中软国际公司内部技术规范

Java语言安全编程规范

中软国际科技服务有限公司

修订声明

参考:华为 Java 语言安全编码规范。

1.

日期	修订版本	修订描述	作者
2017-07-28	1.0	整理创建初稿	陈丽佳
2017-08-02	1.1	统一整理格式，并修正部分错误	陈丽佳

目录

前言

背景

《Java 语言安全编程规》针对 Java 语言编程中的输入校验、异常行为、IO 操作、序列化和反序列化、平台安全与运行安全等方面，描述可能导致安全漏洞或风险的常见编码错误。该规范基于业界最佳实践，参考业界安全编码规范相关著作，例如 *The Cert Secure Coding Standard for Java*、*Sun Secure Coding Guidelines for the Java Programming Language*、*CWE/SANS TOP 25* 和 *OWASP Guide Project*，并总结了公司内部的编程实践。该规范旨在减少 SQL 注入、敏感信息泄露、格式化字符串攻击、命令注入攻击、目录遍历等安全问题的发生。

使用对象

本规范的读者及使用对象主要为使用 Java 语言的研发人员和测试人员等。

适用范围

该规范适用于基于 Java 语言的产品开发。除非有特别说明，所有的代码示例都是基于 JDK1.6 版本。

术语定义

规则：编程时必须遵守的约定
说明：某个规则的具体解释
错误示例：违背某条规则的例子
正确示例：遵循某条规则的例子
例外情况：相应的规则不适用的场景
信任边界：位于信任边界之内的所有组件都是被系统本身直接控制的。所有来自不受控的外部系统的连接与数据，包括客户端与第三方系统，都应该被认为是不可信的，要先在边界处对其校验，才能允许它们进一步与本系统交互。
非信任代码：非产品包中的代码，如通过网络下载到本地虚拟机中加载并执行的代码。

2. 数据校验

规则 1.1 校验跨信任边界传递的不可信数据

说明：程序可能会接收来自用户、网络连接或其他来源的不可信数据，并将这些数据跨信任边界传递到目标系统（如浏览器、数据库等）。由于目标系统可能无法区分处理畸形的不可信数据，未经校验的不可信数据可能会引起某种注入攻击，对系统造成严重影响，因此，必须对不可信数据进行校验，且数据校验必须在信任边界以内进行（如对于 Web 应用，需要在服务端做校验）。数据校验有输入校验和输出校验，对从信任边界外传入的数据进行校验的叫输入校验，对传出到信任边界外的数据进行校验的叫输出校验。

如下描述了四种数据校验策略（任何时候，尽可能使用接收已知合法数据的“白名单”策略）。

接受已知好的数据

这种策略被称为“白名单”或者“正向”校验。该策略检查数据是否属于一个严格约束的、已知的、可接受的合法数据集合。例如，下面的示例代码确保 name 参数只包含字母、数字以及下划线。

```
//...
If (Pattern.Matches ("^ [0-9A-Za-z_] +$", fileName))
{
    throw new IllegalArgumentException("Invalid file name");
}
//..
```

拒绝已知坏的数据

这种策略被称为“黑名单”或者“负向”校验，相对于正向校验，这是一种较弱的校验方式。由于潜在的不合法数据可能是一个不受约束的无限集合，这就意味着你必须一直维护一个已知不合法字符或者模式的列表。如果不定期研究新的攻击方式并对校验的表达式进行日常更新，该校验方式就会很快过时。

```
public String removeJavascript(String input)
{
    Pattern p = Pattern.compile("javascript", Pattern.CASE_INSENSITIVE);
    Matcher m = p.matcher(input);
    return (!m.matches()) ? input : "";
}
```

“白名单”方式净化

对任何不属于已验证合法字符数据中的字符进行净化，然后再使用净化后的数据，净化的方式包括删除、编码、替换。比如，如果你期望接收一个电话号码，那么你可以删除掉输入中所有的非数字字符，“(555)123-1234”，“555.123.1234”，与“555\";DROP TABLE USER;--123.1234”全部会被转换为“5551231234”，然后再对转换的结果进行校验。又比如，对于用户评论栏的文本输入，由于几乎所有的字符都可能被用到，确定一个合法的数据集合是非常困难的，一种解决方案是对所有非字母数字进行编码，如对“I like your web page!”使用 URL 编码，其净化后的输出为“I+like+your+web+page%21”。“白名单”方式净化不仅有利于安全，它也允许接收和使用更宽泛的有效用户输入。

“黑名单”方式净化

为了确保输入数据是“安全”的，可以剔除或者转换某些字符（例如，删除引号、转换成 HTML 实体）。跟“黑名单”校验类似，随着时间推移不合法字符的范围很可能不一样，需要对不合法字符进行日常维护。因此，执行一个单纯针对正确输入的“正向”校验更加简单、高效、安全。

```
public static String quoteApostrophe(String input)
```

```
{
    if (input != null)
    {
        return input.replaceAll("\'", "&rsquo;");
    }
    else
    {
        return null;
    }
```

规则 1.2 禁止直接使用不可信数据来拼接 SQL 语句

说明：SQL 注入是指原始 SQL 查询被恶意动态更改成一个与程序预期完全不同的查询。执行更改后的查询可能会导致信息泄露或者数据被篡改。防止 SQL 注入的方式主要分为两类：

- 使用参数化查询
- 对不可信数据进行校验

参数化查询是一种简单有效的防止 SQL 注入的查询方式，应该被优先考虑使用。另外，参数化查询还能提高数据库访问的性能，例如，SQL Server 与 Oracle 数据库会为其缓存一个查询计划，以便在后续重复执行相同的查询语句时无需编译而直接使用。

错误示例（Java 代码动态构建 SQL）：

```
Statement stmt = null;
ResultSet rs = null;
try
{
    String userName = ctx.getAuthenticatedUserName(); //this is a constant
    String sqlString = "SELECT * FROM t_item WHERE owner='" + userName + "' AND
itemName='" + request.getParameter("itemName") + "'";
    stmt = conn.createStatement();
    rs = stmt.executeQuery(sqlString);
    // ... result set handling
}
catch (SQLException se)
{
    // ... logging and error handling
}
```

上例中查询字符串常量与用户输入将拼接成动态 SQL 查询命令。仅当 `itemName` 不包含单引号时，这条查询语句的行为才会和预期是一致的。如果一个攻击者提交的 `userName` 为 *wiley*，同时 `itemName` 为：

```
name' OR 'a' = 'a
```

那么这个查询将变成：

```
SELECT * FROM t_item WHERE owner = 'wiley' AND itemname = 'name' OR
'a'='a';
```

这里的 `OR 'a'='a'` 条件会导致整个 WHERE 子句的值总为真。所以这个查询便等价于如下非常简单的查询：

```
SELECT * FROM t_item
```

此查询使得攻击者能够绕过原有的条件限制，返回 items 表中所有储存的条目，而不管它们的所有者是谁（原本应该只返回属于当前已认证用户的条目）

正确示例（使用 PreparedStatement 进行参数化查询）：

```
PreparedStatement stmt = null
ResultSet rs = null
try
{
    String userName = ctx.getAuthenticatedUserName(); //this is a constant
```

```
    String itemName = request.getParameter("itemName");
    // ...Ensure that the length of userName and itemName is legitimate
    // ...
    String sqlString = "SELECT * FROM t_item WHERE owner=? AND itemName=?";

    stmt = conn.prepareStatement(sqlString);
    stmt.setString(1, userName);
    stmt.setString(2, itemName);
    rs = stmt.executeQuery();
    // ... result set handling
}
catch (SQLException se)
{
    // ... logging and error handling
}
```

参数化查询在 SQL 语句中使用占位符表示需在运行时确定的参数值，使得 SQL 查询的语义逻辑预先被定义，实际的查询参数值则在程序运行时再确定。参数化查询使得数据库能够区分 SQL 语句中语义逻辑和数据参数，以确保用户输入无法改变预期的 SQL 查询语义逻辑。在 C#中，可以使用 SqlCommand.Parameters.AddWithValue 进行参数化查询。正确示例中，如果攻击者的 itemName 输入为 name' OR 'a' = 'a，参数化查询将会查找 itemName 匹配 name' OR 'a' = 'a 字符串的条目，而不是返回整个表中的所有条目。

错误示例（在存储过程中动态构建 SQL）：

Java 代码：

```
CallableStatement = null
ResultSet results = null;
try
{
    String userName = ctx.getAuthenticatedUserName(); //this is a constant
    String itemName = request.getParameter("itemName");
    cs = conn.prepareCall("{call sp_queryItem(?,?)}");
    cs.setString(1, userName);
    cs.setString(2, itemName);
    results = cs.executeQuery();
    // ... result set handling
}
catch (SQLException se)
{
    // ... logging and error handling
}
```

SQL Server 存储过程：

```
CREATE PROCEDURE sp_queryItem
    @userName varchar(50),
    @itemName varchar(50)
AS
BEGIN
    DECLARE @sql nvarchar(500);
    SET @sql = 'SELECT * FROM t_item
               WHERE owner = ''' + @userName + '''
               AND itemName = ''' + @itemName + '''';
    EXEC(@sql);
END
GO
```

在存储过程中，通过拼接参数值来构建查询字符串，和在应用程序代码中拼接参数一样，同样是有 SQL 注入风险的。

正确示例（在存储过程中进行参数化查询）：

Java 代码：

```
CallableStatement = null
ResultSet results = null;
try
```

```
{
    String userName = ctx.getAuthenticatedUserName(); //this is a constant
    String itemName = request.getParameter("itemName");
    // ... Ensure that the length of userName and itemName is legitimate
    // ...
    cs = conn.prepareCall("{call sp_queryItem(?,?)}");
    cs.setString(1, userName);
    cs.setString(2, itemName);
    results = cs.executeQuery();
    // ... result set handling
}
catch (SQLException se)
{
    // ... logging and error handling
}
```

SQL Server 存储过程：

```
CREATE PROCEDURE sp_queryItem
    @userName varchar(50),
    @itemName varchar(50)
AS
BEGIN
    SELECT * FROM t_item
    WHERE userName = @userName
    AND itemName = @itemName;
END
GO
```

存储过程使用参数化查询，而不包含不安全的动态 SQL 构建。数据库编译此存储过程时，会生成一个 SELECT 查询的执行计划，只允许原始的 SQL 语义被执行，任何参数值，即使是被注入的 SQL 语句也不会被执行。

错误示例（Hibernate：动态构建 SQL/HQL）：

原生 SQL 查询：

```
String userName = ctx.getAuthenticatedUserName(); //this is a constant
String itemName = request.getParameter("itemName");
Query sqlQuery = session.createSQLQuery("select * from t_item where owner = '"
+ userName + "' and itemName = '" + itemName + "'");
List<Item> rs = (List<Item>) sqlQuery.list();
```

HQL 查询：

```
String userName = ctx.getAuthenticatedUserName(); //this is a constant
String itemName = request.getParameter("itemName");
Query hqlQuery = session.createQuery("from Item as item where item.owner = '"
+ userName + "' and item.itemName = '" + itemName + "'");
List<Item> hrs = (List<Item>) hqlQuery.list();
```

即使是使用 Hibernate，如果在动态构建 SQL/HQL 查询时依然使用拼接的方式，同样也会面临 SQL/HQL 注入的问题。

正确示例（Hibernate：参数化查询）：

HQL 中基于位置的参数化查询：

```
String userName = ctx.getAuthenticatedUserName(); //this is a constant
String itemName = request.getParameter("itemName");
Query hqlQuery = session.createQuery("from Item as item where item.owner = ? and
item.itemName = ?");
hqlQuery.setString(1, userName);
hqlQuery.setString(2, itemName);
List<Item> rs = (List<Item>) hqlQuery.list();
```

HQL 中基于名称的参数化查询：

```
String userName = ctx.getAuthenticatedUserName(); //this is a constant
String itemName = request.getParameter("itemName");
Query hqlQuery = session.createQuery("from Item as item where item.owner = :owner
and item.itemName = :itemName");
hqlQuery.setString("owner", userName);
hqlQuery.setString("itemName", itemName);
```

```
List<Item> rs = (List<Item>) hqlQuery.list();
```

原生参数化查询：

```
String userName = ctx.getAuthenticatedUserName(); //this is a constant
String itemName = request.getParameter("itemName");
Query sqlQuery = session.createSQLQuery("select * from t_item where owner = ?
and itemName = ?");
sqlQuery.setString(0, owner);
sqlQuery.setString(1, itemName);
List<Item> rs = (List<Item>) sqlQuery.list();
```

Hibernate 支持 SQL/HQL 参数化查询。上述示例使用了参数化绑定方式不仅能防止 SQL 注入，同时也能提高查询速度。

正反示例（iBATIS SQL 映射）：

iBATIS SQL 映射允许在 SQL 语句中通过#字符指定动态参数，例如：

```
<select id="getItems" parameterClass="MyClass" resultClass="Item">
    SELECT * FROM t_item WHERE owner = #userName# AND itemName = #itemName#
</select>
```

#符号括起来的 userName 和 itemName 两个参数指示 iBATIS 在创建参数化查询时将它们替换成占位符：

```
String sqlString = "SELECT * FROM t_item WHERE owner=? AND itemName=?";
PreparedStatement stmt = connection.prepareStatement(sqlString);
stmt.setString(1, myClassObj.getUserName());
stmt.setString(2, myClassObj.getItemName());
ResultSet rs = stmt.executeQuery();
// ... convert results set to Item objects
```

然而，iBATIS 也允许使用$符号指示使用某个参数来直接拼接 SQL 语句，这种做法是有 SQL 注入漏洞的：

```
<select id="getItems" parameterClass="MyClass" resultClass="items">
    SELECT * FROM t_item WHERE owner = #userName# AND itemName = '$itemName$'
</select>
```

iBATIS 将会为以上 SQL 映射执行类似下面的代码：

```
String sqlString = "SELECT * FROM t_item WHERE owner=? AND itemName='" +
myClassObj.getItemName() + "'";
PreparedStatement stmt = connection.prepareStatement(sqlString);
stmt.setString(1, myClassObj.getUserName());
ResultSet rs = stmt.executeQuery();
// ... convert results set to Item objects
```

在这里，攻击者可以利用 itemName 参数发起 SQL 注入攻击。

正确示例（对不可信输入做校验）：

```
public List<Book> queryBooks(List<Expression> queryCondition)
{
    /* ... */
    try
    {
        StringBuilder sb = new StringBuilder("select * from t_book where ");
        Codec oe = new OracleCodec();
        if (queryCondition != null && !queryCondition.isEmpty())
        {
            for (Expression e : queryCondition)
            {
                String exprString = e.getColumn() + e.getOperator() + e.getValue();
                String safeExpr = ESAPI.encoder().encodeForSQL(oe, exprString);
                sb.append(safeExpr).append(" and ");
            }
            sb.append("1=1");
            Statement stat = connection.createStatement();
            ResultSet rs = stat.executeQuery(sb.toString());
            //other omitted code
        }
    }
    /* ... */
```

```
}
```

虽然参数化查询是防止 SQL 注入最便捷有效的一种方式，但不是 SQL 语句中任何部分在执行前都能够被占位符所替代，因此，参数化查询无法应用于所有场景（有关 SQL 语句执行前可被占位符替代的元素，请参考相应数据库的帮助文档）。当使用执行前不可被占位符替代的不可信数据来动态构建 SQL 语句时，必须对不可信数据进行校验。每种 DBMS 都有其特定的转义机制，通过这种机制来告诉数据库此输入应该被当作数据，而不应该是代码逻辑。因此，只要输入数据被适当转义，就不会发生 SQL 注入问题。对于一些常用数据库中需要注意的特殊字符，请参考附录 A。此示例代码使用了 ESAPI 来做转义。ESAPI （OWASP 企业安全应用程序接口）是一个免费、开源的、网页应用程序安全控件库，程序员使用它能够更简单方便得写出低风险的程序。ESAPI 为大部分常用的数据库提供了安全的转义实现，推荐大家使用。

注：如果传入的是字段名或者表名，建议使用白名单的方式进行校验。

规则 1.3 禁止直接使用不可信数据来拼接 XML

说明：使用未经校验数据来构造 XML 会导致 XML 注入漏洞。如果用户被允许输入结构化的 XML 片段，则他可以在 XML 的数据域中注入 XML 标签来改写目标 XML 文档的结构和内容，XML 解析器会对注入的标签进行识别和解释，引起注入问题。

错误示例（未检查的输入）：

```
private void createXMLStream(BufferedOutputStream outStream, User user) throws
IOException
{
    String xmlString;
    xmlString = "<user><role>operator</role><id>" + user.getUserId()
                + "</id><description>" + user.getDescription() +
"</description></user>";
    outStream.write(xmlString.getBytes());
    outStream.flush();
}
```

恶意用户可能会使用下面的字符串作为用户 ID：

```
"joe</id><role>administrator</role><id>joe"
```

并使用如下正常的输入作为描述字段：

```
"I want to be an administrator"
```

最终，整个 XML 字符串将变成如下形式：

```
<user>
    <role>operator</role>
    <id>joe</id>
    <role>administrator</role>
    <id>joe</id>
    <description>I want to be an administrator</description>
</user>
```

由于 SAX 解析器（org.xml.sax and javax.xml.parsers.SAXParser）在解释 XML 文档时会将第二个 role 域的值覆盖前一个 role 域的值，因此会导致此用户角色由操作员提升为了管理员。

错误示例（XML Schema 或者 DTD 校验 ）：

```
private void createXMLStream(BufferedOutputStream outStream, User user)
          throws IOException
{
    String xmlString;
    xmlString = "<user><id>" + user.getUserId()
          + "</id><role>operator</role><description>"
```

```
            + user.getDescription() + "</description></user>";

    StreamSource xmlStream = new StreamSource(new StringReader(xmlString));

    // Build a validating SAX parser using the schema
    SchemaFactory sf =
SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
    StreamSource ss = new StreamSource(new File("schema.xsd"));
    try
    {
        Schema schema = sf.newSchema(ss);
        Validator validator = schema.newValidator();
        validator.validate(xmlStream);
    }
    catch (SAXException x)
    {
        throw new IOException("Invalid userId", x);
    }
    // the XML is valid, proceed
    outStream.write(xmlString.getBytes());
    outStream.flush();
}
```

如下是 schema.xsd 文件中的 schema 定义：

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="user">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="id" type="xs:string"/>
      <xs:element name="role" type="xs:string"/>
      <xs:element name="description" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```

某个恶意用户可能会使用下面的字符串作为用户 ID：

```
"joe</id><role>Administrator</role><!—"
```

并使用如下字符串作为描述字段：

```
"--><description>I want to be an administrator"
```

最终，整个 XML 字符串将变成如下形式：

```
<user>
    <id>joe</id>
    <role>Administrator</role><!--</id> <role>operator</role> <description>
-->
    <description>I want to be an administrator</description>
</user>
```

用户 ID 结尾处的“<!--”和描述字段开头处的“-->”将会注释掉原本硬编码在 XML 字符串中的角色信息。虽然用户角色已经被攻击者篡改成管理员类型，但是整个 XML 字符串仍然可以通过 schema 的校验。XML schema 或者 DTD 校验仅能确保 XML 的格式是有效的，但是攻击者可以在不打破原有 XML 格式的情况下，对 XML 的内容进行篡改。

正确示例（白名单校验）：

```
private void createXMLStream(BufferedOutputStream outStream, User user) throws
IOException
{
    // Write XML string if userID contains alphanumeric and underscore characters
only
    if (!Pattern.matches("[_a-bA-B0-9]+", user.getUserId()))
    {
        // Handle format violation
    }
    if (!Pattern.matches("[_a-bA-B0-9]+", user.getDescription()))
    {
```

```
        // Handle format violation
    }
    String xmlString = "<user><id>" + user.getUserId()
            + "</id><role>operator</role><description>"
            + user.getDescription() + "</description></user>";
    outStream.write(xmlString.getBytes());
    outStream.flush();
}
```

这个方法使用白名单的方式对输入进行校验，要求输入的 userId 字段中只能包含字母、数字或者下划线。

正确示例（使用安全的 XML 库）：

```
public static void buidlXML(FileWriter writer, User user) throws IOException
{
    Document userDoc = DocumentHelper.createDocument();
    Element userElem = userDoc.addElement("user");
    Element idElem = userElem.addElement("id");
    idElem.setText(user.getUserId());
    Element roleElem = userElem.addElement("role");
    roleElem.setText("operator");
    Element descrElem = userElem.addElement("description");
    descrElem.setText(user.getDescription());
    XMLWriter output  = null;
    try
    {
        OutputFormat format = OutputFormat.createPrettyPrint();
        format.setEncoding("UTF-8");
        output = new XMLWriter(writer, format);
        output.write(userDoc);
        output.flush();
    }
    finally
    {
       try
       {
           output.close();
       }
       catch (Exception e)
       {
          // handle exception
       }
    }
}
```

正确示例中使用 dom4j 来构建 XML，dom4j 是一个定义良好、开源的 XML 工具库。Dom4j 将会对文本数据域进行 XML 编码，从而使得 XML 的原始结构和格式免受破坏。

这个例子中，攻击者如果输入如下字符串作为用户 ID：

"joe</id><role>Administrator</role><!—"

以及使用如下字符串作为描述字段：

"--><description>I want to be an administrator"

则最终会生成如下格式的 XML：

```
<user>
  <id>joe&lt;/id&gt;&lt;role&gt;Administrator&lt;/role&gt;&lt;!—</id>
  <role>operator</role>
  <description>--&gt;&lt;description&gt;I want to be an
administrator</description>
</user>
```

可以看到，“<”与“>”经过 XML 编码后分别被替换成了“&lt”与“&gt”，导致攻击者未能将其角色类型从操作员提升到管理员。

规则 1.4 禁止直接使用不可信数据来记录日志

说明：如果在记录的日志中包含未经校验的不可信数据，则可能导致日志注入漏洞。恶意用户会插入伪造的日志数据，从而让系统管理员误以为这些日志数据是由系统记录的。例如，用户可能通过输入一个回车符或一个换行符（CRLF）来将一条合法日志拆分成两条日志，使得日志内容可能令人误解。将未经净化的用户输入写入日志可能会导致敏感数据泄露，若在日志中写入和存储了某些类型的敏感数据甚至可能违反当地法律法规。例如，如果用户要把一个未经加密的信用卡号插入到日志文件中，那么系统就会违反了 PCI DSS（Payment Card Industry Data Security Standard）标准。

可以对记录到日志中的任何不可信数据进行校验和净化来防止日志注入攻击。

错误示例：

```
if (loginSuccessful)
{
    logger.severe("User login succeeded for: " + username);
}
else
{
    logger.severe("User login failed for: " + username);
}
```

此错误示例代码中，在接收到非法请求时，会记录用户的用户名，由于没有执行任何输入净化，这种情况下就可能会遭受日志注入攻击：

当 username 字段的值是 david 时，会生成一条标准的日志信息：

```
May 15, 2011 2:19:10 PM java.util.logging.LogManager$RootLogger log
SEVERE: User login failed for: david
```

但是，如果记录日志时使用的 username 字段是多行字符串时，如下所示：

```
david
May 15, 2011 2:25:52 PM java.util.logging.LogManager$RootLogger log
SEVERE: User login succeeded for: administrator
```

那么日志中包含了以下可能引起误导的信息：

```
May 15, 2011 2:19:10 PM java.util.logging.LogManager$RootLogger log
SEVERE: User login failed for: david
May 15, 2011 2:25:52 PM java.util.logging.LogManager log
SEVERE: User login succeeded for: administrator
```

正确示例：

```
if (!Pattern.matches("[A-Za-z0-9_]+", username))
{
    // Unsanitized username
    logger.severe("User login failed for unauthorized user");
}
else if (loginSuccessful)
{
    logger.severe("User login succeeded for: " + username);
}
else
{
    logger.severe("User login failed for: " + username);
```

```
}
```

这个正确示例在登录之前会对用户名输入进行净化，从而防止注入攻击。

在通用情况下，如果只需消除换行符和回车符，那么可以在记录日志前调用如下函数即可：

```
public String getcleanedMessage(String message){
        if (message == null) {
            return "";
        }
        message = message.replace('\n', '_').replace('\r', '_');
        return message;
    }
```

规则 1.5 从格式化字符串中排除用户输入

说明：当转换参数与对应的格式符不匹配时，标准类库会抛出异常，这种方式可以减少被恶意攻击的机会。但如果代码中使用来源不可信的数据来构造格式化字符串时，仍有可能被攻击者利用，造成系统信息泄露或者拒绝服务。因此，不能直接将来自不可信源的字符串用于构造格式化字符串。

错误示例：

```
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] may 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);
    }
}
```

这个错误示例展示了一个信息泄露的问题。它将信用卡的失效日期作为输入参数并将其用在格式字符串中。如果没有经过正确的输入校验，攻击者可以通过提供一段包含%1$tm、%1$te和%1$tY之一的输入，来识别出程序中用来和输入做对比验证的日期。

正确示例：

```
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 %2$terd of some month", args[0], c);
    }
}
```

该正确示例将用户输入排除在格式化字符串之外。

规则 1.6 禁止向 Runtime.exec() 方法传递不可信、未净化的数据

说明：使用了未经校验的不可信输入来执行系统命令或者外部程序时，就可能会导致命令和参数注入。对于命令注入漏洞，命令将会以与 Java 应用程序相同的权限执行，它向攻击者提供了类似系统 shell 的功能。在 Java 中，`Runtime.exec()`经常被用来调用一个新的进程，但是这个调用并不会通过命令行 Shell 来执行。因此，无法通过链接或者管道的方式来连续执行多个命令。但是，如果在 `Runtime.exec()`中使用命令行 shell（例如，*cmd.exe* 或 */bin/sh*）来调用一个程序，则可产生命令注入。当通过 `Runtime.exec()`来运行 bat 或者 sh 脚本时，命令行 shell 将自动被调用。例如，`Runtime.getRuntime().exec("test.bat & notepad.exe")`，由于 bat 文件默认是由命令行解释器 cmd.exe 来解释执行的，这里的“&”符号将会被 cmd.exe 当做一个命令分隔符，从而导致 `test.bat` 与 `notepad.exe` 都将会被执行。当参数中包含空格，双引号，以-或者/符号开头表示一个参数开关时，可能会导致参数注入漏洞。与命令和参数注入相关的特殊符号的详细信息，请参考附录 B

错误示例：

```
class DirList
{
   public static void main(String[] args)
   {
       if (args.length == 0)
       {
           System.out.println("No arguments");
           System.exit(1);
       }
       try
       {
           Runtime rt = Runtime.getRuntime();
           Process proc = rt.exec("cmd.exe /c dir " + args[0]);
           // ...
       }
       catch (Exception e)
       {
           // Handle errors
       }
   }
}
```

攻击者可以通过以下命令来利用这个漏洞程序：

```
java DirList "dummy & echo bad"
```

实际将会执行两个命令：

```
dir dummy
echo bad
```

这会试图列举一个不存的文件夹 dummy 中的文件，然后往控制台输出 bad。

正确示例（避免使用 Runtime.exec()）：

```
class DirList
{
   public static void main(String[] args)
   {
       if (args.length == 0)
       {
           System.out.println("No arguments");
           System.exit(1);
       }
       try
```

```
        {
            File dir = new File(args[0]);
            if (!validate(dir)) // the dir need to be validated
            {
                System.out.println("An illegal directory");
            }
            else
            {
                for (String file : dir.list())
                {
                    System.out.println(file);
                }
            }
        }
        catch (Exception e)
        {
            System.out.println("An unexpected exception");
        }
    }
    // Other omitted code…
}
```

如果可以使用标准的 API 替代运行系统命令来完成任务，则应该使用标准的 API。正确示例使用 File.list()方法来列举目录下的内容，而不是调用 Runtime.exec()来运行一个外部进程，从而消除了发生命令注入与参数注入的可能。

正确示例（数据校验）：

```
// ...
if (!Pattern.matches("[0-9A-Za-z@]+", dir))
{
    // Handle error
}
// ...
```

如果无法避免使用 Runtime.exec()，则必须要对输入数据进行检查和净化，防止其中包含命令分隔符，管道，或者重定向操作符（“&&”，“&”，“||”，“|”，“>”，“>>”）等，以及参数开关符（“-”，“/”）。

规则 1.7 验证路径之前应该先将其标准化

说明：绝对路径或者相对路径中可能会包含如符号（软）链接（symbolic [soft] links）、硬链接（hard links）、快捷方式（shortcuts）、影子文件（shadows）、别名（aliases）和连接文件（junctions）等形式，在进行文件验证操作之前必须完整解析这些文件链接。路径中也可能会包含如下所示的文件名，使得验证变得困难：

“.”指目录本身。

在一个目录内，“..”指该目录的上一级目录。

除此之外，还有与特定操作系统和特定文件系统相关的命名约定，也会使验证变得困难。

同一个目录或者文件，可以通过多种路径名来引用它，对其进行标准化，可以使文件路径验证较为容易。

当试图限制用户只能访问某个特定目录中的文件时，或者当基于文件名或者路径名来做安全决策时，验证是必须的。攻击者可能会利用目录遍历（directory traversal）或者等价路径（path equivalence）漏洞的方式来绕过这些限制。目录遍历漏洞使得攻击者能够转移到一个特定目录进行 I/O 操作，等价路径漏洞使得攻击者可以使用与某个资源名不同但是等价的名称来绕过安全检查。

在程序获取一个文件标准路径与打开这个文件之间，会有一个固有的时间竞争窗口。在对标

准化的文件路径进行验证时，文件系统可能被修改，标准化的路径名可能不再指向原始的有效文件。值得庆幸的是，可以使用标准化的路径名来判断引用的文件名是否在安全目录中，来消除条件竞争（更多信息请参考规则 3.5 避免在共享目录操作文件）。如果引用的文件是在一个安全目录之中，很明显攻击者无法篡改文件，也无法利用条件竞争。

错误示例：

```
public static void main(String[] args)
{
       File f = new File(System.getProperty("user.home")
              + System.getProperty("file.separator") + args[0]);
       String absPath = f.getAbsolutePath();
       if (!isInSecureDir(Paths.get(absPath)))
       {
          // Refer to Rule 3.5 for the details of isInSecureDir()
          throw new IllegalArgumentException();
       }
       if (!validate(absPath))
       {
          // Validation
          throw new IllegalArgumentException();
       }
       /* … */
}
```

File.getAbsolutePath()返回文件的绝对路径，但是它不会解析文件链接，也不会消除等价错误。

注意，在 Windows 和 Macintosh 平台中，File.getAbsolutePath()方法确实可以解析符号链接、别名和快捷方式。但是在别的平台上却不能保证这样的行为都有效，或者在未来的实现中均会这样做。

正确示例（getCanonicalPath()）：

```
public static void main(String[] args) throws IOException
{
   File f = new File(System.getProperty("user.home")
             + System.getProperty("file.separator") + args[0]);
   String canonicalPath = f.getCanonicalPath();
   if (!isInSecureDir(Paths.get(absPath)))
   {
      // Refer to Rule 3.5 for the details of isInSecureDir()
      throw new IllegalArgumentException();
   }
   if (!validate(absPath))
   {
      // Validation
      throw new IllegalArgumentException();
   }
   /* ... */
}
```

这个正确示例使用了 File.getCanonicalPath()方法，它能在所有的平台上对所有别名、快捷方式以及符号链接进行一致地解析。特殊的文件名，比如“..”会被移除，这样输入在验证之前会被简化成对应的标准形式。当使用标准形式的文件路径来做验证时，攻击者将无法使用../序列来跳出指定目录。

规则 1.8 安全地从 ZipInputStream 提取文件

说明：从 java.util.zip.ZipInputStream 中解压文件时需要小心谨慎。有两个特别的问题需要避免：一个是解压出的标准化路径文件在解压目标目录之外，另一个是解压的文

件消耗过多的系统资源。前一种情况，攻击者可以从 zip 文件中往用户可访问的任何目录写入任意的数据。后一种情况，在与输入数据所使用资源相比严重不成比例时，就可能产生拒绝服务。由于 Zip 算法有极高的压缩率，即使在解压如 ZIP、GIF、gzip 编码 HTTP 的小文件时，也可能会导致过度的资源消耗，导致 zip 炸弹（zip bomb）。

Zip 算法有非常高的压缩比。例如，一个由字符 a 和字符 b 交替出现的行构成的文件，压缩比可以达到 200：1。使用针对目标压缩算法的输入数据，或者使用更多的输入数据（不针对目标压缩算法的），或者使用其他的压缩方法，甚至可以达到更高的压缩比。

任何被提取条目的目标路径不在程序预期目录之内时（必须先对文件名进行标准化，参照规则 1.7 验证路径之前应该先将其标准化），要么拒绝将其提取出来，要么将其提取到一个安全的位置。Zip 文件中任何被提取条目，若解压之后的文件大小超过一定的限制时，必须拒绝将其解压。具体大小限制由平台的处理性能来决定。

错误示例：

```
static final int BUFFER = 512;
// ...
public final void unzip(String fileName) throws java.io.IOException
{
    FileInputStream fis = new FileInputStream(fileName);
    ZipInputStream zis = new ZipInputStream(new BufferedInputStream(fis));
    ZipEntry entry;
    while ((entry = zis.getNextEntry()) != null)
    {
        System.out.println("Extracting: " + entry);
        int count;
        byte data[] = new byte[BUFFER];
        // Write the files to the disk
        FileOutputStream fos = new FileOutputStream(entry.getName());
        BufferedOutputStream dest = new BufferedOutputStream(fos, BUFFER);
        while ((count = zis.read(data, 0, BUFFER)) != -1)
        {
                dest.write(data, 0, count);
        }
        dest.flush();
        dest.close();
        zis.closeEntry();
     }
     zis.close();
}
```

错误示例中，未对解压的文件名做验证，直接将文件名传递给 FileOutputStream 构造器。它也未检查解压文件的资源消耗情况，允许程序运行到操作完成或者本地资源被耗尽。

错误示例（getSize()）：

```
public static final int BUFFER = 512;
public static final int TOOBIG = 0x6400000; // 100MB
// ...
public final void unzip(String filename) throws java.io.IOException
{
    FileInputStream fis = new FileInputStream(filename);
    ZipInputStream zis = new ZipInputStream(new BufferedInputStream(fis));
    ZipEntry entry;
    try
    {
        while ((entry = zis.getNextEntry()) != null)
        {
            System.out.println("Extracting: " + entry);
            int count;
            byte data[] = new byte[BUFFER];
            // Write the files to the disk, but only if the file is not insanely
big
            if (entry.getSize() > TOOBIG)
```

```
            {
                throw new IllegalStateException(
                      "File to be unzipped is huge.");
            }
            if (entry.getSize() == -1)
            {
                throw new IllegalStateException(
                      "File to be unzipped might be huge.");
            }
            FileOutputStream fos = new FileOutputStream(entry.getName());
            BufferedOutputStream dest = new BufferedOutputStream(fos,
                  BUFFER);
            while ((count = zis.read(data, 0, BUFFER)) != -1)
            {
                dest.write(data, 0, count);
            }
            dest.flush();
            dest.close();
            zis.closeEntry();
        }
    }
    finally
    {
        zis.close();
    }
}
```

错误示例在解压条目之前调用 ZipEntry.getSize()方法判断其大小，以试图解决之前的问题。但不幸的是，恶意攻击者可以伪造 ZIP 文件中用来描述解压条目大小的字段，因此，getSize()方法的返回值是不可靠的，本地资源实际仍可能被过度消耗。

正确示例：

```
static final int BUFFER = 512;
static final int TOOBIG = 0x6400000; // max size of unzipped data, 100MB
static final int TOOMANY = 1024; // max number of files
// ...
private String sanitzeFileName(String entryName, String intendedDir) throws
IOException
{
    File f = new File(intendedDir, entryName);
    String canonicalPath = f.getCanonicalPath();

    File iD = new File(intendedDir);
    String canonicalID = iD.getCanonicalPath();

    if (canonicalPath.startsWith(canonicalID))
    {
        return canonicalPath;
    }
    else
    {
        throw new IllegalStateException(
              "File is outside extraction target directory.");
    }
}
// ...
public final void unzip(String fileName) throws java.io.IOException
{
    FileInputStream fis = new FileInputStream(fileName);
    ZipInputStream zis = new ZipInputStream(new BufferedInputStream(fis));
    ZipEntry entry;
    int entries = 0;
    int total = 0;
    byte[] data = new byte[BUFFER];
    try
    {
        while ((entry = zis.getNextEntry()) != null)
```

```
        {
            System.out.println("Extracting: " + entry);
            int count;
            // Write the files to the disk, but ensure that the entryName is valid,
            // and that the file is not insanely big
            String name = sanitzeFileName(entry.getName(), ".");
            FileOutputStream fos = new FileOutputStream(name);
            BufferedOutputStream dest = new BufferedOutputStream(fos, BUFFER);
            while (total + BUFFER <= TOOBIG && (count = zis.read(data, 0, BUFFER)) !=
-1)
            {
                dest.write(data, 0, count);
                total += count;
            }
            dest.flush();
            dest.close();
            zis.closeEntry();
            entries++;
            if (entries > TOOMANY)
            {
                throw new IllegalStateException("Too many files to unzip.");
            }
            if (total > TOOBIG)
            {
                throw new IllegalStateException(
                        "File being unzipped is too big.");
            }
        }
    }
    finally
    {
        zis.close();
    }
}
```

正确示例中，代码会在解压每个条目之前对其文件名进行校验。如果某个条目校验失败，整个解压过程都将会被终止。实际上也可以忽略跳过这个条目，继续后面的解压过程，甚至也可以将这个条目解压到某个安全位置。除了校验文件名，while 循环中的代码会检查从 zip 存档文件中解压出来的每个文件条目的大小。如果一个文件条目太大，此例中是 100MB，则会抛出异常。最后，代码会计算从存档文件中解压出来的文件条目总数，如果超过 1024 个，则会抛出异常。

3. 异常行为

规则 2.1 不要抑制或者忽略已检查异常

说明：编码人员常常会通过一个空的或者无意义的 catch 块来抑制捕获的已检查异常。每一个 catch 块都应该确保程序只会在继续有效的情况下才会继续运行下去。因此，catch 块必须要么从异常情况中恢复，要么重新抛出适合当前 catch 块上下文的另一个异常以允许最邻近的外层 try-catch 语句块来进行恢复工作。异常会打断应用原本预期的控制流程。例如，try 块中位于异常发生点之后的任何表达式和语句都不会被执行。因此，异常必须被妥当处理。许多抑制异常的理由都是不合理的。例如，当对客户端从潜在问题恢复过来不抱期望时，一种好的做法是让异常被广播出来，而不是去捕获和抑制这个异常。

错误示例：

```
//...
try
{
    //...
}
catch (IOException ioe)
{
    ioe.printStackTrace();
}
//...
```

错误示例中，catch 块只是简单地将异常堆栈轨迹打印出来。虽然打印异常的堆栈轨迹对于定位问题是有帮助的，但是最终的程序运行逻辑等同于抑制异常。注意，即使这个错误示例在发生异常时会打印一个堆栈轨迹，但是程序会继续运行，如同异常从未被抛出过。换句话说，除了 try 块中位于异常发生点之后的表达式和语句不会被执行之外，发生的异常不会影响程序的其他行为。

正确示例（交互）：

```
// ...
volatile boolean validFlag = false;
do
{
    try
    {
        // If requested file does not exist, throws FileNotFoundException
        // If requested file exists, sets validFlag to true
        validFlag = true;
    }
    catch (FileNotFoundException e)
    {
        // Ask the user for a different file name
    }
} while (validFlag != true);
// Use the file
```

正确示例通过要求用户指定另外一个文件名来处理 FileNotFoundException 异常。为了遵循规则 2.2 禁止在异常中泄露敏感信息，用户只允许访问该用户特定的目录中的文件。这可以防止往循环外抛出其他 IOException 异常从而导致泄露文件系统中的敏感信息。

正确示例（Exception Reporter）：

```
public interface Reporter
{
    public void report(Throwable t);
```

```
}

public class ExceptionReporter
{
    // Exception reporter that prints the exception
    // to the console (used as default)
    private static final Reporter printException = new Reporter()
    {
        public void report(Throwable t)
        {
            System.err.println(t.toString());
        }
    };

    // Stores the default reporter.
    // The default reporter can be changed by the user.
    private static Reporter default = printException;

    // Helps change the default reporter back to
    // PrintException in the future
    public static Reporter getPrintException()
    {
        return printException;
    }

    public static Reporter getExceptionReporter()
    {
        return default;
    }

    // May throw a SecurityException (which is unchecked)
    public static void setExceptionReporter(Reporter reporter)
    {
        // Custom permission
        ExceptionReporterPermission perm = new ExceptionReporterPermission(
                "exc.reporter");
        SecurityManager sm = System.getSecurityManager();
        if (sm != null)
        {
            // Check whether the caller has appropriate permissions
            sm.checkPermission(perm);
        }
        // Change the default exception reporter
        default = reporter;
    }
}
```

如何恰当的报告异常情况依赖与具体的上下文。例如，对于 GUI 应用应该以图形界面的方式报告异常，例如弹出一个错误对话框。对于大部分库中的类应该能够客观的决定该如何报告一个异常来保持模块化；它们不能依赖于 System.err、特定的 logger，或者 windowing 环境的可用性。因此，对于报告异常的库类，应该指定一个用来报告此类异常的 API。正确示例中，指定了一个包含 report()方法的用来报告异常的接口，以及一个默认的异常 reporter 类供库类使用。可以定义子类来覆盖这个异常 reporter 类。

库中的类后续便可以在 catch 子句中使用这个异常 reptorter 类：

```
try
{
    // ...
}
catch (IOException warning)
{
    ExceptionReporter.getExceptionReporter().report(warning);
    // Recover from the exception...
}
```

任何使用这个异常 reporter 类的客户代码，只要具备所需的权限许可，就能够覆写这个

ExceptionReporter，使用一个 logger 或者提供一个对话框来报告异常。例如，一个使用 Swing 的 GUI 客户代码，要求使用对话框来报告异常：

```
ExceptionReporter.setExceptionReporter(new ExceptionReporter()
{
    public void report(Throwable exception)
    {
        JOptionPane.showMessageDialog(frame,
              exception.toString, exception.getClass().getName(),
                  JOptionPane.ERROR_MESSAGE);
    }
});
```

正确示例（继承 Exception Reporter 并过滤敏感异常）：

```
class MyExceptionReporter extends ExceptionReporter
{
    public static void report(Throwable t)
    {
        t = filter(t);
        // Do any necessary user reporting (show dialog box or send to console)
    }

    public static Exception filter(Throwable t)
    {
        // Sanitize sensitive data or replace sensitive exceptions with
non-sensitive exceptions (whitelist)
        // Return non-sensitive exception
    }
}
```

出于安全原因（参考规则 2.2 禁止在异常中泄露敏感信息），有时候必须对用户隐藏异常。在这种情况下，一种可行的方式是继承 ExceptionReporter 类，并且在重写默认 report()方法的基础上，增加一个 filter()方法。

例外情况：

1) 在资源释放失败不会影响程序后续行为的情况下，释放资源时发生的异常可以被抑制。释放资源的例子包括关闭文件、网络套接字、线程等等。这些资源通常是在finally块内的try-catch块中被释放，并且在后续的程序运行中都不会再被使用。因此，除非资源被耗尽，否则不会有其他途径使得这些异常会影响程序后续的行为。在充分处理了资源耗尽问题的情况下，只需对异常进行净化和记录日志（以备日后改进）就足够了；在这种情况下没必要做其他额外的错误处理。
2) 如果在特定的抽象层次上不可能从异常情况中恢复过来，则在那个层级的代码就不用处理这个异常，而是应该抛出一个合适的异常，让更高层次的代码去捕获处理，并尝试恢复。对于这种情况，最通常的实现方法是省略掉catch语句块，允许异常被广播出去。

规则 2.2 禁止在异常中泄露敏感信息

说明：敏感数据的范围应该基于应用场景以及产品威胁分析的结果来确定。典型的敏感数据包括口令、银行账号、个人信息、通讯记录、密钥等。如果在传递异常的时候未对其中的敏感信息进行过滤常常会导致信息泄露，而这可能帮助攻击者尝试发起进一步的攻击。攻击者可以通过构造恶意的输入参数来发掘应用的内部结构和机制。不管是异常中的文本消息，还是异常本身的类型都可能泄露敏感信息。例如 FileNotFoundException 异常会透露文件系统的结构信息，而通过异常本身的类型，可以得知所请求的文件不存在。因此，当异常会被传递到信任边界以外时，必须同时对敏感的异常消息和敏感的异常类型进行过滤。附录 C 列出了一些常见的需要注意的异常类型。

错误示例（异常消息和类型泄露敏感信息）：

```
public class ExceptionExample
{
    public static void main(String[] args) throws FileNotFoundException
    {
        // Linux stores a user's home directory path in
        // the environment variable $HOME, Windows in %APPDATA%
        // ... Other omitted code
        FileInputStream fis = new FileInputStream(System.getenv("APPDATA")
                + args[0]);
    }
}
```

此错误示例代码中，当请求的文件不存在时，FileInputStream 的构造器会抛出 FileNotFoundException 异常。这使得攻击者可以不断传入伪造的路径名称来重现出底层文件系统结构。

错误示例（封装后重新抛出敏感异常）：

```
try
{
    FileInputStream fis = new FileInputStream(System.getenv("APPDATA")
                    + args[0]);
}
catch (FileNotFoundException e)
{
    // Log the exception
    throw new IOException("Unable to retrieve file", e);
}
```

即使用户无法访问到日志中记录的异常，但是原始异常仍然会有大量有用信息，攻击者可以借此来发现文件系统结构信息。

错误示例 （异常净化）：

```
class SecurityIOException extends IOException
{
    /* ... */
};

// ...
try
{
    FileInputStream fis = new FileInputStream(System.getenv("APPDATA") +
args[0]);
}
catch (FileNotFoundException e)
{
    // Log the exception
    throw new SecurityIOException();
}
```

相比前面几个错误示例，此示例抛出的异常中泄露的有用信息较少，但是它仍然会透露出指定的文件不可读。程序对于有效的文件路径和不存在的文件路径会有不同的反应，攻击者可以根据程序的行为推断出有关文件系统的敏感信息。未对用户输入做限制，使得系统面临暴力攻击的风险，攻击者可以多次传入所有可能的文件名进行查询来发现有效文件。如果传入一个文件名后程序返回一个净化后的异常，则表明该文件不存在，而如果不抛出异常则说明该文件是存在的。

正确示例（安全策略）：

```
public class ExceptionExample
{
    public static void main(String[] args)
    {
        File file = null;
```

```
        try
        {
            file = new File(System.getenv("APPDATA") +
args[0]).getCanonicalFile();
            if (!file.getPath().startsWith("c:\\homepath"))
            {
                System.out.println("Invalid file");
                return;
            }
        }
        catch (IOException x)
        {
            System.out.println("Invalid file");
            return;
        }
        try
        {
            FileInputStream fis = new FileInputStream(file);
        }
        catch (FileNotFoundException x)
        {
            System.out.println("Invalid file");
            return;
        }
    }
}
```

正确示例中，规定用户只能打开 c:\homepath 目录下的文件，用户不可能发现这个目录以外的任何信息。在这个方案中，如果无法打开文件，或者文件不在合法的目录下，则会产生一条简洁的错误消息。任何 c:\homepath 目录以外的文件信息都被会隐蔽起来。

正确示例（限制输入）：

```
public class ExceptionExample
{
    public static void main(String[] args)
    {
        FileInputStream fis = null;
        try
        {
            switch (Integer.valueOf(args[0]))
            {
                case 1:
                    fis = new FileInputStream("c:\\homepath\\file1");
                    break;
                case 2:
                    fis = new FileInputStream("c:\\homepath\\file2");
                    break;
                // ...
                default:
                    System.out.println("Invalid option");
                    break;
            }
        }
        catch (Throwable t)
        {
            MyExceptionReporter.report(t); // Sanitize any sensitive data
        }
    }
}
```

这个正确示例限制用户只能打开 c:\homepath\file1 与 c:\homepath\file2。同时，它也会过滤在 catch 块中捕获的异常中的敏感信息。

规则 2.3 方法发生异常时要恢复到之前的对象状态

说明：当发生异常的时候，对象一般需要（关键的安全对象则必须）维持其状态的一致性。常用的可用来维持对象状态一致性的手段包括：

- 输入校验（如校验方法的调用参数）
- 调整逻辑顺序，使可能发生异常的代码在对象被修改之前执行
- 当业务操作失败时，进行回滚
- 对一个临时的副本对象进行必要的操作，直到成功完成这些操作后，才把更新提交到原始的对象
- 避免去修改对象状态

错误示例：

```
public class Dimensions
{
    private int length;
    private int width;
    private int height;

    public static final int PADDING = 2;

    public static final int MAX_DIMENSION = 10;

    public Dimensions(int length, int width, int height)
    {
        this.length = length;
        this.width = width;
        this.height = height;
    }

    public int getVolumePackage(int weight)
    {
        length += PADDING;
        width += PADDING;
        height += PADDING;
        try
        {
            validate(weight);
            int volume = length * width * height; // 12 * 12 * 12 = 1728
            // Revert
            length -= PADDING;
            width -= PADDING;
            height -= PADDING;
            return volume;
        }
        catch (Exception t)
        {
            MyExceptionReporter.report(t); // Sanitize any sensitive data
            return -1; // Non-positive error code
        }
    }

    private void validate(int weight) throws InvalidArgumentException
    {
        // do some validation and may throw a exception
    }

    public static void main(String[] args)
    {
        Dimensions d = new Dimensions(10, 10, 10);
        System.out.println(d.getVolumePackage(21)); // Prints -1 (error)
```

```
        System.out.println(d.getVolumePackage(19)); // Prints 2744 instead of
1728
    }
}
```

错误示例中，未有异常发生时，代码逻辑会恢复对象的原始状态。但是如果出现异常时，回滚代码不会被执行，从而导致后续调用 getVolumePackage()但不返回正确的结果。

正确示例（回滚）：

```
// ...
}
catch (Exception t)
{
    MyExceptionReporter.report(t); // Sanitize any sensitive data
    // Revert
    length -= PADDING;
    width -= PADDING;
    height -= PADDING;
    return -1;
}
```

正确示例中，getVolumePackage()方法的 catch 块会恢复对象状态。

正确示例（finally 子句）：

```
public int getVolumePackage(int weight)
{
    length += PADDING;
    width += PADDING;
    height += PADDING;
    try
    {
        validate(weight);
        int volume = length * width * height; // 12 * 12 * 12 = 1728
        return volume;
    }
    catch (Exception t)
    {
        MyExceptionReporter.report(t); // Sanitize any sensitive data
        return -1; // Non-positive error code
    }
    finally
    {
        length -= PADDING;
        width -= PADDING;
        height -= PADDING; // Revert
    }
}
```

正确示例使用一个 finally 子句来执行回滚操作，以保证不管是否发生异常，都会进行回滚。

正确示例（输入校验）：

```
public int getVolumePackage(int weight)
{
    try
    {
        validate(weight); // Validate first
    }
    catch (Exception t)
    {
        MyExceptionReporter.report(t); // Sanitize any sensitive data
        return -1;
    }

    length += PADDING;
    width += PADDING;
    height += PADDING;
```

```
    int volume = length * width * height;
    length -= PADDING;
    width -= PADDING;
    height -= PADDING;
    return volume;
}
```

正确示例在修改对象状态之前执行输入校验。注意，try 代码块中只包含可能会抛出异常的代码，而其他代码都被移到 try 块之外。

正确示例（未修改的对象）：

```
public int getVolumePackage(int weight)
{
    try
    {
        validate(weight);
    }
    catch (Exception t)
    {
        MyExceptionReporter.report(t); // Sanitize any sensitive data
        return -1;
     }

    int volume = (length + PADDING) * (width + PADDING) * (height + PADDING);
    return volume;
}
```

正确示例避免了修改对象状态，从而使得对象状态总是一致的。因此没有必要再进行回滚操作。相比之前的解决方案，更推荐使用这种方式。但是对于一些复杂的代码，这种方式可能无法实行。

4. I/O 操作

规则 3.1 临时文件使用完毕应及时删除

说明：程序员经常会在全局可写的目录中创建临时文件。例如，POSIX 系统下的/tmp 与/var/tmp 目录，Windows 系统下的C:\TEMP 目录。这类目录中的文件可能会被定期清理，例如，每天晚上或者重启时。然而，如果文件未被安全地创建或者用完后还是可访问的，具备本地文件系统访问权限的攻击者便可以利用共享目录中的文件进行恶意操作。删除已经不再需要的临时文件有助于对文件名和其他资源（如二级存储）进行回收利用。每一个程序在正常运行过程中都有责任确保已使用完毕的临时文件被删除。

注意：下面的示例代码已假设文件在创建时指定了合适的访问权限（遵循规则 3.3 在多用户系统中创建文件时指定合适的访问权限）以及被创建在安全目录中（遵循规则 3.5 避免在共享目录操作文件）。这两个要求都可以设法通过 JVM 以外的手段来满足。

错误示例：

```
public class TempFile
{
    public static void main(String[] args) throws IOException
    {
        File f = new File("tempnam.tmp");
        if (f.exists())
        {
            System.out.println("This file already exists");
            return;
        }
        FileOutputStream fop = null;
        try
        {
            fop = new FileOutputStream(f);
            String str = "Data";
            fop.write(str.getBytes());
        }
        finally
        {
            if (fop != null)
            {
                try
                {
                    fop.close();
                }
                catch (IOException x)
                {
                    // handle error
                }
            }
        }
    }
}
```

这个错误示例代码在运行结束时未将临时文件删除。

正确示例（JDK1.7: DELETE_ON_CLOSE）：

```
public class TempFile
{
    public static void main(String[] args)
    {
        Path tempFile = null;
        try
```

```
        {
            tempFile = Files.createTempFile("tempnam", ".tmp");
            try (BufferedWriter writer = Files.newBufferedWriter(tempFile,
                                         Charset.forName("UTF8"),
                                         StandardOpenOption.DELETE_ON_CLOSE))
            {
                // write to the file and use it
            }
            System.out.println("Temporary file write done, file erased");
        }

        catch (IOException x)
        {
            // Some other sort of failure, such as permissions.
            System.err.println("Error creating temporary file");
        }
    }
}
```

这个正确示例创建临时文件时用到了 JDK1.7 的 NIO2 包中的几个方法。它使用 createTempFile()方法新建一个随机的文件名（文件名的构造方式由具体的实现所定义，JDK 缺少相关的文档说明）。文件使用 try-with-resources 构造块来打开，这种方式将会自动关闭文件，而不管是否有异常发生，并且在打开文件时用到了 DELETE_ON_CLOSE 选项，使得文件在关闭时会被自动删除。但是这种方式只适合于 windows 系统，在 linux 系统中参数 StandardOpenOption.DELETE_ON_CLOSE 是无效的，所以建议采用下面的示例，即采用手动删除临时文件：

正确示例（手动删除临时文件）：

```
public class TempFile
{
    public static void main(String[] args) throws IOException
    {
        File f = File.createTempFile("tempnam", ".tmp");
        FileOutputStream fop = null;
        try
        {
            fop = new FileOutputStream(f);
            // write to the file and use it
        }
        finally
        {
            if (fop != null)
            {
                try
                {
                    fop.close();
                }
                catch (IOException x)
                {
                    // handle error
                }
                if (!f.delete())// delete file when finished
                {
                    // log the error
                }
            }
        }
    }
}
```

对于 JDK1.7 之前的版本，可以在临时文件使用完毕之后、系统终止之前，显式地对其进行删除。

规则 3.2 不要将 Buffer 对象封装的数据暴露给不可信代码

说明：java.nio 包中的 Buffer 类，如 IntBuffer, CharBuffer，以及 ByteBuffer 定义了一系列的方法，如 wrap()、slice()、duplicate()、slice()，这些方法会创建一个新的 buffer 对象，但是修改这个新 buffer 对象会导致原始的封装数据也被修改，反之亦然。例如，wrap()方法将原始类型数组包装成一个 buffer 对象并返回。虽然这些方法会创建一个新的 buffer 对象，但是它后台封装的还是之前的给定数组，那么任何对 buffer 对象的修改也会导致封装的数组被修改，反之亦然。将这些 buffer 对象暴露给不可信代码，则会使其封装的数组面临恶意修改的风险。同样的，duplicate()方法会以原始 buffer 封装的数组来额外创建新的 buffer 对象，将此额外新建的 buffer 对象暴露给不可信代码同样会面临原始数据被恶意修改的风险。为了防止这种问题的发生，新建的 buffer 应该以只读视图或者拷贝的方式返回。

错误示例：

```
public class Wrapper
{
    private char[] dataArray;

    public Wrapper ()
    {
        dataArray = new char[10];
        // Initialize
    }

    public CharBuffer getBufferCopy()
    {
        return CharBuffer.wrap(dataArray);
    }
}

public class Duplicator
{
    CharBuffer cb;

    public Duplicator ()
    {
        cb = CharBuffer.allocate(10);
        // Initialize
    }

    public CharBuffer getBufferCopy()
    {
        return cb.duplicate();
    }
}
```

错误示例代码声明了一个 char 数组，然后将此数组封装到一个 buffer 中，最后通过 getBufferCopy()方法将此 buffer 暴露给不可信代码。

正确示例：

```
public class Wrapper
{
    private char[] dataArray;

    public Wrapper ()
    {
        // Initialize
        dataArray = new char[10];
    }
```

```
    // return a read-only view
    public CharBuffer getBufferCopy()
    {
        return CharBuffer.wrap(dataArray).asReadOnlyBuffer();
    }
}

public class Duplicator
{
    CharBuffer cb;

    public Duplicator ()
    {
        // Initialize
        cb = CharBuffer.allocate(10);
    }

    // return a read-only view
    public CharBuffer getBufferCopy()
    {
        return cb.asReadOnlyBuffer();
    }
}
```

这个正确示例以只读 CharBuffer 的方式返回 char 数组的一个只读视图。

规则 3.3 在多用户系统中创建文件时指定合适的访问许可

说明：多用户系统中的指定的文件通常归属于一个特定的用户。文件的主人能够指定系统中哪些用户能够访问该文件的内容。这些文件系统使用权限和许可模型来保护文件访问。当一个文件被创建时，文件访问许可规定了哪些用户可以访问或者操作这个文件。如果创建文件时没有对文件的访问许可做足够的限制，攻击者可能在修改此文件的访问权限之前对其进行读取或者修改。因此，一定要在创建文件时就为其指定访问许可，以防止未授权的文件访问。

错误示例：

```
Writer out = new FileWriter("file");
```

FileOutputStream 与 FileWriter 的构造器方法无法让程序员显式的指定文件的访问权限。在这个错误示例中，所创建文件的访问许可取决于具体的实现机制，可能无法防止未授权的访问。

正确示例（Linux 情况）：

```
Path file = new File("file").toPath();
// Throw exception rather than overwrite existing file
Set<OpenOption> options = new HashSet<OpenOption>();
options.add(StandardOpenOption.CREATE_NEW);
options.add(StandardOpenOption.APPEND);
// File permissions should be such that only user may read/write file
Set<PosixFilePermission> perms = PosixFilePermissions.fromString("rw-------");
FileAttribute<Set<PosixFilePermission>> attr =
PosixFilePermissions.asFileAttribute(perms);
try (SeekableByteChannel sbc = Files.newByteChannel(file, options, attr))
{
   // write data
};
```

正确示例（Windows 情况）：

```
private static void fileAcl() throws IOException
    {

        Path path = new File("D:/aa.txt").toPath();
```

```
        Set<OpenOption> options = new HashSet<OpenOption>();
        options.add(StandardOpenOption.CREATE);
        options.add(StandardOpenOption.WRITE);

        // Lookup for the principal
        UserPrincipalLookupService service = path.getFileSystem()
                .getUserPrincipalLookupService();
        UserPrincipal user = service.lookupPrincipalByName("Users");
        UserPrincipal systemGroup = service
                .lookupPrincipalByGroupName("SYSTEM");

        // Create a new entry
        final AclEntry entry = AclEntry
                .newBuilder()
                .setType(AclEntryType.ALLOW)
                .setPrincipal(user)
                .setPermissions(AclEntryPermission.READ_DATA,
                        AclEntryPermission.READ_ATTRIBUTES,
                        AclEntryPermission.READ_NAMED_ATTRS,
                        AclEntryPermission.READ_ACL,
                        AclEntryPermission.WRITE_DATA,
                        AclEntryPermission.APPEND_DATA,
                        AclEntryPermission.WRITE_ATTRIBUTES,
                        AclEntryPermission.WRITE_NAMED_ATTRS,
                        AclEntryPermission.WRITE_ACL,
                        AclEntryPermission.SYNCHRONIZE).build();

        final AclEntry entrySystem = AclEntry
                .newBuilder()
                .setType(AclEntryType.ALLOW)
                .setPrincipal(systemGroup)
                .setPermissions(AclEntryPermission.READ_DATA,
                        AclEntryPermission.READ_ATTRIBUTES,
                        AclEntryPermission.READ_NAMED_ATTRS,
                        AclEntryPermission.WRITE_DATA).build();

        FileAttribute<List<AclEntry>> aclattrs = null;
        aclattrs = new FileAttribute<List<AclEntry>>()
        {
            public String name()
            {
                return "acl:acl";
            } /* Windows ACL */

            public List<AclEntry> value()
            {
                ArrayList<AclEntry> list = new ArrayList<AclEntry>();
                list.add(entry);
                list.add(entrySystem);
                return list;
            }
        };

        ByteBuffer buffer = ByteBuffer.allocate(1024);
        final String str = "abc";
        byte[] a = str.getBytes();
        buffer.put(a);
        try
        {
            SeekableByteChannel sbc = Files.newByteChannel(path, options,
                    aclattrs);
            // write data
            sbc.write(buffer);
        }
        catch (Exception e)
        {
            e.printStackTrace();
```

```
        }
    }
```

JDK1.7 的 NIO2 包（java.nio）中提供了一些类来管理文件访问许可。另外，许多方法和构造器在创建文件时可以接受一个用来指定文件访问许可的参数，`Files.newByteChannel()`方法可以用来创建一个文件，并规定其访问许可。由于 JDK1.6 以及之前的版本缺少机制来指定默认的访问许可，因此，为了解决这个问题，必须使用 Java 以外的机制，例如使用本地代码（native code）与 Java 本地接口（JNI）。

例外情况：

如果文件是创建在一个安全目录中，而且该目录对于非受信用户是不可读的，那么允许以默认许可创建文件。例如，如果整个文件系统是可信的或者只有可信用户可以访问，就属于这种情况。关于安全目录的定义请参考规则 3.5 避免在共享目录操作文件。

规则 3.4 避免让外部进程阻塞在输入输出流上

说明：`java.lang.Runtime` 类的 `exec()` 方法与相关联的 `ProcessBuilder.start()` 方法可以被用来调用外部程序进程。这些外部程序运行时由 `java.lang.Process` 对象描述。这个对象包含一个输入流，输出流，以及一个错误流。因为这个进程对象可被 Java 程序用来与外部程序通信，外部进程的输入流是一个 `OutputStream` 对象，可以通过 `Process.getOutputStream()` 方法获取。同样的，外部进程的输出流和错误流都可以由输入流对象来代表，分别通过 `Process.getInputStream()` 与 `Process.getErrorStream()` 方法来获取。这些进程可能需要通过其输入流对其提供输入，并且其输出流、错误流或两者同时会产生输出。不正确地处理这些外部程序可能会导致一些意外的异常、DoS，及其他安全问题。一个进程如果试图从一个空的输入流中读取输入，则会一直阻塞，直到为其提供输入。因此，在调用这样的进程时，必须为其提供输入。一个外部进程的输出可能会耗尽该进程输出流与错误流的缓冲区。当发生这种情况时，Java 程序可能会阻塞外部进程，同时阻碍 Java 程序与外部程序的继续运行。注意，许多平台限制了输出流可用的缓冲区大小。因此，在运行一个外部进程时，如果此进程往其输出流发送任何数据，则必须将其输出流清空。类似的，如果进程会往其错误流发送数据，其错误流也必须被清空。

错误示例（exitValue()）：

```
public class Exec
{
    public static void main(String args[]) throws IOException
    {
        Runtime rt = Runtime.getRuntime();
        Process proc = rt.exec("notemaker");
        int exitVal = proc.exitValue();
        //...
    }
}
```

在这个错误示例中，程序未等到 notemaker 进程结束就调用 `exitValue()` 方法，这很可能会导致 `IllegalThreadStateException` 异常。

错误示例（waitFor()）：

```
public class Exec
{
    public static void main(String args[]) throws IOException,
            InterruptedException
    {
```

```
        Runtime rt = Runtime.getRuntime();
        Process proc = rt.exec("notemaker");
        int exitVal = proc.waitFor();
        //...
    }
}
```

此错误示例中，waitFor()方法将会一直阻塞调用线程直到 notemaker 进程终止。这可以防止前一个示例中的 IllegalThreadStateException 异常。但是，程序可能会阻塞，阻塞的时长将不确定。来自 notemaker 进程的输出可能会耗尽其输出流或者错误流的缓冲区，这是因为在等待外部进程结束的过程中任何一个流都未被读取。如果其中任何一个缓冲区被塞满，则会阻塞 notemaker 进程，从而会同时阻止 notemaker 进程与 java 进程继续往下运行。

正确示例：

```
public class Exec
{
    public static void main(String[] args) throws IOException,
            InterruptedException
    {

        Runtime rt = Runtime.getRuntime();
        Process proc = rt.exec("notemaker");

        // Any error message
        StreamGobbler errorGobbler = new StreamGobbler(proc.getErrorStream(),
                System.err);

        // Any output
        StreamGobbler outputGobbler = new StreamGobbler(proc.getInputStream(),
                System.out);

        errorGobbler.start();
        outputGobbler.start();

        int exitVal = proc.waitFor();
        // Any error
        errorGobbler.join(); // Handle condition where the
        outputGobbler.join(); // process ends before the threads finish
    }
}

class StreamGobbler extends Thread
{
    InputStream is;

    PrintStream os;

    StreamGobbler(InputStream is, PrintStream os)
    {
        this.is = is;
        this.os = os;
    }

    public void run()
    {
        try
        {
            int c;
            while ((c = is.read()) != -1)
                os.print((char) c);
        }
        catch (IOException x)
        {
            // handle error
        }
    }
```

```
}
```

这个正确示例产生两个线程来分别读取进程的输出流和错误流。因此，进程将不会无限期地阻塞在这些流之上。当输出流和错误流被分开处理时，它们必须独立被清空。如果未这样做，则可导致程序被无限期阻塞。

例外情况：

对于那些从来不会读取其输入流的进程，不对其提供输入非但无害，且还有益。而对于那些从来不会发送数据到其输出流或者错误流的进程，不对其输出流或者错误流进行清空同样是有益无害的。因此，只要能够保证进程不会使用这些流，那么在程序中可以忽略其输入流、输出流、以及错误流。

规则 3.5 避免在共享目录操作文件

说明：多用户系统允许多个具有不同权限的用户共享一个文件系统。攻击者可以利用许多文件系统的特性和功能，包括文件链接、设备文件和共享文件访问，在不具备权限的情况下对文件进行越权访问，特别是在多个用户可以创建、移动、删除文件的共享目录中操作文件。当程序以较高权限运行时，可能被攻击者利用来提升自己的权限。为了防止漏洞，程序必须仅在安全目录中操作文件。如果对于某个特定用户，只有该用户与系统管理员可以在其中创建、移动、删除文件，那么这个目录就是安全的。

错误示例：

```
String file = request.getParameter("webDAVPath");
InputStream in = null;
try
{
    in = new FileInputStream(file);
    // ...
}
finally
{
    try
    {
        if (in != null)
        {
            in.close();
        }
     }
     catch (IOException x)
    {
        // handle error
    }
}
```

错误示例中，攻击者可以指定一个锁定设备或者一个先入先出（FIFO）文件名，导致程序在打开文件时挂起。

正确示例：

下面的解决方式是针对 POSIX 系统的一个 isInSecureDir()方法实现。这个方法确保传入的文件及其所有上层目录是归当前用户或者管理员所有，任何其他的用户不能对其进行写操作，并且其所有上层目录也不能被任何其他用户删除或者改名（除了系统管理员）。

```
public static boolean isInSecureDir(Path file)
{
    return isInSecureDir(file, null);
}

public static boolean isInSecureDir(Path file, UserPrincipal user)
```

```
{
    return isInSecureDir(file, user, 5);
}

/**
 * Indicates whether file lives in a secure directory relative
 * to the program's user
 * @param file Path to test
 * @param user User to test. If null, defaults to current user
 * @param symlinkDepth Number of symbolic links allowed
 * @return true if file's directory is secure
 */
public static boolean isInSecureDir(Path file, UserPrincipal user,
                                    int symlinkDepth)
{
    if (!file.isAbsolute())
    {
        file = file.toAbsolutePath();
    }
    if (symlinkDepth <= 0)
    {
        // Too many levels of symbolic links
        return false;
    }
    // Get UserPincipal for specified user and superuser
    FileSystem fileSystem = Paths.get(file.getRoot().toString())
            .getFileSystem();
    UserPrincipalLookupService upls =
fileSystem.getUserPrincipalLookupService();
    UserPrincipal root = null;
    try
    {
        root = upls.lookupPrincipalByName("root");
        if (user == null)
        {
            user = upls.lookupPrincipalByName(System.getProperty("user.name"));
        }
        if (root == null || user == null)
        {
            return false;
        }
    }
    catch (IOException x)
    {
        return false;
    }
    // If any parent dirs (from root on down) are not secure,
    // dir is not secure
    for (int i = 1; i <= file.getNameCount(); i++)
    {
        Path partialPath = Paths.get(file.getRoot().toString(),
                file.subpath(0, i).toString());

        try
        {
            if (Files.isSymbolicLink(partialPath))
            {
                if (!isInSecureDir(Files.readSymbolicLink(partialPath),
                        user, symlinkDepth - 1))
                {
                    // Symbolic link, linked to dir not secure
                    return false;
                }
            }
            else
            {
                UserPrincipal owner = Files.getOwner(partialPath);
                if (!user.equals(owner) && !root.equals(owner))
```

```
                {
                    // dir owned by someone else, not secure
                    return false;
                }
                PosixFileAttributes attr = Files.readAttributes(partialPath,
                        PosixFileAttributes.class);
                Set<PosixFilePermission> perms = attr.permissions();
                if (perms.contains(PosixFilePermission.GROUP_WRITE)
                        || perms.contains(PosixFilePermission.OTHERS_WRITE))
                {
                    // someone else can write files, not secure
                    return false;
                }
            }
        }
        catch (IOException x)
        {
            return false;
        }
    }
    return true;
}
```

在对目录进行检查时，非常重要的一点是必须从根目录往叶节点目录遍历，这样做可以防止危险的条件竞争，以防止具有至少一个目录权限的攻击者对目录进行重命名或者重新创建目录的操作，而这个操作正好发生在被篡改目录的权限检查之前和其子目录的权限检查之后。如果路径中包含任何符号链接，这个方法会递归调用链接的目标目录以确保它也是安全的。对于一个符号链接目录，只有当源目录与目标目录都是安全的时候才是安全的。这个方法会检查路径中的每一个目录，以确保所有目录都是属于当前用户或者管理员的，其他任何用户都不能在其中创建、删除、或者重命名文件。在 POSIX 系统中，可通过禁用一个目录的组和全局写访问来防止目录主人和管理员以外的任何用户对其进行更改。

注意，该方法只对遵循 POSIX 文件访问许可机制的文件系统有效，而对于使用其他许可机制的文件系统则可能会引发错误行为。下面的正确示例使用 `isInSecureDir()`方法来确保攻击者不能篡改将要打开和删除的目录。注意，一个目录的路径名称一旦经过`isInSecureDir()`方法的检查，后续所有对那个目录的文件操作都应该使用相同的路径名称。这个正确示例同时检查所请求的文件是一个常规文件，而不是一个符号链接、设备文件或者其他特殊文件。

```
String fileName = /* provided by user */;
Path path = new File(fileName).toPath();
try
{
    if (!isInSecureDir(path))
    {
        System.out.println("File not in secure directory");
        return;
    }
    BasicFileAttributes attr = Files.readAttributes(path,
BasicFileAttributes.class, LinkOption.NOFOLLOW_LINKS);
    // Check
    if (!attr.isRegularFile())
    {
        System.out.println("Not a regular file");
        return;
    }
    // other necessary checks
    try (InputStream in = Files.newInputStream(path))
    {
        // read file
    }
}
catch (IOException x)
{
```

```
    // handle error
}
```

例外情况：

对于运行在单用户系统或者没有共享目录的系统，或者不会发生文件系统漏洞的系统上的程序，则无需在操作文件之前确保其在安全目录之中。

5. 序列化和反序列化

规则 4.1 将敏感对象发送出信任区域前进行签名并加密

说明：敏感数据传输过程中要防止被窃取和恶意篡改。使用安全的加密算法加密传输对象可以保护数据。这就是所谓的对对象进行密封。而对密封的对象进行数字签名则可以防止对象被非法篡改，保持其完整性。在以下场景中，需要对对象密封和数字签名来保证数据安全：

1) 序列化或传输敏感数据
2) 没有使用类似于 SSL 传输通道或者对于有限的事务来说代价太高
3) 敏感数据需要长久保存（比如在硬盘驱动器上）

应该避免使用私有加密算法，因为这样会引入不必要的漏洞。在 `readObject()` 和 `writeObject()` 函数中使用私有加密算法的应用是典型的反面示例。

该规则的代码示例都是基于下面的代码来说明：

```
class SerializableMap<K, V> implements Serializable
{
    final static long serialVersionUID = 45217497203262395L;

    private Map<K, V> map;

    public SerializableMap()
    {
        map = new HashMap<K, V>();
    }

    public V getData(K key)
    {
        return map.get(key);
    }

    public void setData(K key, V data)
    {
        map.put(key, data);
    }
}
public class MapSerializer
{
    public static SerializableMap<String, Integer> buildMap()
    {
        SerializableMap<String, Integer> map = new SerializableMap<String,
Integer>();
        map.setData("John Doe", new Integer(123456789));
        map.setData("Richard Roe", new Integer(246813579));
        return map;
    }

    public static void InspectMap(SerializableMap<String, Integer> map)
    {
        System.out.println("John Doe's number is " + map.getData("John Doe"));
        System.out.println("Richard Roe's number is "
                + map.getData("Richard Roe"));
    }

}
```

错误示例：

```
public static void main(String[] args) throws IOException,
    ClassNotFoundException
{
```

```
    // Build map
    SerializableMap<String, Integer> map = buildMap();
    // Serialize map
    ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream("data"));
    out.writeObject(map);
    out.close();

    // Deserialize map
    ObjectInputStream in = new ObjectInputStream(new FileInputStream("data"));
    map = (SerializableMap<String, Integer>) in.readObject();
    in.close();
    // Inspect map
    InspectMap(map);
}
```

错误代码没有采取任何措施抵御二进制数据传输过程中可能遭遇的字节流操纵攻击。因此，任何人都可以对序列化的流数据实施逆向工程从而恢复 HashMap 中的数据。

错误示例（仅加密）：

```
public static void main(String[] args) throws IOException,
       GeneralSecurityException, ClassNotFoundException
{
    // Build map
    SerializableMap<String, Integer> map = buildMap();
    // Generate sealing key & seal map
    KeyGenerator generator = KeyGenerator.getInstance("AES");
    generator.init(new SecureRandom());
    Key key = generator.generateKey();
    Cipher cipher = Cipher.getInstance("AES");
    cipher.init(Cipher.ENCRYPT_MODE, key);
    SealedObject sealedMap = new SealedObject(map, cipher);
    // Serialize map
    ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream("data"));
    out.writeObject(sealedMap);
    out.close();

    // Deserialize map
    ObjectInputStream in = new ObjectInputStream(new FileInputStream("data"));
    sealedMap = (SealedObject) in.readObject();
    in.close();
    // Unseal map
    cipher = Cipher.getInstance("AES");
    cipher.init(Cipher.DECRYPT_MODE, key);
    map = (SerializableMap<String, Integer>) sealedMap.getObject(cipher);
    // Inspect map
    InspectMap(map);
}
```

该程序未对数据进行签名，因此无法进行可靠性验证。

错误示例（先加密后签名）：

```
public static void main(String[] args) throws IOException,
                            GeneralSecurityException, ClassNotFoundException
{
    // Build map
    SerializableMap<String, Integer> map = buildMap();
    // Generate sealing key & seal map
    KeyGenerator generator = KeyGenerator.getInstance("AES");

    generator.init(new SecureRandom());
    Key key = generator.generateKey();
    Cipher cipher = Cipher.getInstance("AES");
    cipher.init(Cipher.ENCRYPT_MODE, key);
    SealedObject sealedMap = new SealedObject(map, cipher);
    // Generate signing public/private key pair & sign map
```

```
    KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA");
    KeyPair kp = kpg.generateKeyPair();
    Signature sig = Signature.getInstance("SHA256withRSA");
    SignedObject signedMap = new SignedObject(sealedMap, kp.getPrivate(), sig);
        // Serialize map
    ObjectOutputStream out = new ObjectOutputStream(new
                                                    FileOutputStream("data"));
    out.writeObject(signedMap);
    out.close();

    // Deserialize map
    ObjectInputStream in = new ObjectInputStream(new FileInputStream("data"));
    signedMap = (SignedObject) in.readObject();
    in.close();
    // Verify signature and retrieve map
    if (!signedMap.verify(kp.getPublic(), sig))
    {
        throw new GeneralSecurityException("Map failed verification");
    }
    sealedMap = (SealedObject) signedMap.getObject();
    // Unseal map
    cipher = Cipher.getInstance("AES");
    cipher.init(Cipher.DECRYPT_MODE, key);
    map = (SerializableMap<String, Integer>) sealedMap.getObject(cipher);
    // Inspect map
    InspectMap(map);
}
```

这段代码先将对象加密然后为其签名。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。这样一来，由于对象被加密和签名（只有在签名验证通过后才可以解密对象），恶意第三方和正常的接收者均无法得到原始的消息内容。

正确示例 （先签名后加密）：

```
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SealedObject;
// Other import…
public static void main(String[] args) throws IOException,
GeneralSecurityException, ClassNotFoundException
{
    // Build map
    SerializableMap<String, Integer> map = buildMap();
    // Generate signing public/private key pair & sign map
    KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA");
    KeyPair kp = kpg.generateKeyPair();
    Signature sig = Signature.getInstance("SHA256withRSA");
    SignedObject signedMap = new SignedObject(map, kp.getPrivate(), sig);
    // Generate sealing key & seal map
    KeyGenerator generator = KeyGenerator.getInstance("AES");

    generator.init(new SecureRandom());
    Key key = generator.generateKey();
    Cipher cipher = Cipher.getInstance("AES");
    cipher.init(Cipher.ENCRYPT_MODE, key);
    SealedObject sealedMap = new SealedObject(signedMap, cipher);
    // Serialize map
    ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(
                "data"));
    out.writeObject(sealedMap);
    out.close();

    // Deserialize map
    ObjectInputStream in = new ObjectInputStream(new FileInputStream("data"));
    sealedMap = (SealedObject) in.readObject();
```

```
    in.close();
    // Unseal map cipher = Cipher.getInstance("AES");
    cipher.init(Cipher.DECRYPT_MODE, key);
    signedMap = (SignedObject) sealedMap.getObject(cipher);
    // Verify signature and retrieve map
    if (!signedMap.verify(kp.getPublic(), sig))
    {
        throw new GeneralSecurityException("Map failed verification");
    }
    map = (SerializableMap<String, Integer>) signedMap.getObject();
    // Inspect map
    InspectMap(map);
}
```

这段正确的代码先为对象签名然后再加密。这样既能保证数据的真实可靠性，又能防止“中间人攻击”（man-in-middle attacks）。

例外情况：

1) 在特定场景下为已加密对象签名是合理的，比如验证从其他地方接收的加密对象的真实性。这是对于被机密对象本身而非其内容的保证。
2) 签名和加密对于必须跨过信任边界的对象是必需的。始终位于信任边界内的对象不需要签名或加密。例如，如果某网络全部位于信任边界内，始终处于该网络上的对象无需签名或加密。另一个例子是仅在已签名的二进制流下发送对象的情况。

规则 4.2 禁止序列化未加密的敏感数据

说明：虽然序列化可以将对象的状态保存为一个字节序列，之后通过反序列化将字节序列又能重新构造出原来的对象，但是它并没有提供一种机制来保证序列化数据的安全性。因此，敏感数据序列化之后是潜在对外暴露的，可访问序列化数据的攻击者可以借此获取敏感信息并确定对象的实现细节。永远不应该被序列化的敏感信息包括：密钥、数字证书、以及那些在序列化时引用敏感数据的类，防止敏感数据被无意识的序列化导致敏感信息泄露。另外，声明了可序列化标识对象的所有字段在序列化时都会被输出为字节序列，能够解析这些字节序列的代码可以获取到这些数据的值，而不依赖于该字段在类中的可访问性。因此，若其中某些字段包含敏感信息，则会造成敏感信息泄露。

错误示例：

```
public class GPSLocation implements Serializable
{
    private double x; // sensitive field
    private double y; // sensitive field
    private String id;// non-sensitive field

    // other content
}
public class Coordinates
{
    public static void main(String[] args)
    {
        FileOutputStream fout = null;
        try
        {
            GPSLocation p = new GPSLocation(5, 2, "northeast");
            fout = new FileOutputStream("location.ser");
            ObjectOutputStream oout = new ObjectOutputStream(fout);
            oout.writeObject(p);
            oout.close();
        }
        catch (Throwable t)
```

```
        {
            // Forward to handler
        }
        finally
        {
            if (fout != null)
            {
                try
                {
                    fout.close();
                }
                catch (IOException x)
                {
                    // handle error
                }
            }
        }
    }
}
```

示例代码中，假定坐标信息是敏感的，那么将其序列化到数据流中使之面临敏感信息泄露与被恶意篡改的风险。

正确示例（transient）：

```
public class GPSLocation implements Serializable
{
    private transient double x; // transient field will not be serialized
    private transient double y; // transient field will not be serialized
    private String id;

    // other content
}
```

在对某个包含敏感数据的类序列化时，程序必须确保敏感数据不被序列化。包括阻止包含敏感信息的数据成员被序列化，以及不可序列化或者敏感对象的引用被序列化。该示例将相关字段声明为 transient，从而使它们不包括在依照默认的序列化机制应该被序列化的字段列表中。这样既避免了错误的序列化，又防止了敏感数据被意外序列化。

正确示例（serialPersistentFields）：

```
public class GPSLocation implements Serializable
{
    private double x;
    private double y;
    private String id;
    // sensitive fields x and y are not content in serialPersistentFields
    private static final ObjectStreamField[] serialPersistentFields = {new
ObjectStreamField("id", String.class)};
    // other content
}
```

该示例通过定义 serialPersistentFields 数组字段来确保敏感字段被排除在序列化之外，除了上述方案，也可以通过自定义 writeObject()、writeReplace()、writeExternal()这些函数，不将包含敏感信息的字段写到序列化字节流中。

例外情况：

可以序列化已正确加密的敏感数据。

规则 4.3 防止序列化和反序列化被利用来绕过安全管理

说明：序列化和反序列化可能会被利用来绕过安全管理器的检查。一个可序列化类的构造器为了防止不可信代码修改类的内部状态而需要引入安全管理器的检查。这种安全管理器的检

查必须应用到所有能够构建类实例的地方。例如，如果某个类依据安全检查的结果来判定调用者是否能够读取其敏感内部状态，那么这类安全检查也必须在反序列化中应用。这就确保了攻击者无法通过反序列化对象来提取敏感信息。

错误示例：

```
public final class Hometown implements Serializable
{
    private static final long serialVersionUID = 9078808681344666097L;

    // Private internal state
    private String town;

    private static final String UNKNOWN = "UNKNOWN";

    void performSecurityManagerCheck() throws SecurityException
    {
        // verify whether current user has rights to access the file
    }

    void validateInput(String newCC) throws InvalidInputException
    {
        // ...
    }

    public Hometown()
    {
        performSecurityManagerCheck();
        // Initialize town to default value
        town = UNKNOWN;
    }

    // Allows callers to retrieve internal state
    String getValue()
    {
        performSecurityManagerCheck();
        return town;
    }

    // Allows callers to modify (private) internal state
    public void changeTown(String newTown) throws InvalidInputException
    {
        if (town.equals(newTown))
        {
            // No change
            return;
        }
        else
        {
            performSecurityManagerCheck();
            validateInput(newTown);
            town = newTown;
        }
    }

    private void writeObject(ObjectOutputStream out) throws IOException
    {
        out.writeObject(town);
    }

    private void readObject(ObjectInputStream in) throws IOException,
ClassNotFoundException
    {
        in.defaultReadObject();
        // If the deserialized name does not match
        // the default value normally
        // created at construction time, duplicate the checks
```

```
        if (!UNKNOWN.equals(town))
        {
            validateInput(town);
        }
    }
}
```

错误示例中，安全管理器检查被应用在构造器中，但在序列化与反序列化涉及的writeObject()和 readObject()方法中没有用到。这样会允许非信任代码恶意创建类实例。

正确示例：

```
public final class Hometown implements Serializable
{
    // ... all methods the same except the following:
    // writeObject() correctly enforces checks during serialization
    private void writeObject(ObjectOutputStream out) throws IOException
    {
        performSecurityManagerCheck();
        out.writeObject(town);
    }

    // readObject() correctly enforces checks during deserialization
    private void readObject(ObjectInputStream in) throws IOException,
ClassNotFoundException
    {
        in.defaultReadObject();
        // If the deserialized name does not match the default value normally
        // created at construction time, duplicate the checks
        if (!UNKNOWN.equals(town))
        {
            performSecurityManagerCheck();
            validateInput(town);
        }
    }
}
```

正确做法是在所有构造器以及可以修改或检索内部数据的方法中都需要应用安全管理器检查。这样一来，攻击者就不能用反序列化来修改对象的实例或者通过读取字节流来窃取序列化的数据。

6. 平台安全

规则 5.1 使用安全管理器来保护敏感操作

说明：当应用需要加载非信任代码时，必须安装安全管理器，且敏感操作必须经过安全管理器检查，从而防止它们被非信任代码调用。某些常见敏感操作的 Java API，例如访问本地文件、向外部主机开放套接字连接或者创建一个类加载器，已经包括了使用安全管理器检查来实施 JDK 中的某些预定义策略。仅需要安装安全管理器即可保护这些预定义的敏感操作。然而，应用本身也可能包含敏感操作。对于这些敏感操作，除了安装一个安全管理器之外，必须自定义安全策略，并在操作前手动为其增加安全管理器检查。

错误示例：

```
public class SensitiveHash
{
    private Hashtable<Integer, String> ht = new Hashtable<Integer, String>();

    public void removeEntry(Object key)
    {
        ht.remove(key);
    }
}
```

示例代码实例化一个 Hashtable，并定义了一个 `removeEntry()`方法删除其条目。这个方法被认为是敏感的，因为哈希表中包含敏感信息。由于该方法被声明为是 public 且 non-final 的，所以暴露给了恶意调用者。

正确示例：

```
public class SensitiveHash
{
    Hashtable<Integer, String> ht = new Hashtable<Integer, String>();

    void removeEntry(Object key)
    {
        // "removeKeyPermission" is a custom target name for SecurityPermission
        check("removeKeyPermission");
        ht.remove(key);
    }
    private void check(String directive)
    {
        SecurityManager sm = System.getSecurityManager();
        if (sm != null)
        {
            sm.checkSecurityAccess(directive);
        }
    }
}
```

示例使用安全管理器检查来防止 Hashtable 实例中的条目被恶意删除。如果调用者缺少 `java.security.SecurityPermission removeKeyPermission`，一个 `SecurityException`异常将被抛出。`SecurityManager.checkSecurityAccess()`方法检查调用者是否有特定的操作权限。

规则 5.2 防止特权区域内出现非法的数据

说明：`java.security.AccessController` 类是 Java 安全机制的一部分，负责实施可

应用的安全策略。该类静态的 doPrivileged()方法以不严格的安全策略执行一个代码块。doPrivileged()方法将会阻止权限检查在方法调用栈上进一步往下进行。因此，任何包含 doPrivileged()代码块的方法或者类都有责任确保敏感操作访问的安全性。
不要在特权块内操作未经校验的或者非信任的数据。如果违反，攻击者可以通过提供恶意输入来提升自己的权限。在进行特权操作之前，通过硬编码方式进行数据校验，可以减小这种风险。

错误示例：

```
private void privilegedMethod(final String fileName) throws
FileNotFoundException
{
    try
    {
        FileInputStream fis = (FileInputStream) AccessController.doPrivileged(
                new PrivilegedExceptionAction()
        {
            public FileInputStream run() throws FileNotFoundException
            {
                return new FileInputStream(fileName);
            }
        });
        // do something with the file and then close it
    }
    catch (PrivilegedActionException e)
    {
       // forward to handler
    }
}
```

该代码示例接受一个非法的路径或文件名作为参数。攻击者可以通过将受保护的文件路径传入，从而得到特权访问这些文件。

正确示例：

```
private void privilegedMethod(final String fileName) throws
FileNotFoundException, InvalidArgumentException
{

    final String cleanFileName;
    cleanFileName = cleanAFileNameAndPath(fileName);
    try
    {
        FileInputStream fis = (FileInputStream)
        AccessController.doPrivileged(new PrivilegedExceptionAction()
        {
            public FileInputStream run() throws FileNotFoundException
            {
                return new FileInputStream(cleanFileName);
            }
        });
        // do something with the file and then close it
    }
    catch (PrivilegedActionException e)
    {
        // forward to handler and log
    }
}
```

正确示例（内置文件名与路径）：

```
static final String FILEPATH = "/path/to/protected/file/fn.ext";
private void privilegedMethod() throws FileNotFoundException
{
    try
    {
        FileInputStream fis = (FileInputStream)
```

```
        AccessController.doPrivileged(new PrivilegedExceptionAction()
        {
            public FileInputStream run() throws FileNotFoundException
            {
                return new FileInputStream(FILEPATH);
            }
        });
        // do something with the file and then close it
    }
    catch (PrivilegedActionException e)
    {
        // forward to handler and log
    }
}
```

允许一个非特权用户访问受保护文件或其他资源本身就是不安全的设计。可以考虑硬编码资源名称，或者是只允许用户在一个特定的选项列表中进行选择，这些选项会间接映射到对应的资源名称。正确示例同时显式硬编码文件名与限制包含特权块方法中使用的变量。这就确保了恶意文件无法通过利用特权方法被加载。

规则 5.3 禁止基于不信任的数据源做安全检查

说明：不受信任数据源的安全检查可以被攻击者所绕过。在使用非受信数据源时，必须确保被检查的输入和实际被处理的输入相同。如果输入在检查和使用之间发生了变化，便会发生“time-of-check, time-of-use”（TOCTOU）漏洞。正确的做法是保持数据不可变。在做安全检查之前，可以先对不受信任的对象或者参数做防御性拷贝，然后对这份拷贝做安全检查。这样的拷贝必须是深拷贝。如果对象的 `clone()` 方法是一个浅拷贝，仍然可能会带来危害。另外 `clone()` 方法的实现本身可能就是由攻击者所提供。

错误示例：

```
public RandomAccessFile openFile(final java.io.File f)
{
    RandomAccessFile rf = null;
    try
    {
        askUserPermission(f.getCanonicalPath());
        // ...
        rf = AccessController.doPrivileged(new
PrivilegedExceptionAction<RandomAccessFile>()
        {
            public RandomAccessFile run() throws FileNotFoundException
            {
                return new RandomAccessFile(f, "r");
            }
        });
    }
    catch(IOException e)
    {
        // handle error
    }
    catch (PrivilegedActionException e)
    {
        // handle error
    }
    return rf;
}
```

代码示例描述了 JDK1.5 版本 java.io 包中的一个安全漏洞。在此版本中，`java.io.File` 类不是 final 类，它允许攻击者继承合法的 File 类来提供一个非受信参数。在这种方式下，覆盖 `getPath()` 函数以后，第一次被调用的时候安全检查会通过，但是在第二次改变数值的

时候会指向敏感文件，例如/etc/passwd。这就是 TOCTOU 漏洞的一个例子。

攻击者可将 java.io.File 按如下方式扩展：

```
public class BadFile extends java.io.File
{
    private int count;
    // ... Other omitted code
    public String getPath()
    {
        return (++count == 1) ? "/tmp/foo" : "/etc/passwd";
    }
}
```

然后用 BadFile 类型的文件对象调用有漏洞的 openFile()函数。

正确示例：

```
public RandomAccessFile openFile(final java.io.File f)
{
    RandomAccessFile rf = null;
    try
    {
        final java.io.File copy = new java.io.File(f.getPath());
        askUserPermission(copy.getCanonicalPath());
        // ...
        rf = AccessController.doPrivileged(new
PrivilegedExceptionAction<RandomAccessFile>()
        {
            public RandomAccessFile run() throws FileNotFoundException
            {
                return new RandomAccessFile(f, "r");
            }
        });
    }
    catch(IOException e)
    {
        // handle error
    }
    catch (PrivilegedActionException e)
    {
        // handle error
    }
    return rf;
}
```

正确代码示例确保 java.io.File 对象是可信的，不管它是否是 final 型的。该示例使用标准构造器创建了一个新的文件对象。这样可以保证在 File 对象上调用的任何函数均来自标准类库，而不是被攻击者所覆盖过的函数。注意，使用 clone()函数而非 openFile()函数会拷贝攻击者的类，而这是不可取的。

规则 5.4 禁止特权块向非信任域泄漏敏感信息

说明：java.security.AccessController 类是 Java 安全机制的一部分，负责实施可应用的安全策略。该类静态的 doPrivileged()方法以相对宽松的安全策略执行一个代码块。doPrivileged()方法停止在调用链上对权限进行进一步的检查。因此，任何调用 doPrivileged()代码块的方法或者类都有责任确保敏感操作访问的安全性。

doPrivileged()方法一定不能泄露敏感信息或者功能。假如一个 Web 应用程序为 Web 服务维护一个敏感的口令文件，同时也会加载运行不受信任的代码。那么，Web 应用程序可以实施一种安全策略来防止自身的大部分代码和不受信任代码访问该敏感文件。由于必须要提供添加和修改口令的机制，可通过 doPrivileged()特权快来临时允许不受信任代码访问敏感文件来管理密码。这种情况下，任何特权块必须防止不受信任代码访问口令信息。

错误示例：

```
public class PasswordManager
{
    public static void changePassword() throws MyAppException
    {
        // ...
        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 passwordFile = "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
                    return new FileInputStream(passwordFile);
                }
            });
        }
        catch (PrivilegedActionException x)
        {
             // Handle exceptions…
        }
        return fin;
    }
}
```

示例中，doPrivileged()方法被openPasswordFile()方法所调用。openPasswordFile()函数通过特权块代码获取并返回口令文件的FileInputStream流。由于openPasswordFile()方法为public，它可能被不受信任代码所调用，从而引起敏感信息泄漏。

正确示例：

```
public class PasswordManager
{
    public static void changePassword() throws MyAppException
    {
        try
        {
            FileInputStream fin = openPasswordFile();
            // test old password with password in file contents; change password
            // then close the password file
        }
        // Handle exceptions…
    }

    private static FileInputStream openPasswordFile()
            throws FileNotFoundException
    {
        final String passwordFile = "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
                    return new FileInputStream(passwordFile);
                }
            });
        }
        catch (PrivilegedActionException x)
        {
            // Handle exceptions…
        }
        return fin;
    }
}
```

正确代码将 openPasswordFile()声明为 private，非受信调用者可以调用 changePassword()但却不能直接调用 openPasswordFile()函数。

规则 5.5 编写自定义类加载器时应调用超类的 getPermission()函数

说明：当一个自定义的类装载器必须要覆写 getPermissions()方法的时候，必须要在给源代码赋予任何权限之前直接调用基类的 getPermissions 方法来获知默认的系统规则。一个忽略了基类的 getPermissions()方法的自定义类装载器会载入被提升了权限的非受信的类。

错误示例：

```
public class MyClassLoader extends URLClassLoader
{
    @Override
    protected PermissionCollection getPermissions(CodeSource cs)
    {
        PermissionCollection pc = new Permissions();
        // allow exit from the VM anytime
        pc.add(new RuntimePermission("exitVM"));
        return pc;
    }
    // Other code…
}
```

该错误代码示例展示了一个继承自 URLClassLoader 类的自定义类加载器的一部分。它覆盖了 getPermissions()方法，但是并未调用其超类的 getPermissions()方法。因此，该自定义类加载器加载的类具有的权限就会完全独立于系统全局策略文件规定的权限。实际上，该类的权限覆盖了这些系统全局文件规定的权限。

正确示例：

```
public class MyClassLoader extends URLClassLoader
{
    @Override
    protected PermissionCollection getPermissions(CodeSource cs)
    {
        PermissionCollection pc = super.getPermissions(cs);
        // allow exit from the VM anytime
        pc.add(new RuntimePermission("exitVM"));
        return pc;
    }
    // Other code…
}
```

在上面的正确代码示例中，getPermissions()函数调用了 super.getPermissions()。

结果，除了自定义策略外，系统全局的默认安全策略也被应用。

规则 5.6 避免完全依赖URLClassLoader和java.util.jar提供的默认自动签名认证机制

说明：使用 Java 可以给一些部署文件进行打包，例如在 Enterprise JavaBeans（EJB）、MIDlets（J2ME）和 Weblogic Server J2EE 等应用中，一般使用 JAR 文件的方式进行分发。Java Web Start 提供的即点即击的安装也依赖于 JAR 文件格式进行打包。有时厂商会为自己的 JAR 文件进行签名。这可以证明代码的真实性，但却不能保证代码的安全性。

客户代码可能缺乏代码签名的程序化检查。例如，`URLClassLoader` 及其子类实例与 `java.util.jar` 自动验证 JAR 文件的签名，但是自动验证也只是进行完整性检查，由于检查使用的是 JAR 包中未经验证的公钥，因此无法对加载类的真实性进行认证。而开发人员自定义的类加载器也可能缺乏这项检查。合法的 JAR 文件可能会被恶意 JAR 文件所替换，连同其中的公钥和摘要值也被适当的替换和修改。

默认的自动签名验证过程仍然可以使用，但仅仅借助它是不够的。使用默认的自动签名验证过程的系统必须执行额外的检查来确保签名的正确性（(如与一个已知的受信任签名进行比较）。

错误示例：

```
public class JarRunner
{
    public static void main(String[] args) throws IOException,
            ClassNotFoundException, NoSuchMethodException,
            InvocationTargetException
    {

        URL url = new URL(args[0]);

        // Create the class loader for the application jar file
        JarClassLoader cl = new JarClassLoader(url);

        // Get the application's main class name
        String name = cl.getMainClassName();

        // Get arguments for the application
        String[] newArgs = new String[args.length - 1];
        System.arraycopy(args, 1, newArgs, 0, newArgs.length);

        // Invoke application's main class
        cl.invokeClass(name, newArgs);
    }
}

final class JarClassLoader extends URLClassLoader
{
    private URL url;

    public JarClassLoader(URL url)
    {
        super(new URL[] {url});
        this.url = url;
    }

    public String getMainClassName() throws IOException
    {
        URL u = new URL("jar", "", url + "!/");
```

```
        JarURLConnection uc = (JarURLConnection) u.openConnection();
        Attributes attr = uc.getMainAttributes();
        return attr != null ? attr.getValue(Attributes.Name.MAIN_CLASS) : null;
    }

    public void invokeClass(String name, String[] args)
            throws ClassNotFoundException, NoSuchMethodException,
            InvocationTargetException
    {
        Class c = loadClass(name);
        Method m = c.getMethod("main", new Class[] {args.getClass()});
        m.setAccessible(true);
        int mods = m.getModifiers();
        if (m.getReturnType() != void.class || !Modifier.isStatic(mods)
                || !Modifier.isPublic(mods))
        {
            throw new NoSuchMethodException("main");
        }
        try
        {
            m.invoke(null, new Object[] {args});
        }
        catch (IllegalAccessException e)
        {
            System.out.println("Access denied");
        }
    }
}
```

该错误示例代码可以动态执行 JAR 文件中的某个特定类。该程序创建了一个 JarClassLoader，它通过不信任的网络如 Internet 来加载程序更新、插件或补丁。第一个参数是获取代码的 URL， JarRunner 使用反射来调用被加载类的 main()方法。不幸的是，默认情况下，JarClassLoader 使用 JAR 文件中包含的公钥来验证签名。

正确示例：

```
public void invokeClass(String name, String[] args)
        throws ClassNotFoundException, NoSuchMethodException,
        InvocationTargetException, GeneralSecurityException, IOException
{
    Class c = loadClass(name);
    Certificate[] certs =
c.getProtectionDomain().getCodeSource().getCertificates();
    if (certs == null)
    {
        // return, do not execute if unsigned
        System.out.println("No signature!");
        return;
    }

    KeyStore ks = KeyStore.getInstance("JKS");
    ks.load(new FileInputStream(System.getProperty("user.home"
            + File.separator + "keystore.jks")), getKeyStorePassword());
    // get the certificate stored in the keystore with "user" as alias
    Certificate pubCert = ks.getCertificate("user");
    // check with the trusted public key, else throws exception
    certs[0].verify(pubCert.getPublicKey());
    // ... other omitted code
}
```

当本地系统采用不可靠的验证签名时，调用程序必须通过程序化的方式验证签名。具体做法是，程序必须从加载类的代码源（Code-Source）中获取证书链，然后检查证书是否属于某个事先获取并保存在本地密钥库（KeyStore）中的受信任签名者。

7. 运行环境

规则 6.1 禁止给仅执行非特权操作的代码签名

说明：在 Java 中代码签名可以让代码获得更高的权限。许多安全策略允许被签名的代码执行更高权限的操作。代码签名被用来对代码的来源做认证以及验证代码的完整性。它依赖于认证机构（CA）来确认签名者的身份。用户普遍将数字签名与代码的安全执行相关联，相信签名的代码不会带来危害。若签名的代码出现漏洞则会产生问题，因为许多系统被配置为固定的信任某些签名机构，如果系统下载了由这些机构签名的包含漏洞的代码，但是这些系统不会警告和通知其用户。攻击者可以向用户输送带有合法签名的漏洞代码，并利用这些漏洞代码。由于上述问题，对于那些不需要执行特权操作的代码不要对其进行签名，让它们运行在受限的沙箱（sandbox）里面。

例外情况：

Oracle 已经弃用了非签名 Applet 的使用，而且也将停止对它们的支持。签名过的 Applet 传统上都是以所有权限运行的。从 JDK1.7.0 update25 版本开始，Oracle 就开始提供一些机制，允许对 Applet 进行签名，并在没有所有权限的情况下可以运行。这使得至今未签名的 Applet 继续在安全沙箱中运行，尽管他们可能有的已经被签名过。对一个以有限权限运行的 Applet 进行签名，至少在 update25 版本之前都是对原则 6.1 构成一个例外。

规范 6.2 不要使用危险的许可与目标组合

说明：有些许可和目标的组合会导致权限过大，而这些权限本不应该被赋予。另外有些权限必须只赋予给特定的代码。

1. 不要将 `AllPermission` 许可赋予给不信任的代码。
2. `ReflectPermission` 许可与 `suppressAccessChecks` 目标组合会减少 Java 语言的访问检查。被授权的类能够访问任意其他类中任意的字段和方法。因此，不要将 `ReflectPermission` 许可和 `suppressAccessChecks` 目标组合使用。
3. 如果将 `java.lang.RuntimePermission` 许可与 `createClassLoader` 目标组合，将赋予代码创建 `ClassLoader` 对象的权限。这将是非常危险的，因为恶意代码可以创建其自己特有的类加载器并通过类加载来为类分配任意许可。

错误示例：

```
// Grant the klib library AllPermission
grant codebase "file:${klib.home}/j2se/home/klib.jar"
{
   permission java.security.AllPermission;
};
```

在该错误代码示例中，为 klib 库赋予了 `AllPermission` 许可。这个许可是在安全管理器使用的安全策略文件中指定的。

正确示例：

```
grant codebase "file:${klib.home}/j2se/home/klib.jar", signedBy "Admin"
{
   permission java.io.FilePermission "/tmp/*", "read";
   permission java.io.SocketPermission "*", "connect";
};
```

此正确示例展示了一个可用来进行细粒度授权的策略文件

例外情况：

有可能需要为受信任的库代码授予 AllPermission 来使得回调方法按预期运行。例如，对可选的 Java 包（拓展库）赋予 AllPermission 权限是常见并可以接受的做法：

```
// Standard extensions extend the core platform and are granted all permissions
by default
grant codeBase "file:${{java.ext.dirs}}/*"
{
    permission java.security.AllPermission;
};
```

规则 6.3 不要禁用字节码验证

说明：Java 字节码验证器是 JVM 的一个内部组件，负责检测不合规的 Java 字节码。包括确保 class 文件的格式正确性、没有出现非法的类型转换、不会出现调用栈下溢，以及确保每个方法最终都会移除它推入栈的任何内容。用户通常觉得从可信的源获取的 Java class 文件是合规的，所以执行起来也是安全的，误以为字节码验证对于这些类来说是多余的。结果，用户可能会禁用字节码验证，破坏 Java 的安全性以及安全保障。字节码验证器一定不能被禁用。

错误示例：

```
java -Xverify:none ApplicationName
```

字节码验证程序默认会被 JVM 所执行。JVM 命令行参数-Xverify:none 会让 JVM 抑制字节码验证过程。在这个错误代码示例中，就使用了这个参数来禁用字节码验证。

正确示例 （默认开启验证）：

```
java ApplicationName
```

字节码验证默认就是启用的。

正确示例 （显式启用验证）：

```
java -Xverify:all ApplicationName
```

在命令行中配置-Xverify:all 参数要求 JVM 启用字节码验证（尽管可能之前是被禁用的）。

规则 6.4 禁止部署的应用可被远程监控

说明：Java 提供了多种 API 让外部程序来监控运行中的 Java 程序。这些 API 也允许不同主机上的程序远程监控 Java 程序。这样的特征方便对程序进行调试或者对其性能进行调优。但是，如果一个 Java 程序被部署在生产环境中同时允许远程监控，攻击者很容易连接到 JVM 来监视这个 Java 程序的行为和数据，包括所有潜在的敏感信息。攻击者也可以对程序的行为进行控制。 因此，当 Java 程序运行在生产环境中时，必须禁用远程监控。

错误示例（JVMTI）：

```
${JDK_PATH}/bin/java -agentlib:libname=options ApplicationName
```

在该错误示例中，JVM Tool Interface（JVMTI）通过代理来与运行中的 JVM 通信。这些代理通常是在 JVM 启动的时候通过 Java 命令行参数–agentlib 或者-agentpath 来加载的，从而允许 JVMTI 对应用程序进行监控。

错误示例（JVM 监控）：

```
${JDK_PATH}/bin/java -Dcom.sun.management.jmxremote.port=8000
ApplicationName
```

在以上错误示例中，用命令行参数使得 JVM 被允许在 8000 端口上进行远程监控。如果密码强度很弱或者误用 SSL 协议，可能会导致安全漏洞。

正确示例：

```
${JDK_PATH}/bin/java -Djava.security.manager ApplicationName
```

上面的命令行启动 JVM 时，未启用任何代理。避免在生产设备上使用-agentlib， -Xrunjdwp，和-Xdebug 命令行参数，并且安装了默认的安全管理器。

例外情况：

对于一个 Java 程序，如果能保证本地信任边界外没有任何程序可以访问该程序，那么这个程序可通过任意一种技术被远程监控。例如，如果这个程序安装在一个本地网络上，该本地网络是完全可信的而且与所有不可信的网络不连通，包括 Internet，那么远程监控是被允许的。

规则 6.5 将所有安全敏感代码都放在一个 jar 包中，签名再密封

说明：若所有安全敏感代码（例如进行权限控制或者用户名密码校验的代码）没有放到同一个受信任的 JAR 包中，攻击者可以先加载恶意代码（使用相同的类名），然后操纵受信任的敏感代码执行恶意代码，导致受信任代码的执行逻辑被劫持。

错误示例：

```
package trusted;

import untrusted.RetValue;

public class MixMatch
{
   private void privilegedMethod() throws IOException
   {
      try
      {
         final FileInputStream fis =
               AccessController.doPrivileged(new
PrivilegedExceptionAction<FileInputStream>()
         {
            public FileInputStream run() throws FileNotFoundException
            {
               return new FileInputStream("file.txt");
            }
         });
         try
         {
            RetValue rt = new RetValue();
            if (rt.getValue() == 1)
            {
               // do something with sensitive file
            }
         }
         finally
         {
            fis.close();
         }
      }
```

```
        catch (PrivilegedActionException e)
        {
            // forward to handler and log
        }
    }

    public static void main(String[] args) throws IOException
    {
        MixMatch mm = new MixMatch();
        mm.privilegedMethod();
    }
}
// In another JAR file:
package untrusted;

class RetValue
{
    public int getValue()
    {
        return 1;
    }
}
```

攻击者可以提供 RetValue 类的实现，使特权代码使用不正确的返回值。尽管 MixMatch 类包含的都是信任的签名的代码，攻击者仍然可以恶意部署一个经过有效签名 JAR 文件，这个 JAR 文件包含不受信任的 RetValue 类，来进行攻击。

正确示例：

```
package trusted;

public class MixMatch
{
    // ...
}
// In the same signed & sealed JAR file:
package trusted;

class RetValue
{
    int getValue()
    {
        return 1;
    }
}
```

该正确代码示例将所有安全敏感代码放在一个包和 JAR 文件中。同时也将 getValue()方法的访问性降低到包可访问。需要对包进行密封以防止攻击者插入恶意类。按以下方式，在 JAR 文件中的 manifest 文件头部中加入 sealed 属性来对包进行密封：

```
Name: trusted // package name
Sealed: true // sealed attribute
```

例外情况：

如果有以下情况，不相关的特权代码和其关联的安全敏感代码（以下简称“组”）可被放置于不同的密封包甚至不同的 JAR 文件中：

- 任意独立组中包含的代码必须与其他任意组中的代码没有任何动态或者静态的依赖关系。这意味着来自于任何这样一组中的代码都不能直接或间接地调用任意其他组里的代码。
- 任意单个组里的所有代码都放置于一个或多个密封的包中。
- 任意单个组里的所有代码都放置于单独一个签名JAR文件中。

规则 6.6 不要信任环境变量的值

说明：为了最大化可移植性，当同样的值已经在系统属性中可用时，就不要引用环境变量。例如，对于操作系统的用户名，可以通过系统属性 user.name 获取。可移植性问题只是为什么不应该依赖环境变量的原因之一。攻击者可以通过某种机制（比如使用 java.lang.ProcessBuilder）来控制进入到某个程序中的所有环境变量的值。因此，当一个环境变量包含的信息可以通过其他方式获取到，包括系统属性，那么就不得使用该环境变量。如果确实需要用到环境变量，也需要在使用之前对其进行校验。

错误示例：

```
String username = System.getenv("USER");
```

在以上错误代码示例中，使用了一个环境变量来获取用户名。这会带来可移植性问题，因为环境变量使用的方式是不同的。例如，Windows 系统中提供用户名的环境变量是“USERNAME”，而在 Unix 中的则可能是“USER”或者“LOGNAME”或者是二者都有。除此之外，攻击者可以在执行程序的时候为“USER”指定任意的值。

正确示例：

```
String username = System.getProperty("user.name");
```

在以上正确代码示例中，使用 user.name 系统属性来获取用户名。JVM 在初始化的时候将这个系统属性设置为一个正确的用户名，尽管 USER 这个环境变量曾经被设置为一个错误的值或者为空。

规则 6.7 生产代码不能包含任何调试入口点

说明：一种常见的做法就是由于调试或者测试目的在代码中添加特定的后门代码，这些代码并没有打算与应用一起交付或者部署。当这类的调试代码不小心被留在了应用中，这个应用对某些无意的交互就是开放的。这些后门入口点可以导致安全风险，因为在设计和测试的时候并没有考虑到而且处于应用预期的运行情况之外。

被忘记的调试代码最常见的例子比如一个 web 应用中出现的 main()方法。虽然这在产品生产的过程中也是可以接受的，但是在生产环境下，J2EE 应用中的类是不应该定义有 main()的。

错误示例：

```
public class Stuff
{
    // other fields and methods
    public static void main(String args[])
    {
        Stuff stuff = new Stuff();
        // Test stuff
    }
}
```

在这个错误代码示例中，Stuff 类使用了一个 main()函数来测试其方法。尽管对于调试是很有用的，如果这个函数被留在了生产代码中（例如，一个 Web 应用），那么攻击者就可能直接调用 Stuff.main()来访问 Stuff 类的测试方法。

正确示例：

正确的代码示例中将 main()方法从 Stuff 类中移除，这样攻击者就不能利用这个入口点

了。

例外：

但是如果出于程序安装或初始化 shell 脚本调用需要，可以保留 main 方法，但是需要确保在使用完以后对对应的类进行保护或者删除。

8. 其他

规则 7.1 禁止在日志中保存口令、密钥和其他敏感数据

说明：在日志中不能输出口令、密钥和其他敏感信息，口令包括明文口令和密文口令。对于敏感信息建议采取以下方法：

- 不在日志中打印敏感信息。
- 若因为特殊原因必须要打印日志，则用固定长度的星号（*）代替输出的敏感信息。

规则 7.2 禁止使用私有或者弱加密算法

说明：禁止使用私有算法或者弱加密算法（比如 DES，SHA1 等）。应该使用经过验证的、安全的、公开的加密算法。

加密算法分为对称加密算法和非对称加密算法。推荐使用的对称加密算法有：

- AES

推荐使用的非对称算法有：

- RSA

推荐使用的数字签名算法有：

- DSA
- ECDSA

除了以上提到的几种算法之外，还经常使用安全哈希算法（SHA256）等来验证消息的完整性。如果使用哈希算法来存储口令，则必须加入盐值（salt）（可参考规则 7.3 基于哈希算法的口令安全存储必须加入盐值（salt））

对每个推荐的算法，其密钥长度需符合以下最低安全要求：

- AES: 128位
- RSA: 2048位
- DSA: 2048位

另：非对称哈希和对称加密应使用 javax.crypto 包。

规则 7.3 基于哈希算法的口令安全存储必须加入盐值（salt）

说明：实践中，一个口令可以编码为一个哈希值，且无法从哈希值逆向计算出原始的口令。口令是否相等可以通过比较它们的哈希值是否相等来判断。如果一个口令的哈希值储存在一个数据库中，由于哈希算法的不可逆性，攻击者就应该不可能还原出口令。如果说可以恢复口令，那么唯一的方式就是暴力破解攻击，比如计算所有可能口令的哈希值，或是字典攻击，计算出所有常用的口令的哈希值。如果每个口令都只仅经过简单哈希，相同的口令将得到相同的哈希值。仅保存口令哈希有以下两个缺陷：

- 由于“生日判定”，攻击者可以快速找到一个口令，尤其是当数据库中的口令数量较大的时候。
- 攻击者可以使用事先计算好的哈希列表在几秒钟之内破解口令。

为了解决这些问题，可以在进行哈希运算之前在口令中引入盐值。一个盐值是一个固定长度

的随机数。这个盐值对于每个存储入口来说必须是不同的。可以明文方式紧邻哈希后的口令一起保存。在这样的配置下，攻击者必须对每一个口令分别进行暴力破解攻击。这样数据库便能抵御“生日”或者“彩虹表”攻击。

为了减慢哈希的计算速度，推荐进行 n 次迭代操作。虽然对一个口令进行 n 次哈希对于攻击者和典型用户来说的确减慢了哈希，但是典型用户并不会有太大的感知，因为哈希的时间相对于他们与系统互动的总时间来说只是非常小的一个比例。另一方面，攻击者破解时几乎100%的时间花在哈希计算上，所以哈希 n 次以 n 为因子减慢了攻击者的速度而对于典型用户几乎不可察觉。

建议：

- 盐值至少应该包含8字节而且必须是由安全随机数产生。
- 应使用强哈希函数，推荐使用SHA-256或者更加安全的哈希函数。
- 推荐默认进行50000次哈希，至少对有性能限制（比如说嵌套系统）的产品进行5000次以上哈希。

正确示例（PBKDF2）：

```
public static byte[] createHash(char[] password)
       throws NoSuchAlgorithmException, InvalidKeySpecException
{
    SecureRandom random = SecureRandom.getInstance("SHA1PRNG");
    byte[] salt = new byte[8];
    random.nextBytes(salt);
    int iterCount = 50000;
    PBEKeySpec spec = new PBEKeySpec(password, salt, iterCount, 256);
    //PBKDF2WithHmacSHA256 is supportted from JDK1.8
    SecretKeyFactory skf =
SecretKeyFactory.getInstance("PBKDF2WithHmacSHA256");
    byte[] hashed = skf.generateSecret(spec).getEncoded();
    return hashed;
}
```

口令单向 Hash 场景下可以使用 PBKDF2 算法，PBKDF2 是一个密钥导出算法，既可用于导出密钥，也可用于口令保存，并且已在 RFC 2898 标准中定义。它使用最为广泛，能被大多数算法库所支持。注意，SunJCE Provider 从 JDK1.8 版本才开始支持 PBKDF2WithHmacSHA256 算法。对于 JDK1.8 之前的版本，可以接受选择 PBKDF2WithHmacSHA1 算法，或者考虑使用其他可靠 JCE Provider 提供支持的 PBKDF2 与 SHA256 或者更强哈希算法的组合。

规则 7.4 禁止将敏感信息硬编码在程序中

说明：如果将敏感信息（包括口令和加密密钥）硬编码在程序中，可能会将敏感信息暴露给攻击者。任何能够访问到 class 文件的人都可以反编译 class 文件并发现这些敏感信息。因此，不能将信息硬编码在程序中。同时，硬编码敏感信息会增加代码管理和维护的难度。例如，在一个已经部署的程序中修改一个硬编码的口令需要发布一个补丁才能实现。

错误示例：

```
public class IPaddress
{
    private String ipAddress = "172.16.254.1";

    public static void main(String[] args)
    {
        //...
    }

}
```

恶意用户可以使用 javap -c IPaddress 命令来反编译 class 来发现其中硬编码的服务器 IP 地址。反编译器的输出信息透露了服务器的明文 IP 地址：172.16.254.1。

正确示例：

```
public class IPaddress
{
   public static void main(String[] args) throws IOException
   {
      char[] ipAddress = new char[100];
      BufferedReader br = new BufferedReader(new InputStreamReader(
            new FileInputStream("serveripaddress.txt")));

      // Reads the server IP address into the char array,
      // returns the number of bytes read
      int n = br.read(ipAddress);
      // Validate server IP address
      // Manually clear out the server IP address
      // immediately after use
      for (int i = n - 1; i >= 0; i--)
      {
         ipAddress[i] = 0;
      }
      br.close();
   }
}
```

这个正确代码示例从一个安全目录下的外部文件获取服务器 IP 地址。并在其使用完后立即从内存中将其清除可以防止后续的信息泄露。

规则 7.5 使用强随机数

说明：伪随机数生成器（PRNG）使用确定性数学算法来产生具有良好统计属性的数字序列。但是这种数字序列并不具有真正的随机特性。伪随机数生成器通常以一个算术种子值为起始。算法使用该种子值生成一个输出以及一个新的种子，这个种子又被用来生成下一个随机值，以此类推。

Java API 提供了伪随机数生成器（PRNG）——java.util.Random 类。这个伪随机数生成器具有可移植性和可重复性。因此，如果两个 java.util.Random 类的实例创建时使用的是相同的种子值，那么对于所有的 Java 实现，它们将生成相同的数字序列。在系统重启或应用程序初始化时，Seed 值总是被重复使用。在一些其他情况下，seed 值来自系统时钟的当前时间。攻击者可以在系统的一些安全脆弱点上监听，并构建相应的查询表预测将要使用的 seed 值。

因此，java.util.Random 类不能用于安全敏感应用或者敏感数据保护。应使用更加安全的随机数生成器，例如 java.security.SecureRandom 类。

但是 java.security.SecureRandom 类本身有可能会产生阻塞的情况，所以建议在非敏感应用或者非敏感数据保护时使用伪随机数生成器 PRNG。

正确示例：

```
public byte[] genRandBytes(int len)
{
    byte[] bytes = null;
    if (len > 0 && len < 1024)
    {
        bytes = new byte[len];
        SecureRandom random = new SecureRandom();
        random.nextBytes(bytes);
    }
```

```
    return bytes;
}
```

规则 7.6 防止将系统内部使用的锁对象暴露给不可信代码

说明：有两种方法来对共享变量的访问做同步：同步方法和同步块。声明为同步的方法以及在 this 引用上的同步块都使用对象自身的锁（隐式锁）。攻击者可以通过获取一个可访问类对象的隐式锁并无限期持有来该锁来触发条件竞争与死锁，进而引起拒绝服务（DoS）。防御这个漏洞一种方法就是使用私有锁对象。

错误示例：

```
public class SomeObject
{
    // Locks on the object's monitor
    public synchronized void changeValue()
    {
        // . . .
    }
}
// Untrusted code
SomeObject theObject = getTheObject ();
synchronized (someObject)
{
    while (true)
    {
        // Indefinitely delay someObject
        Thread.sleep(Integer.MAX_VALUE);
    }
}
```

非信任代码企图获取对象监控器上的锁，一旦成功，将引入一个无限期的时延来阻止声明为同步的 changeValue()方法获取同一个锁。

正确示例：

```
public class SomeObject
{
    private final Object lock = new Object(); // private final lock object

    public void changeValue()
    {
        synchronized (lock)
        {
            // Locks on the private Object
            // ...
        }
    }
}
```

当使用私有不变锁对象时，攻击者将不可能获取到锁。

规则 7.7 使用 SSLSocket 代替 Socket 来进行安全数据交互

说明：当在不安全的传输通道中传输敏感数据时，程序必须使用 javax.net.ssl.SSLSocket 类，而不能是 java.net.Socket 类。SSLSocket 类提供了诸如 SSL/TLS 等安全协议来保证通道不受监听和恶意篡改的影响。

Socket 不提供但是 SSLSocket 提供的主要保护包括：

- *完整性保护：*SSL防止消息被主动窃取者篡改。

- *认证：*在大多数模式下，SSL都对端进行认证。服务器通常都被认证，如果服务器有要求，客户端也可以被认证。
- *保密性（隐私保护）：*在大多数模式下，SSL对客户端和服务器之间传输的数据进行加密。这样被动窃听器不能监听其中的诸如财务或者个人信息之类的敏感信息。

错误示例：

```
//Exception handling has been omitted for the sake of brevity
class EchoServer
{
   public static void main(String[] args) throws IOException
   {
      //Exception handling has been omitted for the sake of brevity
      //...
      ServerSocket serverSocket = new ServerSocket(9900);
      Socket socket = serverSocket.accept();
      PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
      BufferedReader in = new BufferedReader(new InputStreamReader(
                socket.getInputStream()));
      String line;
      while ((line = in.readLine()) != null)
      {
            System.out.println(line);
            out.println(line);
      }
      // ...
   }
   // ...
}

class EchoClient
{
   public static void main(String[] args) throws UnknownHostException,
         IOException
   {
      // Exception handling has been omitted for the sake of brevity
      // ...
      Socket socket = new Socket(getServerIp(), 9999);
      PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
      BufferedReader in = new BufferedReader(new InputStreamReader(
                socket.getInputStream()));
      BufferedReader stdIn = new BufferedReader(new InputStreamReader(
                System.in));
      String userInput;
      while ((userInput = stdIn.readLine()) != null)
      {
         out.println(userInput);
         System.out.println(in.readLine());
      }
      // ...
   }
   // ...
}
```

正确示例：

```
class EchoServer
{
   public static void main(String[] args) throws IOException
   {
      // Exception handling has been omitted for the sake of brevity
      // ...
      SSLServerSocket SSLServerSocketFactory sslServerSocketFactory =
(SSLServerSocketFactory) SSLServerSocketFactory.getDefault();
      sslServerSocket = (SSLServerSocket)
sslServerSocketFactory.createServerSocket(9999);
      SSLSocket sslSocket = (SSLSocket) sslServerSocket.accept();
```

```
        PrintWriter out = new PrintWriter(sslSocket.getOutputStream(), true);
        BufferedReader in = new BufferedReader(new InputStreamReader(
                    sslSocket.getInputStream()));
        String inputLine;
        while ((inputLine = in.readLine()) != null)
        {
            System.out.println(inputLine);
            out.println(inputLine);
        }
        // ...
    }
    // ...
}
class EchoClient
{
    public static void main(String[] args) throws IOException
    {
        // Exception handling has been omitted for the sake of brevity
        // ...
        SSLSocket SSLSocketFactory sslSocketFactory = (SSLSocketFactory)
SSLSocketFactory.getDefault();
        sslSocket = (SSLSocket) sslSocketFactory.createSocket(getServerIp(),
                    9999);
        PrintWriter out = new PrintWriter(sslSocket.getOutputStream(), true);
        BufferedReader in = new BufferedReader(new InputStreamReader(
                    sslSocket.getInputStream()));
        BufferedReader stdIn = new BufferedReader(new InputStreamReader(
                    System.in));
        String userInput;
        while ((userInput = stdIn.readLine()) != null)
        {
            out.println(userInput);
            System.out.println(in.readLine());
        }
        // ...
    }
    // ...
}
```

该正确代码示例应用 SSLSocket 来使用 SSL/TLS 安全协议保护传输的报文。使用 SSLSocket 的程序如果尝试连接不使用 SSL 的端口，那么这个程序将无限期被阻塞。同理，一个不使用 SSLSocket 的程序如果要同一个使用 SSL 的端口建立连接也将会被阻塞。.

例外情况：

因为 SSLSocket 提供的报文安全传输机制性，将造成巨大的性能开销。在以下情况下，普通的套接字就可以满足需求：

- 套接字上传输的数据不敏感。
- 数据虽然敏感，但是已经过恰当加密（参考规则 4.1将敏感对象发送出信任区域前进行签名并加密）。
- 套接字的网络路径从来不越出信任边界。这种情况只有在特定的情况下才能发生。例如，套接字的两端都在同一个本例网络，而且整个网络都是可信的情况时。

规则 7.8 封装本地方法调用

说明：本地方法在 Java 中定义声明，并使用 C 或者 C++语言实现。由于代码不再遵循 Java 的策略，本地方法虽然增加了可扩展性，但是却牺牲了灵活性和可移植性。本地方法经常被用来执行平台特定的操作，与遗留的库代码对接，以及改善程序性能。为本地方法调用定义一个封装方法可以带来如下的好处：安装适当的安全管理器检查，校验传递给本地代码的参数，校验返回值，防御性的拷贝可变输入，以及净化不可信数据。因此，所有的本地方法都

应该被定义为私有的，然后仅通过一个封装方法来调用。

错误示例：

```
public final class NativeMethod
{
    // public native method
    public native void nativeOperation(byte[] data, int offset, int len);

    // wrapper method that lacks security checks and input validation
    public void doOperation(byte[] data, int offset, int len)
    {
        nativeOperation(data, offset, len);
    }

    static
    {
        // load native library in static initializer of class
        System.loadLibrary("NativeMethodLib");
    }
}
```

在以上错误代码示例中，nativeOperation()方法既是本地的也是公有的。因此，可能会被不可信调用者调用。本地方法调用会绕开安全管理器检查。本示例中包含了doOperation()方法封装器，它调用了 nativeOperation()本地方法，但是却没能进行输入验证和安全管理器检查。

正确示例：

```
public final class NativeMethodWrapper
{
    // private native method
    private native void nativeOperation(byte[] data, int offset, int len);

    // wrapper method performs SecurityManager and input validation checks
    public void doOperation(byte[] data, int offset, int len)
    {
        // permission needed to invoke native method
        securityManagerCheck();

        if (data == null)
        {
            throw new NullPointerException();
        }

        // copy mutable input
        data = data.clone();

        // validate input
        if ((offset < 0) || (len < 0) || (offset > (data.length - len)))
        {
            throw new IllegalArgumentException();
        }

        nativeOperation(data, offset, len);
    }

    static
    {
        // load native library in static initializer of class
        System.loadLibrary("NativeMethodLib");
    }
    // ...
}
```

这个示例代码将本地方法声明为私有的。doOperation()封装器方法进行许可检查，对可变输入数组进行防御性复制，并检查参数的范围。因此，nativeOperation()方法调用

将使用安全的输入参数。注意，验证检查的输出必须满足本地方法对输入的要求。

例外情况：

对于那些不要求安全管理器检查、参数验证或返回值验证、和可变输入防御性复制的本地方法（例如标准 C 函数 `int rand(void)`），不需要对其做封装。

参考资料

1. Fred Long, Dhruv Mohindra, Robert C. Seacord, Dean F. Sutherland, David Svoboda. The CERT Oracle Secure Coding Standard for Java. Addison-Wesley Professional, 2011
2. Secure Coding Guidelines for the Java Programming Language. http://www.oracle.com/technetwork/java/seccodeguide-139067.html
3. CWE&SANS TOP 25, http://www.sans.org/top25-software-errors/
4. OWASP Guide Project, https://www.owasp.org/index.php/OWASP_Guide_Project

附录 A

下表中总结了常用数据库中与 SQL 注入攻击相关的特殊字符：

数据库	特殊字符	描述	转义序列
Oracle	%	百分比：任何包括 0 或更多字符的字符串	/% escape '/'
	_	下划线：任意单个字符	/_ escape '/'
	/	斜线：转义字符	// escape '/'
	'	单引号	''
MySQL	'	单引号	\'
	"	双引号	\"
	\	反斜杠	\\
	%	百分比：任何包括 0 或更多字符的字符串	\%
	_	下划线：任意单个字符	_
DB2	'	单引号	''
	;	冒号	.
SQL Server	'	单引号	''
	[	左中括号：转义字符	[[]
	_	下划线：任意单个字符	[_]
	%	百分比：任何包括 0 或更多字符的字符串	[%]
	^	插入符号：不包括以下字符	[^]

附录 B

下表中总结了 shell 脚本中常用的与命令注入相关的特殊字符：

类型	举例	常见注入模式和结果
管道	|	| shell_command -执行命令并返回命令输出信息
内联	;	; shell_command -执行命令并返回命令输出信息
	&	& shell_command -执行命令并返回命令输出信息
逻辑运算符	$	$(shell_command) -执行命令
	&&	&& shell_command -执行命令并返回命令输出信息
	||	|| shell_command -执行命令并返回命令输出信息
重定向运算符	>	> target_file -使用前面命令的输出信息写入目标文件
	>>	>> target_file -将前面命令的输出信息附加到目标文件
	<	< target_file -将目标文件的内容发送到前面的命令

附录 C

下表列举了一些常见的敏感异常：

异常名称	信息泄露或威胁描述
java.io.FileNotFoundException	泄露文件系统结构和文件名列举。
java.util.jar.JarException	泄露文件系统结构
java.util.MissingResourceException	资源列举
java.security.acl.NotOwnerException	所有人列举
java.util.ConcurrentModificationException	可能提供线程不安全的代码信息
javax.naming.InsufficientResourcesException	服务器资源不足（可能有利于 DoS 攻击）
java.net.BindException	当不信任客户端能够选择服务器端口时造成开放端口列举。
java.lang.OutOfMemoryError	DoS
java.lang.StackOverflowError	DoS
java.sql.SQLException	数据库结构，用户名列举