
不应该用XMLDecoder解析不受信任的数据。反序列化用户输入数据会导致代码执行。这是因为XMLDecoder
支持任意的方法调用。这个功能旨在调用setter方法，但是实际上，这个功能什么方法都能调用。
恶意的xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<java version="1.4.0" class="java.beans.XMLDecoder">
 <object class="java.io.PrintWriter">
   <string>/tmp/Hacked.txt</string>
   <void method="println">
     <string>Hello World!</string>
   </void>
   <void method="close"/>
```

```
 </object>
</java>
```

上面这个xml代码可能会在服务器中创建一个内容为"Hello World!".的文件。

有漏洞的代码：

```
XMLDecoder d = new XMLDecoder(in);
try {
   Object result = d.readObject();
}
[...]
```

解决方案：
解决方案是避免使用XMLDecoder去解析不受信任的用户输入数据

引用：
Dinis Cruz Blog: Using XMLDecoder to execute server-side Java Code on an Restlet application
RedHat blog : Java deserialization flaws: Part 2, XML deserialization
CWE-20: Improper Input Validation

固定IV

漏洞规则：STATIC_IV
每一条消息都应该为它初始化生成一个新的加密向量
有漏洞的代码：

```
private static byte[] IV = new byte[16] {(byte)0,(byte)1,(byte)2,[...]};

public void encrypt(String message) throws Exception {

   IvParameterSpec ivSpec = new IvParameterSpec(IV);
[...]
```

解决方案:

```
public void encrypt(String message) throws Exception {

   byte[] iv = new byte[16];
   new SecureRandom().nextBytes(iv);

   IvParameterSpec ivSpec = new IvParameterSpec(iv);
[...]
```

引用：
Wikipedia: Initialization vector
CWE-329: Not Using a Random IV with CBC Mode
Encryption - CBC Mode IV: Secret or Not?

ECB模式是不安全的

漏洞规则： ECB_MODE

提供了最好机密性的授权加密模式应该替换电码本模式(Electronic Codebook Book
(ECB))，因为ecb没有提供很好的机密性。尤其，在ecb模式下，输入相同的数据，每一次的输出也是相同的。所以，如果用户发送一个密码，它的加密值每次都是相同的。
为了修复这个。一些像Galois/Counter Mode (GCM)也应该被替换
有漏洞的代码

```
Cipher c = Cipher.getInstance("AES/ECB/NoPadding");
c.init(Cipher.ENCRYPT_MODE, k, iv);
byte[] cipherText = c.doFinal(plainText);
```

解决方案：

```
Cipher c = Cipher.getInstance("AES/GCM/NoPadding");
c.init(Cipher.ENCRYPT_MODE, k, iv);
byte[] cipherText = c.doFinal(plainText);
```

引用：
Wikipedia: Authenticated encryption

NIST: Authenticated Encryption Modes
Wikipedia: Block cipher modes of operation
NIST: Recommendation for Block Cipher Modes of Operation

加密容易受到Padding Oracle的影响

漏洞特征：PADDING_ORACLE
具有PKCS5Padding的CBC特定模式容易受到padding
oracle攻击。如果系统暴露了的明文数据与有效padding或无效padding之间的差异。那么攻击者就可能会解密数据。有效padding和无效padding的差别通常可以通过每一
有漏洞的代码：

```
Cipher c = Cipher.getInstance("AES/CBC/PKCS5Padding");
c.init(Cipher.ENCRYPT_MODE, k, iv);
byte[] cipherText = c.doFinal(plainText);
```

解决方案：

```
Cipher c = Cipher.getInstance("AES/GCM/NoPadding");
c.init(Cipher.ENCRYPT_MODE, k, iv);
byte[] cipherText = c.doFinal(plainText);
```

引用：
Padding Oracles for the masses (by Matias Soler)
Wikipedia: Authenticated encryption
NIST: Authenticated Encryption Modes
CAPEC: Padding Oracle Crypto Attack
CWE-696: Incorrect Behavior Order

密码没有完整性

漏洞特征：CIPHER_INTEGRITY
产生的密文容易被对手改变。这就意味着，加密提供者没法发现数据是否遭到篡改。如果加密数据被攻击者控制，那么它可能会被偷偷改掉。
解决方案通常是加密数据通常包含基本的身份验证hash(HMAC)
去签名数据。把HMAC方法和现有的加密方式结合容易出错。尤其，推荐你要首先去验证HMAC，并且如果数据没有被篡改，你才能执行所有的解密操作。
如果没有提供HMAC，下面的模式都是有漏洞的：

- CBC
- OFB
- CTR
  ECB
  下面的片段是一些有漏洞的代码:
  有漏洞的代码
  aes的cbc模式

  ```
  Cipher c = Cipher.getInstance("AES/CBC/PKCS5Padding");
  c.init(Cipher.ENCRYPT_MODE, k, iv);
  byte[] cipherText = c.doFinal(plainText);
  ```

  三次DES的ECB模式

  ```
  Cipher c = Cipher.getInstance("DESede/ECB/PKCS5Padding");
  c.init(Cipher.ENCRYPT_MODE, k, iv);
  byte[] cipherText = c.doFinal(plainText);
  ```

  解决方案：

  ```
  Cipher c = Cipher.getInstance("AES/GCM/NoPadding");
  c.init(Cipher.ENCRYPT_MODE, k, iv);
  byte[] cipherText = c.doFinal(plainText);
  ```

- 在上面这个例子中，GCM模式把HMAC引入到加密数据的结果之中，提供了结果的完整性
  引用：
  Wikipedia: Authenticated encryption
  NIST: Authenticated Encryption Modes
  Moxie Marlinspike's blog: The Cryptographic Doom Principle
  CWE-353: Missing Support for Integrity Check

使用ESAPI加密

漏洞规则：ESAPI_ENCRYPTOR
ESAPI的加密组件在历史上有一些小的漏洞。这里有一个能够快速验证的列表，以保证授权的加密是以期望的方式运行的。

1. 库的版本

这个问题在2.1.0这个版本被修正。在2.0.1版本以下有漏洞可以绕过MAC（CVE-2013-5679）
对于Maven使用者，使用下面的命令可以查看插件的版本。有效的ESAPI将会被输出

```
$ mvn versions:display-dependency-updates
```

输出：

```
[...]
[INFO] The following dependencies in Dependencies have newer versions:
[INFO]   org.slf4j:slf4j-api ................................. 1.6.4 -> 1.7.7
[INFO]   org.owasp.esapi:esapi ............................... 2.0.1 -> 2.1.0
[...]
```

或者可以直接查看配置

```
<dependency>
   <groupId>org.owasp.esapi</groupId>
   <artifactId>esapi</artifactId>
   <version>2.1.0</version>
</dependency>
```

对于Ant使用者，应该使用 esapi-2.1.0.jar 这个jar。

2.配置
在2.1.0这个版本中，在密文定义中，密钥的改变会导致漏洞(CVE-2013-5960)。需要使用一些预防措施。
如果存在以下任何元素，那么ESAPI的加密算法就是有问题的
不安全的配置：

```
Encryptor.CipherText.useMAC=false

Encryptor.EncryptionAlgorithm=AES
Encryptor.CipherTransformation=AES/CBC/PKCS5Padding

Encryptor.cipher_modes.additional_allowed=CBC
```

安全的配置：

```
#Needed
Encryptor.CipherText.useMAC=true

#Needed to have a solid auth. encryption
Encryptor.EncryptionAlgorithm=AES
Encryptor.CipherTransformation=AES/GCM/NoPadding

#CBC mode should be removed to avoid padding oracle
Encryptor.cipher_modes.additional_allowed=
```

引用：
ESAPI Security bulletin 1 (CVE-2013-5679)
Vulnerability Summary for CVE-2013-5679
Synactiv: Bypassing HMAC validation in OWASP ESAPI symmetric encryption
CWE-310: Cryptographic Issues
ESAPI-dev mailing list: Status of CVE-2013-5960

点击收藏 | 2 关注 | 1
上一篇：tcpdump 4.5.1 漏洞分... 下一篇：在Debug中学Tcache
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