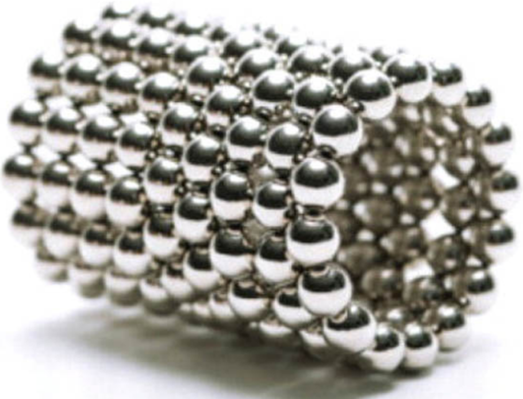
SEI SERIES • A CERT® BOOK

The CERT® Oracle® Secure Coding Standard for Java



Fred Long | Dhruv Mohlndra | Robert C. Seacord
Dean F. Sutherland | David Svoboda

The CERT® Oracle® Secure Coding Standard for Java™

The CERT® Oracle® Secure Coding Standard for Java™

Fred Long
Dhruv Mohindra
Robert C. Seacord
Dean F. Sutherland
David Svoboda

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Software Engineering Institute | Carnegie Mellon

The SEI Series in Software Engineering

Library of Congress Cataloging-in-Publication Data

The CERT Oracle secure coding standard for Java / Fred Long . . . [et al.].
p. cm.—(The SEI series in software engineering)
Includes bibliographical references and index.
ISBN-13: 978-0-321-80395-5 (pbk. : alk. paper)
ISBN-10: 0-321-80395-7 (pbk. : alk. paper)
1. Java (Computer program language) 2. Computer security. 3. Oracle (Computer file) 4. Computer programming—Standards. I. Long, F. W. (Frederick W.), 1947- II. Carnegie-Mellon University. CERT Coordination Center.
QA76.73.J38C44 2012
005.8—dc23
2011027284

ISBN-13: 978-0-321-80395-5
ISBN-10: 0-321-80395-7
Text printed in the United States on recycled paper at Edwards Brothers in Ann Arbor, Michigan.
First printing, September 2011

To my late wife, Ann, for all her love, help, and support over the years.
—Fred Long

To my parents Deepak and Eta Mohindra, my grandmother
Shashi Mohindra, and our very peppy, spotted Dalmatian Google.
—Dhruv Mohindra

To my wife, Alfie, for making this book worthwhile, and
to my parents, Bill and Lois, for making it possible.
—David Svoboda

To my wife, Rhonda, and our children, Chelsea and Jordan.
—Robert C. Seacord

For Libby, who makes everything worthwhile.
—Dean Sutherland

This page intentionally left blank

Contents

This page intentionally left blank

Foreword

James Gosling

Security in computer systems has been a serious issue for decades. This past decade's explosion in the dependence on networks and the computers connected to them has raised the issue to stratospheric levels. When Java was first designed, dealing with security was a key component. And in the years since then, all of the various standard libraries, frameworks, and containers that have been built have had to deal with security too. In the Java world, security is not viewed as an add-on feature. It is a pervasive way of thinking. Those who forget to think in a secure mindset end up in trouble.

But just because the facilities are there doesn't mean that security is assured automatically. A set of standard practices has evolved over the years. *The CERT® Oracle® Secure Coding Standard for Java™* is a compendium of these practices. These are not theoretical research papers or product marketing blurbs. This is all serious, mission-critical, battle-tested, enterprise-scale stuff.

This page intentionally left blank

Preface

An essential element of secure coding in the Java programming language is a well-documented and enforceable coding standard. The CERT Oracle Secure Coding Standard for Java provides rules for secure coding in the Java programming language. The goal of these rules is to eliminate insecure coding practices that can lead to exploitable vulnerabilities. The application of the secure coding standard leads to higher quality systems that are safe, secure, reliable, dependable, robust, resilient, available, and maintainable and can be used as a metric to evaluate source code for these properties (using manual or automated processes).

This coding standard affects a wide range of software systems developed in the Java programming language.

Scope

The CERT Oracle Secure Coding Standard for Java focuses on the Java Standard Edition 6 Platform (Java SE 6) environment and includes rules for secure coding using the Java programming language and libraries. *The Java Language Specification*, 3rd edition [JLS 2005] prescribes the behavior of the Java programming language and served as the primary reference for the development of this standard. This coding standard also addresses new features of the Java SE 7 Platform. Primarily, these features provide alternative compliant solutions to secure coding problems that exist in both the Java SE 6 and Java SE 7 platforms.

Languages such as C and C++ allow undefined, unspecified, or implementation-defined behaviors, which can lead to vulnerabilities when a programmer makes incorrect assumptions about the underlying behavior of an API or language construct. The Java Language Specification goes further to standardize language requirements because Java is designed to be a “write once, run anywhere” language. Even then, certain behaviors are left to the discretion of the implementor of the Java Virtual Machine (JVM) or the Java compiler. This standard identifies such language peculiarities and demonstrates secure coding practices to avoid them.

Focusing only on language issues does not translate to writing secure software. Design flaws in Java application programming interfaces (APIs) sometimes lead to their deprecation. At other times, the APIs or the relevant documentation may be interpreted incorrectly by the programming community. This standard identifies such problematic APIs and highlights their correct use. Examples of commonly used faulty design patterns (anti-patterns) and idioms are also included.

The Java language, its core and extension APIs, and the JVM provide security features such as the security manager, access controller, cryptography, automatic memory management, strong type checking, and bytecode verification. These features provide sufficient security for most applications, but their proper use is of paramount importance. This standard highlights the pitfalls and caveats associated with the security architecture and stresses its correct implementation. Adherence to this standard safeguards the confidentiality, integrity, and availability (CIA) of trusted programs and helps eliminate exploitable security flaws that can result in denial-of-service attacks, time-of-check-to-time-of-use attacks, information leaks, erroneous computations, and privilege escalation.

Software that complies with this standard provides its users the ability to define fine-grained security policies and safely execute trusted mobile code on untrusted systems or untrusted mobile code on trusted systems.

Included Libraries

This secure coding standard addresses security issues primarily applicable to the `lang` and `util` libraries, as well as to the Collections, Concurrency Utilities, Logging, Management, Reflection, Regular Expressions, Zip, I/O, JMX, JNI, Math, Serialization, and XML JAXP libraries. This standard avoids the inclusion of open bugs that have already been fixed or those that lack security ramifications. A functional bug is included only when it is likely that it occurs with high frequency, causes considerable security concerns, or affects most Java technologies that rely on the core platform. This standard is not limited to security issues specific to the Core API but also includes important security concerns pertaining to the standard extension APIs (`javax` package).

Issues Not Addressed

The following issues are not addressed by this standard:

- **Design and Architecture.** This standard assumes that the design and architecture of the product is secure—that is, that the product is free of design-level vulnerabilities that would otherwise compromise its security.
- **Content.** This coding standard does not address concerns specific to only one Java-based platform but applies broadly to all platforms. For example, rules that are applicable to Java Micro Edition (ME) or Java Enterprise Edition (EE) alone and not to Java SE are typically not included. Within Java SE, APIs that deal with the user interface (User Interface Toolkits) or with the web interface for providing features such as sound, graphical rendering, user account access control, session management, authentication, and authorization are beyond the scope of this standard. However, this does not preclude the standard from discussing networked Java systems given the risks associated with improper input validation and injection flaws and suggesting appropriate mitigation strategies.
- **Coding Style.** Coding style issues are subjective; it has proven impossible to develop a consensus on appropriate style rules. Consequently, *The CERT® Oracle® Secure Coding Standard for Java™* recommends only that the user define style rules and apply those rules consistently; requirements that mandate use of any particular coding style are deliberately omitted. The easiest way to consistently apply a coding style is with the use of a code formatting tool. Many integrated development environments (IDEs) provide such capabilities.
- **Tools.** As a federally funded research and development center (FFRDC), the Software Engineering Institute (SEI) is not in a position to recommend particular vendors or tools to enforce the restrictions adopted. Users of this document are free to choose tools; vendors are encouraged to provide tools to enforce these rules.
- **Controversial Rules.** In general, the CERT secure coding standards try to avoid the inclusion of controversial rules that lack a broad consensus.

Audience

The CERT® Oracle® Secure Coding Standard for Java™ is primarily intended for developers of Java language programs. While this standard focuses on the Java Platform SE 6, it should also be informative (although incomplete) for Java developers working with Java ME or Java EE and other Java language versions.

While primarily designed for secure systems, this standard is also useful for achieving other quality attributes such as safety, reliability, dependability, robustness, resiliency, availability, and maintainability.

This standard may also be used by

- Developers of analyzer tools who wish to diagnose insecure or nonconforming Java language programs
- Software development managers, software acquirers, or other software development and acquisition specialists to establish a proscriptive set of secure coding standards
- Educators as a primary or secondary text for software security courses that teach secure coding in Java

The rules in this standard may be extended with organization-specific rules. However, a program must comply with existing rules to be considered conforming to the standard.

Training may be developed to educate software professionals regarding the appropriate application of secure coding standards. After passing an examination, these trained programmers may also be certified as secure coding professionals.

Contents and Organization

The standard is organized into an introductory chapter and 17 chapters containing rules in specific topic areas. Each of the rule chapters contains a list of rules in that section, and a risk assessment summary for the rules. There is also a common glossary and bibliography. This preface is meant to be read first, followed by the introductory chapter. The rule chapters may be read in any order or used as reference material as appropriate. The rules are loosely organized in each chapter but, in general, may also be read in any order.

Rules have a consistent structure. Each rule has a unique identifier, which is included in the title. The title of the rules and the introductory paragraphs define the conformance requirements. This is typically followed by one or more sets of noncompliant code examples and corresponding compliant solutions. Each rule also includes a risk assessment and bibliographical references specific to that rule. When applicable, rules also list related vulnerabilities and related guidelines from the following sources:

- *The CERT® C Secure Coding Standard* [Seacord 2008]
- *The CERT® C++ Secure Coding Standard* [CERT 2011]
- ISO/IEC TR 24772. Information Technology—Programming Languages—Guidance to Avoiding Vulnerabilities in Programming Languages through Language Selection and Use [ISO/IEC TR 24772:2010]
- MITRE CWE [MITRE 2011]

- Secure Coding Rules for the Java Programming Language, version 3.0 [SCG 2009]
- *The Elements of Java™ Style* [Rogue 2000]

Identifiers

Each rule has a unique identifier, consisting of three parts:

- A three-letter mnemonic, representing the section of the standard, is used to group similar rules and make them easier to find.
- A two-digit numeric value in the range of 00 to 99, which ensures each rule has a unique identifier.
- The letter J, which indicates that this is a Java language rule and is included to prevent ambiguity with similar rules in CERT secure coding standards for other languages.

Identifiers may be used by static analysis tools to reference a particular rule in a diagnostic message or otherwise used as shorthand for the rule title.

System Qualities

Security is one of many system attributes that must be considered in the selection and application of a coding standard. Other attributes of interest include safety, portability, reliability, availability, maintainability, readability, and performance.

Many of these attributes are interrelated in interesting ways. For example, readability is an attribute of maintainability; both are important for limiting the introduction of defects during maintenance that can result in security flaws or reliability issues. In addition, readability facilitates code inspection by safety officers. Reliability and availability require proper resource management, which also contributes to the safety and security of the system. System attributes such as performance and security are often in conflict, requiring tradeoffs to be made.

The purpose of the secure coding standard is to promote software security. However, because of the relationship between security and other system attributes, the coding standards may include requirements and recommendations that deal primarily with other system attributes that also have a significant impact on security.

Priority and Levels

Each rule has an assigned priority. Priorities are assigned using a metric based on Failure Mode, Effects, and Criticality Analysis (FMECA) [IEC 60812]. Three values are assigned for each rule on a scale of 1 to 3 for

- Severity—How serious are the consequences of the rule being ignored:

  1 = low (denial-of-service attack, abnormal termination)

  2 = medium (data integrity violation, unintentional information disclosure)

  3 = high (run arbitrary code, privilege escalation)
- Likelihood—How likely is it that a flaw introduced by violating the rule could lead to an exploitable vulnerability:

  1 = unlikely

  2 = probable

  3 = likely
- Remediation cost—How expensive is it to remediate existing code to comply with the rule:

  1 = high (manual detection and correction)

  2 = medium (automatic detection and manual correction)

  3 = low (automatic detection and correction)

The three values are multiplied together for each rule. This product provides a measure that can be used in prioritizing the application of the rules. These products range from 1 to 27. Rules with a priority in the range of 1 to 4 are level 3 rules, 6 to 9 are level 2, and 12 to 27 are level 1. As a result, it is possible to claim level 1, level 2, or complete compliance (level 3) with a standard by implementing all rules in a level, as shown in Figure P–1.

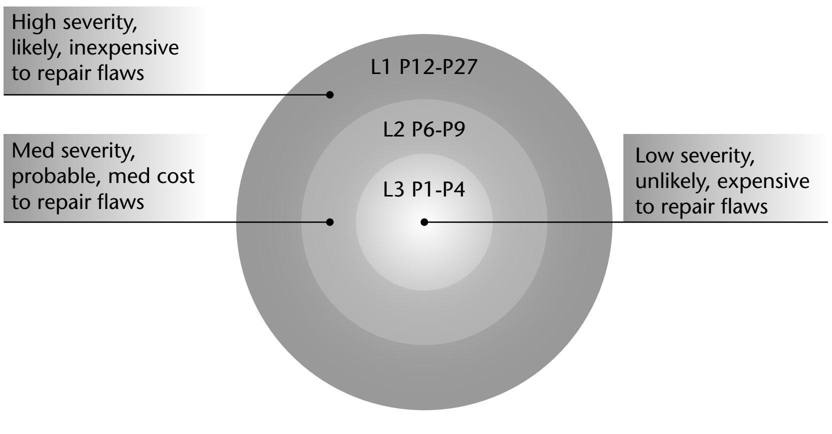

High severity, likely, inexpensive to repair flaws
L1 P12-P27
L2 P6-P9
Med severity, probable, med cost to repair flaws
L3 P1-P4
Low severity, unlikely, expensive to repair flaws

Figure P–1. Levels and priority ranges

The metric is designed primarily for remediation projects and does not apply to new development efforts that are implemented to the standard.

Conformance Testing

Software systems can be validated as conforming to *The CERT® Oracle® Secure Coding Standard for Java™*.

Normative vs. Nonnormative Text

Portions of this coding standard are intended to be normative; other portions are intended as good advice. The normative statements in these rules are the requirements for conformance with the standard. Normative statements use imperative language such as “must,” “shall,” and “require.” Normative portions of each rule must be analyzable, although automated analysis is infeasible for some rules and not required.

The nonnormative portions of a rule describe good practices or useful advice. Nonnormative statements do not establish conformance requirements. Nonnormative statements use verbs such as “should” or phrases such as “is recommended” or “is good practice.” Nonnormative portions of rules may be inappropriate for automated checking because such checking would likely report excessive false positives when applied to existing code. Automated checkers for these nonnormative portions might be useful when analyzing new code (that is, code that has been developed to this coding standard).

All of the rules in this standard have a normative component. Nonnormative recommendations are provided only when

- there is well-known good practice to follow
- the rule describes an approach that, if universally followed, would avoid violations where the normative part of the rule applies and would also be harmless when applied to code where the normative part of the rule is inapplicable

Entirely nonnormative guidelines are excluded from this coding standard. However, the authors of this book are planning a follow-on effort to publish these guidelines.

Automated Analysis

To ensure that the source code conforms to this secure coding standard, it is necessary to check for rule violations. The most effective means of checking is to use one or more analysis tools (analyzers). When a rule cannot be checked by a tool, manual review is required.

Many of the rules in this standard provide some indication as to whether or not existing analyzers can diagnose violations of the rule or even how amenable the rule is to automated analysis. This information is necessarily transitory because existing analyzers evolve and new analyzers are developed.

When choosing a source code analysis tool, it is clearly desirable that the tool be able to enforce as many of the rules in this document as possible. Not all rules are enforceable by automated analysis tools; some will require manual inspection.

Completeness and Soundness

To the greatest extent possible, an analyzer should be both complete and sound with respect to enforceable rules. An analyzer is considered sound (with respect to a specific rule) if it does not give a false-negative result, meaning it is able to find all violations of a rule within the entire program. An analyzer is considered complete if it does not issue false-positive results, or false alarms. The possibilities for a given rule are outlined in Table P–1.

Table P–1. Soundness and completeness

		False Positives	
		Y	N
False Negatives	N	Sound with false positives	Complete and sound
	Y	Unsound with false positives	Unsound

Tools with a high false-positive rate cause developers to waste their time, and they can lose interest in the results and consequently fail to realize value from the true bugs that are lost in the noise. Tools with a high number of false-negatives miss many defects that should be found and can foster a false sense of security. In practice, tools need to strike a balance between the two.

There are many tradeoffs in minimizing false-positives and false-negatives. It is obviously better to minimize both, and there are many techniques and algorithms that do both to some degree.

Analyzers are trusted processes, meaning that reliance is placed on the output of the tools. Consequently, developers must ensure that this trust is warranted. Ideally, this should be achieved by the tool supplier running appropriate validation tests. While it is possible to use a validation suite to test an analyzer, no formal validation scheme exists at this time.

■ CERT Source Code Analysis Laboratory

CERT has created the Source Code Analysis Laboratory (SCALe), which offers conformance testing of software systems to CERT secure coding standards, including The CERT Oracle Secure Coding Standard for Java.

SCALe evaluates client source code using multiple analyzers, including static analysis tools, dynamic analysis tools, and fuzz testing. CERT reports any violations of the secure coding rules to the developer. The developer may repair and resubmit the software for reevaluation.

After the developer has addressed these findings and the SCALe team determines that the product version tested conforms to the standard, CERT issues the developer a certificate and lists the system in a registry of conforming systems.

Successful conformance testing of a software system indicates that the SCALe analysis was unable to detect violations of rules defined by a CERT secure coding standard. Successful conformance testing does not provide any guarantees that these rules are not violated or that the software is entirely and permanently secure. SCALe does not test for unknown code-related vulnerabilities, high-level design and architectural flaws, the code's operational environment, or the code's portability. Conforming software systems can still be insecure, for example, if the software implements an insecure design or architecture.

Some rules in this standard include enumerated exceptions with discussion of the conditions under which each exception applies. When developers invoke an enumerated exception as a reason for deviating from a rule, they must document the relevant exception in the code at or near the point of deviation. A minimally acceptable form of documentation is a stylized comment containing the identifier of the exception being claimed, as in this example:

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

The authors are currently developing a set of Java annotations that will permit programmers to indicate such exceptions in a form that is both human-readable and accessible to static analysis tools. For conformance testing purposes, determination of whether an exception applies in any particular case is made by the SCALe analyst.

Third-Party Libraries

Static analysis tools, such as FindBugs that analyze Java bytecode, can frequently discover violations of this secure coding standard in third-party libraries in addition to custom code. Violations of secure coding rules in third-party libraries are treated in the same manner as if they appeared in custom code.

Unfortunately, developers are not always in a position to modify third-party library code or perhaps even to convince the vendor to modify the code. This means that the system cannot pass conformance testing unless the problem is eliminated (possibly by replacing the library with another library or custom-developed code) or by documenting a deviation. The deviation procedure for third-party library code is the same as for custom code—that is, the developer must show that the violation does not cause a vulnerability. However, the costs may be different. For custom code, it may be more economical to repair the problem, whereas for third-party libraries, it might be easier to document a deviation.

Conformance Testing Process

For each secure coding standard, the source code is found to be provably nonconforming, conforming, or provably conforming against each rule in the standard.

- *Provably nonconforming.* The code is provably nonconforming if one or more violations of a rule are discovered for which no deviation has been allowed.
- *Conforming.* The code is conforming if no violations of a rule are identified.
- *Provably conforming.* The code is provably conforming if the code has been verified to adhere to the rule in all possible cases.

Deviation Procedure

Strict adherence to all rules is unlikely; consequently, deviations associated with specific rule violations are necessary. Deviations can be used in cases where a true positive finding is uncontested as a rule violation but the code is nonetheless determined to be secure. This may be the result of a design or architecture feature of the software or because the particular violation occurs for a valid reason that was unanticipated by the secure coding standard. In this respect, the deviation procedure allows for the possibility that secure coding rules are overly strict. Deviations cannot be used for reasons of performance, usability, or to achieve other nonsecurity attributes in the system. A software system that successfully passes conformance testing must not present known vulnerabilities resulting from coding errors.

Deviation requests are evaluated by the lead assessor; if the developer can provide sufficient evidence that deviation does not introduce a vulnerability, the deviation request is accepted. Deviations should be used infrequently because it is almost always easier to fix a coding error than it is to prove that the coding error does not result in a vulnerability.

Once the evaluation process has been completed, a report detailing the conformance or nonconformance of the code to the corresponding rules in the secure coding standard is provided to the developer.

CERT SCALe Seal

Developers of software that has been determined by CERT to conform to a secure coding standard may use the seal shown in Figure P–2 to describe the conforming software on the developer's website. The seal must be specifically tied to the software passing conformance testing and not applied to untested products, the company, or the organization.

Figure P-2. CERT SCALe Seal

Except for patches that meet the following criteria, any modification of software after it is designated as conforming voids the conformance designation. Until such software is retested and determined to be conforming, the new software cannot be associated with the CERT SCALe Seal.

Patches that meet all three of the following criteria do not void the conformance designation:

- The patch is necessary to fix a vulnerability in the code or is necessary for the maintenance of the software.
- The patch does not introduce new features or functionality.
- The patch does not introduce a violation of any of the rules in the secure coding standard to which the software has been determined to conform.

Use of the CERT SCALe Seal is contingent upon the organization entering into a service agreement with Carnegie Mellon University and upon the software being designated by CERT as conforming. For more information, email securecoding@cert.org.

Acknowledgments

Thanks to everyone who has contributed to making this effort a success.

Contributors

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, and Weam Abu Zaki.

Reviewers

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, and Kenneth A. Williams.

Editors

Pamela Curtis, Shannon Haas, Carol Lallier, Tracey Tamules, Melanie Thompson, Paul Ruggerio, and Pennie Walters.

Addison-Wesley

Kim Boedigheimer, John Fuller, Stephane Nakib, Peter Gordon, Chuti Prasertsith, and Elizabeth Ryan.

Special Thanks

Archie Andrews, David Biber, Kim Boedigheimer, Peter Gordon, Frances Ho, Joe Jarzombek, Jason McNatt, Stephane Nakib, Rich Pethia, and Elizabeth Ryan.

About the Authors

Fred Long is a senior lecturer and director of learning and teaching in the Department of Computer Science, Aberystwyth University in the United Kingdom.

He lectures on formal methods; Java, C++, and C programming paradigms and programming-related security issues. He is chairman of the British Computer Society's Mid-Wales Sub-Branch.

Fred has been a Visiting Scientist at the Software Engineering Institute since 1992. Recently, his research has involved the investigation of vulnerabilities in Java.

Dhruv Mohindra is a senior software engineer at Persistent Systems Limited, India, where he develops monitoring software for widely used enterprise servers. He has worked for CERT at the Software Engineering Institute and continues to collaborate to improve the state of security awareness in the programming community.

Dhruv has also worked for Carnegie Mellon University, where he obtained his master of science degree in information security policy and management. He holds an undergraduate degree in computer engineering from Pune University, India, where he researched with Calsoft, Inc., during his academic pursuit.

A writing enthusiast, Dhruv occasionally contributes articles to technology magazines and online resources. He brings forth his experience and learning from developing and securing service oriented applications, server monitoring software, mobile device applications, web-based data miners, and designing user-friendly security interfaces.

Robert C. Seacord is a computer security specialist and writer. He is the author of books on computer security, legacy system modernization, and component-based software engineering.

Robert manages the Secure Coding Initiative at CERT, located in Carnegie Mellon's Software Engineering Institute in Pittsburgh, Pennsylvania. CERT, among other security-related activities, regularly analyzes software vulnerability reports and assesses the risk to the Internet and other critical infrastructure. Robert is an adjunct professor in the Carnegie Mellon University School of Computer Science and in the Information Networking Institute.

Robert started programming professionally for IBM in 1982, working in communications and operating system software, processor development, and software engineering. Robert also has worked at the X Consortium, where he developed and maintained code for the Common Desktop Environment and the X Window System.

Robert has a bachelor's degree in computer science from Rensselaer Polytechnic Institute.

Dean F. Sutherland is a senior software security engineer at CERT. Dean received his Ph.D. in software engineering from Carnegie Mellon in 2008. Before his return to academia, he spent 14 years working as a professional software engineer at Tartan, Inc. He spent the last six of those years as a senior member of the technical staff and a technical lead for compiler back-end technology. He was the primary active member of the corporate R&D group, was a key instigator of the design and deployment of a new software development process for Tartan, led R&D projects, and provided both technical and project leadership for the 12-person compiler back-end group.

David Svoboda is a software security engineer at CERT. David has been the primary developer on a diverse set of software development projects at Carnegie Mellon since 1991, ranging from hierarchical chip modeling and social organization simulation to automated machine translation (AMT). His KANTOO AMT software, developed in 1996, is still in production use at Caterpillar. He has over 13 years of Java development experience, starting with Java 2, and his Java projects include Tomcat servlets and Eclipse plug-ins. David is also actively involved in several ISO standards groups: the JTC1/SC22/WG14 group for the C programming language and the JTC1/SC22/WG21 group for C++.

Chapter 2

Input Validation and Data Sanitization (IDS)

■ Rules

■ Risk Assessment Summary

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
IDS00-J	high	probable	medium	P12	L1
IDS01-J	high	probable	medium	P12	L1
IDS02-J	medium	unlikely	medium	P4	L3
IDS03-J	medium	probable	medium	P8	L2
IDS04-J	low	probable	high	P2	L3
IDS05-J	medium	unlikely	medium	P4	L3
IDS06-J	medium	unlikely	medium	P4	L3
IDS07-J	high	probable	medium	P12	L1
IDS08-J	medium	unlikely	medium	P4	L3
IDS09-J	medium	probable	medium	P8	L2
IDS10-J	low	unlikely	medium	P2	L3
IDS11-J	high	probable	medium	P12	L1
IDS12-J	low	probable	medium	P4	L3
IDS13-J	low	unlikely	medium	P2	L3

■ IDS00-J. Sanitize untrusted data passed across a trust boundary

Many programs accept untrusted data originating from unvalidated users, network connections, and other untrusted sources and then pass the (modified or unmodified) data across a trust boundary to a different trusted domain. Frequently the data is in the form of a string with some internal syntactic structure, which the subsystem must parse. Such data must be sanitized both because the subsystem may be unprepared to handle the malformed input and because unsanitized input may include an injection attack.

In particular, programs must sanitize all string data that is passed to command interpreters or parsers so that the resulting string is innocuous in the context in which it is parsed or interpreted.

Many command interpreters and parsers provide their own sanitization and validation methods. When available, their use is preferred over custom sanitization techniques because custom developed sanitization can often neglect special cases or hidden complexities in the parser. Another problem with custom sanitization code is that it may not be adequately maintained when new capabilities are added to the command interpreter or parser software.

SQL Injection

A SQL injection vulnerability arises when the original SQL query can be altered to form an altogether different query. Execution of this altered query may result in information leaks or data modification. The primary means of preventing SQL injection are sanitizing and validating untrusted input and parameterizing queries.

Suppose a database contains user names and passwords used to authenticate users of the system. The user names have a string size limit of 8. The passwords have a size limit of 20.

A SQL command to authenticate a user might take the form:

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

If it returns any records, the user name and password are valid.

However, if an attacker can substitute arbitrary strings for `<USERNAME>` and `<PASSWORD>`, they can perform a SQL injection by using the following string for `<USERNAME>`:

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

When injected into the command, the command becomes:

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

If `validuser` is a valid user name, this `SELECT` statement selects the `validuser` record in the table. The password is never checked because `username='validuser'` is true; consequently the items after the `OR` are not tested. As long as the components after the `OR` generate a syntactically correct SQL expression, the attacker is granted the access of `validuser`.

Likewise, an attacker could supply a string for `<PASSWORD>` such as:

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

This would yield the following command:

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

This time, the `'1'='1'` tautology disables both user name and password validation, and the attacker is falsely logged in without a correct login ID or password.

Noncompliant Code Example

This noncompliant code example shows JDBC code to authenticate a user to a system. The password is passed as a `char` array, the database connection is created, and then the passwords are hashed.

Unfortunately, this code example permits a SQL injection attack because the SQL statement `sqlString` accepts unsanitized input arguments. The attack scenario outlined previously would work as described.

```
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
      }
    }
  }
}
```

Compliant Solution (PreparedStatement)

Fortunately, the JDBC library provides an API for building SQL commands that sanitize untrusted data. The `java.sql.PreparedStatement` class properly escapes input strings, preventing SQL injection when used properly. This is an example of component-based sanitization.

This compliant solution modifies the `doPrivilegedAction()` method to use a `PreparedStatement` instead of `java.sql.Statement`. This code also validates the length of the `username` argument, preventing an attacker from submitting an arbitrarily long user name.

```
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
    }
  }
}
```

Use the `set*()` methods of the `PreparedStatement` class to enforce strong type checking. This mitigates the SQL injection vulnerability because the input is properly escaped by automatic entrapment within double quotes. Note that prepared statements must be used even with queries that insert data into the database.

XML Injection

Because of its platform independence, flexibility, and relative simplicity, the extensible markup language (XML) has found use in applications ranging from remote procedure calls to systematic storage, exchange, and retrieval of data. However, because of its versatility, XML is vulnerable to a wide spectrum of attacks. One such attack is called *XML injection*.

A user who has the ability to provide structured XML as input can override the contents of an XML document by injecting XML tags in data fields. These tags are interpreted and classified by an XML parser as executable content and, as a result, may cause certain data members to be overridden.

Consider the following XML code snippet from an online store application, designed primarily to query a back-end database. The user has the ability to specify the quantity of an item available for purchase.

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

A malicious user might input the following string instead of a simple number in the `quantity` field.

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

Consequently, the XML resolves to the following block:

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

A Simple API for XML (SAX) parser (`org.xml.sax` and `javax.xml.parsers.SAXParser`) interprets the XML such that the second price field overrides the first, leaving the price of the item as $1. Even when it is not possible to perform such an attack, the attacker may be able to inject special characters, such as comment blocks and `CDATA` delimiters, which corrupt the meaning of the XML.

Noncompliant Code Example

In this noncompliant code example, a client method uses simple string concatenation to build an XML query to send to a server. XML injection is possible because the method performs no input validation.

```
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();
}
```

Compliant Solution (Whitelisting)

Depending on the specific data and command interpreter or parser to which data is being sent, appropriate methods must be used to sanitize untrusted user input. This compliant solution uses whitelisting to sanitize the input. In this compliant solution, the method requires that the quantity field must be a number between 0 and 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();
}
```

Compliant Solution (XML Schema)

A more general mechanism for checking XML for attempted injection is to validate it using a Document Type Definition (DTD) or schema. The schema must be rigidly defined to prevent injections from being mistaken for valid XML. Here is a suitable schema for validating our XML snippet:

```
<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>
```

The schema is available as the file `schema.xsd`. This compliant solution employs this schema to prevent XML injection from succeeding. It also relies on the `CustomResolver` class to prevent XXE attacks. This class, as well as XXE attacks, are described in the subsequent code examples.

```
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();
  }
```

Using a schema or DTD to validate XML is convenient when receiving XML that may have been loaded with unsanitized input. If such an XML string has not yet been built, sanitizing input before constructing XML yields better performance.

XML External Entity Attacks (XXE)

An XML document can be dynamically constructed from smaller logical blocks called entities. Entities can be internal, external, or parameter-based. External entities allow the inclusion of XML data from external files.

According to XML W3C Recommendation [W3C 2008], Section 4.4.3, "Included If Validating":

> When an XML processor recognizes a reference to a parsed entity, to validate the document, the processor MUST include its replacement text. If the entity is external, and the processor is not attempting to validate the XML document, the processor MAY, but need not, include the entity's replacement text.

An attacker may attempt to cause denial of service or program crashes by manipulating the URI of the entity to refer to special files existing on the local file system, for example, by specifying `/dev/random` or `/dev/tty` as input URIs. This may crash or block the program indefinitely. This is called an XML external entity (XXE) attack. Because inclusion of replacement text from an external entity is optional, not all XML processors are vulnerable to external entity attacks.

Noncompliant Code Example

This noncompliant code example attempts to parse the file `evil.xml`, reports any errors, and exits. However, a SAX or a DOM (Document Object Model) parser will attempt to access the URL specified by the `SYSTEM` attribute, which means it will attempt to read the contents of the local `/dev/tty` file. On POSIX systems, reading this file causes the program to block until input data is supplied to the machine's console. Consequently, an attacker can use this malicious XML file to cause the program to hang.

```
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());
  }
}
```

This program is subject to a remote XXE attack if the `evil.xml` file contains the following:

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

This noncompliant code example may also violate rule ERR06-J if the information contained in the exceptions is sensitive.

Compliant Solution (`EntityResolver`)

This compliant solution defines a `CustomResolver` class that implements the interface `org.xml.sax.EntityResolver`. This enables a SAX application to customize handling of external entities. The `setEntityResolver()` method registers the instance with the corresponding SAX driver. The customized handler uses a simple whitelist for external entities. The `resolveEntity()` method returns an empty `InputSource` when an input fails to resolve to any of the specified, safe entity source paths. Consequently, when parsing malicious input, the empty `InputSource` returned by the custom resolver causes a `java.net.MalformedURLException` to be thrown. Note that you must create an `XMLReader` object on which to set the custom entity resolver.

This is an example of component-based sanitization.

```
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());
  }
}
```

Risk Assessment

Failure to sanitize user input before processing or storing it can result in injection attacks.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
IDS00-J	high	probable	medium	P12	L1

Related Vulnerabilities CVE-2008-2370 describes a vulnerability in Apache Tomcat 4.1.0 through 4.1.37, 5.5.0 through 5.5.26, and 6.0.0 through 6.0.16. When a `RequestDispatcher` is used, Tomcat performs path normalization before removing the query string from the URI, which allows remote attackers to conduct directory traversal attacks and read arbitrary files via a .. (dot dot) in a request parameter.

Related Guidelines

CERT C Secure Coding Standard	STR02-C. Sanitize data passed to complex subsystems
CERT C++ Secure Coding Standard	STR02-CPP. Sanitize data passed to complex subsystems
ISO/IEC TR 24772:2010	Injection [RST]
MITRE CWE	CWE-116. Improper encoding or escaping of output

Bibliography

[OWASP 2005]	
[OWASP 2007]	
[OWASP 2008]	Testing for XML Injection (OWASP-DV-008)
[W3C 2008]	4.4.3, Included If Validating

■ IDS01-J. Normalize strings before validating them

Many applications that accept untrusted input strings employ input filtering and validation mechanisms based on the strings' character data.

For example, an application's strategy for avoiding cross-site scripting (XSS) vulnerabilities may include forbidding `<script>` tags in inputs. Such blacklisting mechanisms are a useful part of a security strategy, even though they are insufficient for complete input validation and sanitization. When implemented, this form of validation must be performed only after normalizing the input.

Character information in Java SE 6 is based on the Unicode Standard, version 4.0 [Unicode 2003]. Character information in Java SE 7 is based on the Unicode Standard, version 6.0.0 [Unicode 2011].

According to the Unicode Standard [Davis 2008a], annex #15, Unicode Normalization Forms:

> When implementations keep strings in a normalized form, they can be assured that equivalent strings have a unique binary representation.
>
> Normalization Forms KC and KD must not be blindly applied to arbitrary text. Because they erase many formatting distinctions, they will prevent round-trip conversion to and from many legacy character sets, and unless supplanted by

formatting markup, they may remove distinctions that are important to the semantics of the text. It is best to think of these Normalization Forms as being like uppercase or lowercase mappings: useful in certain contexts for identifying core meanings, but also performing modifications to the text that may not always be appropriate. They can be applied more freely to domains with restricted character sets.

Frequently, the most suitable normalization form for performing input validation on arbitrarily encoded strings is KC (NFKC) because normalizing to KC transforms the input into an equivalent canonical form that can be safely compared with the required input form.

Noncompliant Code Example

This noncompliant code example attempts to validate the `String` before performing normalization. Consequently, the validation logic fails to detect inputs that should be rejected because the check for angle brackets fails to detect alternative Unicode representations.

```
// 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);
```

The `normalize()` method transforms Unicode text into an equivalent composed or decomposed form, allowing for easier searching of text. The normalize method supports the standard normalization forms described in Unicode Standard Annex #15—Unicode Normalization Forms.

Compliant Solution

This compliant solution normalizes the string before validating it. Alternative representations of the string are normalized to the canonical angle brackets. Consequently, input validation correctly detects the malicious input and throws an `IllegalStateException`.

```
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 {
  // ...
}
```

Risk Assessment

Validating input before normalization affords attackers the opportunity to bypass filters and other security mechanisms. This can result in the execution of arbitrary code.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
IDS01-J	high	probable	medium	P12	L1

Related Guidelines

ISO/IEC TR 24772:2010	Cross-site scripting [XYT]
MITRE CWE	CWE-289. Authentication bypass by alternate name
	CWE-180. Incorrect behavior order: Validate before canonicalize

Bibliography

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

■ IDS02-J. Canonicalize path names before validating them

According to the Java API [API 2006] for class `java.io.File`:

> A path name, whether abstract or in string form, may be either absolute or relative. An absolute path name is complete in that no other information is required to locate the file that it denotes. A relative path name, in contrast, must be interpreted in terms of information taken from some other path name.

Absolute or relative path names may contain file links such as symbolic (soft) links, hard links, shortcuts, shadows, aliases, and junctions. These file links must be fully resolved before any file validation operations are performed. For example, the final target of a symbolic link called `trace` might be the path name `/home/system/trace`. Path names may also contain special file names that make validation difficult:

1. "." refers to the directory itself.
2. Inside a directory, the special file name ".." refers to the directory's parent directory.

In addition to these specific issues, there are a wide variety of operating system–specific and file system–specific naming conventions that make validation difficult.

The process of canonicalizing file names makes it easier to validate a path name. More than one path name can refer to a single directory or file. Further, the textual representation of a path name may yield little or no information regarding the directory or file to which it refers. Consequently, all path names must be fully resolved or *canonicalized* before validation.

Validation may be necessary, for example, when attempting to restrict user access to files within a particular directory or otherwise make security decisions based on the name of a file name or path name. Frequently, these restrictions can be circumvented by an attacker by exploiting a *directory traversal* or *path equivalence* vulnerability. A directory traversal vulnerability allows an I/O operation to escape a specified operating directory. A path equivalence vulnerability occurs when an attacker provides a different but equivalent name for a resource to bypass security checks.

Canonicalization contains an inherent race window between the time the program obtains the canonical path name and the time it opens the file. While the canonical path name is being validated, the file system may have been modified and the canonical path name may no longer reference the original valid file. Fortunately, this race condition can be easily mitigated. The canonical path name can be used to determine whether the referenced file name is in a secure directory (see rule FIO00-J for more information). If the referenced file is in a secure directory, then, by definition, an attacker cannot tamper with it and cannot exploit the race condition.

This rule is a specific instance of rule IDS01-J.

Noncompliant Code Example

This noncompliant code example accepts a file path as a command-line argument and uses the `File.getAbsolutePath()` method to obtain the absolute file path. It also uses the `isInSecureDir()` method defined in rule FIO00-J to ensure that the file is in a secure directory. However, it neither resolves file links nor eliminates equivalence errors.

```
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();
    }
  }
```

The application intends to restrict the user from operating on files outside of their home directory. The `validate()` method attempts to ensure that the path name resides within this directory, but can be easily circumvented. For example, a user can create a link in their home directory that refers to a directory or file outside of their home directory. The path name of the link might appear to the `validate()` method to reside in their home directory and consequently pass validation, but the operation will actually be performed on the final target of the link, which resides outside the intended directory.

Note that `File.getAbsolutePath()` does resolve symbolic links, aliases, and short cuts on Windows and Macintosh platforms. Nevertheless, the *Java Language Specification* (JLS) lacks any guarantee that this behavior is present on *all* platforms or that it will continue in future implementations.

Compliant Solution (`getCanonicalPath()`)

This compliant solution uses the `getCanonicalPath()` method, introduced in Java 2, because it resolves all aliases, shortcuts, and symbolic links consistently across all platforms. Special file names such as dot dot (`..`) are also removed so that the input is reduced to a canonicalized form before validation is carried out. An attacker cannot use `../` sequences to break out of the specified directory when the `validate()` method is present.

```
  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();
    }
  }
```

The getCanonicalPath() method throws a security exception when used within applets because it reveals too much information about the host machine. The getCanonicalFile() method behaves like getCanonicalPath() but returns a new File object instead of a String.

Compliant Solution (Security Manager)

A comprehensive way of handling this issue is to grant the application the permissions to operate only on files present within the intended directory—the user's home directory in this example. This compliant solution specifies the absolute path of the program in its security policy file and grants java.io.FilePermission with target ${user.home}/* and actions read and write.

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

This solution requires that the user's home directory is a secure directory as described in rule FIO00-J.

Noncompliant Code Example

This noncompliant code example allows the user to specify the absolute path of a file name on which to operate. The user can specify files outside the intended directory (/img in this example) by entering an argument that contains ../ sequences and consequently violate the intended security policies of the program.

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

Noncompliant Code Example

This noncompliant code example attempts to mitigate the issue by using the File.getCanonicalPath() method, which fully resolves the argument and constructs a canonicalized path. For example, the path /img/../etc/passwd resolves to /etc/passwd. Canonicalization without validation is insufficient because an attacker can specify files outside the intended directory.

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

Compliant Solution

This compliant solution obtains the file name from the untrusted user input, canonicalizes it, and then validates it against a list of benign path names. It operates on the specified file only when validation succeeds; that is, only if the file is one of the two valid files `file1.txt` or `file2.txt` in `/img/java`.

```
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);
```

The `/img/java` directory must be secure to eliminate any race condition.

Compliant Solution (Security Manager)

This compliant solution grants the application the permissions to read only the intended files or directories. For example, read permission is granted by specifying the absolute path of the program in the security policy file and granting `java.io.FilePermission` with the canonicalized absolute path of the file or directory as the target name and with the action set to `read`.

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

Risk Assessment

Using path names from untrusted sources without first canonicalizing them and then validating them can result in directory traversal and path equivalence vulnerabilities.

Rule	**Severity**	**Likelihood**	**Remediation Cost**	**Priority**	**Level**
IDS02-J	medium	unlikely	medium	P4	L3

Related Vulnerabilities CVE-2005-0789 describes a directory traversal vulnerability in LimeWire 3.9.6 through 4.6.0 that allows remote attackers to read arbitrary files via a `..` (dot dot) in a magnet request.

CVE-2008-5518 describes multiple directory traversal vulnerabilities in the web administration console in Apache Geronimo Application Server 2.1 through 2.1.3 on Windows that allow remote attackers to upload files to arbitrary directories.

Related Guidelines

The CERT C Secure Coding Standard	FIO02-C. Canonicalize path names originating from untrusted sources
The CERT C++ Secure Coding Standard	FIO02-CPP. Canonicalize path names originating from untrusted sources
ISO/IEC TR 24772:2010	Path Traversal [EWR]
MITRE CWE	CWE-171. Cleansing, canonicalization, and comparison errors
	CWE-647. Use of non-canonical URL paths for authorization decisions

Bibliography

[API 2006] Method `getCanonicalPath()`

[Harold 1999]

IDS03-J. Do not log unsanitized user input

A log injection vulnerability arises when a log entry contains unsanitized user input. A malicious user can insert fake log data and consequently deceive system administrators as to the system's behavior [OWASP 2008]. For example, a user might split a legitimate log entry into two log entries by entering a carriage return and line feed (CRLF) sequence, either of which might be misleading. Log injection attacks can be prevented by sanitizing and validating any untrusted input sent to a log.

Logging unsanitized user input can also result in leaking sensitive data across a trust boundary, or storing sensitive data in a manner that violates local law or regulation. For example, if a user can inject an unencrypted credit card number into a log file, the system could violate PCI DSS regulations [PCI 2010]. See rule IDS00-J for more details on input sanitization.

Noncompliant Code Example

This noncompliant code example logs the user's login name when an invalid request is received. No input sanitization is performed.

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

Without sanitization, a log injection attack is possible. A standard log message when `username` is `david` might look like this:

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

If the `username` that is used in a log message was not `david`, but rather a multiline string like this:

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

the log would contain the following misleading data:

```
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
```

Compliant Solution

This compliant solution sanitizes the `username` input before logging it, preventing injection attacks. Refer to rule IDS00-J for more details on input sanitization.

```
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);
}
```

Risk Assessment

Allowing unvalidated user input to be logged can result in forging of log entries, leaking secure information, or storing sensitive data in a manner that violates a local law or regulation.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
IDS03-J	medium	probable	medium	P8	L2

Related Guidelines

ISO/IEC TR 24772:2010	Injection [RST]
MITRE CWE	CWE-144. Improper neutralization of line delimiters
	CWE-150. Improper neutralization of escape, meta, or control sequences

Bibliography

[API 2006]

[OWASP 2008]

[PCI DSS Standard]

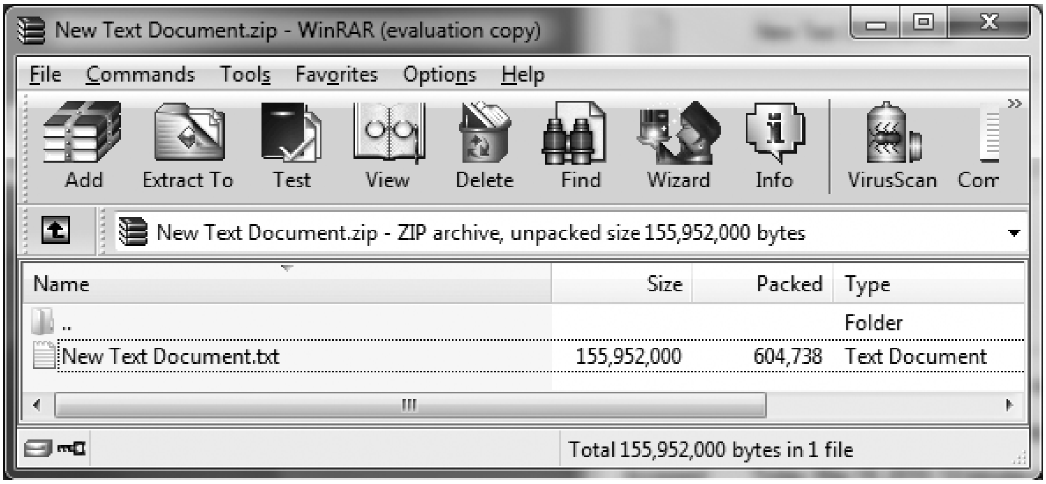
■ IDS04-J. Limit the size of files passed to `ZipInputStream`

Check inputs to `java.util.ZipInputStream` for cases that cause consumption of excessive system resources. Denial of service can occur when resource usage is disproportionately large in comparison to the input data that causes the resource usage. The nature of the zip algorithm permits the existence of *zip bombs* where a small file, such as ZIPs, GIFs, or gzip-encoded HTTP content consumes excessive resources when uncompressed because of extreme compression.

The zip algorithm is capable of producing very large compression ratios [Mahmoud 2002]. Figure 2–1 shows a file that was compressed from 148MB to 590KB, a ratio of more than 200 to 1. The file consists of arbitrarily repeated data: alternating lines of *a* characters and *b* characters. Even higher compression ratios can be easily obtained using input data that is targeted to the compression algorithm, or using more input data (that is untargeted), or other compression methods.

Any entry in a zip file whose uncompressed file size is beyond a certain limit must not be uncompressed. The actual limit is dependent on the capabilities of the platform.

This rule is a specific instance of the more general rule MSC07-J.


New Text Document.zip - WinRAR (evaluation copy)

File Commands Tools Favorites Options Help

Add Extract To Test View Delete Find Wizard Info VirusScan Com

New Text Document.zip - ZIP archive, unpacked size 155,952,000 bytes

Name	Size	Packed	Type
..			Folder
New Text Document.txt	155,952,000	604,738	Text Document

Total 155,952,000 bytes in 1 file

Figure 2–1. Very large compression ratios in a Zip file.

Noncompliant Code Example

This noncompliant code fails to check the resource consumption of the file that is being unzipped. It permits the operation to run to completion or until local resources are exhausted.

```
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();
```

Compliant Solution

In this compliant solution, the code inside the while loop uses the `ZipEntry.getSize()` method to find the uncompressed file size of each entry in a zip archive before extracting the entry. It throws an exception if the entry to be extracted is too large—100MB in this case.

```
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);
  }
```

Risk Assessment

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
IDS04-J	low	probable	high	P2	L3

Related Guidelines

MITRE CWE	CWE-409. Improper handling of highly compressed data (data amplification)
Secure Coding Guidelines for the Java Programming Language, Version 3.0	Guideline 2-5. Check that inputs do not cause excessive resource consumption

Bibliography

[Mahmoud 2002] Compressing and Decompressing Data Using Java APIs

■ IDS05-J. Use a subset of ASCII for file and path names

File and path names containing particular characters can be troublesome and can cause unexpected behavior resulting in vulnerabilities. The following characters and patterns can be problematic when used in the construction of a file or path name:

- Leading dashes: Leading dashes can cause problems when programs are called with the file name as a parameter because the first character or characters of the file name might be interpreted as an option switch.
- Control characters, such as newlines, carriage returns, and escape: Control characters in a file name can cause unexpected results from shell scripts and in logging.
- Spaces: Spaces can cause problems with scripts and when double quotes aren't used to surround the file name.
- Invalid character encodings: Character encodings can make it difficult to perform proper validation of file and path names. (See rule IDS11-J.)
- Name-space separation characters: Including name-space separation characters in a file or path name can cause unexpected and potentially insecure behavior.
- Command interpreters, scripts, and parsers: Some characters have special meaning when processed by a command interpreter, shell, or parser and should consequently be avoided.

As a result of the influence of MS-DOS, file names of the form `xxxxxxxx.xxx`, where `x` denotes an alphanumeric character, are generally supported by modern systems. On some platforms, file names are case sensitive; while on other platforms, they are case insensitive. VU#439395 is an example of a vulnerability in C resulting from a failure to deal appropriately with case sensitivity issues [VU#439395].

This rule is a specific instance of rule IDS00-J.

Noncompliant Code Example

In the following noncompliant code example, unsafe characters are used as part of a file name.

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

A platform is free to define its own mapping of unsafe characters. For example, when tested on an Ubuntu Linux distribution, this noncompliant code example resulted in the following file name:

```
A?
```

Compliant Solution

Use a descriptive file name containing only the subset of ASCII previously described.

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

Noncompliant Code Example

This noncompliant code example creates a file with input from the user without sanitizing the input.

```
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);
  // ...
}
```

No checks are performed on the file name to prevent troublesome characters. If an attacker knew this code was in a program used to create or rename files that would later be used in a script or automated process of some sort, the attacker could choose particular characters in the output file name to confuse the later process for malicious purposes.

Compliant Solution

In this compliant solution, the program uses a whitelist to reject unsafe file names.

```
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);
    // ...
  }
```

All file names originating from untrusted sources must be sanitized to ensure they contain only safe characters.

Risk Assessment

Failing to use only a safe subset of ASCII can result in misinterpreted data.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
IDS05-J	medium	unlikely	medium	P4	L3

Related Guidelines

CERT C Secure Coding Standard	MSC09-C. Character encoding—Use subset of ASCII for safety
CERT C++ Secure Coding Standard	MSC09-CPP. Character encoding—Use subset of ASCII for safety
ISO/IEC TR 24772:2010	Choice of filenames and other external identifiers [AJN]
MITRE CWE	CWE-116. Improper encoding or escaping of output

Bibliography

ISO/IEC 646-1991	ISO 7-bit coded character set for information interchange
[Kuhn 2006]	UTF-8 and Unicode FAQ for UNIX/Linux
[Wheeler 2003]	5.4, File Names
[VU#439395]	

■ IDS06-J. Exclude user input from format strings

Interpretation of Java format strings is stricter than in languages such as C [Seacord 2005]. The standard library implementations throw appropriate exceptions when any conversion argument fails to match the corresponding format specifier. This approach reduces opportunities for malicious exploits. Nevertheless, malicious user input can exploit format strings and can cause information leaks or denial of service. As a result, strings from an untrusted source should not be incorporated into format strings.

Noncompliant Code Example

This noncompliant code example demonstrates an information leak issue. It accepts a credit card expiration date as an input argument and uses it within the format string.

```
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);
  }
}
```

In the absence of proper input validation, an attacker can determine the date against which the input is being verified by supplying an input that includes one of the format string arguments `%1$tm`, `%1$te`, or `%1$tY`.

Compliant Solution

This compliant solution ensures that user-generated input is excluded from format strings.

```
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);
  }
}
```

Risk Assessment

Allowing user input to taint a format string may cause information leaks or denial of service.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
IDS06-J	medium	unlikely	medium	P4	L3

Automated Detection Static analysis tools that perform taint analysis can diagnose some violations of this rule.

Related Guidelines

CERT C Secure Coding Standard	FIO30-C. Exclude user input from format strings
CERT C++ Secure Coding Standard	FIO30-CPP. Exclude user input from format strings
ISO/IEC TR 24772:2010	Injection [RST]
MITRE CWE	CWE-134. Uncontrolled format string

Bibliography

[API 2006]	Class Formatter
[Seacord 2005]	Chapter 6, Formatted Output

■ IDS07-J. Do not pass untrusted, unsanitized data to the `Runtime.exec()` method

External programs are commonly invoked to perform a function required by the overall system. This is a form of reuse and might even be considered a crude form of component-based software engineering. Command and argument injection vulnerabilities occur when an application fails to sanitize untrusted input and uses it in the execution of external programs.

Every Java application has a single instance of class `Runtime` that allows the application to interface with the environment in which the application is running. The current runtime can be obtained from the `Runtime.getRuntime()` method. The semantics of `Runtime.exec()` are poorly defined, so it's best not to rely on its behavior any more than necessary, but typically it invokes the command directly without a shell. If you want a shell, you can use `/bin/sh -c` on POSIX or `cmd.exe` on Windows. The variants of `exec()` that take the command line as a single string split it using a `StringTokenizer`. On Windows, these tokens are concatenated back into a single argument string before being executed.

Consequently, command injection attacks cannot succeed unless a command interpreter is explicitly invoked. However, argument injection attacks can occur when arguments have spaces, double quotes, and so forth, or start with a `-` or `/` to indicate a switch.

This rule is a specific instance of rule IDS00-J. Any string data that originates from outside the program's trust boundary must be sanitized before being executed as a command on the current platform.

Noncompliant Code Example (Windows)

This noncompliant code example provides a directory listing using the `dir` command. This is implemented using `Runtime.exec()` to invoke the Windows `dir` command.

```
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);
    }
  }
}
```

Because `Runtime.exec()` receives unsanitized data originating from the environment, this code is susceptible to a command injection attack.

An attacker can exploit this program using the following command:

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

The command executed is actually two commands:

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

which first attempts to list a nonexistent `dummy` folder and then prints `bad` to the console.

Noncompliant Code Example (POSIX)

This noncompliant code example provides the same functionality but uses the POSIX `ls` command. The only difference from the Windows version is the argument passed to `Runtime.exec()`.

```
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);
    }
  }
}
```

The attacker can supply the same command shown in the previous noncompliant code example with similar effects. The command executed is actually:

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

Compliant Solution (Sanitization)

This compliant solution sanitizes the untrusted user input by permitting only a small group of whitelisted characters in the argument that will be passed to `Runtime.exec()`; all other characters are excluded.

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

Although this is a compliant solution, this sanitization approach rejects valid directories. Also, because the command interpreter invoked is system dependent, it is difficult to establish that this solution prevents command injections on every platform on which a Java program might run.

Compliant Solution (Restricted User Choice)

This compliant solution prevents command injection by passing only trusted strings to `Runtime.exec()`. While the user has control over which string is used, the user cannot provide string data directly to `Runtime.exec()`.

```
// ...
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
}
```

This compliant solution hard codes the directories that may be listed.

This solution can quickly become unmanageable if you have many available directories. A more scalable solution is to read all the permitted directories from a properties file into a `java.util.Properties` object.

Compliant Solution (Avoid `Runtime.exec()`)

When the task performed by executing a system command can be accomplished by some other means, it is almost always advisable to do so. This compliant solution uses the `File.list()` method to provide a directory listing, eliminating the possibility of command or argument injection attacks.

```
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);
      }
    }
  }
}
```

Risk Assessment

Passing untrusted, unsanitized data to the `Runtime.exec()` method can result in command and argument injection attacks.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
IDS07-J	high	probable	medium	P12	L1

Related Vulnerabilities

[CVE-2010-0886]	Sun Java Web Start plugin command line argument injection
[CVE-2010-1826]	Command injection in `updateSharingD`'s handling of Mach RPC messages
[T-472]	Mac OS X Java command injection flaw in `updateSharingD` lets local users gain elevated privileges

Related Guidelines

The CERT C Secure Coding Standard	ENV03-C. Sanitize the environment when invoking external programs
	ENV04-C. Do not call `system()` if you do not need a command processor
The CERT C++ Secure Coding Standard	ENV03-CPP. Sanitize the environment when invoking external programs
	ENV04-CPP. Do not call `system()` if you do not need a command processor
ISO/IEC TR 24772:2010	Injection [RST]
MITRE CWE	CWE-78. Improper neutralization of special elements used in an OS command ("OS command injection")

Bibliography

[Chess 2007]	Chapter 5, Handling Input, "Command Injection"
[OWASP 2005]	
[Permissions 2008]	

■ IDS08-J. Sanitize untrusted data passed to a regex

Regular expressions are widely used to match strings of text. For example, the POSIX `grep` utility supports regular expressions for finding patterns in the specified text.

For introductory information on regular expressions, see the Java Tutorials [Tutorials 08]. The `java.util.regex` package provides the `Pattern` class that encapsulates a compiled representation of a regular expression and the `Matcher` class, which is an engine that uses a `Pattern` to perform matching operations on a `CharSequence`.

Java's powerful regular expression (regex) facilities must be protected from misuse. An attacker may supply a malicious input that modifies the original regular expression in such a way that the regex fails to comply with the program's specification. This attack vector, called a *regex injection*, might affect control flow, cause information leaks, or result in denial-of-service (DoS) vulnerabilities.

Certain constructs and properties of Java regular expressions are susceptible to exploitation:

- Matching flags: Untrusted inputs may override matching options that may or may not have been passed to the `Pattern.compile()` method.
- Greediness: An untrusted input may attempt to inject a regex that changes the original regex to match as much of the string as possible, exposing sensitive information.
- Grouping: The programmer can enclose parts of a regular expression in parentheses to perform some common action on the group. An attacker may be able to change the groupings by supplying untrusted input.

Untrusted input should be sanitized before use to prevent regex injection. When the user must specify a regex as input, care must be taken to ensure that the original regex cannot be modified without restriction. Whitelisting characters (such as letters and digits) before delivering the user-supplied string to the regex parser is a good input sanitization strategy. A programmer must provide only a very limited subset of regular expression functionality to the user to minimize any chance of misuse.

Regex Injection Example

Suppose a system log file contains messages output by various system processes. Some processes produce public messages and some processes produce sensitive messages marked "private." Here is an example log file:

```
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
```

A user wishes to search the log file for interesting messages but must be prevented from seeing the private messages. A program might accomplish this by permitting the user to provide search text that becomes part of the following regex:

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

However, if an attacker can substitute any string for `<SEARCHTEXT>`, he can perform a regex injection with the following text:

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

When injected into the regex, the regex becomes:

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

This regex will match any line in the log file, including the private ones.

Noncompliant Code Example

This noncompliant code example periodically loads the log file into memory and allows clients to obtain keyword search suggestions by passing the keyword as an argument to `suggestSearches()`.

```
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;
    }
  }

}
```

This code permits a trusted user to search for public log messages such as "error." However, it also allows a malicious attacker to perform the regex injection previously described.

Compliant Solution (Whitelisting)

This compliant solution filters out nonalphanumeric characters (except space and single quote) from the search string, which prevents regex injection previously described.

```
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 + ".*)";
      // ...
    }
  }
}
```

This solution also limits the set of valid search terms. For instance, a user may no longer search for "name =" because the = character would be sanitized out of the regex.

Compliant Solution

Another method of mitigating this vulnerability is to filter out the sensitive information prior to matching. Such a solution would require the filtering to be done every time the log file is periodically refreshed, incurring extra complexity and a performance penalty. Sensitive information may still be exposed if the log format changes but the class is not also refactored to accommodate these changes.

Risk Assessment

Failing to sanitize untrusted data included as part of a regular expression can result in the disclosure of sensitive information.

Rule	**Severity**	**Likelihood**	**Remediation Cost**	**Priority**	**Level**
IDS08-J	medium	unlikely	medium	P4	L3

Related Guidelines

MITRE CWE	CWE-625. Permissive regular expression

Bibliography

[Tutorials 08]	Regular Expressions
[CVE 05]	CVE-2005-1949

■ IDS09-J. Do not use locale-dependent methods on locale-dependent data without specifying the appropriate locale

Using locale-dependent methods on locale-dependent data can produce unexpected results when the locale is unspecified. Programming language identifiers, protocol keys, and HTML tags are often specified in a particular locale, usually `Locale.ENGLISH`. It may even be possible to bypass input filters by changing the default locale, which can alter the behavior of locale-dependent methods. For example, when a string is converted to uppercase, it may be declared valid; however, changing the string back to lowercase during subsequent execution may result in a blacklisted string.

Any program which invokes locale-dependent methods on untrusted data must explicitly specify the locale to use with these methods.

Noncompliant Code Example

This noncompliant code example uses the locale-dependent `String.toUpperCase()` method to convert an HTML tag to uppercase. While the English locale would convert “title” to “TITLE,” the Turkish locale will convert “title” to “T?TLE,” where “?” is the Latin capital letter “I” with a dot above the character [API 2006].

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

Compliant Solution (Explicit Locale)

This compliant solution explicitly sets the locale to English to avoid unexpected results.

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

This rule also applies to the `String.equalsIgnoreCase()` method.

Compliant Solution (Default Locale)

This compliant solution sets the default locale to English before proceeding with string operations.

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

Risk Assessment

Failure to specify the appropriate locale when using locale-dependent methods on locale-dependent data may result in unexpected behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
IDS09-J	medium	probable	medium	P8	L2

Bibliography

[API 2006]	Class `String`

■ IDS10-J. Do not split characters between two data structures

Legacy software frequently assumes that every character in a string occupies 8 bits (a Java `byte`). The Java language assumes that every character in a string occupies 16 bits (a Java `char`). Unfortunately, neither the Java `byte` nor Java `char` data types can represent all possible Unicode characters. Many strings are stored or communicated using encodings such as `UTF-8` that support characters with varying sizes.

While Java strings are stored as an array of characters and can be represented as an array of bytes, a single character in the string might be represented by two or more consecutive elements of type `byte` or of type `char`. Splitting a `char` or `byte` array risks splitting a multibyte character.

Ignoring the possibility of supplementary characters, multibyte characters, or combining characters (characters that modify other characters) may allow an attacker to bypass input validation checks. Consequently, characters must not be split between two data structures.

Multibyte Characters

Multibyte encodings are used for character sets that require more than one byte to uniquely identify each constituent character. For example, the Japanese encoding Shift-JIS (shown below) supports multibyte encoding where the maximum character length is two bytes (one leading and one trailing byte).

Byte Type	**Range**
single-byte	`0x00` through `0x7F` and `0xA0` through `0xDF`
lead-byte	`0x81` through `0x9F` and `0xE0` through `0xFC`
trailing-byte	`0x40-0x7E` and `0x80-0xFC`

The trailing byte ranges overlap the range of both the single-byte and lead-byte characters. When a multibyte character is separated across a buffer boundary, it can be interpreted differently than if it were not separated across the buffer boundary; this difference arises because of the ambiguity of its composing bytes [Phillips 2005].

Supplementary Characters

According to the Java API [API 2006] class `Character` documentation (Unicode Character Representations):

> The `char` data type (and consequently the value that a `Character` object encapsulates) are based on the original Unicode specification, which defined characters as fixed-width 16-bit entities. The Unicode standard has since been changed to allow for characters whose representation requires more than 16 bits. The range of legal code points is now `\u0000` to `\u10FFFF`, known as Unicode scalar value.
>
> The Java 2 platform uses the UTF-16 representation in `char` arrays and in the `String` and `StringBuffer` classes. In this representation, supplementary characters are represented as a pair of `char` values, the first from the high-surrogates range, (`\uD800-\uDBFF`), the second from the low-surrogates range (`\uDC00-\uDFFF`).
>
> An `int` value represents all Unicode code points, including supplementary code points. The lower (least significant) 21 bits of `int` are used to represent Unicode code points, and the upper (most significant) 11 bits must be zero. Unless otherwise specified, the behavior with respect to supplementary characters and surrogate char values is as follows:
>
> - The methods that only accept a `char` value cannot support supplementary characters. They treat `char` values from the surrogate ranges as undefined characters. For example, `Character.isLetter('\uD840')` returns `false`, even though this specific value if followed by any low-surrogate value in a string would represent a letter.

- The methods that accept an `int` value support all Unicode characters, including supplementary characters. For example, `Character.isLetter(0x2F81A)` returns `true` because the code point value represents a letter (a CJK ideograph).

Noncompliant Code Example (Read)

This noncompliant code example tries to read up to 1024 bytes from a socket and build a `String` from this data. It does this by reading the bytes in a while loop, as recommended by rule FIO10-J. If it ever detects that the socket has more than 1024 bytes available, it throws an exception. This prevents untrusted input from potentially exhausting the program's memory.

```
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;
}
```

This code fails to account for the interaction between characters represented with a multibyte encoding and the boundaries between the loop iterations. If the last byte read from the data stream in one `read()` operation is the leading byte of a multibyte character, the trailing bytes are not encountered until the next iteration of the `while` loop. However, multibyte encoding is resolved during construction of the new `String` within the loop. Consequently, the multibyte encoding can be interpreted incorrectly.

Compliant Solution (Read)

This compliant solution defers creation of the string until all the data is available.

```
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;
}
```

This code avoids splitting multibyte-encoded characters across buffers by deferring construction of the result string until the data has been read in full.

Compliant Solution (Reader)

This compliant solution uses a `Reader` rather than an `InputStream`. The `Reader` class converts bytes into characters on the fly, so it avoids the hazard of splitting multibyte characters. This routine aborts if the socket provides more than 1024 characters rather than 1024 bytes.

```
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;
  }
```

Noncompliant Code Example (Substring)

This noncompliant code example attempts to trim leading letters from the `string`. It fails to accomplish this task because `Character.isLetter()` lacks support for supplementary and combining characters [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);
  }
```

Noncompliant Code Example (Substring)

This noncompliant code example attempts to correct the problem by using the `String.codePointAt()` method, which accepts an `int` argument. This works for supplementary characters but fails for combining characters [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);
}
```

Compliant Solution (Substring)

This compliant solution works both for supplementary and for combining characters [Hornig 2007]. According to the Java API [API 2006] class `java.text.BreakIterator` documentation:

> The `BreakIterator` class implements methods for finding the location of boundaries in text. Instances of `BreakIterator` maintain a current position and scan over text returning the index of characters where boundaries occur.

The boundaries returned may be those of supplementary characters, combining character sequences, or ligature clusters. For example, an accented character might be stored as a base character and a diacritical mark.

```
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);
  }
}
```

To perform locale-sensitive `String` comparisons for searching and sorting, use the `java.text.Collator` class.

Risk Assessment

Failure to correctly account for supplementary and combining characters can lead to unexpected behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
IDS10-J	low	unlikely	medium	P2	L3

Bibliography

[API 2006]	Classes `Character` and `BreakIterator`
[Hornig 2007]	Problem Areas: Characters

■ IDS11-J. Eliminate noncharacter code points before validation

In some versions prior to Unicode 5.2, conformance clause C7 allows the deletion of noncharacter code points. For example, conformance clause C7 from Unicode 5.1 states [Unicode 2007]:

> C7. When a process purports not to modify the interpretation of a valid coded character sequence, it shall make no change to that coded character sequence other than the possible replacement of character sequences by their canonical-equivalent sequences or the deletion of noncharacter code points.

According to the Unicode Technical Report #36, Unicode Security Considerations [Davis 2008b], Section 3.5, "Deletion of Noncharacters":

> Whenever a character is invisibly deleted (instead of replaced), such as in this older version of C7, it may cause a security problem. The issue is the following: A gateway might be checking for a sensitive sequence of characters, say "delete." If what is passed in is "deXlete," where X is a noncharacter, the gateway lets it through: The sequence "deXlete" may be in and of itself harmless. However, suppose that later on, past the gateway, an internal process invisibly deletes the X. In that case, the sensitive sequence of characters is formed, and can lead to a security breach.

Any string modifications, including the removal or replacement of noncharacter code points, must be performed before any validation of the string is performed.

Noncompliant Code Example

This noncompliant code example accepts only valid ASCII characters and deletes any non-ASCII characters. It also checks for the existence of a `<script>` tag.

Input validation is being performed before the deletion of non-ASCII characters. Consequently, an attacker can disguise a `<script>` tag and bypass the validation checks.

```
// "\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>"
```

Compliant Solution

This compliant solution replaces the unknown or unrepresentable character with Unicode sequence `\uFFFD`, which is reserved to denote this condition. It also does this replacement before doing any other sanitization, in particular, checking for `<script>`. This ensures that malicious input cannot bypass filters.

```
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 {
  // ...
}
```

According to the Unicode Technical Report #36, Unicode Security Considerations [Davis 2008b], "`U+FFFD` is usually unproblematic, because it is designed expressly for this kind of purpose. That is, because it doesn't have syntactic meaning in programming languages or structured data, it will typically just cause a failure in parsing. Where the output character set is not Unicode, though, this character may not be available."

Risk Assessment

Deleting noncharacter code points can allow malicious input to bypass validation checks.

Rule	**Severity**	**Likelihood**	**Remediation Cost**	**Priority**	**Level**
IDS11-J	high	probable	medium	P12	L1

Related Guidelines

MITRE CWE	CWE-182. Collapse of data into unsafe value

Bibliography

[API 2006]	
[Davis 2008b]	3.5, Deletion of Noncharacters
[Weber 2009]	Handling the Unexpected: Character-Deletion
[Unicode 2007]	
[Unicode 2011]	

■ IDS12-J. Perform lossless conversion of String data between differing character encodings

Performing conversions of `String` objects between different character encodings may result in loss of data.

According to the Java API [API 2006], `String.getBytes(Charset)` method documentation:

> This method always replaces malformed-input and unmappable-character sequences with this charset's default replacement byte array.

When a `String` must be converted to bytes, for example, for writing to a file, and the string might contain unmappable character sequences, proper character encoding must be performed.

Noncompliant Code Example

This noncompliant code example [Hornig 2007] corrupts the data when `string` contains characters that are not representable in the specified `charset`.

```
// 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);
}
```

Compliant Solution

The `java.nio.charset.CharsetEncoder` class can transform a sequence of 16-bit Unicode characters into a sequence of bytes in a specific `Charset`, while the `java.nio.charset.Character-Decoder` class can reverse the procedure [API 2006]. Also see rule FIO11-J for more information.

This compliant solution [Hornig 2007] uses the `CharsetEncoder` and `CharsetDecoder` classes to handle encoding conversions.

```
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();
}
```

Noncompliant Code Example

This noncompliant code example [Hornig 2007] attempts to append a string to a text file in the specified encoding. This is erroneous because the `String` may contain unrepresentable characters.

```
// 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();
}
```

Compliant Solution

This compliant solution [Hornig 2007] uses the `CharsetEncoder` class to perform the required function.

```
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();
}
```

Use the `FileInputStream` and `InputStreamReader` objects to read back the data from the file. The `InputStreamReader` accepts an optional `CharsetDecoder` argument, which must be the same as that previously used for writing to the file.

Risk Assessment

Use of nonstandard methods for performing character-set-related conversions can lead to loss of data.

Rule	**Severity**	**Likelihood**	**Remediation Cost**	**Priority**	**Level**
IDS12-J	low	probable	medium	P4	L3

Related Guidelines

MITRE CWE	CWE-838. Inappropriate encoding for output context
	CWE-116. Improper encoding or escaping of output

Bibliography

[API 2006]	Class `String`
[Hornig 2007]	Global Problem Areas: Character Encodings

■ IDS13-J. Use compatible encodings on both sides of file or network I/O

Every Java platform has a default character encoding. The available encodings are listed in the *Supported Encodings* document [Encodings 2006]. A conversion between characters and sequences of bytes requires a character encoding to specify the details of the conversion. Such conversions use the system default encoding in the absence of an explicitly specified encoding. When characters are converted into an array of bytes to be sent as output, transmitted across some communication channel, input, and converted back into characters, compatible encodings must be used on both sides of the conversation. Disagreement over character encodings can cause data corruption.

According to the Java API [API 2006] for the `String` class:

> The length of the new `String` is a function of the charset, and for that reason may not be equal to the length of the byte array. The behavior of this constructor when the given bytes are not valid in the given charset is unspecified.

Binary data that is expected to be a valid string may be read and converted to a string by exception FIO11-EX0.

Noncompliant Code Example

This noncompliant code example reads a byte array and converts it into a `String` using the platform's default character encoding. When the default encoding differs from the encoding that was used to produce the byte array, the resulting `String` is likely to be incorrect. Undefined behavior can occur when some of the input lacks a valid character representation in the default encoding.

```
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
      }
    }
  }
```

Compliant Solution

This compliant solution explicitly specifies the intended character encoding in the second argument to the `String` constructor.

```
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
    }
  }
}
```

Exceptions

IDS13-EX0: An explicit character encoding may be omitted on the receiving side when the data is produced by a Java application that uses the same platform and default character encoding and is communicated over a secure communication channel (see MSC00-J for more information).

Risk Assessment

Failure to specify the character encoding while performing file or network I/O can result in corrupted data.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
IDS13-J	low	unlikely	medium	P2	L3

Automated Detection Sound automated detection of this vulnerability is not feasible.

Bibliography

[Encodings 2006]

This page intentionally left blank

Index