编码安全性问题-案例

# 【违背规则1】数据校验：对来自系统外部的不可信输入做校验

**说明**：对于外部输入需要检验入参的合法性。如果不对数据作合法性验证，很可能导致DoS、内存越界、格式化字符串漏洞、命令注入、SQL注入、数据破坏等问题。

## 1.1案例一

### **【问题描述】**

未对配置文件获取的IP值进行校验，存在安全性风险

### **【错误代码】**

/\*\* 获取IP地址 \*/

**public** String getIBAddress()

{

//从配置文件中获取的IP，未做IP正则检验与白名单校验

**return** getStringValue("ib\_ip", **null**);

}

### **【正确代码】**

如果在使用时可增加IP正则校验和白名单校验，如下所示：

**public** String getIBAddress()

{

String ip= getStringValue("ib\_ip", **null**);

// *IP\_REG* :IP地址的正则表达式常量

**Pattern** p = **Pattern.compile(*IP\_REG*)**;

**Matcher m = p.matcher(ip);**

//增加IP格式校验和白名单校验

**if**(**m.matches()&& InWhiteList(ip)**)

{

**return** ip;

}

**else**

{

//TODO 记录错误日志

}

}

# 【违背规则2】数据校验：对于B/S、C/S模式的系统，对于外部不可信输入，是否做到在服务端对其进行最终校验

**说明**：对于外部输入数据需要进行服务器端检验入参的合法性。

## 2.1案例一

### **【问题描述】**

如果对于外部输入的数据未经处理，直接用于拼接SQL语句，攻击者可以通过构造特殊形式的输入来改变程序中原本要执行的SQL逻辑，形成SQL注入攻击。

### **【错误代码】**

**public** **void** doPrivilegedAction(String username, **char**[] password) **throws** SQLException

{

**try**

{

String pwd = hashPassword(password);

**String sqlString = "SELECT \* FROM db\_user WHERE username = '"**

**+ username + "' AND password = '" + pwd + "'";**

**Statement stmt = connection.createStatement();**

ResultSet rs = stmt.executeQuery(sqlString);

// Authenticate …

}

}

### **【正确代码】**

使用PreparedStatement，输入的参数无法改变原始的SQL语义。

**public** **void** doPrivilegedAction(String username, **char**[] password)

**throws** SQLException

{

**try**

{

String pwd = hashPassword(password);

// Ensure that the length of user name is legitimate

**if** ((username.length()) > 8)

{

// Handle error

}

**String sqlString = "select \* from db\_user where username=? and password=?";**

**PreparedStatement stmt = connection.prepareStatement(sqlString);**

**stmt.setString(1, username);**

**stmt.setString(2, pwd);**

ResultSet rs = stmt.executeQuery();

// Authenticate …

}

}

# 【违背规则3】数据校验：对字符串类型的外部不可信输入，做校验之前需要先对其进行归一化

**说明：**对外部输入字符串校验之前，需要使用java.text.Normalizer的normalize()方法先对其进行归一化（Unicode Normalization）处理。归一化可以确保具有相同意义的字符串具有统一的二进制描述。

## 3.1案例一

### **【问题描述】**

攻击者可以利用非标准的表达方式来绕过输入检查。

### **【错误代码】**

String s = "\uFE64" + "script" + "\uFE65";

Pattern pattern = Pattern.compile("[<>]");

Matcher matcher = pattern.matcher(s);

**if** (matcher.find())

{

**throw** **new** IllegalStateException();

}

如上示例代码的本意是防止外部输入中包含正反尖括号: “<”和“>”。但是在Unicode中正反尖括号还有另外一种非标准的表达方式：< (\uFE64)和 > (\uFE65)。攻击者可以利用这种非标准的表达方式来绕过输入检查。

### **【正确代码】**

String s = "\uFE64" + "script" + "\uFE65";

//先进行归一化处理

// \uFE64 is normalized to < and \uFE65 is normalized to > using NFKC

**s = Normalizer.normalize(s, Form.NFKC);**

// Validate

Pattern pattern = Pattern.compile("[<>]");

Matcher matcher = pattern.matcher(s);

//再进行校验

if (matcher.find())

{

// Found black listed tag

**throw** **new** IllegalStateException();

}

在进行校验之前先对输入的字符串做归一化处理，可以将使用非标准表达方式的字符串转换成统一的标准表达形式，从而使其不能绕过输入校验。

# 【违背规则4】数据校验：对外部输入文件的路径范围做校验时，应该使用其标准路径做比较

**说明：**路径在验证时会有很多干扰因素，同一个文件可能有不同的表达方式，所以在验证路径时需要对路径进行标准化，使得路径表达唯一化后进行无歧义的比较。

## 4.1案例一

### **【问题描述】**

构造一个跨越目录限制的文件路径，例如“../../../password”

### **【错误代码】**

**public** **static** **void** main(String[] args)

{

File f = **new** File(System.getProperty("user.home")

+ System.getProperty("file.separator") + args[0]);

// getAbsolutePath()方法可以获取全路径，但不会将路径中的“..”处理掉

**String absPath = f.getAbsolutePath();**

// Validation

}

### **【正确代码】**

**public** **static** **void** main(String[] args) **throws** IOException

{

File f = **new** File(System.getProperty("user.home")

+ System.getProperty("file.separator") + args[0]);

//在验证路径之前应该用getCanonicalPath()对路径进行标准化，使得路径表达式唯一，无歧义

**String canonicalPath = f.getCanonicalPath();**

// Validation

}

# 【违背规则5】数据校验：对于源自外部的不可信数据，若要将其输出到外部解释器解析与执行，则需先根据数据的使用方式与解释器的类型做相应的编码或转义

**说明**：对于源自外部的不可信数据，若要将其输出到外部解释器解析与执行，则需先根据数据的使用方式与解释器的类型做相应的编码或转义

## 5.1案例一

### **【问题描述】**

程序对于输出到浏览器的包含在HTML/URL/Javascript/css中的不可信数据、输出到DBMS中的包含在SQL中的不可信数据、输出到LDAP服务器的包含在LDAP命令字符串中的不可信数据，需要进行编码或转义。

### **【错误代码】**

无

### **【正确代码】**

（使用OWASP的ESAPI做转义）:

public void doPrivilegedAction(String username, char[] password)

throws SQLException

{

//...

try

{

String pwd = hashPassword(password);

**Codec oe = new OracleEncodec();**

**String susername = ESAPI.encoder().encodeForSQL(oe,username);**

**String spwd = ESAPI.encoder().encodeForSQL(oe,pwd);**

String sqlString = "SELECT \* FROM db\_user WHERE username = '"

+ susername + "' AND password = '" + spwd + "'";

//...

}

# 【违背规则6】数据校验：禁止使用未经校验的输入拼接构建SQL语句

**说明**：如果sql语句未进行校验，攻击者会将一些恶意代码插入到字符串中。然后会通过各种手段将该字符串传递到数据库的实例中进行分析和执行。只要这个恶意代码符合SQL语句的规则，则在代码编译与执行的时候，就不会被系统所发现

## 6.1案例一

### **【问题描述】**

通过测试类型、长度、格式和范围来验证用户输入，过滤用户输入的内容

### **【错误代码】**

public void doPrivilegedAction(String username, char[] password) throws SQLException

{

//...

try

{

String pwd = hashPassword(password);

**String sqlString = "SELECT \* FROM db\_user WHERE username = '"**

**+ username + "' AND password = '" + pwd + "'";**

Statement stmt = connection.createStatement();

ResultSet rs = stmt.executeQuery(sqlString);

// Authenticate …

}

//...

}

### **【正确代码】**

正确示例（预编译语句）:

public void doPrivilegedAction(String username, char[] password)

throws SQLException

{

//...

try

{

String pwd = hashPassword(password);

/\* Ensure that the length of user name is legitimate \*/

if ((username.length()) > 8)

{

// Handle error

}

**String sqlString = "select \* from db\_user where username=? and password=?";**

**PreparedStatement stmt = connection.prepareStatement(sqlString);**

// …

}

//...

}

正确示例（使用OWASP的ESAPI做转义）:

public void doPrivilegedAction(String username, char[] password)

throws SQLException

{

//...

try

{

String pwd = hashPassword(password);

**Codec oe = new OracleEncodec();**

**String susername = ESAPI.encoder().encodeForSQL(oe,username);**

**String spwd = ESAPI.encoder().encodeForSQL(oe,pwd);**

String sqlString = "SELECT \* FROM db\_user WHERE username = '"

+ susername + "' AND password = '" + spwd + "'";

//...

}

# 【违背规则7】数据校验：禁止使用未经校验的输入拼接构建OS命令行语句并通过Runtime.exec()方法执行

**说明**： 如果输入未进行校验，攻击者会将一些恶意代码插入到字符串中,从而获取一些有用信息

## 7.1案例一

### **【问题描述】**

通过测试类型、长度、格式和范围来验证用户输入，过滤用户输入的内容

### **【错误代码】**

class DirList

{

public static void main(String[] args) throws Exception

{

String dir = System.getProperty("dir");

Runtime rt = Runtime.getRuntime();

**Process proc = rt.exec(new String[] {"sh", "-c", "ls " + dir});**

// ...

}

}

### **【正确代码】**

正确示例（避免使用Runtime.exec()）:

class DirList

{

public static void main(String[] args) throws Exception

{

File dir = new File(System.getProperty("dir"));

if (!dir.isDirectory())

{

System.out.println("Not a directory");

}

else

{

for (String file : dir.list())

{

System.out.println(file);

}

}

}

}

正确示例（输入校验）：

// ...

**if (!Pattern.matches("[0-9A-Za-z@.]+", dir))**

{

// Handle error

}

// ...

# 【违背规则8】数据校验：禁止使用非信任输入拼接构建格式化字符串

**说明**：禁止使用非信任输入拼接构建格式化字符串

## 8.1案例一

### **【问题描述】**

禁止使用非信任输入拼接构建格式化字符串

### **【错误代码】**

class Format

{

static Calendar c = new GregorianCalendar(1995, GregorianCalendar.MAY, 23);

public static void main(String[] args)

{

// args[0] is the credit card expiration date

// args[0] can contain either %1$tm, %1$te or %1$tY as malicious arguments

// First argument prints 05 (May), second prints 23 (day)

// and third prints 1995 (year)

// Perform comparison with c, if it doesn't match print the following line

**System.out.printf(args[0] + " did not match! HINT: It was issued on %1$terd of some month", c);**

}

}

### **【正确代码】**

class Format

{

static Calendar c = new GregorianCalendar(1995, GregorianCalendar.MAY, 23);

public static void main(String[] args)

{

// args[0] is the credit card expiration date

// Perform comparison with c,

// if it doesn't match print the following line

**System.out.printf("%s did not match! "**

**+ " HINT: It was issued on %1$terd of some month", args[0], c);**

}

}

# 【违背规则9】数据校验：禁止使用未经校验的输入拼接构建XML数据

**说明：**构造XML节点时，当XML中包含有未经审查的用户输入时，可能会产生XML注入攻击。恶意攻击者伪造XML数据，改变XML的原有结构，达到攻击的目的。

## 9.1案例一

### **【问题描述】**

发送soap消息时，部分内容是由外部参数传入的，可能会产生XML注入攻击。

### **【错误代码】**

Assert.notNullOrEmpty(moduleName);

StringBuilder soapMsg = new StringBuilder();

soapMsg.append("<?xml version=\"1.0\" encoding=\"UTF-8\"?>");

soapMsg.append("<soapenv:Envelope xmlns:soapenv=\"http://schemas.xmlsoap.org/soap/envelope/\">");

// ...

soapMsg.append("<moduleName>");

soapMsg.append(moduleName);

soapMsg.append("</moduleName>");

// ...

### **【正确代码】**

Assert.notNullOrEmpty(moduleName);

if (!moduleName.matches(允许字符列表的正则表达式字符串常量))

{

throw new BMEException(异常码);

}

// ...

soapMsg.append("<moduleName>");

soapMsg.append(moduleName);

soapMsg.append("</moduleName>");

// ...

# 【违背规则10】数据校验：优先考虑采用“白名单”的方式对输入中的字符或内容做校验

**说明：**“黑名单”和“白名单”是进行数据净化的两种途径。“白名单”比“黑名单”更受推荐（优先考虑采用）的原因是，程序员不必花力气去捕捉所有不可接受的字符，只需确保识别了可接受的字符就可以了。

## 10.1案例一

### **【问题描述】**

2013年，Struts2爆出安全漏洞，通过构造特殊的action url（带"action:"、"redirect:"或"redirectAction:"参数），在值栈上执行传入的OGNL表达式，可以在后台执行任意系统命令。

引申阅读：http://struts.apache.org/development/2.x/docs/s2-016.html

部分修复漏洞代码（DefaultActionMapper.java中）采用白名单验证"action:"参数。

### **【错误代码】**

public void execute(String key, ActionMapping mapping)

{

String name = key.substring("action:".length());

// ...

mapping.setName(name);

}

### **【正确代码】**

protected Pattern allowedActionNames = Pattern.compile("[a-zA-Z0-9.\_!/\\-]\*");

public void execute(String key, ActionMapping mapping)

{

String name = key.substring("action:".length());

// ...

mapping.setName(cleanupActionName(name));

}

protected String cleanupActionName(String rawActionName)

{

if (allowedActionNames.matcher(rawActionName).matches())

{

return rawActionName;

}

// ... 省略方法剩余代码。有非白名单中定义的字符，需要过滤掉再返回

}

# 【违背规则11】数据校验：安全地从ZipInputStream中解压抽取文件

**说明：**ZIP压缩格式可以做到很高的压缩率。如果一个zip文件中的条目解压之后的大小超过了一定的上限，则应该拒绝对其进行解压，否则容易导致系统资源占用过多，形成拒绝服务攻击和zip炸弹。该规则不仅仅限定于zip格式，其它jar、rar等格式也是同样如此。

## 11.1案例一

### **【问题描述】**

如果一个zip文件中的条目解压之后的大小超过了一定的上限，则应该拒绝对其进行解压。

### **【错误代码】**

ZipInputStream zis = new ZipInputStream(...);

ZipEntry entry;

while ((entry = zis.getNextEntry()) != null)

{

// 解压zip文件中的条目

// ...

}

### **【正确代码】**

ZipInputStream zis = new ZipInputStream(...);

ZipEntry entry;

while ((entry = zis.getNextEntry()) != null)

{

// 如果zip文件条目过大或者不确定大小，则不解压

if (entry.getSize() > 100MB常量)

{

throw new IllegalStateException("File to be unzipped is huge.");

}

if (entry.getSize() == -1常量)

{

throw new IllegalStateException("File to be unzipped might be huge.");

}

// 解压zip文件中的条目

// ...

}

# 【违背规则12】异常行为：禁止在对外抛出的异常中暴露敏感信息

**说明：**传递异常时往往会导致敏感信息泄漏，如果没有过滤敏感信息，可能有助于攻击者发起进一步攻击，攻击者可能会构造输入参数进一步挖掘应用程序的内部结构和实现机制。

## 12.1案例一

### **【问题描述】**

执行后台DAO方法时，由于数据库问题可能抛出DASException类型（BMEException类的子类）的异常。页面在后台抛出BMEException类型的异常时，会弹出定制的错误对话框，显示异常信息。某些DASException类型的异常会暴露表结构，不能直接抛出。

### **【错误代码】**

// 准备页面显示用到的Message对象

Message msg = new Message();

msg.setErrorState(前后台处理异常定的错误标识);

if (exception instanceof BMEException)

{

// 处理“后台抛出BMEException类型的异常”

BMEException ex = (BMEException) exception;

msg.setErrorMessage(ex.getExceptionInfo().getDesc());

msg.setErrorCode(ex.getExceptionCode());

}

### **【正确代码】**

// 准备页面显示用到的Message对象

Message msg = new Message();

msg.setErrorState(前后台处理异常定的错误标识);

if (exception instanceof BMEException)

{

// 处理“后台抛出BMEException类型的异常”

BMEException ex = (BMEException) exception;

String exCode = ex.getExceptionCode();

if (exCode是否属于DASException的错误码)

{

// exCode属于DASException的错误码，可能暴露表结构，换成一个不暴露底层信息的通用错误码

ex = new BMEException(通用错误码);

}

msg.setErrorMessage(ex.getExceptionInfo().getDesc());

msg.setErrorCode(ex.getExceptionCode());

}

# 【违背规则13】异常行为：禁止将异常信息输出到控制台

**说明：**异常信息输入到控制台只是输出到屏幕，并没有持久化到文件中，异常信息容易丢失。

## 13.1案例一

### **【问题描述】**

如下错误代码只会在屏幕上打印异常信息，而正确代码会把异常信息格式化的打印到日志文件中。

### **【错误代码】**

try

{

int a=Integer.parseInt("xx");

}

catch (NumberFormatException e)

{

System.err.println(e);

}

### **【正确代码】**

try

{

int a=Integer.parseInt("xx");

}

catch (NumberFormatException e)

{

ILog.DEFAULT.error(e, true, e.toString());

}

# 【违背规则14】异常行为：禁止在finally块中使用return、break、continue、以及throw语句造成try块中的异常丢失

**说明：**在try{...}catch(XX){...}finally{...}中，如果finally中return，那么try{..}以及catch{...}中的返回都被finally覆盖。

## 14.1案例一

### **【问题描述】**

错误代码执行结果为：start和finally。

正确代码执行结果为：start、finally和NumberFormatException异常。

### **【错误代码】**

try

{

System.out.println("start");

int num = Integer.parseInt("xx");

System.out.println("end");

}

finally

{

System.out.println("finally");

return;

}

### **【正确代码】**

try

{

System.out.println("start");

int num = Integer.parseInt("xx");

System.out.println("end");

}

finally

{

System.out.println("finally");

}

# 【违背规则15】异常行为：禁止在finally块中抛出异常造成try块中的异常丢失

**说明：**在try{...} finally{...}中，如果finally中throw异常，那么try{..}中的异常会被finally覆盖。

## 15.1案例一

### **【问题描述】**

错误代码执行结果:start、finally和java.lang.IllegalStateException异常。

正确代码执行结果：start、finally和java.lang.NumberFormatException异常。

### **【错误代码】**

try

{

System.out.println("start");

int num = Integer.parseInt("xx");

System.out.println("end");

}

finally

{

System.out.println("finally");

throw new IllegalStateException();

}

### **【正确代码】**

try

{

System.out.println("start");

int num = Integer.parseInt("xx");

System.out.println("end");

}

finally

{

System.out.println("finally");

}

# 【违背规则16】IO操作：临时文件使用完毕之后应该及时删除

**说明：**临时文件使用完后，应该及时删除，避免浪费资源和信息的泄露。

## 16.1案例一

### **【问题描述】**

错误代码：创建临时文件，使用完后没有及时删除，会导致文件长期放置在临时目录中浪费资源。

正确代码：创建临时文件，使用完后都会在finally方法中及时删除。

### **【错误代码】**

File f = null;

FileOutputStream fop = null;

try

{

f = File.createTempFile("tempnam", ".tmp");

fop = new FileOutputStream(f);

String str = "Data";

fop.write(str.getBytes());

fop.flush();

// handle tempnam.tmp logic

}

catch (FileNotFoundException fe)

{

// handle exception

}

catch (IOException e)

{

// handle exception

}

finally

{

if (fop != null)

{

try

{

fop.close();

}

catch (IOException x)

{

// handle error

}

}

}

### **【正确代码】**

File f = null;

FileOutputStream fop = null;

try

{

f = File.createTempFile("tempnam", ".tmp");

fop = new FileOutputStream(f);

String str = "Data";

fop.write(str.getBytes());

fop.flush();

// handle tempnam.tmp logic

}

catch (FileNotFoundException fe)

{

// handle exception

}

catch (IOException e)

{

// handle exception

}

finally

{

if (fop != null)

{

try

{

fop.close();

}

catch (IOException x)

{

// handle error

}

}

// delete file when finished

if (f != null)

{

boolean isDelete = f.delete();

if (!isDelete)

{

// handle log

}

}

}

【违背规则17】IO操作：资源使用完毕之后须及时释放

**说明**：程序在使用完文件句柄、网络连接、数据库连接等资源之后是否做到进行资源释放。

## 17.1案例一

### **【问题描述】**

写文件的方法在建立了与文件的通道后，没有释放资源，应该在finally里进行释放。

### **【错误代码】**

try

{

FileWriter fileWriter = new FileWriter(new File(filePath));

//…………

}

//省略catch

### **【正确代码】**

try

{

FileWriter fileWriter = new FileWriter(new File(filePath));

//…………

}

//省略catch

finally

{

//这里简写成这样（实际不可以这样写），旨在告诉大家要释放资源。----修改如下

If (fileWriter != null)

{

fileWriter.close();

}

}

# 【违背规则18】IO操作：不要将wrap()和duplicate()方法创建的buffer对象暴露给非信任代码

**说明**：如果将wrap()和duplicate()方法创建的buffer对象暴露给非信任代码，可能会导致被恶意修改。

## 18.1案例一

### **【问题描述】**

wrap()和duplicate()方法创建的buffer对象暴露给非信任代码。

### **【错误代码】**

public class Wrap

{

private char[] dataArray;

public Wrap()

{

dataArray = new char[10];

// Initialize

}

public CharBuffer getBufferCopy()

{

return CharBuffer.wrap(dataArray);

}

}

public class Dup

{

CharBuffer cb;

public Dup()

{

cb = CharBuffer.allocate(10);

// Initialize

}

public CharBuffer getBufferCopy()

{

return cb.duplicate();

}

}

### **【正确代码】**

public class Wrap

{

private char[] dataArray;

public Wrap()

{

dataArray = new char[10];

// Initialize

}

public CharBuffer getBufferCopy()

{

return CharBuffer.wrap(dataArray).asReadOnlyBuffer();

}

}

public class Dup

{

CharBuffer cb;

public Dup()

{

cb = CharBuffer.allocate(10);

// Initialize

}

public CharBuffer getBufferCopy()

{

return cb.duplicate().asReadOnlyBuffer();

}

}

# 【违背规则19】IO操作：不要在单个InputStream上创建多个buffered wrapper

**说明：**在单个InputStream上创建多个bufferedwrapper会导致不必要的内存开销,设置了缓存大小时,也有可能会丢失数据。

## 19.1案例一

### **【问题描述】**

在单个InputStream上创建了多个bufferedwrapper导致了不必要的内存开销。

### **【错误代码】**

private static StringReader sr = new StringReader("12345678abcdefgh");

public static char getChar() throws IOException

{

BufferedReader br = new BufferedReader(sr, 4); // wrapper

int input = br.read();

if (input == -1)

{

throw new EOFException();

}

return (char) input;

}

public static void main(String[] args)

{

try

{

char first = getChar();

char last = getChar();

}

//省略catch

}

### **【正确代码】**

private static BufferedInputStream in = new BufferedInputStream(System.in);

public static char getChar() throws EOFException, IOException

{

return (char) in.read();

}

public static void main(String[] args)

{

try

{

char first = getChar();

char last = getChar();

}

//省略catch

}

# 【违背规则20】IO操作：创建文件时指定合理的访问权限

**说明：**除非文件本身在安全目录中，无法被非信任用户所访问，或者文件本身在业务上无需访问权限控制，否则需要指定合理的访问权限。

## 20.1案例一

### **【问题描述】**

创建文件时需指定合理的访问权限。

### **【错误代码】**

Writer out = new FileWriter("file");

### **【正确代码】**

Path file = new File("file").toPath();

Set<OpenOption> options = new HashSet<OpenOption>();

options.add(StandardOpenOption.CREATE\_NEW);

options.add(StandardOpenOption.APPEND);

Set<PosixFilePermission> perms = PosixFilePermissions.fromString("rw-------");

FileAttribute<Set<PosixFilePermission>> attr = PosixFilePermissions.asFileAttribute(perms);

SeekableByteChannel sbc = Files.newByteChannel(file, options, attr);

# 【违背规则21】序列化：对于包含敏感信息的类，避免将其声明为可序列化的

**说明**：类被序列化后，在序列化文件中仍然可以看到被序列化的类的属性值，引发安全风险。

## 21.1案例一

### **【问题描述】**

SensitiveClass类实现可序列化接口，这个类中的敏感信息可能被序列化而产生泄漏。

### **【错误代码】**

class SensitiveClass implements Serializable

{

//class definition

}

### **【正确代码】**

class SensitiveClass

{

//class definition

}

# 【违背规则22】序列化：禁止序列化未加密的敏感信息

**说明**：对于类中的敏感信息字段，检查程序是否做到通过下列其中一种方式,阻止敏感信息字段被序列化：

1. 将敏感信息字段声明为"transient"

2. 自定义writeObject(), writeReplace(), writeExternal()方法，不让敏感信息字段写入序列化流中

3. 定义serialPersistentFields数组，并将敏感信息字段排除在该数组之外。

## 22.1案例一

### **【问题描述】**

如果User在被序列化时，没有加密，那么序列化后，pwd字段可能会产生泄漏。

### **【错误代码】**

public class User implements Serializable

{

private String pwd;

private String username;

public User (String pwd, String username)

{

this.username = username;

this.pwd = pwd;

}

}

### **【正确代码】**

public class User implements Serializable

{

private transient String pwd;

private String username;

public User (String pwd, String username)

{

this.username = username;

this.pwd = pwd;

}

}

# 【违背规则23】序列化：若未使用安全通道，如SSL/TLS，传输包含敏感信息的序列化对象到信任边界之外，则需要对其先签名后加密

**说明：**

敏感数据传输过程中要防止窃取和非法篡改。使用安全的加密算法给数据加密可以防止数据被窃取。而对数据加上数字签名则可以防止数据被非法篡改。

## 23.1案例一

### **【问题描述】**

这段代码先对数据进行加密，然后再对加密后的数据进行签名。这样做无法保证签名来自数据的原始来源。任何恶意的第三方可以截获原始加密签名后的数据，剔除原始的签名，并对密封的数据加上自己的签名。这样的话，即使恶意第三方无法获取原始的数据内容，正常的接收者也无法得到原始的数据。

### **【错误代码】**

public static void main(String[] args)

{

User user = new User(“aaa”, “bbb”);

//对user加密

//对user加密后的数据进行签名

//发送加密&&签名后的数据。

}

### **【正确代码】**

public static void main(String[] args)

{

User user = new User(“aaa”, “bbb”);

//对user签名

//对user签名后的数据加密

//发送加密&&签名后的数据。

}

# 【违背规则24】序列化：防止序列化与反序列化被利用来绕过安全检查

**说明：**

1. 若一个可序列化类中在构造器方法中实施了安全检查，则同样的安全检查是否应用到反序列化方法中，如readObject()、readObjectNoData()

2. 若一个可序列化类的内部属性修改器方法(如setXXX())中实施了安全检查，则同样的安全检查是否应用到反序列化方法中，如readObject()、readObjectNoData()

3. 若一个可序列化类中的内部属性访问器方法(如getXXX())中实施了安全检查，则同样的安全检查是否应用到序列化方法中，如writeObject()。

## 24.1案例一

### **【问题描述】**

security manager checks在构造器中被使用，在序列化-反序列化的writeObject()和readObject()方法中没有用到，这样会允许非信任代码恶意创建实例。

### **【错误代码】**

public final class Hometown implements Serializable

{

private String town;

public Hometown()

{

performSecurityManagerCheck();

town = “XXXXX”;

}

private void writeObject(ObjectOutputStream out) throws IOException

{

out.writeObject(town);

}

private void readObject(ObjectInputStream in) throws IOException

{

in.defaultReadObject();

}

}

### **【正确代码】**

private void writeObject(ObjectOutputStream out) throws IOException

{

performSecurityManagerCheck();

out.writeObject(town);

}

private void readObject(ObjectInputStream in) throws IOException

{

in.defaultReadObject();

performSecurityManagerCheck();

}

# 【违背规则25】平台安全：禁止序列化非静态内部类实例

**说明**：如果一个序列化类是一个非序列化类的内部类时，会试图去序列化该类关联的外部类的实例，导致运行时错误，将内部类声明为静态将解决该问题。将外部类序列化也能解决运行时错误的问题，但暴露了本不应该序列化的外部实例数据。

## 25.1案例一

### **【问题描述】**

序列化类是非序列化类的内部类。让内部类具有static属性可以解决该问题。

### **【错误代码】**

public class OuterSer

{

private int rank;

class InnerSer implements Serializable

{

protected String name;

//...

}

}

### **【正确代码】**

public class OuterSer

{

private int rank;

static class InnerSer implements Serializable

{

protected String name;

//...

}

}

# 【违背规则26】平台安全：使用安全管理器对敏感操作进行保护

**说明**：当与非信任代码（非产品包中的代码）交互时，敏感操作必须由SecurityManager来检查控制。

## 26.1案例一

### **【问题描述】**

对于敏感操作需要安全控制

### **【错误代码】**

class SensitiveHash

{

Hashtable<Integer,String> ht = new Hashtable<Integer,String>();

public void removeEntry(Object key)

{

ht.remove(key);

}

}

### **【正确代码】**

class SensitiveHash

{

Hashtable<Integer,String> ht = new Hashtable<Integer,String>();

void removeEntry(Object key)

{

check("removeKeyPermission");

ht.remove(key);

}

private void check(String directive)

{

SecurityManager sm = System.getSecurityManager();

if (sm != null)

{

sm.checkSecurityAccess(directive);

}

}

}

# 【违背规则27】平台安全：防止AccessController.doPrivileged特权块向信任边界之外泄露敏感资源与敏感功能

**说明：**1.检查特权块是否使用了未经校验的不可信数据

2.检查特权块是否将受限访问的资源或功能通过public方法泄露出去。

## 27.1案例一

### **【问题描述】**

对于特权代码不能使用public关键字来暴露此方法

### **【错误代码】**

public class PasswordManager

{

public static void changePassword() throws FileNotFoundException

{

FileInputStream fin = openPasswordFile();

// test old password with password in file contents; change password

// then close the password file

}

public static FileInputStream openPasswordFile() throws FileNotFoundException

{

final String password\_file = "password";

FileInputStream fin = null;

try

{

fin = AccessController.doPrivileged(new PrivilegedExceptionAction<FileInputStream>()

{

public FileInputStream run() throws FileNotFoundException

{

// Sensitive action; can't be done outside privileged block

FileInputStream in = new FileInputStream(password\_file);

return in;

}

});

}

catch (PrivilegedActionException x)

{

Exception cause = x.getException();

if (cause instanceof FileNotFoundException)

{

throw (FileNotFoundException) cause;

}

else

{

throw new Error("Unexpected exception type", cause);

}

}

return fin;

}

}

### **【正确代码】**

public class PasswordManager

{

public static void changePassword() throws FileNotFoundException {

// ...

}

private static FileInputStream openPasswordFile() throws FileNotFoundException

// ...

}

# 【违背规则28】平台安全：在自定义类加载器中实现getPermissions(CodeSource cs)方法时必须加入调用超类的该方法

**说明：**检查自定义子类加载器实现的getPermissions(CodeSource cs)方法中是否做到先调用super.getPermissions(CodeSource cs)，并将其返回结果加入到子类。方法的返回结果集中。

## 28.1案例一

### **【问题描述】**

检查自定义子类加载器实现的getPermissions(CodeSource cs)方法中是否做到先调用super.getPermissions(CodeSource cs)，并将其返回结果加入到子类。方法的返回结果集中。

### **【错误代码】**

protected PermissionCollection getPermissions(CodeSource cs)

{

PermissionCollection pc = new Permissions();

// allow exit from the VM anytime

pc.add(new RuntimePermission("exitVM"));

return pc;

}

### **【正确代码】**

protected PermissionCollection getPermissions(CodeSource cs)

{

PermissionCollection pc = super.getPermissions(cs);

// allow exit from the VM anytime

pc.add(new RuntimePermission("exitVM"));

return pc;

}

# 【违背规则29】其他：对本地方法调用进行封装，为其定义一个包装器

**说明**：检查程序是否做到封装本地方法调用，对本地方法调用的参数与返回值做必要的校验。

## 29.1案例一

### **【问题描述】**

在调用本地方法时，如果不对输入参数做校验，会发生不可预知的情况发生，导致代码执行不可控，甚至发生严重性的问题。

### **【错误代码】**

public final class NativeMethod

{

public native void nativeOperation(byte[] data, int offset, int len);

}

### **【正确代码】**

public final class NativeMethodWrapper

{

// private native method

private native void nativeOperation(byte[] data, int offset, int len);

// wrapper method performs checks

public void doOperation(byte[] data, int offset, int len)

{

// copy mutable input

data = data.clone();

// validate input

if (offset < 0 || len < 0 || offset > data.length - len)

{

throw new IllegalArgumentException();

}

nativeOperation(data, offset, len);

}

}

# 【违背规则30】其他：对于安全敏感的应用场景，使用强随机数生成器

**说明**：典型的安全敏感应用如生成身份认证验证码、会话id、密钥相关的随机数，在这些场景中是否做到使用强随机数生成器:java.security.SecureRandom。

## 30.1案例一

### **【问题描述】**

对于安全敏感的应用场景，比如重置密码生成随机密码，未使用强随机数生成器

### **【错误代码】**

Random random = new Random();

// 此处使用的伪随机数，存在安全隐患。应该使用更安全的随机数生存器java.security.SecureRandom！

### **【正确代码】**

SecureRandom random = null;

try

{

random = SecureRandom.getInstance("SHA1PRNG");

// 改用强制的更安全的随机数生成器java.security.SecureRandom，提高安全性！

}

catch (NoSuchAlgorithmException e)

{

debugLogger.error("generate SecureRandom find wrong.", e);

…….

}

# 【违背规则31】其他：禁止在代码中硬编码敏感信息，如口令、密钥、数据库连接字符串

**说明：**在代码中硬编码敏感信息，就是将敏感信息直接写在代码中。这样很容泄露敏感信息，并且一旦发生问题，无法及时有效的处理安全问题。

## 31.1案例一

### **【问题描述】**

对数据库连接，用户名密码使用硬编码，会带来敏感信息泄露的安全问题。

### **【错误代码】**

public final Connection getConnection() throws SQLException

{

return DriverManager.getConnection(

"jdbc:mysql://localhost/dbName",

"username", "password");

}

### **【正确代码】**

public final Connection getConnection() throws SQLException

{

String jdbcUrl;

String username;

String password;

// JdbcUrl, username and password are read at runtime from a secure config file

return DriverManager.getConnection(

jdbcUrl, username, password);

}

# 【违背规则32】其他：删除或修改没有使用到的变量或值，删除或修改没有效果的代码

**说明：**由于在编写代码的过程中会不断的修改代码逻辑，所以经常会导致某些定义的变量或方法未被使用，还有一些代码虽然调用了，但未带来任何效果，比如调用一个没有返回值的函数，同时参数传递进去的是基础类型，由于是值传递，所以对程序的运行不会带来任何效果，像诸如此类的代码都属于废代码，应该从项目中删除，好让代码结构更加清晰，同时降低后期的维护成本。

## 32.1案例一

### **【问题描述】**

函数中定义的变量未使用，属于废代码。

### **【错误代码】**

public static boolean isEmptyOrNull(List<Object> list)

{

//属于废代码，应该删除

boolean result = true;

if (list == null || list.isEmpty())

{

return true;

}

return false;

}

### **【正确代码】**

public static boolean isEmptyOrNull(List<Object> list)

{

if (list == null || list.isEmpty())

{

return true;

}

return false;

}

# 【违背规则33】其他：禁止在日志中输出敏感信息，如明文口令/密文口令、密钥等

**说明：在文件日志中输出敏感信息，这样会导致用户信息泄露，引发安全问题。**

## 33.1案例一

### **【问题描述】**

在日志中输出敏感信息，如明文口令/密文口令、密钥等。

### **【错误代码】**

Pattern p = Pattern.compile(reg, 2);

Matcher m = p.matcher(line);

if (m.find())

{

String username = m.group(1);

String password = m.group(2);

logger.info((new StringBuilder("connect ")).append(username).append("/").append(password).toString());

}

### **【正确代码】**

Pattern p = Pattern.compile(reg, 2);

Matcher m = p.matcher(line);

if (m.find())

{

String username = m.group(1);

String password = "\*\*\*\*\*\*";

logger.info((new StringBuilder("connect ")).append(username).append("/").append(password).toString());

}

# 【违背规则34】其他：对于内存中的敏感信息，使用完之后应该及时清除

**说明：对于内存中的敏感信息如：密码，如果一直缓存不清除，可能导致信息被盗取。**

## 34.1案例一

### **【问题描述】**

对于内存中的敏感信息，使用完之后未及时清除。

### **【错误代码】**

String input = passwordField.getText();

//…

### **【正确代码】**

char[] input = passwordField.getPassword();

//…

//Zero out the possible password, for security.

for (int i = 0; i < input.length; i++)

{

input[i] = 0;

}

# 【违背规则35】其他：与非信任代码交互的类的同步操作要使用私有的、无法被非信任代码访问与修改的锁

**说明：防止在与非安全代码交互时，通过非私有代码的访问进行修改。**

## 35.1案例一

### **【问题描述】**

与非信任代码交互的类的同步操作要使用了私有的、无法被非信任代码访问与修改的锁

### **【错误代码】**

// Locks on the object's monitor

public synchronized void changeValue()

{

// ...

}

### **【正确代码】**

// private final lock object

private final Object lock = new Object();

public void changeValue()

{

// Locks on the private Object

synchronized (lock)

{

/ ...

}

}