java代码审计手书(一）

niexinming / 2018-11-21 07:35:00 / 浏览数 6959

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

可预测的伪随机数发生器

漏洞特征：PREDICTABLE_RANDOM
在某些关键的安全环境中使用可预测的随机数可能会导致漏洞，比如，当这个值被作为：

- csrf token；如果攻击者可以预测csrf的token值的话，就可以发动csrf攻击
- 重置密码的token（通过邮件发送）；如果重置密码的token被替换的话，那么就会导致用户账户被接管，因为攻击者会猜测到重置密码的链接。
- 其他包含秘密的信息

修复这个漏洞最快的方式是用强随机数生成器（ 比如：`java.security.SecureRandom` ）替换掉
`java.util.Random`

有漏洞的代码：

```
String generateSecretToken() {
   Random r = new Random();
   return Long.toHexString(r.nextLong());
}
```

解决方案

```
import org.apache.commons.codec.binary.Hex;

String generateSecretToken() {
   SecureRandom secRandom = new SecureRandom();

   byte[] result = new byte[32];
   secRandom.nextBytes(result);
   return Hex.encodeHexString(result);
}
```

引用：

Cracking Random Number Generators - Part 1 (http://jazzy.id.au)
CERT: MSC02-J. Generate strong random numbers
CWE-330: Use of Insufficiently Random Values
Predicting Struts CSRF Token (Example of real-life vulnerability and exploitation)

可预测的伪随机数发生器（Scala）

漏洞特征：PREDICTABLE_RANDOM_SCALA
在某些关键的安全环境中使用可预测的随机数可能会导致漏洞，比如，当这个值被作为：

- csrf token；如果攻击者可以预测csrf的token值的话，就可以发动csrf攻击
- 重置密码的token（通过邮件发送）；如果重置密码的token被替换的话，那么就会导致用户账户被接管，因为攻击者会猜测到重置密码的链接。
- 其他包含秘密的信息

修复这个漏洞最快的方式是用强随机数生成器（ 比如：`java.security.SecureRandom` ）替换掉
`java.util.Random`
有漏洞的代码：

```
import scala.util.Random

def generateSecretToken() {
   val result = Seq.fill(16)(Random.nextInt)
   return result.map("%02x" format _).mkString
}
```

解决方案：

```
import java.security.SecureRandom

def generateSecretToken() {
   val rand = new SecureRandom()
   val value = Array.ofDim[Byte](16)
   rand.nextBytes(value)
   return value.map("%02x" format _).mkString
}
```

Cracking Random Number Generators - Part 1 (http://jazzy.id.au)
CERT: MSC02-J. Generate strong random numbers
CWE-330: Use of Insufficiently Random Values
Predicting Struts CSRF Token (Example of real-life vulnerability and exploitation)

没有做任何安全检查的servlet 参数

漏洞特征：SERVLET_PARAMETER
Servlet 会从各种函数中获取到GET和POST的值。这些被获取的值肯定是不安全的。在进入到敏感的api函数之前你可能需要验证和过滤这些值：

- sql 查询（可能导致sql注入）
- 文件操作（ 可能会导致目录穿越 ）
- 命令执行（ 可能会导致命令注入 ）
- html解析（可能会导致xss)
- 其他的

引用：
CWE-20: Improper Input Validation

没有做任何安全检查Content-Type 头

漏洞特征：SERVLET_CONTENT_TYPE
服务器端程序通过客户端收集http的Content-Type的值。这个值可能会影响影响应用的安全性

引用：
CWE-807: Untrusted Inputs in a Security Decision

没有做任何安全检查Hostname 头

漏洞特征： SERVLET_SERVER_NAME
服务器端程序通过客户端收集http的hostname 的值。这个值可能会影响影响应用的安全性。`ServletRequest.getServerName()`和`HttpServletRequest.getHeader("Host")`的行为很相似，都是从http头部中获取到host的值

```
GET /testpage HTTP/1.1
Host: www.example.com
[...]
```

默认情况下，web容器可能会直接将请求重定向到你的应用程序中。这就允许用户把恶意的请求放入http的host头中。我建议你不要信任来自客户端的任何输入。

引用：
CWE-807: Untrusted Inputs in a Security Decision

没有做任何安全检查的session cookie值

漏洞特征： SERVLET_SESSION_ID
`HttpServletRequest.getRequestedSessionId()`(
http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletRequest.html#getRequestedSessionId()
)函数返回cookie中JSESSIONID的值。这个值通常被session 管理器访问，而不是开发者代码。
传递给客户端的值通常是字母数字（ 例如：JSESSIONID=jp6q31lq2myn ），无论如何，这个值可以被客户端改变，下面的http请求展示了潜在的危险

```
GET /somePage HTTP/1.1
Host: yourwebsite.com
User-Agent: Mozilla/5.0
Cookie: JSESSIONID=Any value of the user's choice!!??'''">
```

像这样，JSESSIONID应该仅被使用判断是否与存在的session ID相匹配，如果不存在对应的session ID，那么这个用户就可能会是未授权用户。此外， session ID的值应该从来不被记录，如果记录了，那么日志文件中就会包含有效的且在激活状态的session IDs，这样就会允许内部员工可以通过日志记录来劫持任意在线用户。

引用：

OWASP: Session Management Cheat Sheet
CWE-20: Improper Input Validation

没有做任何安全检查的查询字符串

漏洞特征： SERVLET_QUERY_STRING
查询字符串是get请求中参数名和参数值的串联，可以传入预期之外的参数。 比如URL请求：/app/servlet.htm?a=1&b=2 ，查询字符串就是a=1&b=2
通过函数 HttpServletRequest.getParameter() 接收每一个传递进来的参数的值，通过 HttpServletRequest.getQueryString()
这个函数获取到的值应该被看做不安全的。你应该在查询字符串进入敏感函数之前去充分的效验和过滤它们。

引用：
CWE-20: Improper Input Validation

没有做任何安全检查的HTTP头

漏洞特征：SERVLET_HEADER
http请求头很容易会被用户所修改。通常，不要假想请求来自于没有被黑客修改的常规浏览器。我建议你，不要相信客户端传递进来的http头部值

引用：
CWE-807: Untrusted Inputs in a Security Decision

没有做任何安全检查的Referer值

漏洞特征：SERVLET_HEADER_REFERER
行为：

- 如果请求来自于恶意用户，那么Referer的值会是任意的情况。
- 如果请求来自于另一个安全的源（https），那么Referer头就是空的。

建议：

- 访问控制不应该基于此标头的值。
- csrf保护不应该仅基于此值。（ 因为这个选项 ）

引用：
CWE-807: Untrusted Inputs in a Security Decision

没有做任何安全检查的User-Agent值

漏洞特征： SERVLET_HEADER_USER_AGENT
"User-Agent" 很容易被客户端伪造，不建议基于不同的User-Agent（比如爬虫的UA）来适配不同的行为。

引用：
CWE-807: Untrusted Inputs in a Security Decision

潜在的cookie中包含敏感数据

漏洞特征： COOKIE_USAGE
存储在客户端中cookie的数据不应该包含敏感数据或者与session相关的数据。大多数情况下，敏感数据应该仅仅存储在session中，并且通过通过用户的session值去访问。(HttpServletRequest.getSession())
客户端cookie应该是比特定会话维持时间更长且独立于特殊会话

引用：
CWE-315: Cleartext Storage of Sensitive Information in a Cookie

潜在的路径穿越（文件读取）

漏洞特征：PATH_TRAVERSAL_IN
一个文件被打开，然后读取文件内容，这个文件名来自于一个输入的参数。如果没有过滤这个传入的参数，那么本地文件系统中任意文件都会被读取。
这个规则识别潜在的路径穿越漏洞。在许多场景中，用户无法控制文件路径，如果有工具报告了这个问题，那么这个就是误报
有漏洞代码：

```
@GET
@Path("/images/{image}")
@Produces("images/*")
```

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

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

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

解决方案：

```
import org.apache.commons.io.FilenameUtils;

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

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

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

引用：
WASC: Path Traversal
OWASP: Path Traversal
CAPEC-126: Path Traversal
CWE-22: Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')

潜在的路径穿越（文件写）

漏洞特征：PATH_TRAVERSAL_OUT
一个文件被打开，然后读取文件内容，这个文件名来自于一个输入的参数。如果没有过滤这个传入的参数，那么本地文件系统中任意文件都会被修改。
这个规则识别潜在的路径穿越漏洞。在许多场景中，用户无法控制文件路径，如果有工具报告了这个问题，那么这个就是误报
引用：
WASC: Path Traversal
OWASP: Path Traversal
CAPEC-126: Path Traversal
CWE-22: Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')

潜在的路径穿越（文件读取）

漏洞特征：SCALA_PATH_TRAVERSAL_IN
一个文件被打开，然后读取文件内容，这个文件名来自于一个输入的参数。如果没有过滤这个传入的参数，那么本地文件系统中任意文件都会被读取。
这个规则识别潜在的路径穿越漏洞。在许多场景中，用户无法控制文件路径，如果有工具报告了这个问题，那么这个就是误报
有漏洞代码：

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

解决方案：

```
import org.apache.commons.io.FilenameUtils;

def getWordList(value:String) = Action {
 val filename = "public/lists/" + FilenameUtils.getName(value)

 if (!Files.exists(Paths.get(filename))) {
   NotFound("File not found")
```

```
 } else {
   val result = Source.fromFile(filename).getLines().mkString // Fix
   Ok(result)
 }
}
```

引用：
WASC: Path Traversal
OWASP: Path Traversal
CAPEC-126: Path Traversal
CWE-22: Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')

潜在的命令注入

漏洞特征：COMMAND_INJECTION
高亮部分的api被用来执行系统命令，如果输入这个api的数据没有被过滤，那么就会导致任意命令执行
有漏洞的代码：

```
import java.lang.Runtime;

Runtime r = Runtime.getRuntime();
r.exec("/bin/sh -c some_tool" + input);
```

引用：
OWASP: Command Injection
OWASP: Top 10 2013-A1-Injection
CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')

潜在的命令注入(Scala)

漏洞特征：COMMAND_INJECTION
高亮部分的api被用来执行系统命令，如果输入这个api的数据没有被过滤，那么就会导致任意命令执行
有漏洞的代码：

```
def executeCommand(value:String) = Action {
   val result = value.!
   Ok("Result:\n"+result)
}
```

引用：
OWASP: Command Injection
OWASP: Top 10 2013-A1-Injection
CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')

文件类函数没有过滤空字符

漏洞特征：WEAK_FILENAMEUTILS
一些文件类中方法没有过滤空字节（0x00）
如果空字节被注入到文件名之中，如果这个文件被放进系统之中，那么系统则只会读取空字符之前的文件名，字符串就会被空字符截断，甚至java本身也不能关注空字符或者

给出两点建议去修复这个问题：

- 升级到7 update 40 或者最近的版本，或者java 8 +，因为空字节注入这个问题已经被这些版本的java所解决
- 要严格验证用户输入的文件名是否是有效的（例如不能包含空字符，不能包含路径字符）

如果你知道你使用的现有的java版本可以避免空字符注入问题，你可以忽略上面的问题。

引用：
WASC-28: Null Byte Injection
CWE-158: Improper Neutralization of Null Byte or NUL Character

证书管理器接受任何证书

漏洞特征： WEAK_TRUST_MANAGER
空的证书管理器通常可以更轻松的连接到没有根证书的主机上。结果就是，就会更容易受到中间人攻击，因为客户端信任所有的证书。
一个证书管理器应该允许信任指定的一种证书（例如：基于信任库）。下面是一种可行的实现方法：
有漏洞的代码：

```
class TrustAllManager implements X509TrustManager {

   @Override
   public void checkClientTrusted(X509Certificate[] x509Certificates, String s) throws CertificateException {
       //Trust any client connecting (no certificate validation)
   }

   @Override
   public void checkServerTrusted(X509Certificate[] x509Certificates, String s) throws CertificateException {
       //Trust any remote server (no certificate validation)
   }

   @Override
   public X509Certificate[] getAcceptedIssuers() {
       return null;
   }
}
```

解决方案(基于证书库的证书管理器）：

```
KeyStore ks = //Load keystore containing the certificates trusted

SSLContext sc = SSLContext.getInstance("TLS");

TrustManagerFactory tmf = TrustManagerFactory.getInstance("SunX509");
tmf.init(ks);

sc.init(kmf.getKeyManagers(), tmf.getTrustManagers(),null);
```

引用：
WASC-04: Insufficient Transport Layer Protection
CWE-295: Improper Certificate Validation

HostnameVerifier 接收任何签名证书

漏洞规则：WEAK_HOSTNAME_VERIFIER
因为证书会被很多主机重复使用，接收任意证书的HostnameVerifier经常被使用。结果就是，就会更容易受到中间人攻击，因为客户端信任所有的证书。
一个证书管理器应该允许信任指定的一种证书（例如：基于信任库）。应该创建通配符证书，可以允许多个子域下证书。下面是一种可行的实现方法：
有漏洞的代码：

```
public class AllHosts implements HostnameVerifier {
   public boolean verify(final String hostname, final SSLSession session) {
       return true;
   }
}
```

解决方案(基于证书库的证书管理器）：

```
KeyStore ks = //Load keystore containing the certificates trusted

SSLContext sc = SSLContext.getInstance("TLS");

TrustManagerFactory tmf = TrustManagerFactory.getInstance("SunX509");
tmf.init(ks);

sc.init(kmf.getKeyManagers(), tmf.getTrustManagers(),null);
```

引用：
WASC-04: Insufficient Transport Layer Protection
CWE-295: Improper Certificate Validation

发现JAX-RS REST服务器端

漏洞规则： JAXRS_ENDPOINT
这些函数是REST Web Service 的一部分(JSR311).
这个网站的安全性应该被分析。例如：

- 权限认证，如果强制实施，就应该被测试
- 访问控制，如果强制实施，就应该被测试

- 输入应该被追踪，因为可能会有潜在的漏洞
- 聊天程序应该使用SSL
- 如果服务器支持存储私人数据（例如，通过POST），应该调查它是否对csrf有防御

引用：
OWASP: REST Assessment Cheat Sheet
OWASP: REST Security Cheat Sheet
OWASP: Web Service Security Cheat Sheet
OWASP: Cross-Site Request Forgery)
OWASP: CSRF Prevention Cheat Sheet
CWE-20: Improper Input Validation

发现Tapestry页面

漏洞规则： TAPESTRY_ENDPOINT
在应用启动的时候，Tapestry会被发现。Tapestry应用的每一个页面又后端java类和相关的Tapestry标记语言构成（a.tml
文件）。当请求到达的时候，GET/POST参数会被映射到后端的java类之中。映射可以使用fieldName完成：

```
[...]
   protected String input;
[...]
```

或者显示注释的定义：

```
[...]
   @org.apache.tapestry5.annotations.Parameter
   protected String parameter1;

   @org.apache.tapestry5.annotations.Component(id = "password")
   private PasswordField passwordField;
[...]
```

这个页面被映射到视图中[/resources/package/PageName].tml.

在应用中的每一个Tapestry页面应该被调查，确保所有的输入都能被自动的映射，并在这些参数被使用之前都是有效的。
引用：
Apache Tapestry Home Page
CWE-20: Improper Input Validation

发现Wicket的web页面

漏洞特征：WICKET_ENDPOINT
这个类代表一个Wicket
web页面。输入的数据会被来自实例中的PageParameters读取，然后把它们送入后端处理程序。当前页面会被映射到视图之中[/package/WebPageName].html.
在应用中的每一个Wicket页面应该被调查，确保所有的输入都能被自动的映射，并在这些参数被使用之前都是有效的。
引用：
Apache Wicket Home Page
CWE-20: Improper Input Validation

MD2, MD4 和 MD5都是脆弱的哈希函数

漏洞特征：WEAK_MESSAGE_DIGEST_MD5
不建议使用MD2, MD4 和 MD5这个摘要算法。应该使用PBKDF2作为密码的摘要算法。
md5哈希算法的安全性被严重损害。现已存在一种碰撞攻击，这种攻击可以用奔腾2.6 GHz
4核处理器在几秒内碰撞出另一个哈希相同的字符串。进一步来说，还有选择前缀碰撞攻击（chosen-prefix collision attack
），这种攻击能在一个小时之内找到两个前缀相同的哈希，只要现有计算机的计算水平就可以达到。

"SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, and SHA-512/256:
所有散列计算程序都支持这些哈希函数的使用。
NISI:通信传输：[传输中建议使用的加密算法和密钥长度](http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-131Ar1.pdf)

PBKDF的主要思想是减缓字典生成的时间或者增加攻击者攻击每一个密码的时间。攻击者会有一个密码表去爆破PBKDF所使用的迭代计数器和salt。因为攻击者必须花费大量
NISI:基于密码的密钥的加密建议

有漏洞的代码：

```
MessageDigest md5Digest = MessageDigest.getInstance("MD5");
   md5Digest.update(password.getBytes());
   byte[] hashValue = md5Digest.digest();
```

解决方案：

```
public static byte[] getEncryptedPassword(String password, byte[] salt) throws NoSuchAlgorithmException, InvalidKeySpecExcepti
   PKCS5S2ParametersGenerator gen = new PKCS5S2ParametersGenerator(new SHA256Digest());
   gen.init(password.getBytes("UTF-8"), salt.getBytes(), 4096);
   return ((KeyParameter) gen.generateDerivedParameters(256)).getKey();
}
```

解决方案（java 8 和之后的版本）

```
public static byte[] getEncryptedPassword(String password, byte[] salt) throws NoSuchAlgorithmException, InvalidKeySpecExcepti
   KeySpec spec = new PBEKeySpec(password.toCharArray(), salt, 4096, 256 * 8);
   SecretKeyFactory f = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA256");
   return f.generateSecret(spec).getEncoded();
}
```

引用：
[1] On Collisions for MD5: Master Thesis by M.M.J. Stevens
[2] Chosen-prefix collisions for MD5 and applications: Paper written by Marc Stevens
Wikipedia: MD5
NIST: Transitions: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths
NIST: Recommendation for Password-Based Key Derivation
Stackoverflow: Reliable implementation of PBKDF2-HMAC-SHA256 for Java
CWE-327: Use of a Broken or Risky Cryptographic Algorithm

SHA-1 是脆弱的哈希算法

漏洞特征： WEAK_MESSAGE_DIGEST_SHA1
不建议使用SHA-1算法去加密密码、做数字签名和其他用途。应该使用PBKDF2作为密码的摘要算法。

“SHA-1用于生成电子签名：
SHA-1可能仅仅用于NIST指导的特殊协议的电子签名的生成。但是在其他的应用中，SHA-1 不应该用于电子签名

SHA-1用于电子签名的验证：
对于电子签名的验证，SHA-1可以被用于传统应用
"SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, and SHA-512/256:
所有散列计算程序都支持这些哈希函数的使用。
NISI:通信传输：[传输中建议使用的加密算法和密钥长度](http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-131Ar1.pdf）

PBKDF的主要思想是减缓字典生成的时间或者增加攻击者攻击每一个密码的时间。攻击者会有一个密码表去爆破PBKDF所使用的迭代计数器和salt。因为攻击者必须花费大量
NISI:基于密码的密钥的加密建议

有漏洞的代码：

```
MessageDigest sha1Digest = MessageDigest.getInstance("SHA1");
sha1Digest.update(password.getBytes());
byte[] hashValue = sha1Digest.digest();

byte[] hashValue = DigestUtils.getSha1Digest().digest(password.getBytes());
```

解决方案：

```
public static byte[] getEncryptedPassword(String password, byte[] salt) throws NoSuchAlgorithmException, InvalidKeySpecExcepti
   PKCS5S2ParametersGenerator gen = new PKCS5S2ParametersGenerator(new SHA256Digest());
   gen.init(password.getBytes("UTF-8"), salt.getBytes(), 4096);
   return ((KeyParameter) gen.generateDerivedParameters(256)).getKey();
}
```

解决方案（java 8 及以后的版本）

```
public static byte[] getEncryptedPassword(String password, byte[] salt) throws NoSuchAlgorithmException, InvalidKeySpecExcepti
   KeySpec spec = new PBEKeySpec(password.toCharArray(), salt, 4096, 256 * 8);
   SecretKeyFactory f = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA256");
   return f.generateSecret(spec).getEncoded();
}
```

引用：
Qualys blog: SHA1 Deprecation: What You Need to Know
Google Online Security Blog: Gradually sunsetting SHA-1
NIST: Transitions: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths
NIST: Recommendation for Password-Based Key Derivation

Stackoverflow: Reliable implementation of PBKDF2-HMAC-SHA256 for Java
CWE-327: Use of a Broken or Risky Cryptographic Algorithm

DefaultHttpClient的默认构造函数与TLS 1.2不兼容

漏洞特征： DEFAULT_HTTP_CLIENT
有漏洞的代码：

```
HttpClient client = new DefaultHttpClient();
```

解决方案：
用建议的构造函数去升级你的代码并且配置jvm中https.protocols选项，使其包含TLSv1.2:
使用SystemDefaultHttpClient 代替

- 示例代码：

  ```
  HttpClient client = new SystemDefaultHttpClient();
  ```

  基于SSLSocketFactory类创建一个HttpClient，通过 getSystemSocketFactory()) 获得一个SSLScoketFactory实例，用这个实例去初始化一个HttpClient

  基于SSLConnectionSocketFactory类创建一个HttpClient，通过 getSystemSocketFactory())
  获得一个SSLScoketFactory实例，用这个实例去初始化一个HttpClient

  使用HttpClientBuilder，在调用build()之前调用useSystemProperties()
  示例代码：

  ```
  HttpClient client = HttpClientBuilder.create().useSystemProperties().build();
  ```

- HttpClients,调用 createSystem()去创建一个实例

示例代码：

```
HttpClient client = HttpClients.createSystem();
```

引用：
Diagnosing TLS, SSL, and HTTPS

脆弱的SSLContext

漏洞特征： SSL_CONTEXT
有漏洞的代码：

```
SSLContext.getInstance("SSL");
```

解决方案：
用下面的代码升级你的代码，并且配置jvm的https.protocols选项，使其包含TLSv1.2

```
SSLContext.getInstance("TLS");
```

引用：
Diagnosing TLS, SSL, and HTTPS

习惯使用的信息摘要算法

自己实现消息摘要算法是不靠谱的。
NIST建议使用SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, or SHA-512/256。
“SHA-1用于生成电子签名：
SHA-1可能仅仅用于NIST指导的特殊协议的电子签名的生成。但是在其他的应用中，SHA-1 不应该用于电子签名

SHA-1用于电子签名的验证：
对于电子签名的验证，SHA-1可以被用于传统应用
"SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, and SHA-512/256:
所有散列计算程序都支持这些哈希函数的使用。
NISI:通信传输：传输中建议使用的加密算法和密钥长度
有漏洞的代码：

```
MyProprietaryMessageDigest extends MessageDigest {
   @Override
   protected byte[] engineDigest() {
       [...]
```

```
        //Creativity is a bad idea
        return [...];
    }
}
```

使用其中一种信息摘要算法去升级你的代码。这些算法非常强大，能足够满足你的安全需求。
解决方案示例：

```
MessageDigest sha256Digest = MessageDigest.getInstance("SHA256");
sha256Digest.update(password.getBytes());
```

引用：
NIST Approved Hashing Algorithms
CWE-327: Use of a Broken or Risky Cryptographic Algorithm

读取文件的缺陷

漏洞特征： FILE_UPLOAD_FILENAME
通过篡改FileUpload API 提供的文件名，客户端可以任意访问系统中的文件
比如：

```
"../../../config/overide_file"

"shell.jsp\u0000expected.gif"
```

所以，上面的这些值应该没有做任何过滤就直接进入到了文件系统api之中。如果可能，应用应该生成自己的文件名，并且使用它们。
即使这样，被提供的文件名也要去验证它们的有效性，以确保它们没有包含未授权的路径（比如.八)和未授权的文件。

引用：
Securiteam: File upload security recommendations
CWE-22: Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
WASC-33: Path Traversal
OWASP: Path Traversal
CAPEC-126: Path Traversal
CWE-22: Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')

正则dos

漏洞特征： REDOS
正则表达式(regexs)经常导致拒绝服务攻击（(DOS)。这是因为当正则表达式引擎分析一些字符串的时候会消耗大量的时间，而这也取决于正则是怎么写的。
比如，对于正则^(a+)+$,如果输入"aaaaaaaaaaaaaaaX"，就会让正则表达式引擎分析65536种不同的路径。
所以，可能只要客户端发送一个请求就可以让服务器端消耗巨大的计算资源。问题可能就是类似于这样的正则表达式，由于括号内的+ (or a)和括号外的+ (or a)
，当输入相同字符串的时候，可能会有两种不同的处理方式。以这样的方式去写正则，+号会消耗字符'a'。为了修复这样问题，正则表达式应该被重写，目的是消除歧义.比如
引用：
Sebastian Kubeck's Weblog: Detecting and Preventing ReDoS Vulnerabilities
[1] OWASP: Regular expression Denial of Service
CWE-400: Uncontrolled Resource Consumption ('Resource Exhaustion')

上一篇：[基于DOM的XSS]不要过分的依... 下一篇：区块链安全—详谈合约攻击（三）
1. 1 条回复

tb3344523_**** 2018-12-12 10:55:19

findbugs插件的规则？

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