HZ BOOKS 华章科技
PEARSON

资深专家撰写，Java之父James A. Gosling鼎力推荐，Java领域关于Java安全编码最权威、最全面、最详细的著作

不仅从语言角度系统而详细地阐述Java安全编码的要素、标准、规范和最佳实践，而且从架构设计的角度分析Java API存在的设计缺陷和可能存在的安全风险，以及应对策略

华章程序员书库

The CERT Oracle Secure Coding Standard for Java

Java安全编码标准

Fred Long　Dhruv Mohindra
Robert C. Seacord　Dean F. Sutherland　著
David Svoboda
计文柯 杨晓春　译

机械工业出版社
China Machine Press

华章程序员书库

Java 安全编码标准

The CERT Oracle Secure Coding Standard for Java

（美） Fred Long
Dhruv Mohindra
Robert C. Seacord
Dean F. Sutherland
David Svoboda 著

计文柯 杨晓春 译

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

Java安全编码标准 / 朗（Long，F.）等著；计文柯，杨晓春译. —北京：机械工业出版社，2013.6
（华章程序员书库）
书名原文：The CERT Oracle Secure Coding Standard for Java

ISBN 978-7-111-42818-3

I. J… II. ① 朗… ② 计… ③ 杨… III. Java语言—程序设计 IV. TP312

中国版本图书馆CIP 数据核字（2013）第120651号

本书版权登记号：图字：01-2011-7733

本书是Java安全编码领域最权威、最全面、最详细的著作，Java之父James A. Gosling推荐。不仅从语言角度系统而详细地阐述Java安全编码的要素、标准、规范和最佳实践，而且从架构设计的角度分析了Java API存在的设计缺陷和可能存在的安全风险，以及应对的策略和措施。可以将本书作为Java安全方面的工具书，根据自己的需要，找到自己感兴趣的规则进行阅读和理解，或者在实际开发中遇到安全问题时，根据书中列出的大致分类对规则进行索引和阅读，也可以通读全书的所有规则，系统地了解Java安全规则，增强对Java安全特性、语言使用、运行环境特性的理解。本书能指导Java软件工程师设计出高质量的、安全的、可靠的、强大的、有弹性的、可用性和可维护性高的软件系统。

本书内容非常全面，包括基于Java SE 6平台的一系列应用于Java语言和类库的安全编码规则，并且对这一系列规则进行了分类，包括输入数据验证、声明和初始化、表达式、数值类型和操作、面向对象、方法使用、异常处理、可见性和原子性、锁、线程、输入输出、序列化、平台安全特性、Java运行环境等重要方面，对每一个方面所涉及的安全编码要素、规范和标准进行了详细阐释。

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）
责任编辑：秦 健
印刷
2013年6月第1版第1次印刷
186mm×240mm • 33.25印张
标准书号：ISBN 978-7-111-42818-3
定 价：99.00元

译 者 序

随着软件技术的发展，特别是互联网的发展，在很多应用成为互联网应用之后，软件系统本身变成了一个开放的系统，其中也包括潜在用户数据对系统外的开放，不管是开发人员还是软件用户，除了关注软件功能实现本身，对软件安全性的关注也越来越多。

在软件开发中，由于 Java 语言具有开放性和可移植性，它逐渐成为构建应用的主要开发环境和语言之一，而对安全性的关注也开始成为一个热点话题。正如在本书序中，Java 语言之父 James Golsing 说的：“在 Java 的世界里，安全性并没有被视为一个附加功能。尽管这是一种普遍的思维方式，但如果不带着安全性去考虑问题，还是会陷入麻烦之中。”从 Golsing 的观点中，我们可以看出，对代码安全性的设计和考虑是需要系统学习的，因为这是一种对思维方式的反省。本书就是为了帮助开发人员加强对 Java 语言安全性的认识而推出的技术专著，书中结合 Java 应用的开发和 Java 语言的使用，系统地阐述了使用 Java 语言时的相关安全规则。本书内容非常全面，包括基于 Java 标准版 6.0 平台（Java SE 6）环境中一系列应用于 Java 编程语言和类库的安全编码规则，并且对这一系列规则进行了分类，包括输入数据验证、声明和初始化、表达式、数值类型和操作、面向对象、方法使用、异常处理、可见性和原子性、锁、线程、输入输出、序列化、平台安全特性、Java 运行环境等各个基本的分类部分。读者可以将本书作为 Java 安全方面的工具书，根据自己的需要，找到感兴趣的规则进行阅读和学习，或者在开发工作和软件设计中，当遇到安全问题时，根据书中列出的大致分类对规则进行检索和阅读，另外，也可以通读全书的所有规则，系统地了解 Java 安全规则，增强对 Java 安全特性、语言使用、运行环境特性的理解。在国内的技术书籍中，像本书这样系统阐述 Java 安全的书籍并不多见，因此本书值得对 Java 安全感兴趣的读者系统阅读。

译者力争以通俗通畅的中文再现原著的经典知识，希望尽自己的努力，帮助读者学习这部经典著作。但由于译者水平有限，不管是在 Java 安全的专业领域知识，还是在对原文的理解上，甚至在中文遣词造句的表达上，在翻译的作品中，肯定会在诸多方面存在疏漏之处，还请读者不吝赐教。您的意见、建议将帮助我们改善本书的质量。欢迎将关于本书的任何问题和看法发邮件到 jiwenke@gmail.com，与我们交流本书相关的信息。再次感谢！

序

James Gosling

近十年来，在计算机系统中考虑安全性已经是一个严肃的问题。过去十年的网络的爆炸性增长和我们对计算机网络互联的依赖，使安全问题提升到了一个新的层次。在 Java 的最初设计中，对安全性的处理是一个关键部分。从那时起，所有的各式各样的标准类库、框架和容器都对安全性问题有所考虑。在 Java 的世界里，安全性并没有被视为一个附加功能。尽管这是一种普遍的思维方式，但如果不带着安全性去考虑问题，还是会陷入麻烦之中。

并不仅仅因为有了基础设施的帮助，我们就可以想当然地认为安全性已经被自动考虑了。近些年来，业界发展出了一系列的安全标准和最佳实践。这本书就是一本这些实践的经验总结。它们并不是理论研究论文或者说产品市场营销的噱头。它们都是在重项目中经过严格检测的可以在企业级规模应用的产物。

前　言

在 Java 编程语言中，关键的安全编码要素是采用良好的文档和强制的编码规范。本书提供了在 Java 语言中的一系列安全编码规则。这些规则的目标是消除不安全的编码实践，因为这些不安全的因素会导致可利用的漏洞。如果应用这些安全编码标准，可以帮助设计出安全的、可靠的、健壮的、有弹性的、可用性和可维护性高的高质量系统，并且这些安全编码规范还可作为评估源代码质量特性的一个指标（不管使用的是手动的还是自动化的过程）。

对于使用 Java 编程语言开发的软件系统，这个编码规范具有广泛的影响。

本书范围

本书主要关注 Java 标准版 6.0 平台（Java SE 6）环境，其中包含一系列应用于 Java 编程语言和类库的安全编码标准。在《Java 语言规范（第 3 版）》（The Java Language Specification, 3rd edition, JLS 2005）对 Java 编程语言的行为进行了描述，该规范主要是开发本标准的主要参考资料。本标准同样涉及 Java SE 7 平台中的新特性。这些新特性主要为解决存在于 Java SE 6 和 Java SE 7 平台的安全问题提供了替代性的兼容解决方案。

C 和 C++ 语言允许一些不确定、未指定或者在实现时才确定的行为，当程序员错误假设使用的 API 和开发语言的内在行为时，它们都会导致安全漏洞。《Java 语言规范》进一步规范了标准化需求，这是因为 Java 被设计成一个“一次开发，到处运行”的语言。甚至 Java 虚拟机（JVM）或者 Java 编译器的一些行为可以由实现者自行决定。针对这些语言的特殊性，本标准也提出了安全的编码指导，以避免这些特殊性带来的问题。

仅仅关注于语言本身并不能编写出安全的软件。在 Java 应用编程接口（API）中存在的设计缺陷，在某些时候会导致这些 API 过时。在另外一些时候，这些 API 或者相关的文档还可能会被开发社区错误地解读。本标准指出了这些有问题的 API 并且强调了它们的正确使用方法，同样也包括那些广泛使用的错误设计模式（反模式）和使用习惯的例子。

Java 语言的核心和扩展 API 以及 JVM 提供的安全特性包括安全管理器、访问控制器、加密、自动内存管理、强类型检查、字节码验证等。虽然这些安全特性为大部分的应用提供了足够的安全性保证，但是对这些特性的正确使用也是非常重要的。本标准突出与设计安全架构相关的隐患和注意事项，并强调其正确的实施。坚持这个标准可以保障受信任程序的保密性、完整性和可用性（Confidentiality，Integrity，and Availability，CIA），并且有助于消除安全漏洞，这些安全漏洞会导致拒绝服务攻击、time-of-check-to-time-of-use 攻击、信息泄露、差错计算以及权限升级这些问题。

若软件遵循这些标准，则其使用者有能力定义细粒度的安全策略，并且在不受信的系统中执行受信的移动代码，或者在受信的系统中执行非受信的移动代码。

本书包含的类库

本书讨论的安全问题主要应用于 lang 和 util 类库，同时涉及 Collections、并发包、Logging、Management、反射、正则表达式、Zip、I/O、JMX、JNI、Math、Serialization 以及 XML JAXP 等类库。本标准不涉及已经发现的缺陷，不管这些缺陷已经得到修复还是因为设计本身就缺少安全性的考虑。本标准也包括功能缺陷，但只有这个缺陷发生频率高，造成相当大的安全问题，或它影响了基于核心平台的所有 Java 技术，它才会被考虑进来。本标准不仅涉及核心 API 的安全标准，而且包括一系列重要的安全考虑，涉及标准的扩展 API（javax 包）。

本书不会涉及的问题

本标准不会涉及以下问题：

- **设计和架构**。本标准假设产品的设计和架构是安全的，也就是说，产品有足够的设计水平，而不致产生危及其安全的漏洞。
- **内容**。本标准不关注那些在某种 Java 平台上出现的特殊问题，而关注广泛出现在所有平台上的问题。举例来说，如果一条规则只单独适用于 Java 微型版（Micro Edition, ME）或 Java 企业版（Enterprise Edition, EE），而不适用于 Java SE 版，通常不会包括在本标准内。在 Java SE 版中，那些处理 UI 或者在 Web 接口中提供的特性，比如声音、图形渲染、用户登录控制、会话管理、认证和授权等，不包含在本标准内。然而，这并不妨碍本标准讨论在网络中运行的 Java 系统由于不正确的用户输入验证而带来的风险、引入的问题和适当的缓解策略。
- **编码风格**。编码风格的问题是主观的，事实已给证明，我们不可能在制定一个适当的样式规则问题上达成共识。因而，本书只推荐了一些用户定义样式的规则，并且一致性地应用这些规则而已，并不强制使用某一种特定的编码规则。如果希望始终如一应用编码风格，最简单的方法是使用代码格式化工具。许多集成开发环境（IDE）提供这样的功能。
- **工具**。美国软件工程研究所（SEI）作为联邦政府资助的研发机构，不会推荐任何特定的厂商和工具，用来强制采纳标准。本书的用户可以自由选择工具，我们鼓励各个厂商为执行这些规则提供相应的工具。
- **有争议的规则**。一般情况下，本书会尽量避免列入那些缺乏广泛共识的有争议规则。

本书读者对象

本书主要是为 Java 语言开发人员服务的。本标准重点关注 Java 标准版 6.0 平台，同时可以为使用 Java ME 和 Java EE 或者其他 Java 语言版本的开发者提供参考。

尽管主要考虑系统的安全性，但是本标准对达到其他质量标准方面也是有参考价值的，比如可靠性、可用性、健壮性、可扩展性、可用性和可维护性等。

本标准还可以用于：

- 分析工具的开发人员，他们希望能够诊断出那些不安全和不合规则的 Java 程序。
- 软件开发经理、软件采购商或者其他希望建立专有的安全编码规范的软件开发及采购人员。
- 讲授 Java 安全编码标准的教师，可以使用本标准作为主要的或者参考的软件安全课程教材。

本标准中的一些规则可以扩展为某些组织的特定规则。但这种扩展应该与已有的规则一致，

并且和本标准兼容。

关于更好地使用这些安全编码标准，可以开设培训课程。当通过培训课程的考试之后，经过培训的人员可以认证为安全编码方面的专业人士。

内容和组织

本书包括一个介绍性的第 1 章和其他 17 章（每一章包含了特定领域的规则）。每一章的规则以一个列表的形式出现，并且包含对这些规则的风险评估摘要。最后是一个术语表和参考资源。可以按顺序阅读，先从本前言开始，然后是概述部分。对于规则部分的章节没有特定的阅读顺序，它们也可以作为参考资料进行查阅。每章的规则都是组织松散的，在一般情况下，可以以任意顺序阅读。

阐述规则时采用了一致的结构。每一个规则都有一个包含在标题中的唯一标识符，在规则的标题和介绍性文字部分阐述符合规则的要求。通常，在后面会有一个或一组不符合规则的代码示例，并且会给出相应的符合规则的方案。同时给出了每个规则的风险评估和具体参考书目。有的规则也列出了相关的漏洞和准则，它们来自于：

- 《CERT C 语言安全编码标准》（The CERT C Secure Coding Standard）[Seacord 2008]
- 《CERT C++ 语言安全编码标准》（The CERT C++ Secure Coding Standard）[CERT 2011]
- ISO/IEC TR 24772。信息技术–编程语言–通过语言选择和使用以避免在程序语言中出现安全漏洞的指南 [ISO/IEC TR 24772:2010]
- MITRE CWE [MITRE 2011]
- 《Java 语言安全编码规则第 3 版》[SCG 2009]
- 《Java 编程风格要素》（The Elements of Java Style）[Rogue 2000]

标识符

每个规则都包含一个唯一标识符，这个标识符包括 3 个部分：

- 3 个字母的助记符，代表标准的一部分，用来编组类似的规则，方便找到。
- 两位数值范围为 00 ～ 99 的数字，以确保每个规则都有一个唯一标识符。
- 字母 J 表明这是 Java 语言的规则，之所以包含 J，是因为可以防止出现类似 CERT 的安全编码标准也适用其他语言的情况，从而避免产生歧义。

标识符可以被静态分析工具使用，用于诊断信息的参考规则，也可以直接使用规则标题的简称。

系统质量

在考虑选择和应用编码标准的时候，安全性（security）是一个必须考虑的系统属性。另外，我们感兴趣的属性还包括无害性（safety）、可移植性（portability）、可靠性（reliability）、可用性（availability）、可维护性（maintainability）、可读性（readability）和性能（performance）等。

许多系统属性会以有趣的方式相关联。比如，可读性是可维护性的一个属性；它们对于限制在维护阶段引入缺陷都很重要，而这些缺陷往往会导致安全问题或可靠性问题。另外，可读性会为质量保证人员的代码评审提供帮助。可靠性和可用性需要正确的资源管理，而这种资源管理对提高系统的无害性和安全性有很大的帮助作用。诸如性能和安全性这样的系统特性却是相互冲突

的，它们需要有所取舍和相互平衡。

安全编码标准的目的是提高软件的安全性。然而，基于安全和其他系统属性之间的关系，如果其他系统属性也存在对安全性的显著影响，本标准也会包括对它们进行处理的要求和建议。

优先级和层级

每一个规则都分配有优先级。优先级分配使用的度量方式基于 FMECA 模式（Failure Mode, Effects, and Criticality Analysis, FMECA）[IEC 60812]。为每个规则指定 1 ～ 3 的 3 个值：

- 严重性——如果忽略该规则，后果的严重程度：
  1 = 低（拒绝访问攻击，非正常终止）
  2 = 中（破坏数据一致性，非意愿性的数据泄露）
  3 = 高（运行非指定代码，权限升级）
- 可能性——如果违反该规则，引入缺陷而导致系统出现可被利用漏洞的可能性有多大：
  1 = 不可能
  2 = 可能
  3 = 很有可能
- 整改代价——如果对现有代码进行整改，使其满足规则所需要付出的代价：
  1 = 高（手动发现及修改）
  2 = 中（自动发现及手动修改）
  3 = 低（自动发现及修改）

对每一条规则，我们把这 3 个值相乘。该结果提供了一个度量，通过这个度量可以判断应用该规则的优先程度。这个优先程度范围为 1 ～ 27。优先级在 1 ～ 4 内的规则是 3 级规则（L3），6 ～ 9 是第 2 级（L2），12 ～ 27 是 1 级规则（L1）。因此，有可能制定一个标准，实施该标准使得在一个级别的所有规则都需要得到满足，这个级别可以是 1 级、2 级或 3 级，如图 1 所示。

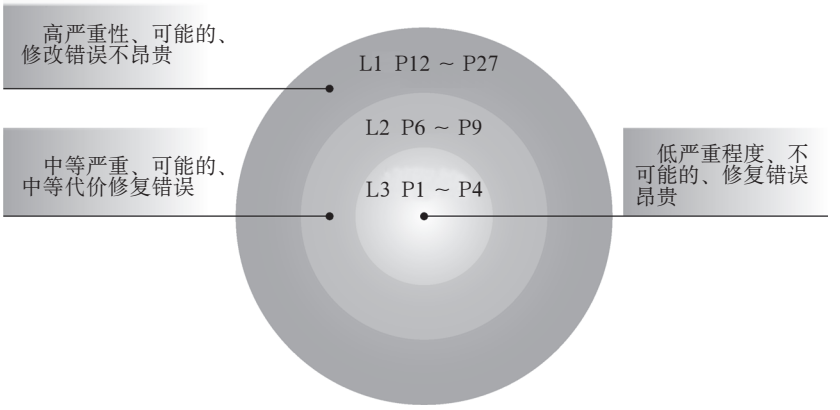

图 1　层级和优先等级

设计该度量的主要目的是表示为修正项目而付出的努力，并不表示为了实现该标准而需要付出的额外精力。

符合性测试

软件系统可以通过验证，从而符合本书中规定的规则。

规范性文字与非规范性文字

在本书中，属于编码标准的部分被视为规范性的内容，其他部分则作为建议给出。在这些规则中，规范性的描述是符合标准的要求。这些规范性描述使用强制式的语言表述，如“必须”、“应当”、“要求”。尽管对于一些规则而言，实现自动化分析并不具备可能性且并非必要，但是每一个规则的规范性部分必须是可分析的。

非规范性部分描述规则的最佳实践或有益的建议。非规范性描述并不设立符合性要求，它常常使用动词“应该”或者短语如“推荐”、“好的做法”。规则的非规范性部分可能不太适合做自动化检测，主要是因为如果在现有代码中做这种检测，可能存在过多误报。当分析新的代码时（新的代码指按照该编码规范开发的代码），自动化检测也许能够发挥作用。

本书中所有的规则都有一个非规范性部分。这个部分仅应用于以下场合：

- 遵循广为人知的最佳实践。
- 如果这条规则描述的方式被广泛采用，对以下两种情况都不产生影响：一是不会违反规则的规范性部分，二是该方法在应用到代码时，是无害的，虽然那些代码并不适用于那些规范性部分。

整个非规范性指导不包含在本编码规则内。然而，本书的作者正在计划发布这些非规范性指导。

自动化分析

为了确保源代码符合本安全编码标准，需要做必要的检查。检查的最有效方法是使用一个或多个分析工具（分析仪）。当一个规则不能由工具检查时，则必须采用人工审核方式。

本书中的许多规则会提示是否已经存在分析工具。这些分析工具可以诊断是否违反规则，甚至可以提示该规则是否可以做自动化检测。这部分的信息可能会有些变化，因为现有的检测工具正在发展中，同时也正开发出新的检测工具。

当选择一个源代码分析工具的时候，理想的状态是，这个工具能够适用于本规范尽可能多的规则。并不是所有的规则都需要自动化检测工具，还是有一些规则需要通过手动检测才能完成的。

完整性和可信性

对于规则来说，一个分析工具在最大程度上应该是完整的，并且是可信的。如果在检测中不会给出假阴性结果，也就说，它能够在一个程序中检测出违反该规则的所有问题，那么称其为可信的。如果不会给出假阳性结果或者误报，则认为该工具的检测是完整的。对于特定的规则，它可能产生的结果如下所示：

表 1　可信性和完整性

		假阳性	
		Y	N
假阴性	N	假阳性可信	可信并完整
	Y	假阳性不可信	不可信

如果工具有太多的假阳性结果，那么它会浪费开发人员的时间，让他们失去对结果的兴趣，结果是无法得到有用的价值，找到真正的缺陷，因为这些缺陷都淹没在噪声当中了。如果工具有太多假阴性结果，那么会错过许多缺陷，而形成一种虚假的安全感。在实践中，我们需要在这两种情况之间找到平衡点。

在减少假阴性和假阳性上有许多权衡。很显然，最好能够两者都减少，这在一定程度上可以通过许多技巧和算法来帮助实现。

分析应该是一个可信的过程，这意味着我们可以依赖于分析工具的输出结果。因此，开发人员必须保证存在这种信任。在理想情况下，这可以通过工具供应商运行认可的测试来完成。但也存在这种情况，就是若没有正式的验证方案，使用验证套件测试分析工具也是可能的。

CERT 源代码分析实验室

CERT 创建的源代码分析实验室（Source Code Analysis Laboratory，SCALe），为软件系统提供了适用于 CERT 安全编码标准的一致性测试，这种测试包括用于 Java 的 CERT Oracle 安全编码标准。

SCALe 使用多种分析手段评估用户的源代码，包括静态分析工具、动态分析工具、模糊测试等。CERT 会向开发人员报告违反安全编码规范的地方。开发人员可以修复这些缺陷，然后再提交软件进行第二次评估。

在开发人员解决 SCALe 测试出的问题之后，SCALe 团队会决定这个版本的软件是否符合 CERT 标准，并且给开发人员颁发证书，并将该软件系统列入通过该标准的列表。

一个成功的一致性测试表明，SCALe 分析并不能检测出所有违反 CERT 标准中定义的规则。成功的一致性测试也无法保证不会违反这些规则，并且也不能保证这些整个软件永远是安全的。SCALe 也不能够测试那些未知的安全漏洞，不能找出概要设计和架构设计的问题，以及代码的运行环境或者可移植性的问题。通过测试的软件仍然可能是不安全的，例如，如果软件实现的是一个不安全的架构和设计。

在本书的规则讨论中，包括了规则的特例以及这些特例产生的情况。当开发人员从一条该规则中引用这些特例时，他们必须在代码中标识出相关的特例信息。至少需要通过注释的方式在注释中标出该特例，如下所示：

```
// MET12-EX0 applies here
```

作者目前正在开发一套 Java 注释，它将允许程序员以可读的和静态分析工具可以访问的方式标识这种特例。通过一致性测试，可以确定异常是否适用于其他情况，这是由SCALe的分析师来完成的。

第三方类库

静态分析工具，如 FindBugs 这个 Java 字节码分析工具，经常可以在第三方类库和自定义代

码中发现违反安全编码标准的行为。出现在第三方类库中违反安全规则的处理方式是和自定义代码一样的。

遗憾的是，开发人员并不是总能修改第三方类库代码或者说不能说服供应商修改代码。在这种情况下，系统就不能通过一致性测试，除非解决这些问题（将替换该类库，或者开发一个自己的类库）或者将这些问题以文档形式记录下来。对第三方类库中出现的这些问题的处理方式和自定义代码的处理方式是一样的，也就是说，开发人员必须自己能够证明虽违反这些规则中但不会导致安全漏洞。然而它们的代价是各不相同的，对于自定义代码，修改问题可能更经济，而对于第三方类库，文档化问题可能更容易。

一致性测试流程

对于每一个编码标准，源代码分为证实不符、符合和完全符合这几种情况，它们适用于本书中的每一条规则。

- **证实不符**。如果出现一条或多条违例，并且不允许出现偏差，那么称为证实不符。
- **符合**。如果没有发现一条违例，则称为符合。
- **完全符合**。如果验证满足规则的所有情况，那么认为该代码完全符合。

偏差规则

严格遵守所有的规则是不可能的，因此，定义违反特定规则的偏差是必要的。偏差应用在下面这个场合，在这个场合中，一个真阳性的问题作为违例提出，但并不能就此判断代码就是不安全的。这可能是软件的设计和架构特性导致的结果，或者存在这个违例，安全编码规范并没有考虑到这种情况。在这种情况下，定义允许的偏差可以防止标准过度严格。但偏差不能用于解释系统性能、可用性和其他不安全性方面的问题。成功通过一致性测试的一套软件系统，必须不存在编码错误导致的已知漏洞。

偏差要求由主任评估师进行评估，如果开发人员可以提供足够的证据，表示该偏差不会引入安全漏洞，那么可以接纳这个偏差请求。偏差不应该经常使用，因为修复代码缺陷总比证明代码没有缺陷要容易，并且不会导致系统安全漏洞。

一旦已完成评估过程，会有报告详细说明代码是否符合或者不符合安全编码标准的相应规则，并会提供给开发人员。

CERT SCALe 认证

被 CERT 认可符合安全编码规范的开发组织可以在其网站上使用如图 2 所示的标识。该标识必须指定通过一致性测试的软件，该标识不适用于未经测试的软件、公司和组织。

除了满足以下条件的软件的补丁，对软件进行任何修改之后，都需要进行一致性设计。直至重新测试该软件，并且认定满足一致性要求之后，这个打过补丁的软件才能使用 CERT SCALe 标识。

满足以下 3 个标准的软件补丁不违反一致性设计：

图 2　CERT SCALe 标识

- 该补丁是在代码中修复安全漏洞所必需的，并且是软件维护所必需的。
- 该补丁不引入新的功能特性。
- 该补丁不引入违反软件之前通过的那些安全编码规范规则的问题。

使用CERT SCALe标识之后，可以列入软件组织列表，与卡耐基-梅隆大学签订服务协议，软件会被认定为符合CERT标准。更多信息请联系：securecoding@cert.org。

致　　谢

向为本书成功付梓而付出努力的每个人表示感谢。

贡献者

Siddarth Adukia、Lokesh Agarwal、Ron Bandes、Scott Bennett、Kalpana Chatnani、Steve Christey、Jose Sandoval Chaverri、Tim Halloran、Thomas Hawtin、Fei He、Ryan Hofler、Sam Kaplan、Georgios Katsis、Lothar Kimmeringer、Bastian Marquis、Michael Kross、Masaki Kubo、Christopher Leonavicius、Bocong Liu、Efstathios Mertikas、Aniket Mokashi、David Neville、Todd Nowacki、Vishal Patel、Jonathan Paulson、Justin Pincar、Michael Rosenman、Brendan Saulsbury、Eric Schwelm、Tamir Sen、Philip Shirey、Jagadish Shrinivasavadhani、Robin Steiger、Yozo Toda、Kazuya Togashi、John Truelove、Theti Tsiampali、Tim Wilson 和 Weam Abu Zaki。

评审者

Daniel Bögner、James Baldo Jr.、Hans Boehm、Joseph Bowbeer、Mark Davis、Sven Dietrich、Will Dormann、Chad R. Dougherty、Holger Ebel、Paul Evans、Hari Gopal、Klaus Havelund、David Holmes、Bart Jacobs、Sami Koivu、Niklas Matthies、Bill Michell、Philip Miller、Nick Morrott、Attila Mravik、Tim Peierls、Kirk Sayre、Thomas Scanlon、Steve Scholnick、Alex Snaps、David Warren、Ramon Waspitz 和 Kenneth A. Williams。

编辑

Pamela Curtis、Shannon Haas、Carol Lallier、Tracey Tamules、Melanie Thompson、Paul Ruggerio 和 Pennie Walters。

Addison-Wesley 出版社

Kim Boedigheimer、John Fuller、Stephane Nakib、Peter Gordon、Chuti Prasertsith 和 Elizabeth Ryan。

特别感谢

Archie Andrews、David Biber、Kim Boedigheimer、Peter Gordon、Frances Ho、Joe Jarzombek、Jason McNatt、Stephane Nakib、Rich Pethia 和 Elizabeth Ryan。

目　录

译者序
序
前言
致谢

第1章　概述 …… 1

1.1　错位的信任 …… 1
1.2　注入攻击 …… 2
1.3　敏感数据泄露 …… 3
1.4　效能泄露 …… 5
1.5　拒绝服务 …… 6
1.6　序列化 …… 8
1.7　并发性、可见性和内存 …… 8
1.8　最低权限原则 …… 14
1.9　安全管理器 …… 15
1.10　类装载器 …… 16
1.11　小结 …… 16

第2章　输入验证和数据净化（IDS） …… 17

规则 …… 17
风险评估概要 …… 17
2.1　IDS00-J净化穿越受信边界的非受信数据 …… 18
2.2　IDS01-J验证前标准化字符串 …… 26
2.3　IDS02-J在验证之前标准化路径名 …… 28
2.4　IDS03-J不要记录未经净化的用户输入 …… 31
2.5　IDS04-J限制传递给ZipInputStream的文件大小 …… 33
2.6　IDS05-J使用ASCII字符集的子集作为文件名和路径名 …… 35
2.7　IDS06-J从格式字符串中排除用户输入 …… 37
2.8　IDS07-J不要向Runtime.exec() 方法传递非受信、未净化的数据 …… 38

2.9 IDS08-J净化传递给正则表达式的非受信数据 …… 41
2.10 DS09-J如果没有指定适当的locale，不要使用locale相关方法处理与locale相关的数据 …… 44
2.11 IDS10-J不要拆分两种数据结构中的字符串 …… 45
2.12 IDS11-J在验证前去掉非字符码点 …… 50
2.13 IDS12-J在不同的字符编码中无损转换字符串数据 …… 51
2.14 IDS13-J在文件或者网络I/O两端使用兼容的编码方式 …… 53

第3章 声明和初始化（DCL） …… 56

规则 …… 56
风险评估概要 …… 56
3.1 DCL00-J防止类的循环初始化 …… 56
3.2 DCL01-J不要重用Java标准库的已经公开的标识 …… 59
3.3 DCL02-J将所有增强for语句的循环变量声明为final类型 …… 60

第4章 表达式（EXP） …… 63

规则 …… 63
风险评估概要 …… 63
4.1 EXP00-J不要忽略方法的返回值 …… 63
4.2 EXP01-J不要解引用空指针 …… 65
4.3 EXP02-J使用两个参数的Arrays.equals()方法来比较两个数组的内容 …… 67
4.4 EXP03-J不要用相等操作符来比较两个基础数据类型的值 …… 67
4.5 EXP04-J确保使用正确的类型来自动封装数值 …… 72
4.6 EXP05-J不要在一个表达式中对同一变量进行多次写入 …… 73
4.7 EXP06-J不要在断言中使用有副作用的表达式 …… 76

第5章 数值类型与运算（NUM） …… 78

规则 …… 78
风险评估概要 …… 78
5.1 NUM00-J检测和避免整数溢出 …… 79
5.2 NUM01-J不要对同一数据进行位运算和数学运算 …… 85
5.3 NUM02-J确保除法运算和模运算中的除数不为0 …… 88
5.4 NUM03-J使用可容纳无符号数据合法取值范围的整数类型 …… 89
5.5 NUM04-J不要使用浮点数进行精细计算 …… 90
5.6 NUM05-J不要使用非标准化数 …… 92
5.7 NUM06-J使用strictfp修饰符确保跨平台浮点运算的一致性 …… 94
5.8 NUM07-J不要尝试与NaN进行比较 …… 97

5.9 NUM08-J检查浮点输入特殊的数值 …… 98
5.10 NUM09-J不要使用浮点变量作为循环计数器 …… 100
5.11 NUM10-J不要从浮点字元构造BigDecimal对象 …… 101
5.12 NUM11-J不要比较或者审查以字符串表达的浮点数值 …… 102
5.13 NUM12-J确保将数值转换成较小类型时不会产生数据丢失或曲解 …… 103
5.14 NUM13-J转换基本整数类型至浮点类型时应避免精度损失 …… 107

第6章 面向对象（OBJ） …… 110

规则 …… 110
风险评估概要 …… 110
6.1 OBJ00-J只有受信子类能对具有不变性的类和方法进行扩展 …… 111
6.2 OBJ01-J声明数据成员为私有并提供可访问的封装器方法 …… 116
6.3 OBJ02-J当改变基类时，保存子类之间的依赖关系 …… 118
6.4 OBJ03-J在新代码中，不要混用具有泛型和非泛型的原始数据类型 …… 124
6.5 OBJ04-J为可变类提供复制功能，并通过此功能允许将实例传递给非受信代码 …… 128
6.6 OBJ05-J在返回引用之前，防御性复制私有的可变的类成员 …… 132
6.7 OBJ06-J对可变输入和可变的内部组件创建防御性复制 …… 136
6.8 OBJ07-J不允许敏感类复制其自身 …… 138
6.9 OBJ08-J不要在嵌套类中暴露外部类的私有字段 …… 141
6.10 OBJ09-J比较类而不是类名称 …… 143
6.11 OBJ10-J不要使用公有静态的非final变量 …… 144
6.12 OBJ11-J小心处理构造函数抛出异常的情况 …… 146

第7章 方法（MET） …… 153

规则 …… 153
风险评估概要 …… 153
7.1 MET00-J验证方法参数 …… 154
7.2 MET01-J不要使用断言验证方法参数 …… 156
7.3 MET02-J不要使用弃用的或过时的类和方法 …… 157
7.4 MET03-J进行安全检测的方法必须声明为private或final …… 158
7.5 MET04-J不要增加被覆写方法和被隐藏方法的可访问性 …… 160
7.6 MET05-J确保构造函数不会调用可覆写的方法 …… 161
7.7 MET06-J不要在clone()中调用可覆写的方法 …… 163
7.8 MET07-J不要定义类方法来隐藏基类或基类接口中声明的方法 …… 165
7.9 MET08-J确保比较等同的对象能得到相等的结果 …… 167
7.10 MET09-J定义了equlas()方法的类必须定义hashCode()方法 …… 174
7.11 MET10-J实现compareTo()方法时遵守常规合约 …… 176

7.12 MET11-J确保比较中的关键码是不可变的 …… 178
7.13 MET12-J不要使用析构函数 …… 182

第8章 异常行为（ERR） …… 187

规则 …… 187
风险评估概要 …… 187
8.1 ERR00-J不要消除或忽略可检查的异常 …… 187
8.2 ERR01-J不能允许异常泄露敏感信息 …… 192
8.3 ERR02-J记录日志时应避免异常 …… 196
8.4 ERR03-J在方法失败时恢复对象先前的状态 …… 197
8.5 ERR04-J不要在finally程序段非正常退出 …… 201
8.6 ERR05-J不要在finally程序段中遗漏可检查异常 …… 202
8.7 ERR06-J不要抛出未声明的可检查异常 …… 205
8.8 ERR07-J不要抛出RuntimeException、Exception或Throwable …… 209
8.9 ERR08-J不要捕捉NullPointerException或任何它的基类 …… 210
8.10 ERR09-J禁止非受信代码终止JVM …… 216

第9章 可见性和原子性（VNA） …… 219

规则 …… 219
风险评估概要 …… 219
9.1 VNA00-J当需要读取共享基础数据类型变量时，需要保证其可见性 …… 219
9.2 VNA01-J保证对一个不可变对象的共享引用的可见性 …… 222
9.3 VNA02-J保证对于共享变量的组合操作是原子性的 …… 225
9.4 VNA03-J即使每一个方法都是相互独立并且是原子性的，也不要假设一组调用是原子性的 …… 230
9.5 VNA04-J保证串联在一起的方法调用是原子性的 …… 235
9.6 VNA05-J保证在读写64位的数值时的原子性 …… 239

第10章 锁（LCK） …… 241

规则 …… 241
风险评估概要 …… 241
10.1 LCK00-J通过私有final锁对象可以同步那些与非受信代码交互的类 …… 242
10.2 LCK01-J不要基于那些可能被重用的对象进行同步 …… 246
10.3 LCK02-J不要基于那些通过getClass()返回的类对象来实现同步 …… 249
10.4 LCK03-J不要基于高层并发对象的内置锁来实现同步 …… 252
10.5 LCK04-J即使集合是可访问的，也不要基于集合视图使用同步 …… 253
10.6 LCK05-J对那些可以被非受信代码修改的静态字段，需要同步进入 …… 255

10.7 LCK06-J不要使用一个实例锁来保护共享静态数据 …… 256
10.8 LCK07-J使用相同的方式请求和释放锁来避免死锁 …… 258
10.9 LCK08-J在异常条件时，保证释放已经持有的锁 …… 266
10.10 LCK09-J不要执行那些持有锁时会阻塞的操作 …… 270
10.11 LCK10-J不要使用不正确形式的双重锁定检查惯用法 …… 273
10.12 LCK11-J当使用那些不能对锁策略进行承诺的类时，避免使用客户端锁定 …… 277

第11章 线程API（THI） …… 282

规则 …… 282
风险评估概要 …… 282
11.1 THI00-J不要调用Thread.run() …… 282
11.2 THI01-J不能调用ThreadGroup方法 …… 284
11.3 THI02-J通知所有等待中的线程而不是单一线程 …… 287
11.4 THI03-J始终在循环中调用wait()和await()方法 …… 292
11.5 THI04-J确保可以终止受阻线程 …… 295
11.6 THI05-J不要使用Thread.stop()来终止线程 …… 300

第12章 线程池（TPS） …… 304

规则 …… 304
风险评估概要 …… 304
12.1 TPS00-J使用线程池处理流量突发以实现降低性能运行 …… 304
12.2 TPS01-J不要使用有限的线程池来执行相互依赖的任务 …… 307
12.3 TPS02-J确保提交至线程池的任务是可中断的 …… 312
12.4 TPS03-J确保线程池中正在执行的任务不会失败而不给出任何提示 …… 315
12.5 TPS04-J使用线程池时，确保ThreadLocal变量可以重新初始化 …… 318

第13章 与线程安全相关的其他规则（TSM） …… 323

规则 …… 323
风险评估概要 …… 323
13.1 TSM00-J不要使用非线程安全方法来覆写线程安全方法 …… 323
13.2 TSM01-J不要让this引用在创建对象时泄漏 …… 326
13.3 TSM02-J不要在初始化类时使用后台线程 …… 332
13.4 TSM03-J不要发布部分初始化的对象 …… 336

第14章 输入输出（FIO） …… 342

规则 …… 342
风险评估概要 …… 342

14.1 FIO00-J不要操作共享目录中的文件 …… 343
14.2 FIO01-J使用合适的访问权限创建文件 …… 351
14.3 FIO02-J发现并处理与文件相关的错误 …… 352
14.4 FIO03-J在终止前移除临时文件 …… 354
14.5 FIO04-J在不需要时关闭资源 …… 357
14.6 FIO05-J不要使用wrap()或duplicate()创建缓存，并将这些缓存暴露给非受信代码 … 361
14.7 FIO06-J不能在一个单独的InputStream上创建多个缓存区封装器 …… 364
14.8 FIO07-J不要让外部进程阻塞输入和输出流 …… 367
14.9 FIO08-J对读取一个字符或者字节的方法，使用int类型的返回值 …… 370
14.10 FIO09-J不要使用write()方法输出超过0～255的整数 …… 372
14.11 FIO10-J使用read()方法保证填充一个数组 …… 373
14.12 FIO11-J不要将原始的二进制数据作为字符数据读入 …… 375
14.13 FIO12-J为小端数据的读写提供方法 …… 376
14.14 FIO13-J不要在受信边界之外记录敏感信息 …… 379
14.15 FIO14-J在程序终止时执行正确的清理动作 …… 381

第15章 序列化（SER） …… 387

规则 …… 387
风险评估概要 …… 387
15.1 SER00-J在类的演化过程中维护其序列化的兼容性 …… 388
15.2 SER01-J不要偏离序列化方法的正确签名 …… 390
15.3 SER02-J在将对象向信任边界之外发送时，需要签名并且封装敏感对象 …… 392
15.4 SER03-J不要序列化未经加密的敏感数据 …… 397
15.5 SER04-J不要允许序列化和反序列化绕过安全管理器 …… 401
15.6 SER05-J不要序列化内部类实例 …… 404
15.7 SER06-J在反序列化时，对私有的可变的组件进行防御性复制 …… 405
15.8 SER07-J不要对实现定义的不可变因素使用默认的序列化格式 …… 406
15.9 SER08-J在从拥有特性的环境中进行反序列化之前最小化特权 …… 410
15.10 SER09-J不要从readObject()方法中调用可以被覆写的方法 …… 413
15.11 SER10-J在序列化时，避免出现内存和资源泄漏 …… 414
15.12 SER11-J防止覆盖外部化的对象 …… 415

第16章 平台安全性（SEC） …… 417

规则 …… 417
风险评估概要 …… 417
16.1 SEC00-J不要允许特权代码块越过受信边界泄露敏感信息 …… 417
16.2 SEC01-J不要在特权代码块中使用污染过的变量 …… 420

16.3 SEC02-J不要基于非受信源进行安全检查 …… 422
16.4 SEC03-J不要在允许非受信代码装载任意类之后装载受信类 …… 424
16.5 SEC04-J使用安全管理器检查来保护敏感操作 …… 426
16.6 SEC05-J不要使用反射来增加类、方法和字段的可访问性 …… 429
16.7 SEC06-J不要依赖于默认的由URLClassLoader和java.util.jar提供的自动化签名检查 …… 434
16.8 SEC07-J当编写一个自定义的类装载器时调用基类的getPermissions()方法 …… 437
16.9 SEC08-J定义基于原生方法的封装器 …… 438

第17章 运行环境（ENV） …… 441

规则 …… 441
风险评估概要 …… 441
17.1 ENV00-J不要签名只执行非特权操作的代码 …… 441
17.2 ENV01-J将所有安全敏感的代码置于单独一个jar包中，并且在签名之后封装它 …… 443
17.3 ENV02-J不要信任环境变量的值 …… 446
17.4 ENV03-J不要赋予危险的权限组合 …… 448
17.5 ENV04-J不要关闭字节码验证功能 …… 451
17.6 ENV05-J不要部署一个被远程监视的应用 …… 452

第18章 其他（MSC） …… 457

规则 …… 457
风险评估概要 …… 457
18.1 MSC00-J在交换安全数据时使用SSLSocket而不是Socket …… 457
18.2 MSC01-J不要使用空的无限循环 …… 461
18.3 MSC02-J生成强随机数 …… 462
18.4 MSC03-J不要硬编码敏感信息 …… 464
18.5 MSC04-J防止内存泄漏 …… 466
18.6 MSC05-J不要耗尽堆空间 …… 473
18.7 MSC06-J当一个遍历正在进行时，不要修改它对应的集合 …… 477
18.8 MSC07-J防止多次实例化单例对象 …… 481

术语表 …… 490

参考资源 …… 497

第1章 概　述

关于软件漏洞和软件漏洞利用的报告越来越多，并继续以惊人的速度增长。相当多的这类报告导致了技术安全上的警报。这一威胁日益严重影响了公司、教育机构、政府和个人，为了解决这一问题，开发出来的系统应该是不包含那些软件安全漏洞的。

编码错误会导致大多数软件安全漏洞。比如，2004 年在国家安全漏洞数据库（National Vulnerability Database）中记录的近 2500 个漏洞，有 64% 是由错误的编程引起的 [Heffley 2004]。

相对来说，Java 是一种较为安全的语言。因为它没有显式指针操作；对数组和字符串边界有自动检查机制；如果尝试引用一个空指针会抛出系统异常；算术运算是明确定义的且与平台无关，类型转换也是如此。内置的字节码验证器可以确保这些检查会在正确的地方执行。此外，Java 还提供全面的、细粒度的安全机制，这个安全机制控制着对单个文件、套接字和其他敏感资源的访问。为从这种安全机制中获益，JVM（Java Virtual Machine, Java 虚拟机）特地设计了一个安全管理器。这个安全管理器是类 java.lang. Security Manager（或它的子类）的一个对象，我们可以基于它进行编程，但通过命令行参数来操作它是更为常见的使用方式。

尽管 Java 在编程上是安全的，但这并不意味可以一劳永逸。本章的剩余部分会介绍一些例子，通过这些例子我们可以看到，在某些情况下，程序可能会被利用而产生安全问题，同时会介绍怎样借助规则的帮助来减轻安全攻击对程序的影响。并不是所有的规则都适用于所有的 Java 语言所编写的程序；通常，规则的适用性取决于软件的部署情况和受信的假设。

1.1　错位的信任

软件程序往往包含多个组件作为子系统，其中每个组件会在一个或多个受信域中运行。例如，一个组件可以访问文件系统，但不允许访问网络，而另一组件可以访问网络，但不能访问文件系统。非信任解耦（distrustful decomposition）及权限分离（privilege separation）[Dougherty 2009] 是安全设计模式的例子，它意味着首先需要减少需要授权的代码的数量，这就要在设计系统时，使用不互信的组件，并保证组件在特定的权限下运行。

当软件组件需要遵循安全策略时，可以允许它们跨越受信的边界来传输数据，对于任何一个组件，它们不能指定其自身的受信级别。受信的边界应该由应用程序的部署者来确定，要完成这件事情，则需要一个全系统级别的安全策略的帮助。安全审核员可以使用这样的安全定义，通过安全定义来确定该软件能否充分支持应用系统安全目标的实现。

一个 Java 程序可以包含自行开发和第三方开发的代码，Java 被设计成允许执行非受信代码；因此，第三方代码可以运行在它自己的受信域中。可以认为使用公共的 API 接口的第三方代码已

经处在受信边界之中。任何跨越受信边界的数据都应接受验证，除非产生这些数据的代码本身已经能够提供权限有效性的保证。一个用户或使用者可以在数据流入受信边界时，省略对数据的验证，当然前提是其受信边界是适当的。在所有其他的情况下，进入数据必须接受验证。

1.2　注入攻击

当一个组件从超出该组件受信边界的外部数据源接收数据的时候，这些数据可能是恶意的，并且会导致注入攻击，如图 1-1 所示。

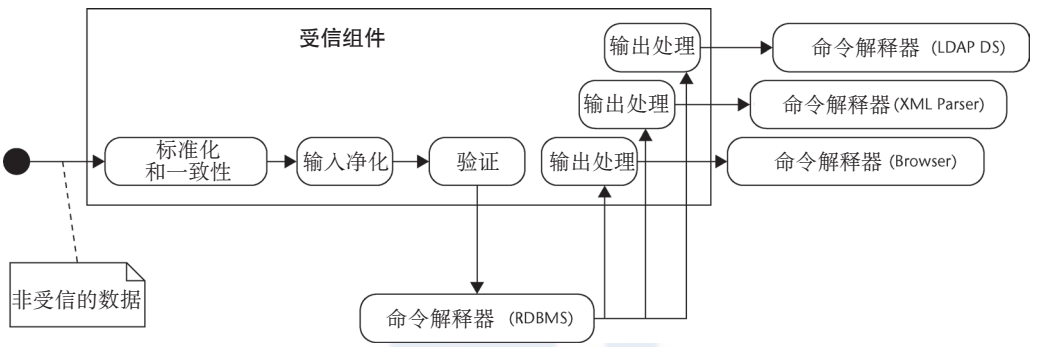

图 1-1　注入攻击

程序必须采取以下几个步骤，从而通过受信边界而收到的数据确保是适当的并且是没有恶意的。这些步骤包括：

验证（Validation）：验证是指这样一个过程，该过程可以保证输入的数据处在预先设定好的有效的程序输入范围之内。这就要求这些输入符合类型和数值范围的要求，并且对各个类别和子系统来说，输入不会发生变化。

净化（Sanitization）：在许多情况下，数据是从另一个受信域直接传递到组件当中来的。数据净化的过程是指，通过这个过程，可以确保数据符合接收该数据的子系统对数据的要求。数据净化也涉及如何确保数据符合安全性的要求，在这些要求中，需要考虑数据泄漏问题，或者需考虑在数据输出跨出受信边界的时候，敏感数据的暴露问题。净化可以通过消除不必要的字符输入来完成，比如对字符进行删除、更换、编码或转义操作。净化可能在数据输入（输入净化）之后或数据跨受信边界传输之前（输出净化）进行。数据净化和输入验证可能是并存的，并且是相互补充的。关于数据净化的详细情况说明，请参见规则 IDS01-J。

标准化（Canonicalization）和归一化（Normalization）：标准化的过程是指将输入最小损失地还原成等价的最简单的已知形式。归一化的过程是一个有损转换的过程，在这个过程中，输入数据会转化成用最简单的（可预期）形式来表现。标准化和归一化必须在数据验证之前进行，从而防止攻击者利用验证例程来除去不合法的字符，并由此构建一个不合法的（或者是有潜在恶意的）字符序列。关于这方面的更多信息，请参见规则 IDS02-J。归一化应当在所有用户输入都收集完整之后进行。不要归一化那些不完整的数据，或者将经过归一化处理的数据和没有经过归一化处理的数据合并起来。

有一种情况需要特殊考虑，就是复杂的子系统接受代表特定操作命令或指令的字符串数据的问题。组件接受这些字符串数据的时候，在这些字符串数据中，可能包含的特殊字符会触发命令或动作，从而造成软件的安全漏洞。

以下是一些可以对命令进行解释或者对指令进行操作的组件的例子：

- 操作系统的命令解释器（见规则 IDS07-J）
- 具备 SQL 兼容接口的数据库
- XML 解析器
- XPath 评估器
- 基于轻量级目录访问协议（Lightweight Directory Access Protocol, LDAP）的目录服务
- 脚本引擎
- 正则表达式（regex）编译器

在必须要把数据发送到处在另一个受信域的组件的时候，发送者必须确保数据经过适当的编码处理，从而在穿过受信边界的时候，满足数据接收者受信边界的要求。例如，尽管一个系统已经被恶意代码或数据渗透了，但如果系统的输出是经过适当转义和编码处理的，那么还是可以避免许多攻击的。

1.3 敏感数据泄露

系统的安全策略需要确定哪些信息是敏感的。敏感数据可能包括用户信息，比如社会保障或信用卡号码、密码或私钥。在不同等级受信域的组件中共享数据的时候，我们称这些数据是跨越受信边界的。因为在 Java 环境中，允许在同一个程序中的处在不同受信域的两个组件进行数据通信，从而会出现那些跨受信边界的数据传输。所以，如果在域中存在一个授权用户，而该用户没有数据接收权限，那么系统必须保证这些数据不会发送给处于该域的组件。如果有可能的话，这种处理只简单地禁止数据传输，也有可能对穿越受信边界的数据进行处理，将敏感数据过滤掉，如图 1-2 所示。

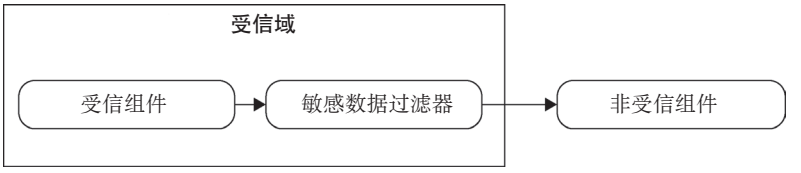

图 1-2 数据过滤

在很多情况下，Java 软件组件会输出敏感信息。以下规则可以帮助减轻敏感信息泄露的风险：

规 则
ERR01-J 不能允许异常泄露敏感数据
FIO13-J 不要在受信边界之外记录敏感信息
IDS03-J 不要记录未经净化的用户输入

（续）

规　　则
MSC03-J 不要硬编码敏感信息
SER03-J 不要序列化未经加密的敏感数据
SER04-J 不要允许序列化和反序列化绕过安全管理器
SER06-J 在反序列化时，对私有的可变的组件进行防御性复制

在 Java 中，接口、类和类成员（如字段和方法）是访问受控的。这种访问受控由访问修饰符（public、private 或 protected）来标识，或者可以通过不使用访问控制符来标识（使用默认的访问控制，也称为 package-private 访问）。

Java 的类型安全是指，在一个字段中被声明为 private 或者 protected 的时候，或者是在默认的保护情况下（package），它是不允许被全局访问的。然而，有一些 Java 平台的设计会让这种保护失效，从而导致出现安全漏洞，比如对 Java 反射机制的错误使用。对于 Java 高手来说，这些漏洞的出现并不奇怪，因为它们在文档中都已做了详细的描述，但一不小心，还是会掉入陷阱。比如说，如果一个字段被声明为 public，那么它可以被 Java 程序中的任意一段代码直接访问，也可以被 Java 程序中的任意一段代码修改（除非这个数据成员同样被声明为 final）。很显然，敏感数据是不允许保存在 public 域中的，因为如果有人可以直接访问该程序运行所在的 JVM 的时候，就会招致问题。

表 1-1 展现了一个简化的关于访问控制规则的视图。x 表示在这个地方允许特定的访问。例如，在“class”中，如果标识了 x，表示在同一类中声明的代码可以访问该成员。同样，“package”一栏中，如果标识了 x，那么表示，这个成员可以被任何定义在同一个包中的类（或者子类）访问，并且该成员可以在由同一个类装载器载入的类（或者子类）中定义。而同一个类加载器的条件仅适用于 package-private 成员访问的情况。

表 1-1　访问控制规则

访问控制关键字	Class	Package	Subclass	World
Private	x			
None	x	x	x①	
Protected	x	x	x②	
Public	x	x	x	x

① 同一个包中的子类同样可以访问那些没有指定访问标识符的数据成员（默认或者包私有的可见性）。对于访问的另一个额外的要求是，子类必须由那些包含包私有数据成员的类的类装载器载入。不同包中的子类不能访问这样的包私有的数据成员。

② 如果要引用一个protected数据成员，访问代码必须包含在定义了这个protected数据成员的类中，或者在定义这个类的子类中。允许子类访问，即使在没有考虑子类所在包位置的情况下。

类和类成员只能给予可能的最低的访问权限，这样的话，恶意代码威胁系统安全的可能性就会大大减少。只要有可能，应当避免使用接口来开放那些包含（或调用）敏感代码的方法；接口应当只允许可以公开访问的方法，而这些方法往往是这个类公开的应用编程接口（API）的一部

分。（注意，这和 Bloch 推荐的优先考虑 API 接口设计的看法相反 [Bloch 2008，Item 16]）。存在一种例外情况，就是在实现一个不可修改的接口的时候，可以开放一个可变对象的公共的不可变部分（见规则 OBJ04-J）。此外，值得注意的是，即使一个非 final 类的可见性是 package-private，它仍然容易被错误使用，特别是当它包含公共方法的时候。对于那些执行了所有必要的安全检查，并对所有输入进行了净化的方法，它们同样是可以通过接口开放出来的。

对顶级类来说，protected 的访问控制是无效的，虽然对嵌套类来说，可以声明它为 protected。对那些非 final 的公共类的字段来说，它们是不能被声明为 protected 的，这是为了防止在另一个包中该类的子类的非受信代码可以访问该成员。此外，如果 protected 成员是这个类的 API 的一部分，则需要继续得到支持。规则 OBJ01-J 要求声明私有字段。

当一个类、接口、方法或者一个字段作为 API 的一部分进行发布的时候，比如说对网络服务接入点来说，它可以被声明为 pubic。其他类和成员应当要么声明为 package-private，要么声明为 private。例如，如果一个类在安全性上的考虑并不是至关重要，它应该定义一个 public 静态工厂来实现实例控制，这可以通过使用 private 的构造器来实现。

1.4 效能泄露

效能（capability）指的是在授权中可以进行沟通而不会被忘记的标识。效能这个词是由 Dennis 和 Van Horn[Dennis 1966] 引入的。它指向的是一个数值，这个数值指向的对象拥有一系列的对象访问权限。在一个基于效能的操作系统中的用户程序，必须使用效能来访问对象。

每一个 Java 对象都有不会被遗忘的标识。因为 Java 的 == 操作符会对引用等进行测试，它会被用来测试这个标识。因为有这个不会被遗忘的标识，所以允许使用一个对象引用作为标记，从而作为某种形式的动作需要的授权的一个不会被遗忘的证据 [Mettler 2010a]。

授权会包含在对象引用中，这些对象引用被作为效能来使用。授权指那些运行代码带来的任何效果，而这些运行代码也许会带来其他的副作用。授权不仅包括那些对于外部资源例如文件或者网络套接字的操作，而且包含了那些在程序不同部分进行共享的可变数据结构的操作 [Mettler 2010b]。

对于那些方法会执行敏感操作的对象的引用而言，它们会允许持有者执行这些操作的效能（或者要求一个对象代表这些持有者执行这些操作）。所以，这样的引用自身必须要被视为敏感数据，并且不能泄露给非受信代码。

一个很有可能让人感到惊讶的泄露效能和数据的源头是内隐类，这些类可以访问它们包含的类的所有字段。Java 字节码缺少内置的对内隐类的支持，因此，内隐类必须编译成带有样式名称的普通类，如 OuterClass$InnerClasss。因为内隐类必须能访问它们的包含类的私有字段，所以对这些字段的访问控制，就被转换到对字节码中的包的访问来进行。因而，手写的字节码可以访问这些名义上的私有字段（请以“Java 字节码工程的安全方面”[Schonefeld2002] 作为例子进行参考）。

以下是考虑效能的规则：

规　则
ERR09-J 禁止非受信代码终止 JVM
MET04-J 不要增加被覆写方法和被隐藏方法的可访问性
OBJ08-J 不要在嵌套类中暴露外部类的私有字段

（续）

规　则
SEC00-J 不要允许特权代码块越过受信边界泄露敏感信息
SEC04-J 使用安全管理器检查来保护敏感操作
SER08-J 在从拥有特性的环境中进行反序列化之前最小化特权

1.5　拒绝服务

拒绝服务攻击试图使计算机的资源不可获得，或者对需要使用该计算机资源的用户来说，会造成资源不足的情况。通常这种攻击是持续服务的服务器系统需要重点关注的，它与桌面应用程序有很大区别；然而，拒绝服务的问题可以出现在所有类别的应用上。

1.5.1　资源耗尽型的拒绝服务

拒绝服务可能出现在，相对于输入数据需要的资源消耗来说，使用比例上更为巨大的资源。通过客户端软件检查资源是否被过度消耗，并希望用户来处理与资源相关的问题是不合理的。但是，存在这样的客户端软件，它们可以检查那些可能会导致持久拒绝服务的输入，比如将文件系统进行填充的操作。

《Java 安全编码指南》（Secure Coding Guidelines for the Java Programming Language）[SCG 2009] 中列举了可能的攻击的例子：

- 请求一个大的矢量图片，如 SVG 文件或者字体文件。
- “Zip 炸弹”（Zip bomb）那些如 ZIP、GIF 或那些经过 gzip 编码的 HTML 内容，会因为解压而消耗大量的资源。
- “XML 解析炸弹”，在解析的时候，在扩展 XML 所包含的实体时，有可能会使得 XML 文档急剧增长。可以通过设置 XMLConstants.FEATURE_SECURE_PROCESSING 功能并设置合理的限度值解决前面的问题。
- 过度使用的磁盘空间。
- 在一个散列表中插入了多个密钥，而这些密钥使用相同的散列码。这样会导致最差的性能（O（n^2））而不是平均性能（O(n)）。
- 发起许多连接，服务器为每个连接分配大量的资源（例如传统的洪水攻击）。

针对拒绝服务攻击并防止出现资源耗尽的规则，有以下几点：

规　则
FIO03-J 在终止前移除临时文件
FIO04-J 在不需要时关闭资源
FIO07-J 不要让外部进程阻塞输入和输出流
FIO14-J 在程序终止时执行正确的清理动作
IDS04-J 限制传递给 ZipInputStream 的文件大小
MET12-J 不要使用解析函数
MSC04-J 防止内存泄漏
MSC05-J 不要耗尽堆空间

（续）

规　则
SER10-J 在序列化时避免出现内存和资源泄露
TPS00-J 使用线程池处理流量突发以实现降低性能运行
TPS01-J 不要使用有限的线程池来执行相互依赖的任务
VNA03-J 即使每一个方法都是相互独立并且是原子性的，也不要假设一组调用是原子性的

1.5.2 与并发相关的拒绝服务

某些拒绝服务攻击是通过试图引入并发问题来实现的，如线程死锁、线程饥饿和竞态。针对拒绝服务攻击并防止出现并发问题的规则，有以下几点：

规　则
LCK00-J 通过私有 final 锁对象可以同步那些与非受信代码交互的类
LCK01-J 不要基于那些可能被重用的对象进行同步
LCK07-J 使用相同的方式请求和释放锁来避免死锁
LCK08-J 在异常条件下，保证释放已持有的锁
LCK09-J 不要执行那些持有锁时会阻塞的操作
LCK11-J 当使用那些不能对锁策略进行承诺的类时，避免使用客户端锁定
THI04-J 确保可以终止受阻线程
TPS02-J 确保提交至线程池的任务是可中断的
TSM02-J 不要在初始化类时使用后台线程

1.5.3 其他的拒绝服务攻击

其他预防拒绝服务攻击的规则如下：

规　则
ERR09-J 禁止非受信代码终止 JVM
IDS00-J 净化穿越受信边界的非受信数据
IDS06-J 从格式字符串中排除用户输入
IDS08-J 净化传递给正则表达式的非受信数据

1.5.4 拒绝服务的前提

其他的规则会处理安全漏洞，这些安全漏洞会导致拒绝服务攻击，但它们自己并不足以导致拒绝服务：

规　则
ERR01-J 不能允许异常泄露敏感信息
ERR02-J 记录日志时应避免异常
EXP01-J 不要解引用空指针
FIO00-J 不要操作共享目录中的文件
NUM02-J 确保除法运算和模运算中的除数不为 0

1.6 序列化

序列化使得 Java 程序中对象的状态能被抓取并写入到字节流中 [Sun 2004b]。这使得该对象的状态能够保存下来，所以可以在未来需要的时候恢复（通过反序列化）。序列化可以通过网络使用 RMI 的方式调用 Java 方法，这些对象被编组（序列化），在分布部署的虚拟机之间进行交换，然后进行解组（反序列化）。序列化还广泛使用在 Java Beans 中。

可以用以下方式序列化对象：

```
ObjectOutputStream oos = new ObjectOutputStream(
    new FileOutputStream("SerialOutput"));
oos.writeObject(someObject);
oos.flush();
```

可以用以下的方式反序列化对象：

```
ObjectInputStream ois = new ObjectInputStream(
    new FileInputStream("SerialOutput"));
someObject = (SomeClass) ois.readObject();
```

序列化能够捕捉对象中的所有非暂态字段，包括那些正常情况下不能读取的非公有字段，这通过 object 类的 Serializable 接口来实现。序列化的值写入字节流之后，如果字节流是可读的，那么原先那些正常情况下不能访问的字段的值也就可以通过推导得出。此外访问，当反序列化一个类时，原来保存的值可能会被修改或者伪造。

引进一个安全管理器并不能防止正常情况下无法访问的字段被序列化和反序列化（如果字节流需要被存储或发送，那么必须给予它从文件或网络进行读写的权限）。网络流量（包括 RMI）可以得到保护，但是，这需要使用 SSL/TLS 协议。

在序列化和反序列化对象时需要进行特殊处理的类可以实现有以下签名的方法 [API 2006]：

```
private void writeObject(java.io.ObjectOutputStream out)
  throws IOException;
private void readObject(java.io.ObjectInputStream in)
  throws IOException, ClassNotFoundException;
```

如果一个 Serializable 类没有重写 writeObject() 方法的实现，这个对象会使用默认的方法来完成序列化，它会序列化所有的 public、protected、package-private，以及 private 字段，除了 transient 字段。同样，当 Serializable 类没有重写 readObject() 方法的实现时，这个对象会反序列化所有的 public、protected，以及 private 字段，除了那些 transient 字段。这个问题会在规则 SER01-J 中深入探讨。

1.7 并发性、可见性和内存

可以在不同线程之间共享的内存称为共享内存（shared memory）或内存堆（heap memory）。本节使用变量（variable）这个名词来代表字段和数组元素 [JLS2005]。在不同的线程中共享的变量称为共享变量。所有的实例字段、静态字段以及数组元素作为共享变量存储在共享内存中。局部变量、形式方法参数以及异常例程参数是从来不能在线程之间共享的，不会受到内存模型的影响。

在现代多处理器共享内存的架构下，每个处理器有一个或多个层次的缓存，会定期地与主存

储器进行协同，如图 1-3 所示。

之所以开放对共享变量的数据写入会带来问题，是因为在共享变量中的数值是会被缓存的，而且把这些数值写入主存会有延迟。然后，其他的线程可能会读取这个变量的过时的数值。

更多的顾虑不仅在于并行执行的代码通常是交错的，同时也在于编译器或运行时系统会对执行语句进行重新排序来优化性能。这会导致执行次序很难从源代码中得到验证。不能记录可能的重排序，这是一个常见的数据竞态的来源。

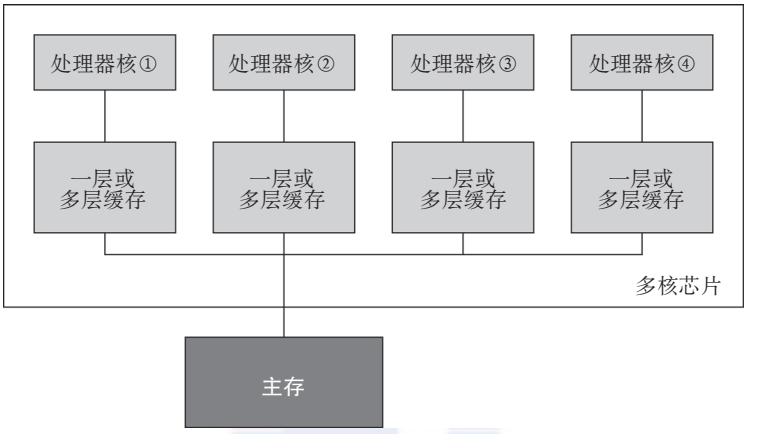

图 1-3 多处理器架构中的共享内存

举例来说，a 和 b 是两个全局（共享）变量或实例域，而 r1 和 r2 是两个局部变量，r1 和 r2 是不能被其他的线程读取的。在初始化状态下，令 a=0，b=0。

线程 1	线程 2
a = 10;	b = 20;
r1 = b;	r2 = a;

在线程 1 中，完成两个赋值：a=10 和 r1=b，这两个赋值是不相关的，所以编译器在编译的时候，运行时系统可以任意安排它们的执行次序。这两个赋值在线程 2 中也有可能是任意安排次序的。尽管可能看起来难以理解，但是 Java 的内存模型允许读取一个刚刚写入的数值，而这次写入显然与执行的次序有关。

以下是在实际赋值时可能的执行次序：

执行次序（时间）	线程号	赋值操作	赋值	备注
1	t_1	a = 10;	10	
2	t_2	b = 20;	20	
3	t_1	r1 = b;	0	读入 b 的初始值 0
4	t_2	r2 = a;	0	读入 a 的初始值 0

在这个次序中，r1 和 r2 分别读取变量 b 和 a 的初始值，但它们实际上希望得到的是更新过

的值：20 和 10，而以下是实际赋值时另一种可能的执行次序：

执行次序（时间）	线程号	赋值操作	赋值	备注
1	t_1	r1 = b；	20	读取后来写入的值（在第 4 步）20
2	t_2	r2 = a；	10	读取后来写入的值（在第 3 步）10
3	t_1	a = 10；	10	
4	t_2	b = 20；	20	

在这个次序中，r1 和 r2 读取 b 和 a 的值，b 和 a 的值分别是在步骤 4 和步骤 3 赋予的，甚至是在对应的这些步骤被执行之前的那些语句赋予的。

如果能够明确代码可能的执行次序，那么对于代码的正确性，会有更大的把握。

当语句在一个线程中依次执行的时候，由于存在缓存，会使最新的数值没有在主存中体现。

《Java 语言规范》（Java Language Specification JLS）中定义了 Java 内存模型（Java Memory Model, JMM），它为 Java 开发人员提供了一定程度的保障。JMM 在定义的行为包括变量的读写、锁定和解锁的监视、线程的开始和会合。JMM 对在程序中的所有动作定义了一种称为 happens-before 的部分次序化动作。它能保证一个线程执行时动作 B 可以看到动作 A 的执行结果，例如，可以说 A 和 B 的关系是一种 happens-before 的关系，A 在 B 之前发生。

根据 JSL17.4.5 节对“Happens-before”的描述：

1）对监视器的解锁需要 happens-before 每一个接下来的对监视器的锁定。

2）对一个 volatile 域的写入需要 happens-before 每一个接下来的对该域的读取。

3）对一个线程的 Thread.start() 调用需要 happens-before 对这个启动线程的任何操作。

4）对一个线程的所有操作，需要 happens-before 从该线程 Thread.join() 引起的其他任何线程的正常返回。

5）任何对象的默认初始化需要 happens-before 该程序的任何其他操作（除了初始化的写入操作之外）。

6）一个线程对另一个线程的中断需要 happens-before 被中断线程检测到该中断。

7）一个对象的构造方法的结束需要 happens-before 这个对象的销毁器的开始。

在两个操作不存在 happens-before 关系的时候，JVM 可以对它们的执行重新排序。当一个变量被至少一个线程写入，并且被至少一个线程读取的时候，如果这些读写不存在 happens-before 关系，数据的竞态会出现。正确同步的程序是不会出现数据的竞态的。JMM 可以通过同步程序来保证其次序是一致的。次序一致是指任何执行结果都是一样的，比如当所有的线程按照任何特定的顺序对一个共享数据执行读写，这个序列中对每个线程的操作都是程序指定的顺序 [Tanenbaum 2003] 时，它们的执行结果都是同样的。换句话说：

1）每个线程都执行读和写操作，并把这些操作按照线程执行的次序进行排列（线程顺序）。

2）以某种方式安排这些操作，使它们在执行次序上是 happens-before 关系。

3）读操作必须返回最新写入的数据，在整个程序执行次序中，可以保证序列化的一致性。

4）这意味着任何线程都可以看到同样的对共享变量进行访问的次序。

如果程序的次序被遵从并且所有数据读取符合内存模型，那么对于实际的指令执行和内存读

写次序来说，会有所不同。这使得开发人员可以理解他们编写的程序的语义，并且允许编译器开发者和虚拟机的实现有不同的优化方式 [JPL 2006]。

这一系列并发原语可以帮助开发人员对多线程程序的语义有所理解。

1.7.1 关键词 volatile

如果声明一个共享变量为 volatile，那么可以保证它的可见性并且限制对访问它的操作进行重新排序。比如递增该变量的时候，不能保证 volatile 访问这个操作组合的原子性。因而，当必须保证操作组合的原子性时，是不能够使用 volatile 的（可以参考 CON02-J 规则获取详细信息）。

声明一个 volatile 变量即建立一种 happens-before 关系，例如，当一个线程写入一个 volatile 变量后，随后读取该变量的线程总会看到这个写入线程。写入这个 volatile 变量之前执行的语句 happens-before 任何对这个 volatile 变量的读操作。

考虑两个线程执行以下语句的情况，如图 1-4 所示。

线程 1 和线程 2 存在 happens-before 关系，因为线程 2 不能在线程 1 结束之前开始。

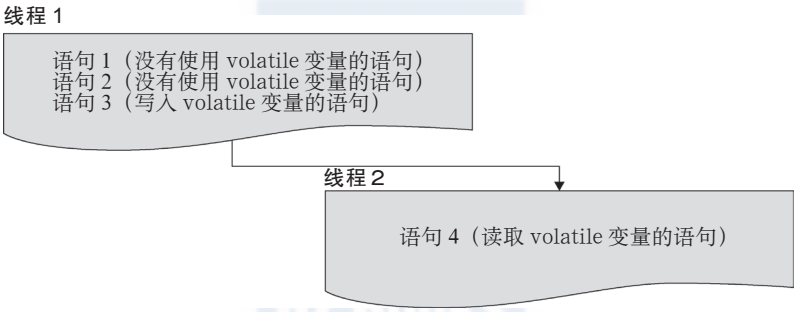

图 1-4 volatile 的读和写操作

在这个例子中，语句 3 写入一个 volatile 变量，语句 4（在线程 2 中）读取该 volatile 变量。这个读取可以得到语句 3 最新写入的数据（对同一个变量 v）。

对 volatile 的读与写操作不能重新排序，要么依次读写它，要么使用非 volatile 变量。当线程 2 读取 volatile 变量的时候，它会得到所有的写入结果，而这些写入结果是在线程 1 写入该 volatile 变量之前发生的。由于需要相对有力的对 volatile 特性的保证，其性能开销几乎和同步是一样的。

在前面的例子中，并不能保证同一个程序中的两条语句按照它们在程序中出现的次序执行。如果在这两个语句之间不存在 happens-before 关系的话，它们可能会被编译器以任意的次序进行重排。

表 1-2 总结了所有对 volatile 和非 volatile 变量进行重排序的可能性。其中的 load/store 操作可以和 read/write 操作对应 [Lea 2008]。

注意，如果在变量上加上 volatile 关键词的话，它会保证可见性和执行次序。也就是说，它仅应该适用于原始字段和对象引用。如果实际成员是一个对象引用本身，可以保证这一点；如果是一个对象，而它的引用是一个 volatile 类型，那么这一点是不能保证的。因而，声明一个对象引用是 volatile 不足以保证对所引用的成员的改变是可见的。这样的话，一个线程可能不能读取

另一个线程对这个引用对象成员字段最新写入。此外，当一个引用是可变的并且不是线程安全，那么其他线程可能只能看到一个对象只是部分地被创建，或者这个对象会处在一个（临时）不一致的状态当中 [Goetz 2007]。然而，当一个引用是不可变的时候，声明该引用是 volatile 已经足够保证该引用成员的可见性。

表 1-2　volatile 和非 volatile 变量之间可能的重排序

可以重排序	第二项操作			
第一项操作	正常载入	正常存储	volatile 载入	volatile 存储
正常载入	是	是	是	否
正常存储	是	是	是	否
volatile 载入	否	否	否	否
volatile 存储	是	是	否	否

1.7.2　同步

一个正确进行同步的程序，可以保证执行的一致次序，并且不会产生数据竞态情况。下面的例子通过使用一个非 volatile 变量 x 和一个 volatile 变量 b 来说明如何没有正确同步。

线程 1	线程 2
x = 1	r1 = y
y = 2	r2 = x

在这个例子中，有两种序列上一致的执行次序。

步骤（时间）	线程号	语句	备注
1	t_1	x =1	写入非 volatile 变量
2	t_1	y = 2	写入 volatile 变量
3	t_2	r1 = y	读取 volatile 变量
4	t_2	r2 = x	读取非 volatile 变量

以及

步骤（时间）	线程号	语句	备注
1	t_2	r1 = y	读取 volatile 变量
2	t_2	r2 = x	读取非 volatile 变量
3	t_1	x =1	写入非 volatile 变量
4	t_1	y = 2	写入 volatile 变量

在第一种情况下，步骤 1 和步骤 2 总是发生在步骤 3 和步骤 4 之前，这是一种 happens-before 关系。然而，第二种顺序一致的执行情况在任何步骤之间都缺乏 happens-before 关系。因此，这个例子中存在数据的竞态。

正确的可见性可以保证多个线程在访问共享数据时，得到相互之间的结果，但是不能在每一个线程读写数据的时候、建立起次序。正确的同步可以实现正确的可见性，并且可以保证线程写以特定的次序访问数据。例如，从下面的代码可以看出，线程 1 的所有操作都在线程 2 的所有操

作之前执行，这时保证了一个一致的执行次序。

```
class Assign {
 public synchronized void doSomething() {
   // If in Thread 1, perform Thread 1 actions
   x = 1;
   y = 2;
   // If in Thread 2, perform Thread 2 actions
   r1 = y;
   r2 = x;
 }
}
```

使用同步的时候，不需要声明变量 y 是 volatile 的。同步涉及取得锁、执行操作、释放锁等过程。在前面的例子中，doSomething() 方法需要获得一个类对象 Assign 的内部锁。这个例子同样可以使用块同步来实现。

```
class Assign {
 public void doSomething() {
  synchronized (this) {
    // If in Thread 1, perform Thread 1 actions
    x = 1;
    y = 2;
    // If in Thread 2, perform Thread 2 actions
    r1 = y;
    r2 = x;
  }
 }
}
```

这两个例子都使用了内部锁。对象的内部锁也可以用作监视器。释放一个对象的内部锁总是在下一次获得该对象的内部锁之前发生，这是一种 happens-before 关系。

1.7.3 java.util.concurrent 类

原子类 volatile 变量对保证可见性很有用。但是，它不能保证原子性。同步可以保证原子性，但它们会产生上下文切换的额外开销，并且经常会出现锁竞争。java.util.concurrent.atomic 包中的原子类提供了一种机制，可以在同时需要保证原子性时，在大多数环境下减少锁竞争。根据 Goetz 及其同事的研究，“在低和中等的竞争时，原子操作提供更好的可扩展性；在高竞争时，锁操作可以避免高竞争”[Goetz 2006a]。

atomic 类开放了通用的功能接口，因而开发人员可以充分利用现代处理器的 compare-swap 指令提供的执行效率。例如，AtomicInteger.incrementAndGet() 方法支持对一个变量的原子加，其他高层方法如 java.util.concurrent.atomic.Atomic*.compareAndSet()（其中 Atomic 可以是 Integer、Long 或 Boolean 类型）为开发者提供了一个简单的抽象接口，通过这个接口，同样可以方便地使用处理器级别的指令。

在 java.util.concurrent 辅助包中，倾向于使用 volatile 变量，而不是使用传统的诸如 synchronized 同步方法，如 synchronized 关键字和 volatile 变量，因为这些辅助包抽象了底层的细节，提供了一个更简单且更少错误的 API，这样更容易扩展，并在一定的策略下可以加强其作用。

执行器框架 通过使用执行器框架，java.util.concurrent 包提供了任务并发执行的机制。这些任

务可以是由实现了 Runnable 或者 Callable 接口的类来封装的一个逻辑执行单元。这个执行器框架将任务提交与底层的任务管理和调度细节分离开。它还提供了线程池机制，通过这个线程池，在系统需要同时处理超过其处理能力的请求时，系统不致崩溃。

执行器框架的核心接口是 Executor 接口，它扩展自 ExecutorService 接口。ExecutorService 接口提供了线程池的终止机制，并且可以获得任务的返回值。ExecutorService 还被 ScheduledExecutorService 扩展了，这个 ScheduledExecutorService 接口提供了可以让运行中的任务周期性或延时执行。Executor 类提供了若干工厂和辅助方法，通过这些方法可以提供 Executor、ExecutorService 和其他接口需要的通用配置。例如，Executors.newFixedThreadPool() 方法可以返回确定大小的线程池，为在线程池中并发执行的任务数目确定一个上限，并在线程池满载时，维护一个任务队列。线程池的基本（实际）实现是由 ThreadPoolExecutor 类来完成的。这个类可以被实例化来定制任务执行策略。

显式锁　java.util.concurrent 包中的 ReentrantLock 类提供了隐含锁所没有的功能特性。举例来说，调用 ReentrantLock.tryLock() 方法会立即返回持有锁的另一个线程对象。在 JMM 的定义中，获取或释放一个 ReentrantLock 对象与获取或释放一个隐含锁是一样的。

1.8　最低权限原则

根据最低权限原则，系统的每一个程序和用户只应赋予能够完成它们操作所需要的最低权限 [Saltzer 1974，Saltzer 1975] 。“构建安全的网站”（Build Security In Website）[DHS 2006] 这篇文章补充说明了这个原则。以最低权限运行可以降低因为代码中的安全漏洞而带来的安全问题的严重性。

体现最低权限原则的规则如下所示：

规　则
ENV03-J 不要赋予危险的权限组合
SEC00-J 不要允许特权代码越过受信边界泄露敏感信息
SEC01-J 不要在特权代码块中使用污染过的变量

定义权限的所谓安全策略必须有严格的控制。当 Java 程序有安全管理器时，通过默认的安全策略文件进行权限控制；然而，Java 灵活的安全模型允许用户赋予应用更多的权限，这可以通过自定义安全策略来完成。

Java 代码若想提高权限的话，需要使用代码签名。很多安全策略允许具备签名的代码以提高后的权限来执行。只有那些需要提升权限的代码需要签名，对其他代码来说，是不需要签名的（请参考规则 ENV00-J。）

在同一个 JAR 文件内，已签名的代码会与未签名的类共存。建议将所有的特权代码打成一个包（详情请参考规则 ENV01-J）。此外，根据安全策略，可以基于代码或签名器为代码赋予特定权限。

特权操作权限应该只提供给那些最少的需要特权的代码。Java 的 AccessController 机制允许只有需要的代码可以获得权限提升。当一个类需要主张其权限时，在 doPrivileged() 代码块中，执行特权代码即可。AccessController 机制需要与安全策略一起发挥作用。由于用户可能不清楚

安全模型的相关细节，所以并不能根据它们的需求正确配置安全策略，在 doPrivileged() 代码块中出现的特权代码必须是最少的，从而避免出现安全漏洞。

1.9 安全管理器

SecurityManager 是 Java 中定义安全策略的类。当一个程序在没有安装安全管理器的环境中运行时，它是不受限制的，它可以使用任何 Java API 提供的类和方法。当使用安全管理器时，它会明确哪些不安全和敏感的操作是允许的。任何违反安全策略的操作都会导致抛出 SecurityException 异常，代码可以向安全管理器查询某些动作是否允许。同时，安全管理器可以控制受信的 Java API 能够执行的功能。当非受信的代码不允许读取系统类的时候，应该赋予这些类最低的权限，以防止特定包中的那些受信类被它们使用。而 accessClassInPackage 权限只提供那些所需要的功能。

某一类应用中存在着一些预先设定好的安全管理器。applet 安全管理器就能够管理所有的 Java applet。它拒绝除了赋予重要权限的所有 applet，以防止不经意的系统修改、信息泄露以及用户模拟。

安全管理器不仅仅只限于对客户端应用的保护。对于 Web 服务器，如 Tomcat 和 WebSphere，可以将安全管理器用来隔离 servlets 和 JSP（Java Server Page）代码中出现的恶意木马，以及保护敏感的系统资源，使其不能被随便访问。

对于在命令行状态下运行的 Java 应用，一个自定义的安全管理器会被设置一个特殊的标识。同样，也可以通过编程式的方法安装一个安全管理器。这样就可以建立一个默认的沙箱，通过安全策略来允许或者拒绝那些敏感操作。

在 Java 2 平台之前，安全管理器是一个抽象类。在 Java 2 平台之后，由于不作为抽象类出现，所以应用时不需要显式重写它的方法。如果编程式地创建安全管理器，代码需要有运行态的权限来运行 createSecurityManager 方法，从而实例化安全管理器并且通过运行 setSecurityManager 来安装 SecurityManager。当已经安装安全管理器时，将会检查这些权限。当存在一个全局默认的安全管理器时，例如在虚拟宿主环境，或者在一个单独的宿主环境中，需要用一个定制的安全管理器来代替默认的安全管理器，从而拒绝以前那些能够通过的权限的情况。

安全管理器和 AccessController 类紧密相关。前者是一个访问控制中枢，而后者是访问控制算法的实际实现。安全管理器支持以下几种情况：

- 提供后向兼容性：历史遗留的代码通常包含许多定制化的安全管理器的实现，因为最开始安全管理器是被设计为抽象的。
- 设定自定义的策略：设计一个安全管理器的子类，以设定自定义的安全策略（例如，多层次、粗略或者细粒度的）。

在考虑实现和使用自定义的安全管理器的时候，对应于默认的安全管理器，在 Java Security Architecture Specification[SecuritySpec 2008] 中提到：

定制一个安全管理器（通过设计它的子类）应当是最后的手段，并且需要特别谨慎，我们鼓励在应用代码中使用 AccessController 。此外，一个定制的安全管理器，比如它会在调用标准的安全检查之前做当天的时间检查，它可以而且应该适当的利用 AccessControllerwhenever 所提供的算法。

许多 Java SE API 在进行敏感操作之前会通过调用安全管理器来进行默认的检查。例如，在调用者不具备读文件权限的时候，java.io.FileInputStream 的构造函数会抛出 SecurityException 异常。因为 SecurityException 是 RuntimeException 的子类，一些 API 方法的声明并不需要声明它们抛出 RuntimeException 异常，但是一些 API 方法不能这样。例如，在 java.io.FileReader 中就没有声明它会抛出 SecurityException 异常。除非在 API 方法文档里特别指明，否则应该避免依赖对安全管理器是否存在进行的检查。

1.10　类装载器

java.lang.ClassLoader 类及其子类可以让新代码动态地加载到 JVM 中。每一个类都提供了装载它的 ClassLoader 的链接。此外，每一个类装载器类都有一个装载它的父类装载器，类装载器的顶端称为根类装载器。因为 ClassLoader 被设计成抽象的，所以它不能被实例化。所有的继承自 SecureClassLoader 的类装载器都继承自 ClassLoader。SecureClassLoader 对其方法进行的权限安全检查，在它的子类中也会同样进行。SecureClassLoader 定义了 getPermissions() 方法，通过这个方法可以指明被类装载器装载的类所具有的权限。通过这种方式提供的保护机制，可以限制非受信的代码装载额外的类。

遗憾的是，被不同的类装载器载入的类总是不同的。为了那些非受信代码的安全性，package-private（即默认的）访问权限可以认为和 private 访问权限是一样的。

1.11　小结

尽管作为一种相对安全的语言，Java 语言及其类库还是在很大程度上存在着一些编程问题，从而造成系统安全漏洞。如果假设 Java 本身提供的功能特性可以减少一般的软件问题，并足够 Java 程序安全得不需要进行进一步检测，那么我们就大错特错了。因为任何在软件实现中出现的缺陷都会产生严重的安全影响，绷紧安全性这根弦是非常关键的，这样，当我们进行系统开发和部署时，就可以避免出现软件的安全漏洞问题。

为了减少因为编程错误所带来的安全漏洞的可能性，Java 开发人员应当遵循本编码标准中的安全编码规则，并遵循其他适用的安全编码指南。

第2章 输入验证和数据净化（IDS）

规则

规　则
IDS00-J 净化穿越受信边界的非受信数据
IDS01-J 验证前规范化字符串
IDS02-J 在验证之前标准化路径名
IDS03-J 不要记录未经净化的用户输入
IDS04-J 限制传递给 ZipInputStream 的文件大小
IDS05-J 使用 ASCII 字符集的子集作为文件名和路径名
IDS06-J 从格式字符串中排除用户输入
IDS07-J 不要向 Runtime.exec() 方法传递非受信、未净化的数据
IDS08-J 净化传递给正则表达式的非受信数据
IDS09-J 如果没有指定适当的 locale，不要使用 locale 相关方法处理与 locale 相关的数据
IDS10-J 不要拆分两种数据结构中的字符串
IDS11-J 在验证前去掉非字符码点
IDS12-J 在不同的字符编码中无损转换字符串数据
IDS13-J 在文件或者网络 I/O 两端使用兼容的编码方式

风险评估概要

规则	严重性	可能性	弥补代价	优先级	级别
IDS00-J	高	可能	中等	P12	L1
IDS01-J	高	可能	中等	P12	L1
IDS02-J	中	不可能	中等	P4	L3
IDS03-J	中	可能	中等	P8	L2
IDS04-J	低	可能	高等	P2	L3
IDS05-J	中	不可能	中等	P4	L3
IDS06-J	中	不可能	中等	P4	L3
IDS07-J	高	可能	中等	P12	L1
IDS08-J	中	不可能	中等	P4	L3

（续）

规则	严重性	可能性	弥补代价	优先级	级别
IDS09-J	中	可能	中等	P8	L2
IDS10-J	低	不可能	中等	P2	L3
IDS11-J	高	可能	中等	P12	L1
IDS12-J	低	可能	中等	P4	L3
IDS13-J	低	不可能	中等	P2	L3

2.1　IDS00-J 净化穿越受信边界的非受信数据

许多程序会接受来自未经验证的用户数据、网络连接，以及其他来自于非受信域的不可信数据，然后它们会将这些数据（经过修改或不经修改）穿过受信边界送到另一个受信域中。通常，这些数据是以字符串的形式出现的，并且它们会有既定的内部数据结构，这种数据结构必须能够被子系统所解析。同时，必须净化这些数据，因为子系统可能并不具备处理恶意输入信息的能力，这些未经净化数据输入很有可能包含注入攻击。

值得注意的是，程序必须对传递给命令行解释器或者解析器的所有字符串数据进行净化，这样才能保证解析器处理后或者解释器处理后的字符串是无害的。

许多命令行解释器和解析器提供了自己的数据净化和数据验证的方法。当存在这样的方法时，应当优先考虑它们，因为相对而言，自定义的净化方法会忽略一些特殊的情况，或者会忽略在解析器中所隐含着的那些复杂性。另一个问题是，当有新的功能添加到命令行解释器或者添加到解析器软件的时候，自定义的处理方法可能并不能得到很好的维护。

2.1.1　SQL 注入

当初始的 SQL 查询被修改成另一个完全不同形式的查询的时候，就会出现 SQL 注入漏洞。执行这一被修改过的查询，可能会导致信息泄露或者数据被修改。防止 SQL 注入漏洞的主要方法是，净化并验证非受信输入，同时采用参数化查询的方法。

假设一个数据库具有用户名和密码数据，它可以用这些数据来对系统用户进行认证。每个用户名使用长度为 8 的字符串表示，密码使用长度为 20 的字符串表示。

一个用来验证用户的 SQL 命令如下所示：

```
SELECT * FROM db_user WHERE username='<USERNAME>' AND
                            password='<PASSWORD>'
```

如果它可以返回任何记录，那么意味着用户名和密码是合法的。

然而，如果攻击者能够替代 <USERNAME> 和 <PASSWORD> 中的任意字符串，它们可以使用下面的关于 < USERNAME > 的字符串进行 SQL 注入：

```
validuser' OR '1'='1
```

当将其注入到命令时，命令就会变成：

```
SELECT * FROM db_user WHERE username='validuser' OR '1'='1' AND
password=<PASSWORD>
```

如果 validuser 是一个有效的用户名，那么这条选择语句会选择出表中的 validuser 记录。这

个操作中不会使用密码，因为 username='validuser' 的判断条件为真；因此，不会检查那些在 OR 后面的条件。所以，只要在 OR 后面的部分是一个语法正确的 SQL 表达式，攻击者就可以由此获得 validuser 的访问权限。

同样，攻击者可以为 <PASSWORD> 提供字符串：

```
' OR '1'='1
```

这将会产生以下的命令：

```
SELECT * FROM db_user WHERE username='' AND password='' OR '1'='1'
```

这一次，‘1’ = ‘1’ 是永远成立的，它使得用户名和密码的验证是无效的，并且攻击者可以不需要正确的用户名或密码就能登录。

2.1.2 不符合规则的代码示例

在这个不符合规则的代码示例中，系统使用 JDBC 代码来认证用户。密码通过 char 数组传入，建立数据库连接，然后进行哈希编码。

遗憾的是，这段代码会出现 SQL 注入问题，因为在 SQL 语句中，sqlString 允许输入未经净化的输入参数。如前所述的攻击场景将再次出现。

```
class Login {
  public Connection getConnection() throws SQLException {
    DriverManager.registerDriver(new
          com.microsoft.sqlserver.jdbc.SQLServerDriver());
    String dbConnection =
      PropertyManager.getProperty("db.connection");
    // can hold some value like
    // "jdbc:microsoft:sqlserver://<HOST>:1433,<UID>,<PWD>"
    return DriverManager.getConnection(dbConnection);
  }

  String hashPassword(char[] password) {
    // create hash of password
  }

  public void doPrivilegedAction(String username, char[] password)
                                 throws SQLException {
    Connection connection = getConnection();
    if (connection == null) {
      // handle error
    }
    try {
      String pwd = hashPassword(password);

      String sqlString = "SELECT * FROM db_user WHERE username = '"
                         + username +
                         "' AND password = '" + pwd + "' ";
      Statement stmt = connection.createStatement();
      ResultSet rs = stmt.executeQuery(sqlString);

      if (!rs.next()) {
        throw new SecurityException(
          "User name or password incorrect"
```

```
        );
      }

      // Authenticated; proceed
    } finally {
      try {
        connection.close();
      } catch (SQLException x) {
        // forward to handler
      }
    }
  }
}
```

2.1.3　符合规则的方案（PreparedStatement）

幸运的是，在 JDBC 类库中，提供了能够构建 SQL 命令并且处理非受信数据的 API。在 java.sql.PreparedStatement 类中，可以对输入字符串进行转义，如果使用正确的话，可以防止 SQL 注入。下面的例子显示了基于组件的净化过程：

这个符合规则的方案修改了 doPrivilegedAction() 方法，使用 PreparedStatement 类来代替 java.sql.Statement。并且这段代码会验证 username 参数的长度，防止如果提交一个长用户名的时候可能会出现的攻击。

```
public void doPrivilegedAction(
  String username, char[] password
) throws SQLException {
  Connection connection = getConnection();
  if (connection == null) {
    // Handle error
  }
  try {
    String pwd = hashPassword(password);

    // Ensure that the length of user name is legitimate
    if ((username.length() > 8) {
      // Handle error
    }

    String sqlString =
      "select * from db_user where username=? and password=?";
    PreparedStatement stmt = connection.prepareStatement(sqlString);
    stmt.setString(1, username);
    stmt.setString(2, pwd);
    ResultSet rs = stmt.executeQuery();
    if (!rs.next()) {
      throw new SecurityException("User name or password incorrect");
    }

    // Authenticated, proceed
  } finally {
    try {
      connection.close();
    } catch (SQLException x) {
```

```
        // forward to handler
      }
    }
  }
```

通过使用 PreparedStatement 类的 set*() 方法，可以进行强类型检查。这样可以减少 SQL 注入漏洞，因为自动化例程会正确地转义双引号内的输入数据。需要注意的是，那些将数据插入数据库的查询也会使用 PreparedStatement 。

XML 注入

由于 XML 语言具有平台无关性、灵活性和相对简洁的特点，它适用于从远端过程调用到系统化存储、交换以及获取数据的各种场合。然而，因为 XML 的多功能性，所以 XML 也是被广泛攻击的对象。其中一种攻击称为 XML 注入。

如果用户有能力使用结构化 XML 文档作为输入，那么他能够通过在数据字段中插入 XML 标签来重写这个 XML 文档的内容。当 XML 解析器对这些标签进行解析和归类的时候，会将它们作为可执行的内容，从而导致重写许多数据成员。

下面是一段从一个在线商店应用中摘取出来的 XML 代码，主要用来查询后台数据库。用户可以指定该次购买的物品的数量。

```
<item>
  <description>Widget</description>
  <price>500.0</price>
  <quantity>1</quantity>
</item>
```

恶意用户可以在 quantity 域中输入以下字符串来代替一个简单的数字：

```
1</quantity><price>1.0</price><quantity>1
```

结果会导致生成如下 XML 文档：

```
<item>
  <description>Widget</description>
  <price>500.0</price>
  <quantity>1</quantity><price>1.0</price><quantity>1</quantity>
</item>
```

通过用于 XML 的简单 API（SAX）解析器（org.xml.sax 和 javax.xml.parsers.SAXParser）可以解释该 XML 文件，这时第二个 price 域会覆盖第一个 price 域，从而使商品的价格被设置为 $1。甚至于存在这样的可能，攻击者可以构造这样一个攻击：插入特殊字符，比如插入注释块或者 CDATA 分隔符，从而就可以扭曲 XML 文档正常表达的意思。

2.1.4 不符合规则的代码示例

在下面这段不符合规则的代码示例中，一个客户方法使用简单的字符串链接来创建一个 XML 查询，然后将其发送到服务器。在这时就有可能出现 XML 注入问题，因为这个方法并没有进行任何输入验证。

```
private void createXMLStream(BufferedOutputStream outStream,
                             String quantity) throws IOException {
  String xmlString;
  xmlString = "<item>\n<description>Widget</description>\n" +
```

```
                "<price>500.0</price>\n" +
                "<quantity>" + quantity + "</quantity></item>";
    outStream.write(xmlString.getBytes());
    outStream.flush();
  }
```

2.1.5　符合规则的方案（白名单）

根据特定的数据和接收这些数据的命令解释器或者解析器的情况，必须使用一个适当的方法来净化这些非受信的用户输入。这个符合规则的方案使用白名单来净化输入数据。在这个方案中，处理方法要求输入的数字必须为 0 ～ 9。

```
  private void createXMLStream(BufferedOutputStream outStream,
                               String quantity) throws IOException {
    // Write XML string if quantity contains numbers only.
    // Blacklisting of invalid characters can be performed
    // in conjunction.

    if (!Pattern.matches("[0-9]+", quantity)) {
      // Format violation
    }

    String xmlString = "<item>\n<description>Widget</description>\n" +
                       "<price>500</price>\n" +
                       "<quantity>" + quantity + "</quantity></item>";
    outStream.write(xmlString.getBytes());
    outStream.flush();
  }
```

2.1.6　符合规则的方案（XML 模板）

一个更为通用的检查 XML 以防止注入的方法是，可以使用文档类型定义（Document Type Definition，DTD）或模板（schema）来进行验证。模板必须严格定义，从而防止通过注入的方式将一个合法的 XML 文档变成错误的文档。以下是一个用来验证 XML 文档片段的合适的模板：

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="item">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="description" type="xs:string"/>
      <xs:element name="price" type="xs:decimal"/>
      <xs:element name="quantity" type="xs:integer"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```

在 schema.xsd 文件中可以得到 XML 模板。在这个符合规则的方案中，可以采用这个模板来防止 XML 注入。它同时需要使用 CustomResolver 类来防止 XXE 攻击。关于这个类和所谓的 XXE 攻击，可以参见如下代码示例中的描述。

```
  private void createXMLStream(BufferedOutputStream outStream,
                               String quantity) throws IOException {
```

```
    String xmlString;
    xmlString = "<item>\n<description>Widget</description>\n" +
                "<price>500.0</price>\n" +
                "<quantity>" + quantity + "</quantity></item>";
    InputSource xmlStream = new InputSource(
      new StringReader(xmlString)
    );

    // Build a validating SAX parser using our schema
    SchemaFactory sf
      = SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
    DefaultHandler defHandler = new DefaultHandler() {
       public void warning(SAXParseException s)
         throws SAXParseException {throw s;}
       public void error(SAXParseException s)
         throws SAXParseException {throw s;}
       public void fatalError(SAXParseException s)
         throws SAXParseException {throw s;}
      };
    StreamSource ss = new StreamSource(new File("schema.xsd"));
    try {
      Schema schema = sf.newSchema(ss);
      SAXParserFactory spf = SAXParserFactory.newInstance();
      spf.setSchema(schema);
      SAXParser saxParser = spf.newSAXParser();
      // To set the custom entity resolver,
      // an XML reader needs to be created
      XMLReader reader = saxParser.getXMLReader();
      reader.setEntityResolver(new CustomResolver());
      saxParser.parse(xmlStream, defHandler);
    } catch (ParserConfigurationException x) {
      throw new IOException("Unable to validate XML", x);
    } catch (SAXException x) {
      throw new IOException("Invalid quantity", x);
    }

    // Our XML is valid, proceed
    outStream.write(xmlString.getBytes());
    outStream.flush();
  }
```

当 XML 可能已经载入还未处理的输入数据时，一般情况下使用 XML 模板或者 DTD 验证 XML。如果还没有创建这样的 XML 字符串，那么在创建 XML 之前处理输入，这种方式性能较高。

XML 外部实体攻击（XXE）

一个 XML 文档可以从一个被称为实体的很小的逻辑块开始动态构建。实体可以是内部的、外部的或基于参数的。外部实体允许将外部文件中的 XML 数据包含进来。

根据 XML W3C 4.4.3 小节的建议 [W3C 2008]：“包含所需的验证”部分。

> 当一个 XML 处理器找到一个已经解析过的实体的引用的时候，为了验证这个文档，处理器必须包含它的替代字段。如果该实体是外部的，而且处理器不打算验证这个 XML 文档，那么处理器可以（但不必）包含该实体的替代字段。

攻击者通过操作实体的 URI，使其指向特定的在当前文件系统中保存的文件，从而造成拒绝服务攻击或者程序崩溃，比如，指定 /dev/random 或者 /dev/tty 作为输入 URI。这可能永久阻塞程序或者造成程序崩溃。这就称为 XML 外部实体攻击（XML external entity, XXE）。因为包含来作为外部实体的替代文本并不是必需的，并不是所有的 XML 解析器都存在这样的外部实体攻击的安全漏洞。

2.1.7　不符合规则的代码示例

下面这个不符合规则的代码示例尝试对 evil.xml 文件进行解析，并报告相关错误，然后退出。然而，SAX 或者 DOM（Document Object Model, 文档对象模型）解析器会尝试访问在 SYSTEM 属性中标识的 URL，这意味着它将读取本地 /dev/tty 文件的内容。在 POSIX 系统中，读取这个文件会导致程序阻塞，直到可以通过计算机控制台得到输入数据为止。结果是，攻击者可以使用这样的恶意 XML 文件来导致系统挂起。

```
class XXE {
 private static void receiveXMLStream(InputStream inStream,
                                       DefaultHandler defaultHandler)
    throws ParserConfigurationException, SAXException, IOException {
   SAXParserFactory factory = SAXParserFactory.newInstance();
   SAXParser saxParser = factory.newSAXParser();
   saxParser.parse(inStream, defaultHandler);
 }

 public static void main(String[] args)
    throws ParserConfigurationException, SAXException, IOException {
   receiveXMLStream(new FileInputStream("evil.xml"),
                    new DefaultHandler());
 }
}
```

如果 evil.xml 文件中包含以下文本，程序会受到远程 XXE 攻击。

```
<?xml version="1.0"?>
<!DOCTYPE foo SYSTEM "file:/dev/tty">
<foo>bar</foo>
```

如果包含在异常里的信息是敏感的，则这个不符合规则的代码示例同样违反了规则 ERR06-J。

2.1.8　符合规则的方案（EntityResolver）

这个符合规则的方案定义了一个 CustomResolver 类，这个类实现了 org.xml.sax.EntityResolver 接口。它可以让 SAX 应用定制对外部实体的处理。setEntityResolver() 方法可以将对应的 SAX 驱动实例注册进来。这个定制的处理器使用的是一个为外部实体定义的简单的白名单。当输入不能解析任何指定的、安全的实体源路径的时候，resolveEntity() 方法会返回一个空的 InputSource 对象。结果是，当解析恶意输入时，这个由自定义的解析器返回的空的 InputSource 对象会抛出 java.net.MalformedURLException 异常。需要注意的是，你必须创建一个 XMLReader 对象，以便通过这个对象来设置自定义的实体解析器。

下面是一个基于组件的净化示例。

```
class CustomResolver implements EntityResolver {
  public InputSource resolveEntity(String publicId, String systemId)
    throws SAXException, IOException {

    // check for known good entities
    String entityPath = "/home/username/java/xxe/file";
    if (systemId.equals(entityPath)) {
      System.out.println("Resolving entity: " + publicId +
                         " " + systemId);
      return new InputSource(entityPath);
    } else {
      return new InputSource(); // Disallow unknown entities
                                // by returning a blank path
    }
  }
}

class XXE {
  private static void receiveXMLStream(InputStream inStream,
                                       DefaultHandler defaultHandler)
      throws ParserConfigurationException, SAXException, IOException {
    SAXParserFactory factory = SAXParserFactory.newInstance();
    SAXParser saxParser = factory.newSAXParser();

    // To set the Entity Resolver, an XML reader needs to be created
    XMLReader reader = saxParser.getXMLReader();
    reader.setEntityResolver(new CustomResolver());
    reader.setErrorHandler(defaultHandler);

    InputSource is = new InputSource(inStream);
    reader.parse(is);
  }

  public static void main(String[] args)
      throws ParserConfigurationException, SAXException, IOException {
    receiveXMLStream(new FileInputStream("evil.xml"),
                     new DefaultHandler());
  }
}
```

2.1.9 风险评估

如果不在系统处理或存储用户输入之前净化数据，会导致注入攻击。

规则	严重性	可能性	弥补代价	优先级	级别
IDS00-J	高	可能	中等	P12	L1

相关的漏洞 CVE-2008-2370 描述了在 Aapache Tomat 从 4.1.0 到 4.1.37，5.5.0 到 5.5.26 以及 6.0.0 到 6.0.16 这些版本中的安全漏洞。当使用 RequestDispatcher 时，Tomcat 会进行路径标准化，然后从 URI 中移除查询字串，这样会导致攻击者可以进行远程目录遍历攻击，并且在请求参数中通过 ..（两个点）来读取任意文件。

2.1.10　相关规范

CERT C 安全编码标准	STR02-C 净化传递给复杂子系统的数据
CERT C++ 安全编码标准	STR02-CPP 净化传递给复杂子系统的数据
ISO/IEC TR 24772:2010	注入 [RST]
MITRE CWE	CWE-116 不正确的输出编码或者转义

2.1.11　参考书目

[OWASP 2005]	
[OWASP 2007]	
[OWASP 2008]	对 XML 注入进行测试（OWASP-DV-008）
[W3C 2008]	4.4.3 包含，如果需要验证的话

2.2　IDS01-J 验证前标准化字符串

许多应用能够接收非受信的输入字符串，但需要对基于字符串的字符数据进行过滤和验证处理。

例如，为了避免跨站脚本（Cross-Site Scripting, XSS）的安全漏洞问题，某些应用会采用在输入中禁止 <script> 标签的策略。这种黑名单式的机制是一种有效的安全策略，尽管它们对于输入验证和净化来说是不够的。这种方式的验证必须并且只能在对输入进行标准化以后进行。

Java SE 6 中的字符编码是基于 Unicode 编码标准 4.0 的 [Unicode 2003] 。而在 Java SE7 中，其字符编码是基于 Unicode 编码标准 6.0.0 的 [Unicode 2011]。

根据 Unicode 编码标准 [Davis 2008a] 的 15 号附件，Unicode 的标准化格式为：

> 当在一个标准化形式中需要保留字符串时，它们需要确定同样的字符串有一个唯一的二进制表示。不能将 KC 和 KD 两种标准化形式盲目地应用于任意文本。因为它们会消除许多格式上的差别，它们可以用来防止在遗留字符集之间循环转换，同时，除非被格式化标记替代，否则它们可能会去除那些对文本语义来说很重要的差别。最好认为这些标准化格式就像大小写字符映射一样：在需要确认核心含义的某些场合下是有用的，但是，它也会对文本做出一些意想不到的修改。使用更为严格的字符集，可以让它们更为自由地应用到相关领域中。

通常，对任意编码字符串进行输入验证的最适合的标准化格式是 KC（NFKC）格式，因为标准化成 KC 可以将输入转换为等价的标准格式，这种格式可以安全地和需要的输入格式对比。

2.2.1　不符合规则的代码示例

如下所示的这个不符合规则的代码示例想要在进行标准化之前对字符串进行验证。结果是没有检测出应当拒绝的输入而使得验证逻辑无法进行，这是因为在进行尖括号检查时，代码不能分辨它的另一种 Unicode 表示方式。

```
// String s may be user controllable
// \uFE64 is normalized to < and \uFE65 is normalized to > using NFKC
String s = "\uFE64" + "script" + "\uFE65";

// Validate
Pattern pattern = Pattern.compile("[<>]"); // Check for angle brackets
```

```
Matcher matcher = pattern.matcher(s);
if (matcher.find()) {
  // Found black listed tag
  throw new IllegalStateException();
} else {
  // ...
}

// Normalize
s = Normalizer.normalize(s, Form.NFKC);
```

这个 normalize() 方法将 Unicode 文本转换为等价的组合或者分拆的形式，从而让对文本的搜索更为简单。因为该标准化方法支持在 Unicode 编码标准的 15 号附录中描述的标准标准化形式，所以被称为基于 Unicode 的标准化格式。

2.2.2 符合规则的方案

这个符合规则的解决方案在字符串验证前会将其进行标准化，它将字符串的另一种表示标准化为标准的尖括号。接着，输入验证可以正确地检测出恶意输入，并抛出 IllegalStateException 异常。

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

// Normalize
s = Normalizer.normalize(s, Form.NFKC);

// Validate
Pattern pattern = Pattern.compile("[<>]");
Matcher matcher = pattern.matcher(s);
if (matcher.find()) {
 // Found black listed tag
  throw new IllegalStateException();
} else {
  // ...
}
```

2.2.3 风险评估

在标准化前进行输入验证会使攻击者有机会绕过数据过滤和其他安全机制。结果会导致执行任意代码。

规则	严重性	可能性	弥补代价	优先级	级别
IDS01-J	高	可能	中等	P12	L1

2.2.4 相关规范

ISO/IEC TR 24772:2010	跨站点脚本 [XYT]
MITRE CWE	CWE-289 使用别名来绕过验证过程
	CWE-180 不正确的行为次序：在标准化数据之前进行数据验证

2.2.5　参考书目

[API 2006]
[Davis 2008a]
[Weber 2009]

2.3　IDS02-J 在验证之前标准化路径名

根据 Java API[API 2006] 文档对 java.io.File 类的描述：

> 一个路径名，不管是抽象的还是字符串形式的，可以是相对路径也可以是绝对路径。使用绝对路径名，因为在定位一个路径表示的文件时，已经不需要其他信息了，因而可以认为是完整的。相比之下，一个相对路径名必须要增加其他的路径信息才能进行解释。

绝对路径名或者相对路径名会包含文件链接，比如符号（软）链接、硬链接、快捷方式、影子、别名和联名。这些文件链接在文件验证操作进行之前必须完全解析。例如，一个命名为 trace 的符号链接最终可以是指向路径名 /home/system/trace。该路径名可能同时会包含特殊的文件名，这些文件名会让验证变得困难。

1）“.”指目录本身。

2）在目录内，一个特殊的文件名“..”指该目录的上一级目录。

除了这些特殊的问题之外，还有很多是和操作系统有关的，并且是和文件系统有关的命名转换，从而让验证变得困难。

标准化文件名比验证路径名要容易得多。很多路径名会指向一个目录或文件。此外，考虑到路径名指向的目录或文件，一个路径名的文本表示会产生很少或没有信息。结果是，所有的路径名必须被完整解析或者在验证之前进行标准化（canonicalized）。

验证可能是必要的，例如，当需要限定用户访问在某个特定目录下的文件，或者需要基于文件名或者路径名的时候，需要做出对应的安全决策。通常，这些限制被攻击者利用目录遍历（directory traversal）或者等价路径（path equivalence）这样的方式规避而产生漏洞。一个目录遍历漏洞会让 I/O 操作转到另一个特定的操作目录中去。当攻击者提供一个不同但是有等价名称的资源，并绕过安全检查的时候，就会由此产生一个路径等价漏洞。

标准化包括内在的竞态窗口，这个窗口在程序获得标准路径名的时间和打开文件的时间之间产生。在验证标准化的路径名的时候，文件系统可能已经被修改，并且标准化的路径名不再指向原有的合法文件。幸运的是，可以很容易消除这些竞态。标准化的路径名可以用来确定引用的文件名是否在一个安全的目录中（参考 FIO00-J 可以得到更多的信息）。如果引用的文件在一个安全的目录之内，根据定义，攻击者就不能篡改它也不能利用这些竞态条件。

该规则是规则 IDS01-J 的一个实例。

2.3.1　不符合规则的代码示例

以下不符合规则的代码示例可以从命令行参数接收文件路径，并使用 File.getAbsolutePath() 方法来获得绝对路径。它同时会使用 isInSecureDir() 方法，这个方法在规则 FIO00-J 中进行定义，它可以用来保证文件在一个安全的目录中。然而，它并不能解析文件链接，也不能消除这些同类

的错误。

```
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))) {
    throw new IllegalArgumentException();
  }
  if (!validate(absPath)) { // Validation
    throw new IllegalArgumentException();
  }
}
```

应用需要限制用户操作那些位于 home 目录之外的文件。validate() 方法可以保证路径名在这个目录当中，但这个方法可以很容易被绕过。例如，用户可以在他的 home 目录中创建一个链接，这个链接指向位于 home 目录之外的目录或文件。当 validate() 方法处理这个链接的路径名的时候，仍然还认为它是在 home 目录中，结果是可以顺利通过验证，但在实际的操作中，它会操作，位于 home 目录之外的最终路径指向。

注意，在 Windows 和 Macintosh 平台中，File.getAbsolutePath() 方法可以解析符号链接、别名和快捷方式。尽管如此，在 Java 语言规范中却不能保证这样的行为在所有的平台上都有效，或者在未来的实现中均会如此。

2.3.2 符合规则的方案（getCanonicalPath()）

该符合规则的解决方案使用 getCanonicalPath() 方法，这个方法是在 Java 2 中引入的，因为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解析。那些特殊文件名如 .. 同样被去除了，所以在执行验证之前，输入就被缩减成一种标准化形式。当存在 validate（）方法时，攻击者无法通过使用 ../ 序列进入特定的目录。

```
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(canonicalPath))) {
    throw new IllegalArgumentException();
  }
  if (!validate(canonicalPath)) { // Validation
   throw new IllegalArgumentException();
  }
}
```

当在 applet 中使用时，getCanonicalPath() 方法会抛出安全异常，因而会泄露太多关于宿主机的信息。getCanonicalFile() 方法和 getCanonicalPath() 类似，但它会返回一个新的 File 对象而不是一个 String。

2.3.3 符合规则的方案（安全管理器）

处理这类问题的一个综合方法是，给应用赋予相应的权限，而这些权限只对特定目录下的文

件有效，比如用户的 home 目录。该方案只需要在程序的安全管理策略文件中指明程序的绝对路径，并且将 java.io.FilePermission 以及读写权限赋予目录 ${user.home}/* 的路径。

```
grant codeBase "file:/home/programpath/" {
  permission java.io.FilePermission "${user.home}/*", "read, write";
};
```

该方案要求用户的 home 目录是一个安全目录，这会在规则 FIO00-J 中描述。

2.3.4　不符合规则的代码示例一

这段代码示例允许用户指定所操作的文件名的绝对路径，改用包含 ../ 序列的参数来指定位于特定目录之外的文件（在这个示例中是 /img 路径），从而违反了该程序的安全策略。

```
FileOutputStream fis =
  new FileOutputStream(new File("/img/" + args[0]));
// ...
```

2.3.5　不符合规则的代码示例二

该代码示例希望解决使用 File.getCanonicalPath() 方法的问题，该方法完全解析参数并构造出标准化路径。例如，路径 /img/../etc/passwd 可以解析为 /etc/passwd。没有经过验证的标准化是不够的，因为攻击者可以使用指定目录之外的特定文件。

```
File f = new File("/img/" + args[0]);
String canonicalPath = f.getCanonicalPath();
FileOutputStream fis = new FileOutputStream(f);
// ...
```

2.3.6　符合规则的方案

该方案从非受信的用户输入中获取文件名，对其进行标准化，然后基于起始路径名列表对其进行验证。当验证成功时，才会操作指定文件，也就是说，仅当该文件是在 /img/java 目录下的 file1.txt 文件或者 file2.txt 文件中的一个时才行。

```
File f = new File("/img/" + args[0]);
String canonicalPath = f.getCanonicalPath();

if (!canonicalPath.equals("/img/java/file1.txt") &&
    !canonicalPath.equals("/img/java/file2.txt")) {
   // Invalid file; handle error
}

FileInputStream fis = new FileInputStream(f);
```

/img/java 目录必须是一个安全目录，不存在任何竞态条件。

2.3.7　符合规则的方案（安全管理器）

该方案赋予应用相应的权限读取指定目录或文件。比如，读权限的赋予，可以通过在安全策略文件中为该程序指定一个绝对路径名，并设置 java.io.FilePermission 为一个文件或目录的标准化绝对路径，然后将动作设为 read。

```
// All files in /img/java can be read
grant codeBase "file:/home/programpath/" {
  permission java.io.FilePermission "/img/java", "read";
};
```

2.3.8 风险评估

当使用来自非受信源的路径名时，如果不首先进行标准化，然后进行验证，那么会导致目录遍历和路径等价漏洞。

规则	严重性	可能性	弥补代价	优先级	级别
IDS02-J	中	不可能	中等	P4	L3

相关漏洞 CVE-2005-0789 描述了在 LimeWire 3.9.6 ～ 4.6.0 节中的目录遍历漏洞，它的存在让远程攻击者可以通过在请求中通过 .. 路径名读取任意文件。

CVE-2008-5518 描述了多种目录遍历漏洞，在 Apache Geronimo 应用服务器 2.1 到 2.1.3 的 Windows 版本的 Web 管理员控制台中，它允许远程攻击者向任意目录上传文件。

2.3.9 相关规范

CERT C 安全编码标准	FIO02-C 将来自于非受信源的路径名标准化
CERT C++ 安全编码标准	FIO02-CPP 将来自于非受信源的路径名标准化
ISO/IEC TR 24772:2010	路径遍历 [EWR]
MITRE CWE	CWE-171 净化、标准化以及比较错误
	CWE-647 针对验证决定，使用非标准化的 URL 路径

2.3.10 参考书目

[API 2006] getCanonicalPath() 方法

[Harold 1999]

2.4 IDS03-J 不要记录未经净化的用户输入

当日志条目包含未经净化的用户输入时会引发记录注入漏洞。恶意用户会插入伪造的日志数据，从而让系统管理员以为是系统行为 [OWASP 2008]。例如，用户在将冗长的日志记录拆分成两份时，日志中使用的回车和换行符可能会被误解。记录注入攻击可以通过对任何非受信的发送到日志的输入进行净化和验证来阻止。

将未经净化的用户输入写入日志同样会导致向受信边界之外泄露敏感数据，或者在存储敏感数据的时候，违反了本地的安全规则。举例来说，如果一个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 规则 [PCI 2010]。关于详情，可以参考规则 IDS00-J 对净化用户输入的详细介绍。

2.4.1 不符合规则的代码示例

从这个不符合规则的代码示例中，可以看到在接收到非法请求的时候，会记录用户的用户名。这时没有执行任何输入净化。

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

如果没有净化，那么可能会出现日志注入攻击。当出现 username 是 david 时，标准的日志信息如下所示：

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

如果日志信息中使用的 username 不是 david 而是多行字符串，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$RootLogger log
SEVERE: User login failed for: david
May 15, 2011 2:25:52 PM java.util.logging.LogManager$RootLogger log
SEVERE: User login succeeded for: administrator
```

2.4.2　符合规则的方案

这个符合规则的方案在登录之前会净化用户名输入，从而防止注入攻击。具体的输入处理的更多细节请参考规则 IDS00-J。

```
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);
}
```

2.4.3　风险评估

允许日志记录未经验证的用户输入，会导致错误的日志信息，从而泄露安全信息，或者会保存敏感信息，但这种保存方式是不合规则的。

规则	严重性	可能性	弥补代价	优先级	级别
IDS03-J	中	可能	中等	P8	L2

2.4.4　相关规范

ISO/IEC TR 24772:2010	注入 [RST]
MITRE CWE	CWE-144 对行分隔符的不正确取消 CWE-150 对转义、元数据或者控制序列的不正确取消

2.4.5 参考书目

[API 2006]
[OWASP 2008]
[PCI DSS 标准]

2.5 IDS04-J 限制传递给 ZipInputStream 的文件大小

通过对 java.util.ZipInputStream 的输入进行检查，可以防止消耗过多的系统资源。当资源使用大大多于输入数据所使用的资源的时候，就会出现拒绝服务问题。当需要解压一个小文件，比如 zip、gif 或者 gzip 编码的 HTTP 内容，会消耗过多的资源时，并且在压缩率极高的情况下，Zip 算法的实现本身就会导致 zip 炸弹（zip bomb）的出现。

zip 算法能产生很高的压缩比率 [Mahmoud 2002]。 图 2-1 显示了一个文件可以从 148MB 压缩到 590KB，压缩比率高达 200:1。这个文件包含了多次出现的重复数据，比如交替出现的字符 a 和 b。如果输入数据符合压缩算法的要求，或者可以使用更多的输入数据，或者可以使用其他的压缩方法，甚至还能达到更高的压缩比例。

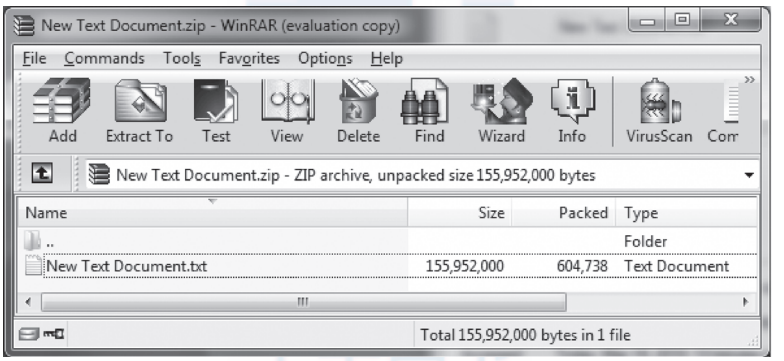

图 2-1 在一个 zip 文件中非常大的压缩比例

在 zip 文件中的任何数据，如果其解压后的文件大小超出了一个特定的限制，那么就不应该将它们解压。而这个限制实际上取决于平台的能力。

这条规则是更普遍的规则 MSC07-J 的一个特例。

2.5.1 不符合规则的代码示例

下面是一个不符合规则的代码示例，它不能检查解压一个文件时所消耗的资源。它会允许操作完成，或者直至本地资源耗尽为止。

```
static final int BUFFER = 512;
// ...

// external data source: filename
BufferedOutputStream dest = null;
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());
  dest = new BufferedOutputStream(fos, BUFFER);
  while ((count = zis.read(data, 0, BUFFER)) != -1) {
    dest.write(data, 0, count);
  }
  dest.flush();
  dest.close();
}
zis.close();
```

2.5.2　符合规则的方案

在这个方案中，while 循环中的代码会通过使用 ZipEntry.getSize() 方法在解压之前，得到 zip 文档中需解压文件的大小。如果该解压文件过大，比如说超过 100MB，那么抛出异常。

```
static final int TOOBIG = 0x6400000; // 100MB

  // ...

  // write the files to the disk, but only if 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());
  dest = new BufferedOutputStream(fos, BUFFER);
  while ((count = zis.read(data, 0, BUFFER)) != -1) {
    dest.write(data, 0, count);
  }
```

2.5.3　风险评估

规则	严重性	可能性	弥补代价	优先级	级别
IDS04-J	低	可能	高等	P2	L3

2.5.4　相关规范

MITRE CWE	CWE-409 对高比例压缩的数据的不正确处理（数据放大）
Java 语言安全编码规范 第 3 版	规范 2-5 对输入进行检查，从而不会产生过多的资源消耗

2.5.5　参考书目

[Mahmoud 2002]　使用 Java API 对数据进行压缩和解压缩

2.6 IDS05-J 使用 ASCII 字符集的子集作为文件名和路径名

如果文件名和路径名中包含了特殊字符，就会有问题，并且它会引起不可预期的系统行为，从而导致安全漏洞。在构建文件名或者路径名的时候，下面的字符和模式是有问题的。

- 以破折号开头：当调用一个程序时，使用的是它的文件名，若文件名以破折号开头，那么会导致问题，因为这样的文件名的第一个字符会被解析为选项标志。
- 控制字符，例如换行符、回车与 Esc：在执行 shell 脚本和记录日志的时候，如果在文件名中采用控制字符，那么可能会导致意想不到的问题。
- 空格：脚本中的空格会导致问题，在没用双引号包围文件名的时候空格也会引发问题。
- 不合法的字符编码：字符编码会使正常验证文件名和路径名变得困难（参见规则 IDS11-J）。
- 命名空间分隔符：文件名和路径名中包括了命名空间分隔符，会引起潜在的安全问题和不可预期的后果。
- 命令行解释器、脚本和解析器：某些字符对于命令行解释器、shell 或者解析器来说，是有特殊含义的。应该避免使用它们。

受 MS-DOS 的影响，文件名一般采用 xxxxxxxx.xxx 的形式，这里 x 代表字母数字字符。这种方式受到现代系统的广泛支持。在某些平台上，文件名是大小写敏感的，而在其他平台上，文件名是大小写不敏感的。 VU#439395 是一个在 C 语言中存在安全漏洞的例子，在这个例子中，因为没有正确处理大小写敏感而产生了问题 [VU#439395]。

这条规则是规则 IDS00-J 的特例。

2.6.1 不符合规则的代码示例

在以下不符合规则的代码示例中，使用了不安全的字符作为文件名的一部分。

```
File f = new File("A\uD8AB");
OutputStream out = new FileOutputStream(f);
```

一个平台可以自由地定义它自己的不安全字符的映射集合。比如，当在 Ubuntu Linux 系统中测试时，这种不合规的代码例子会产生以下的文件名：

```
A?
```

2.6.2 符合规则的方案

使用一个描述性的文件名，并且该文件名只使用 ASCII 字符集的子集。

```
File f = new File("name.ext");
OutputStream out = new FileOutputStream(f);
```

2.6.3 不符合规则的代码示例

这个不符合规则的代码示例创建了一个文件，其中使用了未经净化的用户输入。

```
public static void main(String[] args) throws Exception {
  if (args.length < 1) {
    // handle error
  }
```

```
  File f = new File(args[0]);
  OutputStream out = new FileOutputStream(f);
  // ...
}
```

该文件名没有经过任何检查来防止使用问题字符，当攻击者正好知道这段程序代码用来创建或重命名文件，而这些文件以后会用于脚本或某些自动化过程的话，他可以选择特殊的字符作为输出文件夹名，从而恶意迷惑后续过程。

2.6.4　符合规则的方案

在这个符合规则的方案中，程序使用了白名单，从而拒绝了不安全的文件名。

```
public static void main(String[] args) throws Exception {
  if (args.length < 1) {
    // handle error
  }
  String filename = args[0];

  Pattern pattern = Pattern.compile("[^A-Za-z0-9%&+,.:=_]");
  Matcher matcher = pattern.matcher(filename);
  if (matcher.find()) {
    // filename contains bad chars, handle error
  }
  File f = new File(filename);
  OutputStream out = new FileOutputStream(f);
  // ...
}
```

所有来自于非受信源的文件名必须经过净化，以保证它们只含有安全的字符。

2.6.5　风险评估

如果没有使用安全的 ASCII 字符子集，那么将会导致数据被错误解释。

规则	严重性	可能性	弥补代价	优先级	级别
IDS05-J	中	不可能	中等	P4	L3

2.6.6　相关规范

CERT C 安全编码标准	MSC09-C 字符编码——基于安全的原因，使用 ASCII 字符集的子集
CERT C++ 安全编码标准	MSC09-CPP 字符编码——基于安全的原因，使用 ASCII 字符集的子集
ISO/IEC TR 24772:2010	对文件名和其他外部标识符的选择 [AJN]
MITRE CWE	CWE-161 对输出的不正确编码或者不正确转义

2.6.7　参考书目

ISO/IEC 646-1991	ISO 在信息交换中，使用 7 位编码字符集合
[Kuhn 2006]	UTF-8 编码以及用于 UNIX/Linux 系统的 Unicode FAQ

[Wheeler 2003] 5.4 文件名
[VU#439395]

2.7 IDS06-J 从格式字符串中排除用户输入

对 Java 格式字符串的解释要比对在像 C 语言这样的语言中更严格 [Seacord 2005]。当任何转换参数不能匹配相应的格式符时，标准类库实现会抛出一个相应的异常。这种方法降低了被恶意利用的可能性。然而，恶意用户输入可以利用格式字符串，并且造成信息泄露或者拒绝服务。因此，不能在格式字符串中使用非受信来源的字符串。

2.7.1 不符合规则的代码示例

这个不符合规则的代码示例展示了可能出现信息泄露的问题。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。

```
class Format {
  static Calendar c =
   new GregorianCalendar(1995, GregorianCalendar.MAY, 23);
  public static void main(String[] args) {
    // args[0] is the credit card expiration date
    // args[0] can contain either %1$tm, %1$te or %1$tY as malicious
    // arguments
    // First argument prints 05 (May), second prints 23 (day)
    // and third prints 1995 (year)
    // Perform comparison with c, if it doesn't match print the
    // following line
    System.out.printf(args[0] +
    " did not match! HINT: It was issued on %1$terd of some month", c);
  }
}
```

如果没有经过正确输入验证，攻击者通过在输入语句中包含以下参数之一：%1$tm、%1$te 或 %1$tY，就可以判断验证中所用来和输入相对比的日期。

2.7.2 符合规则的方案

该方案能够保证用户产生的输入会被排除在格式字符串之外。

```
class Format {
  static Calendar c =
    new GregorianCalendar(1995, GregorianCalendar.MAY, 23);
  public static void main(String[] args) {
    // args[0] is the credit card expiration date
    // Perform comparison with c,
    // if it doesn't match print the following line
    System.out.printf ("%s did not match! "
        + " HINT: It was issued on %1$terd of some month", args[0], c);
  }
}
```

2.7.3 风险评估

允许用户输入污染格式字符串会导致信息泄露或者拒绝服务。

规则	严重性	可能性	弥补代价	优先级	级别
IDS06-J	中	不可能	中等	P4	L3

自动化检测　完成污染分析的静态分析工具可以用来判断是否违背该规则。

2.7.4　相关规范

CERT C 安全编码标准	FIO30-C 从格式字符串中去除用户输入
CERT C++ 安全编码标准	FIO30-CPP 从格式字符串中去除用户输入
ISO/IEC TR 24772:2010	注入 [RST]
MITRE CWE	CWE-134 不受控制的格式字符串

2.7.5　参考书目

[API 2006]	类格式化器
[Seacord 2005]	第 6 章 输出格式化

2.8　IDS07-J 不要向 Runtime.exec() 方法传递非受信、未净化的数据

外部程序通常被系统调用来完成某种需要的功能。这是一种重用的形式，也被认为是一种简单基于组件的软件工程方法。在应用没有净化非受信的输入并且在执行外部程序时使用这种数据，就会导致产生命令和参数注入漏洞。

每一个 Java 应用都有一个唯一的 Runtime 类的实例，通过它可以提供一个应用和应用运行环境的接口。当前的 Runtime 对象可以通过 Runtime.getRuntime() 方法获得。Runtime.getRuntime() 的语义定义并不严格，所以最好仅仅使用它的那些必要的行为，然而，在通常情况下，它应当被命令直接调用，而不应通过 shell 来调用。如果需要使用 shell 来调用它，可以在 POSIX 中使用 /bin/sh 或者在 Windows 平台中使用 cmd.exe 的方式。作为 exec() 的另一种形式，它会使用 StringTokenizer 来分隔从命令行读入的字符串。在 Windows 平台中，在处理这些符号时，它们会被连接成一个单一的参数字符串。

因而，除非显式地调用命令行解释器，否则是不会产生命令行注入攻击的。然而，当参数中包含那些以空格、双引号或者其他以 - / 开头的用来表示分支的字符时，就可能发生参数注入攻击。

这条规则是规则 IDS00-J 的特例。任何源于程序受信边界之外的字符串数据，在当前平台作为命令来执行之前，都必须经过净化。

2.8.1　不符合规则的代码示例（Windows）

该代码示例使用 dir 命令列出目录列表。这是通过 Runtime.exec() 方法调用 Windows 的 dir 命令来实现的。

```
class DirList {
  public static void main(String[] args) throws Exception {
    String dir = System.getProperty("dir");
    Runtime rt = Runtime.getRuntime();
    Process proc = rt.exec("cmd.exe /C dir " + dir);
    int result = proc.waitFor();
    if (result != 0) {
```

```
        System.out.println("process error: " + result);
      }
      InputStream in = (result == 0) ? proc.getInputStream() :
                                       proc.getErrorStream();
      int c;
      while ((c = in.read()) != -1) {
        System.out.print((char) c);
      }
    }
  }
```

因为 Runtime.exec() 方法接受源于运行环境的未经净化的数据，所以这些代码会引起命令注入攻击。

攻击者可以通过以下命令利用该程序：

```
java -Ddir='dummy & echo bad' Java
```

该命令实际上执行的是两条命令：

```
cmd.exe /C dir dummy & echo bad
```

第一条命令会列出并不存在的 dummy 文件夹，并且在控制台上输出 bad。

2.8.2 不符合规则的代码示例（POSIX）

这一个代码示例的功能与 2.8.1 节介绍的类似，只是它使用了 POSIX 中的 ls 命令。与 Windows 版本的唯一区别在于传入 Runtime.exec() 的参数。

```
class DirList {
  public static void main(String[] args) throws Exception {
    String dir = System.getProperty("dir");
    Runtime rt = Runtime.getRuntime();
    Process proc = rt.exec(new String[] {"sh", "-c", "ls " + dir});
    int result = proc.waitFor();
    if (result != 0) {
      System.out.println("process error: " + result);
    }
    InputStream in = (result == 0) ? proc.getInputStream() :
                                     proc.getErrorStream();
    int c;
    while ((c = in.read()) != -1) {
      System.out.print((char) c);
    }
  }
}
```

攻击者可以通过使用与上例中一样的命令取得同样的效果。这个命令实际执行的是：

```
sh -c 'ls dummy & echo bad'
```

2.8.3 符合规则的方案（净化）

符合规则的方案会对非受信的用户输入进行净化，这种净化只允许一小组列入白名单的字符出现在参数中，并传给 Runtime.exec() 方法，其他所有的字符都会被排除掉。

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

```
    // Handle error
  }
  // ...
```

尽管这是一个符合规则的方案，这个净化方案会拒绝合法的目录。同时，因为命令行解释器调用是依赖于系统的，所以很难有一个方案可以应付所有 Java 程序可以运行的平台上的命令行注入。

2.8.4　符合规则的方案（限定用户选择）

这个符合规则的方案通过只向 Runtime.exec() 方法输入那些受信的字符串来防止命令注入。当用户可以控制使用哪一个字符串时，用户就可以不向 Runtime.exec() 方法直接提供字符串数据。

```
// ...
String dir = null;
// only allow integer choices
int number = Integer.parseInt(System.getproperty("dir"));
switch (number) {
  case 1:
    dir = "data1"
    break; // Option 1
  case 2:
    dir = "data2"
    break; // Option 2
  default: // invalid
    break;
}
if (dir == null) {
  // handle error
}
```

这个方案将可能罗列出的目录直接写出来了。

如果有许多可用目录这个方案很快会变得不好管理。一个更富可伸缩性的方案是从一个属性文件中读取所有允许的目录至 java.util.Properties 对象中。

2.8.5　符合规则的方案（避免使用 Runtime.exec()）

当通过执行系统命令可以完成的任务可以用其他方式完成时，最好避免用执行系统命令的方式。这个符合规则的方案使用 File.list() 方法来提供目录列表，从而消除了命令或参数注入攻击发生的可能性。

```
import java.io.File;

class DirList {
  public static void main(String[] args) throws Exception {
    File dir = new File(System.getProperty("dir"));
    if (!dir.isDirectory()) {
      System.out.println("Not a directory");
    } else {
      for (String file : dir.list()) {
        System.out.println(file);
      }
    }
  }
}
```

2.8.6 风险评估

向 Runtime.exec() 方法传递非受信的、未经净化的数据会导致命令和参数注入攻击。

规则	严重性	可能性	弥补代价	优先级	级别
IDS07-J	高	可能	中等	P12	L1

相关漏洞

[CVE-2010-0886]	Sun Java Web Start 插件命令行的参数注入问题
[CVE-2010-1826]	在处理 Mach RPC 消息时，方法 updateSharingD 中的命令行注入问题
[T-472]	在 updateSharingD 中的 Mac OS X Java 命令行注入问题，可以让本地用户权限得到提升

2.8.7 相关规范

CERT C 安全编码标准	ENV03-C 当调用外部程序时，需要对环境进行净化 ENV04-C 如果不需要命令行处理器，那么不要调用方法 system()
CERT C++ 安全编码标准	ENV03-CPP 当调用外部程序时，需要对环境进行净化 ENV03-CPP 如果不需要命令行处理器，那么不要调用方法 system()
ISO/IEC TR 24772:2010	注入 [RST]
MITRE CWE	CWE-78 在一个 OS 命令行中不正确地将特殊元素取消掉（OS 命令行注入）

2.8.8 参考书目

[Chess 2007]	第 5 章 处理输入“命令行注入”
[OWASP 2005]	
[Permissions 2008]	

2.9 IDS08-J 净化传递给正则表达式的非受信数据

正则表达式在匹配文本字符串时被广泛使用。比如，在 POSIX 中，grep 命令就支持正则表达式，使用它可以在指定文本中搜索模式。如果要了解正则表达式的基本情况，请参考 Java 教程 [Tutorials 08]。java.util.regex 包提供了 Pattern 类，这个类封装了一个编译过的正则表达式和一个 Matcher 类，通过 Matcher 类这个引擎，可以使用 Pattern 在 CharSequence 中进行匹配操作。

在 Java 中，必须注意不能误用强大的正则表达式功能。攻击者也许会通过提供一个恶意输入对初始的正则表达式进行修改，比如使其不符合程序规定的正则表达的要求。这种攻击方式称为正则注入（regex injection），它可以影响程序控制流、导致信息泄露，并引起拒绝服务漏洞。

以下这些是在使用正则表达式时，容易被利用的地方。

- 匹配标志：非受信的输入可能覆盖匹配选项，然后可能会也可能不会传给 Pattern.

compile() 方法。

- 贪婪：一个非受信的输入可能试图注入一个正则表达式，通过它来改变初始的那个正则表达式，从而匹配尽可能多的字符串，从而暴露敏感信息。
- 分组：程序员会用括号包括一部分的正则表达式以完成一组动作中某些共同的部分。攻击者可能通过提供非受信的输入来改变这种分组。

非受信的输入应该在使用前净化，从而防止发生正则表达式注入。当用户必须指定正则表达式作为输入时，必须注意需要保证初始的正则表达式没有被无限制修改。在用户输入字符串提交给正则解析之前，进行白名单字符处理（比如字母和数字）是一个很好的输入净化策略。开发人员必须仅仅提供最有限的正则表达式功能给用户，从而减少被误用的可能。

正则表达式注入示例

假设一个系统日志文件包含一系列系统过程的输出信息。其中一些过程会产生公开的信息，而另一些会产生被标识为“private”的敏感信息。以下是这个日志文件的例子：

```
10:47:03 private[423] Successful logout name: usr1 ssn: 111223333
10:47:04 public[48964] Failed to resolve network service
10:47:04 public[1] (public.message[49367]) Exited with exit code: 255
10:47:43 private[423] Successful login name: usr2 ssn: 444556666
10:48:08 public[48964] Backup failed with error: 19
```

用户希望搜索日志文件以寻找感兴趣的信息，然而又必须防止读取私有数据。程序可能会采用以下的方式，它允许用户将搜索文本作为下面所示正则表达式的一部分：

```
(.*? +public\[\d+\] +.*<SEARCHTEXT>.*)
```

然而，攻击者可以用任何字符串替代 <SEARCHTEXT>，这样他就可以通过下面的文本实现正则表达式注入：

```
.*)|(.*
```

注入后的正则表达式为：

```
(.*? +public\[\d+\] +.*.*)|(.*.*)
```

这个正则表达式会匹配日志文件中的所有信息，包括那些私有的信息。

2.9.1　不符合规则的代码示例

这个不符合规则的代码示例将日志文件周期性地载入内存，它通过将关键词作为参数传给 suggestSearches() 方法来获得关键词搜索建议。

```
public class Keywords {
  private static ScheduledExecutorService scheduler
     = Executors.newSingleThreadScheduledExecutor();
  private static CharBuffer log;
  private static final Object lock = new Object();

  // Map log file into memory, and periodically reload
  static
    try {
      FileChannel channel = new FileInputStream(
         "path").getChannel();
```

```
        // Get the file's size and map it into memory
        int size = (int) channel.size();
        final MappedByteBuffer mappedBuffer = channel.map(
           FileChannel.MapMode.READ_ONLY, 0, size);

        Charset charset = Charset.forName("ISO-8859-15");
        final CharsetDecoder decoder = charset.newDecoder();

       log = decoder.decode(mappedBuffer); // Read file into char buffer
        Runnable periodicLogRead = new Runnable() {
          @Override public void run() {
            synchronized(lock) {
              try {
                log = decoder.decode(mappedBuffer);
              } catch (CharacterCodingException e) {
                // Forward to handler
              }
            }
          }
        };
        scheduler.scheduleAtFixedRate(periodicLogRead,
                                      0, 5, TimeUnit.SECONDS);
      } catch (Throwable t) {
        // Forward to handler
      }
    }

    public static Set<String> suggestSearches(String search) {
      synchronized(lock) {
        Set<String> searches = new HashSet<String>();

        // Construct regex dynamically from user string
        String regex = "(.*? +public\\[\\d+\\] +.*" + search + ".*)";

        Pattern keywordPattern = Pattern.compile(regex);
        Matcher logMatcher = keywordPattern.matcher(log);
        while (logMatcher.find()) {
          String found = logMatcher.group(1);
          searches.add(found);
        }
        return searches;
      }
    }

  }
```

这段代码允许受信用户在公共日志信息中搜索“error”关键词。然而，这样会导致前面提到的通过正则注入进行的恶意攻击。

2.9.2 符合规则的方案（白名单方法）

这个符合规则的方案过滤搜索字符串中的非字母数字的字符（除了空格和单引号外），这种方案可以阻止前面介绍的正则表达式注入。

```
  public class Keywords {
```

```
  // ...
  public static Set<String> suggestSearches(String search) {
    synchronized(lock) {
      Set<String> searches = new HashSet<String>();

      StringBuilder sb = new StringBuilder(search.length());
      for (int i = 0; i < search.length(); ++i) {
        char ch = search.charAt(i);
        if (Character.isLetterOrDigit(ch) ||
            ch == ' ' ||
            ch == '\ ") {
          sb.append(ch);
        }
      }
      search = sb.toString();

      // Construct regex dynamically from user string
      String regex = "(.*? +public\\[\\d+\\] +.*" + search + ".*)";
      // ...
    }
  }
}
```

这个方案同样对有效的搜索项进行限制。例如，用户不再可以使用类似“name=”这样的搜索，因为 = 字符会被净化掉。

2.9.3　符合规则的方案

另一个减少这种漏洞的方法是，在匹配前过滤敏感信息。这样的方案要求每一次日志文件定期更新时都要进行过滤，从而会产生额外的复杂性和性能损失。如果日志格式变化了，但类的设计没有反映这些变化，那么敏感数据依然可能被暴露。

2.9.4　风险评估

在正则表达式中，没有对非受信的数据进行净化，会导致敏感信息的泄露。

规则	严重性	可能性	弥补代价	优先级	级别
IDS08-J	中	不可能	中等	P4	L3

2.9.5　相关规范

MITRE CWE　　CWE-625 有权限的正则表达式

2.9.6　参考书目

[Tutorials 08]　正则表达式

[CVE 05]　CVE-2005-1949

2.10　IDS09-J 如果没有指定适当的 locale，不要使用 locale 相关方法处理与 locale 相关的数据

当 locale 没有明确指定的时候，使用 locale 相关的方法处理与 locale 相关的数据会产生意想

不到的后果。编程语言的标示符、协议关键字以及 HTML 标签通常会指定 Locale.ENGLISH 作为一个特定的 locale。当改变默认的 locale 的时候，很有可能使数据绕过检查，从而改变 locale 相关方法的行为。比如，当把一个字符串转换为大写字符时，可认为它是合法的，然而，当把它接着转换回小写字符时，可能就会产生黑名单字符串。

任何一个程序调用 locale 相关方法时，如果使用非受信数据，必须显式指明使用这些方法的 locale。

2.10.1 不符合规则的代码示例

这个代码示例中使用了 locale 相关的 String.toUpperCase() 方法来将 HTML 标签转换为大写字符。在英文 locale 中，会将“title”转换为“TITLE”，在土耳其 locale 中，会将“title”转换为“T?TLE”，其中的“?”是拉丁文字母“I”，它在字符上是有一个点的 [API 2006]。

```
"title".toUpperCase();
```

2.10.2 符合规则的方案（显式的 locale）

该方案显式地将 locale 设置为英文，从而避免产生意想不到的问题。

```
"title".toUpperCase(Locale.ENGLISH);
```

这条规则同样适用于 String.equalsIgnoreCase() 方法。

2.10.3 符合规则的方案（默认 locale）

符合规则的方案在对字符串数据进行处理之前，把默认的 locale 设置为 English。

```
Locale.setDefault(Locale.ENGLISH);
"title".toUpperCase();
```

2.10.4 风险评估

如果没有指定合适的 locale，那么当使用 locale 相关方法处理与 locale 相关的数据时，会产生意外的行为。

规则	严重性	可能性	弥补代价	优先级	级别
IDS09-J	中	可能	中等	P8	L2

2.10.5 参考书目

[API 2006] String 类

2.11 IDS10-J 不要拆分两种数据结构中的字符串

在历史遗留系统中，常常假设字符串中的每一个字符使用 8 位（一个字节，Java 中的 byte）。而 Java 语言使用 16 位表示一个字符（Java 中的 Char 类型）。遗憾的是，不管是 Java 的 byte 类型还是 char 类型数据，都不能表示所有的 Unicode 字符。许多字符串使用例如 UTF-8 编码的方式存储和通信，而在这种编码中，字符长度是可变的。

当 Java 字符串以字符数组的方式存储时，它可以用一个字节数组来表示，字符串里的一个

字符可以用两个连续的或更多的 byte 类型或者 char 类型表示。如果拆分一个 char 类型或 byte 类型的数组，将会对多字节的字符产生风险。

如果忽略那些补充字符（supplementary character）多字节字符或者整合字符（修改其他字符的那些字符），攻击者可能绕过输入验证。因此，不应该拆分两种数据结构中的字符。

2.11.1　多字节字符

在一些字符集中会使用多字节字符编码，这些字符集要求用一个以上的字节来唯一标识每一个字符。比如，在日文 Shift-JIS 编码中，就支持多字节编码，其最大的字符长度为 2 个字节（一个起始字节，一个结尾字节）。

字节类型	范围
单字节	0x00 ~ 0x7F，0xA0 ~ 0xDF
起始字节	0x81 ~ 0x9F，0xE0 ~ 0xFC
结尾字节	0x40 ~ 0x7E，0x80 ~ 0xFC

对结尾字节而言，它可能覆盖单个字节或者多字节字符的起始字节。当一个多字节字符被分拆时，特别是跨不同的缓冲区边界分拆时，它会产生不同的解释，如果没有按照正常的缓冲区边界进行分拆的话。这种差异一般由构造字符时使用的字节有二义性造成。

2.11.2　补充字符

根据 Java API[API 2006] 对 Character 类的描述（Unicode 的字符表示）：

Char 数据类型（和 Character 对象封装的值）需要依赖于初始的 Unicode 编码定义，其中将其定义为长度为 16 位的字符编码。Unicode 编码后来经过改动，允许使用多于 16 位来表示字符编码。合法的字符编码位于 \u0000 ～ \u10FFFF，这就是我们所熟悉的 Unicode 字符编码值。

Java 2 平台在 char 数组、String 和 StringBuffer 类中使用 UTF-16 编码表示。在这种字符表示中，补充字符会用一对 char 值来表示，第一个高位字符范围是 \uD800 ～ \uDBFF，第二个低位字符范围是 \uDC00 ～ \uDFFF。

一个 int 值可以表示所有的 Unicode 编码字符，包括那些补充码。int 类型中的最低 21 位是用来表示 Unicode 编码的，其他的 11 个高位必须为 0。除非特指，关于补充码和字符值有下面的规则：

- 那些只能接受 char 值的方法是不支持补充码的。替代范围内的 char 值会被认为是未定义的字符。比如，Character.isLetter('\uD840') 会返回 false，即使这种特殊字符紧跟着任何表示字母的字符串的低位替代值。
- 接受 int 值的方法支持所有的 Unicode 字符，包括字符。比如，Character.isLetter(0x2F81A) 会返回 true，因为这个码点值代表一个字母（在 CJK 编码中）。

2.11.3　不符合规则的代码示例（读取）

这个不符合规则的代码示例会从一个套接字中读取 1024 个字节，并且使用这些数据创建一

个字符串。它同时使用一个 while 循环来读取这些字节，就像在 FIO10-J 中推荐的那样。当检测到套接字中的数据多于 1024 个字节时，它就会抛出异常。这样的机制能够防止非受信的输入耗尽程序的内存。

```
public final int MAX_SIZE = 1024;

public String readBytes(Socket socket) throws IOException {
  InputStream in = socket.getInputStream();
  byte[] data = new byte[MAX_SIZE+1];
  int offset = 0;
  int bytesRead = 0;
  String str = new String();
  while ((bytesRead = in.read(data, offset, data.length - offset))
          != -1) {
    offset += bytesRead;
    str += new String(data, offset, data.length - offset, "UTF-8");
    if (offset >= data.length) {
      throw new IOException("Too much input");
    }
  }
  in.close();
  return str;
}
```

这个代码示例没有考虑到使用多字节编码的字符和循环选代边界之间的关系。如果在最后一个通过 read() 方法读取的数据流中存在一个对字节编码的首字节，那么其他的字节只能在循环的下一次处理。然而，可以通过在一个循环中创建一个新的字符串来解决多字节编码问题。这样的话，多字节编码就可能会被错误地解释。

2.11.4 符合规则的方案（读取）

该符合规则的方案将字符串的创建推迟到接收完所有的数据时才完成。

```
public final int MAX_SIZE = 1024;

public String readBytes(Socket socket) throws IOException {
  InputStream in = socket.getInputStream();
  byte[] data = new byte[MAX_SIZE+1];
  int offset = 0;
  int bytesRead = 0;
  while ((bytesRead = in.read(data, offset, data.length - offset))
          != -1) {
    offset += bytesRead;
    if (offset >= data.length) {
      throw new IOException("Too much input");
    }
  }
  String str = new String(data, "UTF-8");
  in.close();
  return str;
}
```

这段代码避免了将跨不同缓冲区的多字节编码字符分隔的问题，使用的方法是，直到读取完

所有的数据，才开始创建字符串。

2.11.5　符合规则的方案（Reader）

这个符合规则的方案使用的是 Reader 而不是 InputStream。这个 Reader 类会快速地将字节数据转换为字符数据，所以能够避免分隔多字节字符的问题。当套接字使用多于 1024 个字符而不是恰好使用 1024 字节时，这个例程会自动退出。

```
public final int MAX_SIZE = 1024;

public String readBytes(Socket socket) throws IOException {
  InputStream in = socket.getInputStream();
  Reader r = new InputStreamReader(in, "UTF-8");
  char[] data = new char[MAX_SIZE+1];
  int offset = 0;
  int charsRead = 0;
  String str = new String(data);
  while ((charsRead = r.read(data, offset, data.length - offset))
         != -1) {
    offset += charsRead;
    str += new String(data, offset, data.length - offset);
    if (offset >= data.length) {
      throw new IOException("Too much input");
    }
  }
  in.close();
  return str;
}
```

2.11.6　不符合规则的代码示例（子字符串）

这个不符合规则的代码示例想截取字符串中的首字母。它的做法不对，因为使用了 Character.isLetter() 方法，这个方法不能处理补充字符和合并字符 [Hornig 2007]。

```
// Fails for supplementary or combining characters
public static String trim_bad1(String string) {
  char ch;
  int i;
  for (i = 0; i < string.length(); i += 1) {
    ch = string.charAt(i);
    if (!Character.isLetter(ch)) {
      break;
    }
  }
  return string.substring(i);
}
```

2.11.7　不符合规则的代码示例（子字符串）

这个不符合规则的代码示例想要纠正使用 String.codePointAt() 的错误，这个方法使用了 int 类型作为输入参数。这对补充字符来说是正确的，但对于合并字符而言却是错误的 [Hornig 2007]。

```
// Fails for combining characters
public static String trim_bad2(String string) {
```

```
    int ch;
    int i;
    for (i = 0; i < string.length(); i += Character.charCount(ch)) {
      ch = string.codePointAt(i);
      if (!Character.isLetter(ch)) {
        break;
      }
    }
    return string.substring(i);
  }
```

2.11.8 符合规则的方案（子字符串）

这个方案可以处理补充字符和合并字符 [Hornig 2007]。根据 Java API[API 2006] 文档对 java.text.BreakIterator 的说明：

> BreakIterator 实现了能够在文本边界内定位的方法。BreakIterator 的对象实例维护了当前的位置信息，并且会对文本进行扫描，当遇到文本边界的时候，它会返回字符所在的位置索引。

返回的边界可能是那些补充字符、合并字符。例如，一个音节字符可以被存储为一个基础字符加上一个用来区分的符号。

```
  public static String trim_good(String string) {
    BreakIterator iter = BreakIterator.getCharacterInstance();
    iter.setText(string);
    int i;
    for (i = iter.first(); i != BreakIterator.DONE; i = iter.next()) {
      int ch = string.codePointAt(i);
      if (!Character.isLetter(ch)) {
        break;
      }
    }
    // Reached first or last text boundary
    if (i == BreakIterator.DONE) {
      // The input was either blank or had only (leading) letters
      return "";
    } else {
      return string.substring(i);
    }
  }
```

如果要对 locale 敏感的字符串比较、搜索和排序，可以使用 java.text.Collator 类。

2.11.9 风险评估

如果没有考虑到补充字符和合并字符的话，将会导致不可预计的行为。

规则	严重性	可能性	弥补代价	优先级	级别
IDS10-J	低	不可能	中等	P2	L3

2.11.10 参考书目

[API 2006] Character 类和 BreakIterator 类

[Hornig 2007] 问题区域：字符

2.12　IDS11-J 在验证前去掉非字符码点

在早于 Unicode 5.2 的版本中，条款 C7 允许删除非字符码点。比如，在 Unicode 5.1 版本的 C7 条款中：

> 条款 C7 当一个进程声称不会修改一个合法编码字符序列的意思时，它不应该改变该字符编码序列，除了有可能使用标准化字符编码来取代字符序列，或是删除非字符编码之外。

根据 Unicode 技术报告第 36 号的 3.5 节“删除非字符编码”，在考虑 Unicode 的安全的时候 [Davis 2008b]：

> 不管一个字符是否被直接删除（不是替代），例如在 C7 的旧版本中提及的，将会导致安全问题。这个问题是这样的：一个网关可能会对所有的敏感字符串进行处理，比如“delete”。如果传递过来的是“delXlete”。这里“X”是非字符，网关会让它通过：序列“deXlete”可能通过并无害。然而，假设在通过网关之后，一个内部进程直接删除了 X。这时，就形成了一个敏感的字符序列，并且会造成安全威胁。

修改任何一个字符串，包括对非字符数据的移除或者替代，必须在对该字符串进行验证之前进行。

2.12.1　不符合规则的代码示例

这个代码示例只接受合法的 ASCII 字符，并且它会删除所有的非 ASCII 码字符。同时，它也会检查是否存在 <script> 标签。

在对输入进行验证之后，会删除那些非 ASCII 码字符。接着，攻击者就可以伪装一个 <script> 标签，从而绕过验证检查。

```
// "\uFEFF" is a non-character code point
String s = "<scr" + "\uFEFF" + "ipt>";
s = Normalizer.normalize(s, Form.NFKC);
// Input validation
Pattern pattern = Pattern.compile("<script>");
Matcher matcher = pattern.matcher(s);
if (matcher.find()) {
  System.out.println("Found black listed tag");
} else {
  // ...
}

// Deletes all non-valid characters
s = s.replaceAll("^\\p{ASCII}]", "");
// s now contains "<script>"
```

2.12.2　符合规则的方案

这个方案使用 Unicode 字符序列 \uFFFD 来替代那些未知的或者不可表示的字符。同时，这样的替代是在其他净化之前完成的，比如对 <script> 标签的检查，因而它能够保证恶意输入不能绕过数据过滤器。

```
String s = "<scr" + "\uFEFF" + "ipt>";

s = Normalizer.normalize(s, Form.NFKC);
// Replaces all non-valid characters with unicode U+FFFD
s = s.replaceAll("^\\p{ASCII}]", "\uFFFD");

Pattern pattern = Pattern.compile("<script>");
Matcher matcher = pattern.matcher(s);
if (matcher.find()) {
  System.out.println("Found blacklisted tag");
} else {
  // ...
}
```

根据 Unicode 技术报告第 36 号，考虑 Unicode 编码的安全问题 [Davis 2008b]，“U+FFFD 通常是没有问题的，因为它就是设计成这样使用的。也就是说，不管是在程序开发语言，还是在结构数据中，因为不会有任何语义上的含义，通常会在解析时出现错误。当输出字符集不是 Unicode 编码时，这个字符可能是不存在。”

2.12.3 风险评估

将非字符编码删除会允许恶意输入绕过验证检查。

规则	严重性	可能性	弥补代价	优先级	级别
IDS11-J	高	可能	中等	P12	L1

2.12.4 相关规范

MITRE CWE	CWE-182 将数据填塞到不安全的值中

2.12.5 参考书目

[API 2006]	
[Davis 2008b]	3.5 节 删除非字符串编码
[Weber 2009]	处理意外情况：字符串删除
[Unicode 2007]	
[Unicode 2011]	

2.13 IDS12-J 在不同的字符编码中无损转换字符串数据

在 String 对象之间进行转换时，如果涉及不同的编码类型，可能会导致数据丢失。

根据 Java API[API 2006] 对 String.getBytes(Charset) 方法的描述：

> 该方法总会替代那些错误格式的输入和不可映射的字符序列，把它们替换成这些字符的字节数组。

当必须将一个 String 转化为字节数组时，例如写入一个文件，并且在这个字符串中含有不可映射的字符序列的时候，就必须进行正确的字符编码。

2.13.1　不符合规则的代码示例

当 String 包含那些指定在 charset 中不能正常表示的字符时，这个不符合规则的代码示例 [Hornig 2007] 会破坏数据。

```
// Corrupts data on errors
public static byte[] toCodePage_bad(String charset, String string)
  throws UnsupportedEncodingException {
  return string.getBytes(charset);
}

// Fails to detect corrupt data
public static String fromCodePage_bad(String charset, byte[] bytes)
  throws UnsupportedEncodingException {
  return new String(bytes, charset);
}
```

2.13.2　符合规则的方案

java.nio.charset.CharsetEncoder 类可以将一个 16 位的 Unicode 字符转换为指定 Charset 的一组字节数据。java.nio.charset.Character-Decoder 用来完成相反的过程 [API 2006]。具体的详情可以参考规则 FIO11-J。

这个方案 [Hornig 2007] 使用了 CharsetEncoder 和 CharsetDecoder 两个类来处理编码转换问题。

```
public static byte[] toCodePage_good(String charset, String string)
  throws IOException {

  Charset cs = Charset.forName(charset);
  CharsetEncoder coder = cs.newEncoder();
  ByteBuffer bytebuf = coder.encode(CharBuffer.wrap(string));
  byte[] bytes = new byte[bytebuf.limit()];
  bytebuf.get(bytes);
  return bytes;
}

public static String fromCodePage_good(String charset,byte[] bytes)
  throws CharacterCodingException {

  Charset cs = Charset.forName(charset);
  CharsetDecoder coder = cs.newDecoder();
  CharBuffer charbuf = coder.decode(ByteBuffer.wrap(bytes));
  return charbuf.toString();
}
```

2.13.3　不符合规则的代码示例

这个代码示例 [Hornig 2007] 想要将一个字符串以特殊的编码方式附加到一个文本文件上，但这样会导致错误，因为 String 中可能包含无法正确表示的字符。

```
// Corrupts data on errors
public static void toFile_bad(String charset, String filename,
                              String string) throws IOException {
```

```
  FileOutputStream stream = new FileOutputStream(filename, true);
  OutputStreamWriter writer = new OutputStreamWriter(stream, charset);
  writer.write(string, 0, string.length());
  writer.close();
}
```

2.13.4 符合规则的方案

这个方案 [Hornig 2007] 使用 CharsetEncoder 类来完成所需要的功能。

```
public static void toFile_good(String filename, String string,
                               String charset) throws IOException {

  Charset cs = Charset.forName(charset);
  CharsetEncoder coder = cs.newEncoder();
  FileOutputStream stream = new FileOutputStream(filename, true);
  OutputStreamWriter writer = new OutputStreamWriter(stream, coder);
  writer.write(string, 0, string.length());
  writer.close();
}
```

可以使用 FileInputStream 和 InputStreamReader 对象来从文件中读取数据。这个 InputStreamReader 对象接受可选的 CharsetDecoder 作为参数，但这个参数必须和前面写入文件时保持一致。

2.13.5 风险评估

使用非标准化的方法来处理和字符集转换相关的问题，通常会导致数据丢失。

规则	严重性	可能性	弥补代价	优先级	级别
IDS12-J	低	可能	中等	P4	L3

2.13.6 相关规范

MITRE CWE　　CWE-838 对输出场景的不正确的编码
　　CWE-116 不正确的编码或者输出转义

2.13.7 参考书目

[API 2006]　　String 类
[Hornig 2007]　　全局问题区域：字符编码

2.14 IDS13-J 在文件或者网络 I/O 两端使用兼容的编码方式

每一个 Java 平台都有自己默认的编码方式。“Supported Encodings”文档中列举了一些可能出现的编码方式 [Encodings 2006]。在字符和一组字节序列之间进行编码转换时，需要通过字符编码方式指定转换细节。当没有显式指定编码方式时，会使用系统默认的编码方式完成这种转换。首先将字符转换为一个字节数组，然后将数组送到输入设备，经过通信通道进行传输，由输入设备接收，最后转换为字符。在这两次转换中，要使用兼容并且一致的编码。如果没有使用一致的编码，会破坏数据。

根据 Java API[API 2006] 对 String 类的描述：

一个新的 String 的长度是与字符集有关的，并且由于这个原因，它可能会与字节数组的长度不同。当指定的字节不适用于指定的字符集时，String 构造器表现出来的行为是不确定的。

二进制数据被认为是合法的字符串，它会被读取并转换为字符串，这是规则 FIO11-EX0 特例。

2.14.1　不符合规则的代码示例

下面的代码示例读取一个字节数组，然后使用平台默认的字符编码将其转换为字符串。当默认的字符编码和用来产生字节数据的编码不同的时候，产生的字符串可能是不正确的。当某些输入在默认的编码中没有有效的字符表示时，也会产生不确定的行为。

```
FileInputStream fis = null;
try {
  fis = new FileInputStream("SomeFile");
  DataInputStream dis = new DataInputStream(fis);
  byte[] data = new byte[1024];
  dis.readFully(data);
  String result = new String(data);
} catch (IOException x) {
  // handle error
} finally {
  if (fis != null) {
    try {
      fis.close();
    } catch (IOException x) {
      // Forward to handler
    }
  }
}
```


2.14.2　符合规则的方案

该方案在 String 构造函数的第二个参数中明确指定了需要的字符编码。

```
FileInputStream fis = null;
try {
  fis = new FileInputStream("SomeFile");
  DataInputStream dis = new DataInputStream(fis);
  byte[] data = new byte[1024];
  dis.readFully(data);
  String encoding = "SomeEncoding"; // for example, "UTF-16LE"
  String result = new String(data, encoding);
} catch (IOException x) {
  // handle error
} finally {
  if (fis != null) {
    try {
      fis.close();
    } catch (IOException x) {
      // Forward to handler
    }
  }
}
```

2.14.3 特例

IDS13-EX0：当 Java 应用使用相同平台和默认的字符编码来产生数据，并且通过一个加密的通信通道（详情请参考 MSC00-J）进行传输时，可能会在接收侧忽略一个显式的字符编码。

2.14.4 风险评估

对文件或者网络 I/O 进行操作时，如果没有明确指定字符编码，那么会导致数据被破坏。

规则	严重性	可能性	弥补代价	优先级	级别
IDS13-J	低	不可能	中等	P2	L3

自动化检测 通过自动检测来发现这个漏洞是不可行的。

2.14.5 参考书目

[Encodings 2006]

第 3 章
声明和初始化（DCL）

规则

规　则
DCL00-J 防止类的循环初始化
DCL01-J 不要重用 Java 标准库的已经公开的标识
DCL02-J 将所有增强 for 语句的循环变量声明为 final 类型

风险评估概要

规则	严重性	可能性	弥补代价	优先级	级别
DCL01-J	低	不可能	中等	P2	L3
DCL02-J	低	不可能	中等	P2	L3
DCL03-J	低	不可能	低等	P3	L3

3.1　DCL00-J 防止类的循环初始化

在 Java 语言规范（Java Language Specification，JLS）第 12.4 节“对类和接口的初始化”中提到 [JLS 2005]：

> 对类进行的初始化包括执行该类的 static 静态初始化方法和初始化该类中的静态数据成员（类变量）。

换句话说，一个静态数据成员的出现会触发类的初始化。然而，一个静态数据成员可能会依赖于其他类的初始化，这样有可能形成一个初始化循环。在 JLS 的 8.3.2.1 节“类变量的初始化”中也提到 [JLS 2005]：

> 在运行态中，使用编译期的常量来初始化的 final 的 static 变量，是最先初始化的。

这个 JLS 的描述会让人误解。误解之处在于，对于某些对象实例而言，变量就算是 static final 的，它们的初始化也可能会安排在后期进行。声明一个字段为 static final 并不能够保证它在被读之前已经完全初始化。

一般来说，程序，特别是对安全敏感的程序，必须消除所有的类初始化循环。

3.1.1 不符合规则的代码示例（类内循环）

不符合规则的代码示例如下，它存在一个涉及多个类的类初始化循环。

```
public class Cycle {
  private final int balance;
  private static final Cycle c = new Cycle();
  // Random deposit
  private static final int deposit = (int) (Math.random() * 100);

  public Cycle() {
    balance = deposit - 10; // Subtract processing fee
  }

  public static void main(String[] args) {
    System.out.println("The account balance is: " + c.balance);
  }
}
```

Cycle 类声明了一个 private static final 类变量，这个变量会在创建 Cycle 对象的时候进行初始化。静态的初始化方法可以保证只调用一次，而这次调用是在第一次使用静态类变量或者在第一次调用构造函数之前发生。

程序员希望通过在存款账户中减去处理费用来计算账户余额。然而，对类变量 c 的初始化在 deposit 变量被初始化之前发生，因为在编码上，它在 deposit 域的初始化之前出现。因此，当对变量 C 进行静态初始化时，Cycle 类的构造函数会读取 deposit 的数值，这个 deposit 数值会是 0 而并非一个随机值。结果是，账户余额计算下来的值是 –10。

JLS 允许实现忽略这种可能出现的循环的初始化 [Bloch 2005a]。

3.1.2 符合规则的方案（类内循环）

这个方案改变了类 Cycle 的初始化次序，所以对数据成员的初始化是不会造成任何依赖循环的。特别是对 c 的初始化，因为在编码上处于对 deposit 初始化之后发生，因而它会在 deposit 被完全初始化之后进行初始化。

```
public class Cycle {
  private final int balance;
  // Random deposit
  private static final int deposit = (int) (Math.random() * 100);
  // Inserted after initialization of required fields
  private static final Cycle c = new Cycle();
  public Cycle() {
    balance = deposit - 10; // Subtract processing fee
  }

  public static void main(String[] args) {
    System.out.println("The account balance is: " + c.balance);
  }
}
```

当涉及许多字段的时候，并不容易发现这样的初始化循环。因此，确保控制流不产生这样的循环就非常重要。

尽管这个方案可以防止初始化循环，但它也依赖于声明次序，因而也是脆弱的。程序的维护者可能不知道这种声明的次序必须被保持来保证程序的正确性。因而，这种依赖必须在代码的文档中清楚地说明。

3.1.3　不符合规则的代码示例（类间循环）

这个不符合规则的代码示例声明了两个类，这两个类都有静态变量，这些变量的值是互相依赖的。当把这两个类放在一起的时候，这个循环是很显然的；而当把它们分开的时候，却很容易忽略这点。

```
class A {
  public static final int a = B.b + 1;
  // ...
}

class B {
  public static final int b = A.a + 1;
  // ...
}
```

因为对这些类的初始化次序是可变的，所以导致的结果是，会计算出不同的 A.a 和 B.b 的值。当先初始化 A 类时，A.a 的值是 2，而 B.b 的值是 1。当先初始化 B 类时，这两个值就要反过来了。

3.1.4　符合规则的方案（类间循环）

与规则一致的方案打破了这个类间循环，这是通过消除其中一个依赖来完成的。

```
class A {
  public static final int a = 2;
  // ...
}
// class B unchanged: b = A.a + 1
```

当打破这个循环时，初始化的值是不变的，总是 A.a=2 且 B.b=3，并且它与初始化次序无关。

3.1.5　风险评估

初始化循环会导致不可预测的结果。

规则	严重性	可能性	弥补代价	优先级	级别
DCL00-J	低	不可能	中等	P2	L3

3.1.6　相关规范

CERT C++ 安全编码标准　DCL14-CPP 不要假设编译单元之间的初始化次序

ISO/IEC TR 24772:2010　变量的初始化 [LAV]

3.1.7　参考书目

[JLS 2005]　8.3.2.1 节 对类变量的初始化

12.4 节 对类和接口的初始化

[Bloch 2005a]　疑难问题 49 大于生命

[MITRE 2009] CWE-665 不正确的初始化

3.2 DCL01-J 不要重用 Java 标准库的已经公开的标识

不要重用那些在 Java 标准库中已经使用过的公共的标识、公共的工具类、接口或者包。

当一个程序员使用和公开类相同的名字时，如 Vector 对后来的维护者来说，他可能不知道这个标识并不是指 java.util.Vector ，并且可能会无意地使用这个自定义的 Vector 类而不是原有的 java.util.Vector 类。使得这个自定义的 Vector 会遮蔽 java.util.Vector 类，正如在 JLS 的 6.3.2 节中提到的那样。从而会导致不可预期的程序行为。

良好定义的 import 语句可以解决这个问题。然而，当重用的命名定义是从其他包中导入时，使用 type-import-on-demand declaration（详细参见 JLS 的 7.5.2 节“Type-Import-on-Demand Declaration”[JLS 2005]）会将程序员弄糊涂，因为他需要确定哪一个定义是想要的。另外，因为我们通常会使 IDE 来自动包括 import 语句，所以一种常见的操作是在编写代码后才生成这些 import 语句，但这种做法容易导致错误。在 Java 包含的 import 引用路径中，如果在预期的类出现之前出现了一个自定义的类，那么不会进一步搜索，这样就会毫无知觉地使用了错误的类。

3.2.1 不符合规则的代码示例（类名称）

这个不符合规则的代码实现了一个类，这个类重用了 java.util.Vector 的名称。它在 isEmpty() 中使用了一个不同的条件判断，尝试通过重写 java.util.Vector 中对应的方法来达到与遗留代码接口的目的。如果维护者混淆了这个 isEmpty() 和 java.util.Vector. isEmpty()，就会出现不可预知的行为。

```
class Vector {
  private int val = 1;

  public boolean isEmpty() {
    if (val == 1) {   // compares with 1 instead of 0
      return true;
    } else {
      return false;
    }
  }
  // other functionality is same as java.util.Vector
}

// import java.util.Vector; omitted
public class VectorUser {
  public static void main(String[] args) {
    Vector v = new Vector();
    if (v.isEmpty()) {
      System.out.println("Vector is empty");
    }
  }
}
```

3.2.2 符合规则的方案（类名称）

这个符合规则的方案为这个类使用了不同的名字，防止这个类作为任何潜在的 Java 标准类库中的类名称的遮蔽情况出现。

```
class MyVector {
  // other code
}
```

当程序员和开发团队可以控制那些被模仿的原始类，更好的方法就是改变对它们的设计策略，这些策略可以根据 Bloch 的《Effective Java》[Bloch 2008] 一书中第 16 条，更倾向于接口而不是抽象类的方法来实现。将原始类改变成接口，这可以让 MyVector 类得以通过声明它是我们假设的接口 Vector 的实现。这可以让使用 MyVector 的代码与使用原始 Vector 实现的代码互相兼容。

3.2.3　风险评估

重用公有的标识会降低代码的可读性和可维护性。

规则	严重性	可能性	弥补代价	优先级	级别
DCL01-J	低	不可能	中等	P2	L3

自动化检测　对于重用了 Java 标准类库中公共类和接口的名字的问题，它们可以被自动检测工具很容易地检测到。

3.2.4　相关规范

CERT C 安全编码标准	PRE04-C 不要重用一个标准的头文件名
CERT C++ 安全编码标准	PRE04-CPP 不要重用一个标准的头文件名

3.2.5　参考书目

[JLS 2005]	6.3.2 节 模糊的声明 6.3.1 节 影子声明 7.5.2 节 按需的类型导入声明 14.4.3 节 使用局部变量的影子命名
[FindBugs 2008]	
[Bloch 2005a]	疑难问题 67 所有的串导出
[Bloch 2008]	第 16 个项目更倾向于使用接口而不是抽象类

3.3　DCL02-J 将所有增强 for 语句的循环变量声明为 final 类型

Java 5 平台（也因 for-each 风格出名）引入了增强的 for 语句，它用来对对象集合进行迭代。与基本的 for 语句不同，在基本的 for 语句中，给循环变量赋值是不能对循环的迭代次序有所影响。但在增强的 for 语句中，给循环变量赋值就可以有影响，而不是像程序员通常认为的那样。这使我们认识到应避免给在 for 循环中的循环变量赋值。

详情请见 JLS 的 14.14.2 节“增强的 for 语句”[JLS 2005]。

一个增强的 for 循环通常采用以下这种形式：

```
for (ObjType obj : someIterableItem) {

  // ...
}
```

这等价于以下形式的基本 for 循环：

```
for (Iterator myIterator = someIterableItem.iterator();
     myIterator.hasNext();) {
 ObjType obj = myIterator.next();
 // ...
}
```

所以，一个给循环变量赋值的动作，等价于修改循环体的局部变量的值，这个变量的初始值会被循环迭代器引用。这种修改不一定是错误的，但它会模糊循环的行为，或者意味着对增强型 for 语句的内在实现，这在理解上有问题。

可以将所有的 for 语句中的循环变量声明为 final。这个 final 声明可以让 Java 编译器做一个标志，并且拒绝对这个循环变量的任何赋值。

3.3.1 不符合规则的代码示例

这个不符合规则的代码示例想要使用增强型的 for 循环处理对象集合。此外，它还希望跳过对集合中某一个元素的处理。

```
Collection<ProcessObj> processThese = // ...

for (ProcessObj processMe: processThese) {
  if (someCondition) { // found the item to skip
    someCondition = false;
    processMe = processMe.getNext(); // attempt to skip to next item
  }
  processMe.doTheProcessing(); // process the object
}
```

这种跳过到下一个集合元素的想法看起来是可以实现的，因为已经成功赋值，并且 processMe 变量的值也更新了。然而，和基本型的 for 循环不同，这个赋值并没有改变循环执行的迭代次序。因此，虽然需要跳过的元素跳过了，但是它之后的元素被处理了两次。

注意，如果声明 processMe 为 final，在试图对它进行这样的赋值时，就会产生编译器错误。

3.3.2 符合规则的方案

这个符合规则的方案可以正确地处理在集合中的每一个对象，并且每一个对象只会被处理一次。

```
Collection<ProcessObj> processThese = // ...

for (final ProcessObj processMe: processThese) {
  if (someCondition) { // found the item to skip
    someCondition = false;
    continue; // skip by continuing to next iteration
  }
  processMe.doTheProcessing(); // process the object
}
```

3.3.3 风险评估

给增强型的 for 循环的循环变量赋值，不影响整体的迭代次序，这样会导致程序员产生困惑，

并且会让数据的状态不一致。

规则	严重性	可能性	弥补代价	优先级	级别
IDS00-J	低	不可能	低等	P3	L3

自动化检查　这条规则可以很容易通过使用静态分析来实行。

3.3.4　参考书目

[JLS 2005]　14.14.2 节 增强的 for 语句