SECURE CODING ASSISTANT: ENFORCING SECURE CODING PRACTICES
USING THE ECLIPSE DEVELOPMENT ENVIRONMENT

A Project

Presented to the faculty of the Department of Computer Science
California State University, Sacramento

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

by

Benjamin Allen White

SPRING
2016
SECURE CODING ASSISTANT: ENFORCING SECURE CODING PRACTICES
USING THE ECLIPSE DEVELOPMENT ENVIRONMENT

A Project

by

Benjamin Allen White

Approved by:

____________________________, Committee Chair
Dr. Jun Dai

____________________________, Second Reader
Dr. Cui Zhang

____________________________
Date
Student: Benjamin Allen White

I certify that this student has met the requirements for format contained in the University format manual, and that this project is suitable for shelving in the Library and credit is to be awarded for the project.

Dr. Jinsong Ouyang
Date

Department of Computer Science
Abstract

of

SECURE CODING ASSISTANT: ENFORCING SECURE CODING PRACTICES USING THE ECLIPSE DEVELOPMENT ENVIRONMENT

by

Benjamin Allen White

Developing secure software in a world where companies like Anthem Blue Cross, Twitter, Facebook, and Target have had massive amounts of data stolen by hackers is as challenging as it is important. Insecure coding practices are major contributors to software security vulnerabilities. Even though several static analysis tools are available that can search for and identify security holes in software applications, this process runs too late and any remediation will be more costly after large portions of the software have been built. The early detection tools that do exist are closed source and utilize proprietary software vulnerability rule sets.

What is missing is an open-sourced secure coding enforcement tool utilizing well-documented rules that software developers can use to predict potential pitfalls, learn from their mistakes and aid in the construction of secure programs as they build them. To address the need, we have designed a new tool called Secure Coding Assistant for the Eclipse Development Environment that semi-automates several secure coding rules set forth by the CERT division of the Software Engineering Institute at Carnegie Mellon University. The tool detects violations of the CERT rules for the Java programming
language but it is easily extensible to other languages supported by Eclipse. It is an open-
source tool with an emphasis on educating software developers in secure coding
practices. The tool is disseminated via github at

http://benw408701.github.io/SecureCodingAssistant/.

_______________________, Committee Chair
Dr. Jun Dai

_______________________
Date
ACKNOWLEDGEMENTS

I would like to thank my project advisor Dr. Jun Dai for the countless hours spent in his office reading and re-reading every sentence and fine-tuning every detail. I would also like to thank my second reader Dr. Cui Zhang who made this project possible. The concept of a secure coding assistant based on the CERT library was wholly hers. This has been an amazing project to work on and without her it would not have been possible.

I also thank my boss Lisa Steele. Working full time while pursuing this degree has not been easy. With her support and the support of my company I had the flexible hours and financial aid needed to pursue this degree. I will never forget the generosity that she and everyone else that I work with have shown me.

Most importantly I would like to thank my friends and family. My friends for understanding while I’ve been distant the last few years and my family for their love and support. My parents, Kevin and Sharon White, are my inspiration. Most importantly I want to thank my partner and the love of my life Brosnan Rhodes. I would have never finished this without him. He is my rock and his love and understanding are astonishing. He provided the encouragement I needed to make it through and there were times in this process that I don’t know how he managed to tolerate me. I can’t thank everyone enough for what they have done to make this possible.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledgements ........................................................................ vii</td>
</tr>
<tr>
<td>List of Tables ................................................................................. x</td>
</tr>
<tr>
<td>List of Figures ................................................................................ xi</td>
</tr>
<tr>
<td>Chapter</td>
</tr>
<tr>
<td>INTRODUCTION ................................................................................. 1</td>
</tr>
<tr>
<td>RELATED WORK .............................................................................. 4</td>
</tr>
<tr>
<td>METHODOLOGY ............................................................................. 7</td>
</tr>
<tr>
<td>DESIGN ......................................................................................... 10</td>
</tr>
<tr>
<td>IMPLEMENTATION .......................................................................... 14</td>
</tr>
<tr>
<td>5.1 Rule Selection ........................................................................... 14</td>
</tr>
<tr>
<td>5.2 Plugin Implementation Details ................................................... 15</td>
</tr>
<tr>
<td>5.3 Compilation Participants .......................................................... 17</td>
</tr>
<tr>
<td>5.4 The Abstract Syntax Tree ............................................................ 18</td>
</tr>
<tr>
<td>5.5 Utility Library .......................................................................... 20</td>
</tr>
<tr>
<td>5.6 Rule Logic ................................................................................ 22</td>
</tr>
<tr>
<td>EVALUATION ................................................................................. 25</td>
</tr>
<tr>
<td>6.1 Efficiency ................................................................................ 25</td>
</tr>
<tr>
<td>6.2 Accuracy .................................................................................. 26</td>
</tr>
<tr>
<td>6.2.1 CERT Validation ................................................................... 26</td>
</tr>
<tr>
<td>6.2.2 False Positive Study ............................................................... 27</td>
</tr>
<tr>
<td>6.2.3 False Negative Study .............................................................. 32</td>
</tr>
<tr>
<td>viii</td>
</tr>
</tbody>
</table>
7. LIMITATIONS, CONCLUSION AND FUTURE WORK .............................................. 36

Appendix A. CERT Rule Selection ........................................................................ 38

Appendix B. Application Source Code .................................................................. 51

Appendix C. CERT Validation Source Code ............................................................ 101

References .............................................................................................................. 109
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Tables</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Static analysis tools that scan for security vulnerabilities compared</td>
<td>5</td>
</tr>
<tr>
<td>2. Utility Library methods by problem solved</td>
<td>21</td>
</tr>
<tr>
<td>3. The IRule interface</td>
<td>22</td>
</tr>
<tr>
<td>4. Plugin efficiency analysis</td>
<td>26</td>
</tr>
<tr>
<td>5. SecuriBench test results</td>
<td>30</td>
</tr>
<tr>
<td>6. False positive analysis</td>
<td>31</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figures</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. CVE’s published year over year.</td>
<td>2</td>
</tr>
<tr>
<td>2. High-level flow of Secure Coding Assistant.</td>
<td>11</td>
</tr>
<tr>
<td>3. Sample flow of SQL injection violation detection.</td>
<td>12</td>
</tr>
<tr>
<td>4. Secure Coding Assistant update site.</td>
<td>16</td>
</tr>
<tr>
<td>5. The AST representation of a code segment.</td>
<td>19</td>
</tr>
<tr>
<td>6. Iterating through rule collection.</td>
<td>23</td>
</tr>
<tr>
<td>7. Implementation of the IDS00-J rule.</td>
<td>24</td>
</tr>
<tr>
<td>8. Sample code segment violating IDS00-J.</td>
<td>27</td>
</tr>
<tr>
<td>9. Sample code segment showing proper usage of executing a SQL command as shown in IDS00-J.</td>
<td>27</td>
</tr>
<tr>
<td>10. SQL injection example from OWASP website.</td>
<td>33</td>
</tr>
<tr>
<td>11. Output of SQL injection detection.</td>
<td>33</td>
</tr>
<tr>
<td>12. Validate before canonicalize example from CWE.</td>
<td>34</td>
</tr>
<tr>
<td>13. Comparison of classes by name from CWE.</td>
<td>35</td>
</tr>
</tbody>
</table>
Chapter 1

INTRODUCTION

Finding secure coding standards is not difficult but following them is. In a 2011 study [1], Veracode analyzed over 6,750 web applications and found that a third of these had SQL code injection vulnerabilities. According to the study, secure coding experts documented how to address these vulnerabilities over a decade ago and it involves something as simple as parameterized SQL statements [1]. A study in India [2] found that less than 1% of engineering students are skilled in secure programming. Even the most “security aware” professionals are writing their code first then adding security as an afterthought [3]. The evidence indicates that there is an overwhelming lack of knowledge and experience when it comes to developing secure software.

Coding for security and especially software security is an extremely important issue. In 2013 Facebook, Twitter and Apple were all targets of large-scale attacks. The Twitter attack detailed in the Journal of Internet Law [4] resulted in 250,000 accounts compromised and stolen usernames, passwords and other personal information. Later that year Target was a victim of a security breach and as many as 40 million credit and debit card accounts were compromised [5]. Home Depot’s 2,157 stores fell prey to a data security breach in 2014 [6]. CNN [7] also reported two alarming attacks on our government. In July 2014, the Department of Energy was hacked and the attackers stole 100,000 records of personally identifiable information. Earlier in the year, hackers hit the Army Corps of Engineers and took information on 85,000 dams across the nation. Lastly,
the medical industry has been a large target as well and we see in [8] that Anthem Blue Cross had a staggering “millions” of personal health records stolen. If these companies had software systems developed to a higher degree of secure coding standards, then these incidents would have been less likely to have occurred.

![Common Vulnerability Enumeration Count Year over Year](image)

**Figure 1. CVE’s published year over year.**

Every year there are thousands of newly documented software vulnerabilities. The Common Vulnerability Enumeration (CVE) is a database of known security vulnerabilities that is commonly cross-referenced by security tools and is one of the most recognizable vulnerability databases today. The list of vulnerabilities may be accessed online at [9] in a raw format. Figure 1 is a compilation of these documented
vulnerabilities after removing all items marked as “reject,” “reserved” or “deprecated.”

The remaining published vulnerabilities count in the thousands year over year, starting with a mere 1,500 in 1999 when CVE was founded and leaping past 7,000 in 2014. There is no possible way that a software developer could be expected to learn thousands of CVE’s and understand how to write secure code that is resilient to them. The Software Engineering Institute (SEI) of Carnegie Mellon University has made it so that they do not have to. SEI’s CERT division documents secure coding rules and recommendations that are language-specific and help protect against these thousands of known vulnerabilities [10]. For instance, there are only 160 secure coding Java rules published by CERT as opposed to the tens of thousands of published CVE’s.

The goal of the Secure Coding Assistant is to alert developers when they have violated a CERT rule, educate them on proper secure coding practices and provide an open-source tool to the development community. Though other tools exist that implement some of the CERT rules, the Secure Coding Assistant is the only tool that specializes in CERT rules and is open source. The initial version of the Secure Coding Assistant detects 21 secure coding rules for Java, see section 5.1 Rule Selection, but is designed to allow for the addition of rules and programming languages.
Chapter 2

RELATED WORK

There are many tools available to developers for building secure applications. In Table 1, several of these tools are compared. The first three are widely used commercial tools and the remaining eight represent a comprehensive list of vulnerability detection plugins for the Eclipse Development Environment. Like the Secure Coding Assistant, several of these tools provide early-detection mechanisms. All of these tools are static analysis tools.

Static analysis tools as described by Díaz and Bermejo [11] all follow the same basic workflow: transform the code, analyze certain properties and display results. In this same article, they also touch on the static analysis counterpart, dynamic analysis, which analyzes properties of an application while it is running. Dynamic tools are capable of detecting vulnerabilities that static tools cannot but provide even later detection than static tools and are not included in the comparison table. The cost of catching and fixing vulnerabilities later in the software lifecycle is so significant that companies such as Microsoft have begun to hold “developers [personally] liable for the security and integrity of the code they write [12].” The solution is to minimize the security vulnerabilities in application code at the time the developer is writing it. The process described in [11] must be running in the background while the developer is typing their code. The tools in Table 1 listed as “Early” detection provide the live feedback necessary for the developer to write secure code and learn as they do it. The tools listed as “Late”
detection require the developer to finish writing their code, launch the tool and load the code, then review the results and make changes to the source as necessary. The Secure Coding Assistant falls into the “Early” category since it alerts the developer when they are at risk of violating a secure coding rule. This type of tool provides feedback at the earliest possible stage in the development process and teaches developers secure coding practices at the same time.

Table 1. Static analysis tools that scan for security vulnerabilities compared.

<table>
<thead>
<tr>
<th>Company</th>
<th>Tool</th>
<th>Early/Late Detection</th>
<th>Open/Closed Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vericode</td>
<td>White Box Testing/Binary Static Analysis</td>
<td>Late</td>
<td>Closed</td>
</tr>
<tr>
<td>HP</td>
<td>Fortify Static Code Analyzer</td>
<td>Late</td>
<td>Closed</td>
</tr>
<tr>
<td>WhiteHat Security</td>
<td>Sentinel Source</td>
<td>Late</td>
<td>Closed</td>
</tr>
<tr>
<td>Klockwork</td>
<td>Klockwork Insight</td>
<td>Early</td>
<td>Closed</td>
</tr>
<tr>
<td>Cigital, Inc.</td>
<td>SecureAssist</td>
<td>Early</td>
<td>Closed</td>
</tr>
<tr>
<td>The Code Master</td>
<td>Early Security Vulnerability Detector (ESVD)</td>
<td>Early</td>
<td>Closed</td>
</tr>
<tr>
<td>Towson University</td>
<td>Static Security Vulnerability Analyzer</td>
<td>Early</td>
<td>Closed</td>
</tr>
<tr>
<td>Contrast Security</td>
<td>Contrast for Eclipse</td>
<td>Late</td>
<td>Closed</td>
</tr>
<tr>
<td>Sonar Source</td>
<td>SonarLint</td>
<td>Early</td>
<td>Closed</td>
</tr>
<tr>
<td>Checkmarx</td>
<td>CxSuite</td>
<td>Late</td>
<td>Closed</td>
</tr>
<tr>
<td>Red Lizard Software</td>
<td>Goanna Studio</td>
<td>Early</td>
<td>Closed</td>
</tr>
</tbody>
</table>

Even though several of the tools available provide an early detection mechanism, none of them are open source and only one, Goanna Studio by Red Lizard Software [13], mentions validation against the CERT secure coding rules. Goanna Studio also only provides support for the C and C++ programming languages and is closed source whereas the Secure Coding Assistant currently supports Java but extensible to others and is open
source. In addition, even though Goanna Studio lists CERT as being one of the sources for secure coding rules, it also lists several others and it does not indicate specifically which rule is violated and the rule source when a violation is detected. There are other tools such as the Early Security Vulnerability Detector (ESVD) and Static Security Vulnerability Analyzer, available at [14, 15], which were developed by graduate students and share many of the same goals as the Secure Coding Assistant but neither of them are open source and with the research complete and published will likely no longer be maintained. At this time there does not exist an open-source tool that the development community can build upon and optimize as new vulnerabilities are documented and new detection methods are discovered.
Chapter 3

METHODOLOGY

In addition to being an open-source development tool that evolves with public contribution, the Secure Coding Assistant has two goals. The first is to provide software developers with instant feedback as they write their source code. Similar to the way a word processor would alert a writer when they have a grammar or spelling mistake, the Secure Coding Assistant provides messages to the developer that are easy to understand and integrate well into their workflow. The second is to educate on the CERT secure coding practices. Initially the C# programming language was considered but the lack of well-documented secure programming rules for C# was the driving force behind developing a tool that focuses on the CERT secure coding rules for Java [10].

A decision was also made regarding which development environment to initially support. Currently NetBeans and Eclipse are both very popular development environments for the Java programming language. They both provide plugin development tools and support multiple programming languages. Eclipse, however, is widely used by the student body at California State University Sacramento and since the goal was to develop a tool that could be used by the students, the decision was made to develop a plugin for Eclipse.

Some of the static analysis tools described earlier are categorized as “early” detection tools. This means that they provide feedback to the programmers as they are typing their code. The other tools require the developer to complete a section of code, send it to the tool and then receive feedback. The Secure Coding Assistant follows the
early detection methodology and provides live feedback as the source code is being typed. This type of feedback is already quite common in a development environment. Modern development environments provide live syntax checking that validates that the code adheres the grammatical rules of the programming language. For instance, a rule in many programming languages is that the addition operator requires two operands on either side that are numerical expressions. For example, trying to add the number 1 to the text string “hello” would result in a syntax error under these rules as would putting the addition operator at the beginning of the operands rather than in-between. This type of instant feedback saves the developer time and helps them write syntactically correct program code before compilation rather than waiting and having to fix those problems later. This is exactly why the Secure Coding Assistant needs to be designed similarly.

The types of mistakes that lead to insecure program code are best identified and corrected early, while the developer is writing their code which reduces time and cost of remediation later in the development process.

The second goal of the Secure Coding Assistant, education, is made possible through the CERT website and their thorough documentation of the Java secure coding rules [10]. The alerts that the programmers receive must provide a comprehensive message that clearly indicates what rule was violated and what measures they can take towards remediation. The CERT website provides this information along with various examples of secure code violations and proposed fixes and is incorporated into the Secure
Coding Assistant alert messages. This creates a natural learning environment for secure coding practices during the development process.
Chapter 4
DESIGN

The Secure Coding Assistant continuously runs in the background of the development environment and looks for violations to secure coding rules. The high-level flow is outlined in Figure 2. The workflow assumes that a syntax tree of the code segment being analyzed has been built. A syntax tree is a representation of the source code that is easily traversed by a tool like the Secure Coding Assistant or any other tool that participates in parsing a programming language and is used extensively in the implementation. Changes to the syntax tree initiate the code analysis process. Once the process begins, any existing secure coding violations tied to the tree are cleared before the tree is traversed. Each node of the tree is analyzed and if the node contains a rule violation then a new marker is created in the source code where the rule violation is detected. The rule violation logic is the only component that is language-specific. Markers alert the programmer that a violation has occurred and contain the name of the rule, a description from the CERT website [10] and the recommendation from CERT to fix the violation. After the syntax tree traversal is completed the application returns to the initial start state and waits to run again. The markers that remain in the source code display in a tooltip fashion. As the programmer makes changes to the source code the tool runs again in the background, removes all existing markers and only adds new ones if violations exist. In this manner, rule violations that have been fixed will no longer show.
Since Eclipse supports multiple programming languages, the design allows for future support of any programming language supported by Eclipse. To accomplish this, the only portion of the workflow that is language-specific is the rule violation detection. This component of the workflow is outlined in bold in Figure 2. An example of a Java-specific rule is IDS00-J [16] which is “Prevent SQL injection” which shows how the Java PreparedStatement.setString() method is the most effective way to sanitize data being passed to a SQL query string. In Figure 3 this Java secure coding rule is translated into a workflow to detect violations. The logic used for the rule violation assumes that if a SQL query is being run then there must be at least one parameter that is

Figure 2. High-level flow of Secure Coding Assistant.
obtained from the user so there must be at least one call to

`PreparedStatement.setString()`. There is a possibility of a false positive
when the query string does not require any parameters and a false negative when the
programmer uses the `setString()` method once but not for subsequent parameters,
but the emphasis is on the general case since capturing the number of user parameters
required would not be feasible.

Figure 3. Sample flow of SQL injection violation detection.
Additional secure programming rules are all crafted in a similar fashion using logic that looks for a node which is a call to a method, instantiation of an object, inheritance from a base class or some other structure identified in the CERT rule. Once identified, the context in which that node is executed is evaluated programmatically. For instance, many of the data sanitization rules require that strings be normalized before they are processed. Detecting a rule violation of this type reduces to finding a call to the method that processes the data then checking to see if a call to a normalize method occurred prior and in the same scope.
Chapter 5

IMPLEMENTATION

5.1 Rule Selection

The CERT website references 185 secure coding rules for the Java programming language [10]. Before selecting which rules to include in the tool, each rule was reviewed and classified as to whether or not automation would be possible. Some rules cannot be automated since they require knowledge of the problem domain. NUM03-J, for instance, states that integer types in Java cannot be used to represent unsigned data [17]. Java programs that need to interoperate with languages like C and C++ must use integer types that can represent the range of unsigned data. This type of rule is very difficult to detect using an automated tool. The tool would need to know that the application is going to be used with components that use unsigned data. The only feasible way to detect this type of vulnerability is to have knowledge of the intended use of the code segment which is not practical for an automated tool. Furthermore, there are entire categories that require some type of metadata for an automated tool to function. An example of this is the “Thread-Safety” category. Without knowledge that a code segment is intended to be run in a multi-threaded environment the tool cannot adequately detect rule violations. Rules like these are infeasible to implement using a tool like the Secure Coding Assistant.

Many of the rules on the CERT website clearly state if they are automatable or not [10]. Others do not say. Out of the total 160 rules available there are 85 that either state explicitly that they may be automated or appear to be automatable. Also, the CERT
website divides the secure coding rules into 20 categories. Three out of the 20 categories do not contain any rules that can be automated leaving 17 categories with eligible rules to automate. Rules were chosen from these categories based on the severity of the potential vulnerability and an effort was made to sample from as many rule categories as possible. A total of 21 rules were chosen covering 15 categories which represents 88% of the eligible categories and 25% of the total eligible rules.

5.2 Plugin Implementation Details

Eclipse provides a Plugin Development Environment (PDE) that gives plugin developers the ability to extend and customize the development environment. The plugin structure itself is defined using a markup language that contains information on what attributes of the environment are being customized. For instance, a plugin that adds a custom command to one of the menus would extend org.eclipse.ui.menus as well as org.eclipse.ui.commands. Along with the extension points there are usually other attributes that are defined as well such as the menu name or the name of the class that contains an execution path when the command is invoked. The Secure Coding Assistant extends two points, the first is org.eclipse.jdt.core.compilationParticipant and the second is org.eclipse.core.resources.problemmarker.

The first extension point, compilationParticipant, allows the plugin to participate in the compilation process. Part of this extension point also requires a class definition that
extends a super class, called CompilationParticipant, which receives notifications at various stages of the compilation process. The second extension point, problemmarker, allows for the creation of custom markers. A marker in Eclipse is a warning, alert, task, or error that developers use to track issues in their code or leave reminders to revisit sections of the source code. The Secure Coding Assistant used generic “warning” markers in early stages of development but the need to track the various severity levels of the CERT secure coding rules necessitated the change to a customized marker. With a customized marker the base marker type can be extended to have additional fields of any types. The secure coding rule violation markers have an additional enumerated field for severity to capture the three severity levels reflected on the CERT website [10].

Figure 4. Secure Coding Assistant update site.
In addition to the plugin definition itself, Eclipse provides the ability to package similar plugins as a “feature” and deploy them on an update site. Deploying plugins through an update site is beneficial for anyone wishing to install the plugin for two reasons. First, Eclipse has an “Install New Software” feature in the Help menu that you can use to install new plugins using the update site. Since the feature is installed through the Eclipse update tool, it is equally convenient to uninstall the plugin if it is no longer needed. Secondly, using the update site lets users quickly check for updates using the “Check for Updates” feature which is also in the Help menu. The update site is online at http://bwprojects.org/SecureCodingPlugin and the display of the update tool is shown in Figure 4. Note that the current version number 1.0.0.201510152028 reflects the date and time of the build (10/15/2015 at 8:28 PM) which is generated automatically by Eclipse when releasing a new build of the feature.

5.3 Compilation Participants

The compilationParticipant extension point directs compilation events to any custom class that extends the CompilationParticipant super class. The SecureCompilationParticipant is such a class and is the top-most component in the design of the Secure Coding Assistant. The SecureCompilationParticipant’s reconcile() method (inherited from CompilationParticipant) is called every time a “reconcile” event occurs. This type of event is triggered by any modification to the application source code. The only parameter to the reconcile() method is an object that conveys information
about the context of the reconciliation such as the name of the file that was modified, the types of changes that occurred and a copy of the Abstract Syntax Tree (AST).

The SecureCompilationParticipant uses the following approach to handling reconciliation events: look for a change to the AST, if there was a change then get a reference to the new AST, clear existing markers, then traverse the new AST and create new markers as necessary. In general, compilation participants are not compiling the source code into an assembly language but only responding to compilation events which include the dynamic syntax checking that is so common in modern development environments like Eclipse. This is exactly what the Secure Coding Assistant is doing, checking the source code, but rather than checking the syntax the emphasis is on the semantics and how the compiled application would affect the application security. The SecureCompilationParticipant is the backbone to the Secure Coding Assistant plugin and is responsible for managing rule detection and marker creation and management.

5.4 The Abstract Syntax Tree

An Abstract Syntax Tree (AST) is a common representation of a block of source code and is primarily used in the study of compiler design. Syntax trees are traversed depth-first and define the order of operations. A simple example is shown in Figure 5 where a block of code contains two statements, the first being a declaration of a variable x of type integer and the second the assignment of the value 5 to x.
The Eclipse development environment provides a Java Development Tools (JDT) library. The core components of this library are found in `org.eclipse.jdt.core`. These tools contain a Java language compiler and many other helpful compilation tools including the AST representation of the source code that is being compiled. At the time that the `reconcile()` method is called the AST has already been built since it is required by any compilation participants. Eclipse also provides a mechanism for traversing the syntax tree, the ASTVisitor class. Any application that wishes to traverse the syntax tree and execute code at any given node in the tree may implement a class that extends ASTVisitor and override one of the many `visit()` methods. ASTVisitor defines a `visit()` method for each type of node (method declaration, assignment,
method invocation, etc.) as well as a `preVisit()` and `postVisit()` method which occurs before and after visiting every node. Note that even though the `visit()` method is defined for each node type, the `preVisit()` and `postVisit()` methods are defined once, generically, for any node type. The Secure Coding Assistant uses the `preVisit()` method in its `SecureNodeAnalyzer` class which attaches to the AST from the `SecureCompilationParticipant`. There is also a second custom ASTVisitor that is used by the Utility Library that supports the rule detection methods. This ASTVisitor is called `ASTNodeProcessor` and defines several `visit()` methods that are used to gather data on the context of a node while evaluating whether or not it contains a CERT rule violation.

5.5 Utility Library

The rule detection logic for many of the CERT rules can be reduced to several sub-problems, these problems are shown in Table 2 along with the methods from the Utility Library that have been developed to solve the given problem. These methods use the `ASTNodeProcessor` to traverse the AST a second time and gather data on the nodes that occur before and after the node being processed. With this library of reusable code, future rules may be built much easier.
Table 2. Utility Library methods by problem solved.

<table>
<thead>
<tr>
<th>Problem Solved</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Was a call made to method x?</td>
<td>calledMethod()</td>
</tr>
<tr>
<td>Was method x called prior to method y?</td>
<td>calledPrior()</td>
</tr>
<tr>
<td>Was a variable x modified after a call to method y?</td>
<td>modifiedAfter()</td>
</tr>
<tr>
<td>Was class c instantiated with argument a?</td>
<td>containsInstanceCreation()</td>
</tr>
<tr>
<td>What block b encloses node n?</td>
<td>getEnclosingNode()</td>
</tr>
<tr>
<td>Is argument a in a list of arguments l?</td>
<td>argumentMatch()</td>
</tr>
<tr>
<td>Retrieve method declaration d from a superclass when method m is overriding it.</td>
<td>getSuperClassDeclaration()</td>
</tr>
</tbody>
</table>

The Utility Library evolved throughout the implementation process. When a rule was chosen for implementation, the pseudo-code for the high-level rule logic was added as comments to the source code. If a step in the pseudo code appeared to be common enough to be reusable in other rules, then it was added to the utility library rather than implemented in the block of rule logic itself. Even though the Utility Library operates alongside the rule detection logic which is language-specific, the parameters of the methods in the library are designed to be used for multiple programming languages.

There were also several instances where method overloading was helpful. For instance, \texttt{calledMethod()} was implemented three times. Once to check to see if a method is called from a given class, again to see if it is called from a base class and lastly to see if it is called with particular arguments.
5.6 Rule Logic

Each rule implements the interface IRule and uses the “protected” class modifier so they cannot be instantiated directly. A call to RuleFactory.getAllRules() returns an ArrayList of references to each rule that has been fully implemented. The IRule interface provides a level of abstraction that can be used in marker creation and node checking since all rules share the same fundamental properties. A rule, for instance, may be violated at a particular node and has several properties like the rule name, level of severity and the recommendation when a violation is detected. These fundamental properties implemented by all secure coding rules in the tool are shown in Table 3.

Table 3. The IRule interface.

<table>
<thead>
<tr>
<th>Method Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean violated(ASTNode)</td>
<td>Checks to see if the rule has been violated in a given node</td>
</tr>
<tr>
<td>String getRuleText()</td>
<td>The description of the rule that was violated</td>
</tr>
<tr>
<td>String getRuleName()</td>
<td>The name of the rule violated</td>
</tr>
<tr>
<td>String getRuleRecommendation()</td>
<td>The recommended action that will satisfy the rule</td>
</tr>
<tr>
<td>int securityLevel()</td>
<td>The security level of the violated rule, values are defined as LOW, MEDIUM, and HIGH in the Global.Markers class</td>
</tr>
</tbody>
</table>

The IRule.violated() method has one parameter, the node that is being evaluated, and returns true if a rule violation was detected at the node location and false otherwise. This level of abstraction makes iterating through a large set of rules very straightforward as shown in Figure 6. In this code segment a collection of rules, built by RuleFactory.getAllRules(), each tests a node in the syntax tree. Since this is in
the overridden preVisit method, it is run against each node in the syntax tree in a depth-first traversal.

```java
@Override
public void preVisit(ASTNode node) {
  // Iterate through rules
  for (IRule rule : m_rules)
    if (rule.violated(node))
      m_insecureCodeSegments.add(new InsecureCodeSegment(node, rule, m_context));
}
```

**Figure 6. Iterating through rule collection.**

Rule violation detection is reduced to implementing the logic for the rule’s violated() method. Figure 7 shows the implementation for the IDS00-J rule which states that a SQL query in Java must use calls to the PreparedStatement.setString() method to properly sanitize and place query parameters in a query string before sending it to the database for execution [16]. The implementation is simplified by using Utility Library. First the node is checked to see if it is an invocation of a method. If it is, then the one of the methods from the Utility Library are used to check if PreparedStatement.executeQuery() is called and if it is the rule is violated only if there is not a call to PreparedStatement.setString() prior to PreparedStatement.executeQuery(). Compare this to the high-level flow in Figure 3.
Figure 7. Implementation of the IDS00-J rule.
Chapter 6

EVALUATION

6.1 Efficiency

The Eclipse development environment has a responsiveness monitoring tool that will log delays over a certain threshold. The efficiency analysis for the Secure Coding Assistant was done by setting the monitor threshold to 10 milliseconds then loading 5 SecuriBench source code files 3 times with the plugin enabled and 3 times with the plugin disabled. When testing with the plugin enabled the alert list was cleared after each test. After each load, the total delay was recorded and then the total delay for all three loads were averaged together. The difference between the average load time without the plugin and the average load time with the plugin was recorded as the increase in load. The results of the study in Table 4 show that the plugin added an additional 0.03 to 0.20 seconds to the load time for each source file. The additional processing is in a separate thread so the impact to the user is minimal. While the plugin is processing the source file the alert window is filling with secure coding alerts which does not interfere with the user's ability to scroll through the file and make edits. There appeared to be a correlation between the amount of additional processing time and the number of detected alerts. The last column in the table shows the additional time per alert and ranges from 2 to 4.5 milliseconds.
Table 4. Plugin efficiency analysis.

<table>
<thead>
<tr>
<th>Application</th>
<th>Source File</th>
<th>Alerts</th>
<th>Increase (sec.)</th>
<th>Time per Alert (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pebble</td>
<td>SimpleBlog.java</td>
<td>46</td>
<td>0.2037</td>
<td>4.428</td>
</tr>
<tr>
<td>roller</td>
<td>WebLogEntryFormAction.java</td>
<td>16</td>
<td>0.0713</td>
<td>4.458</td>
</tr>
<tr>
<td>webgoat</td>
<td>CreateDB.java</td>
<td>49</td>
<td>0.1923</td>
<td>3.925</td>
</tr>
<tr>
<td>snipsnap</td>
<td>ConfigurationMap.java</td>
<td>23</td>
<td>0.0270</td>
<td>1.174</td>
</tr>
<tr>
<td>snipsnap</td>
<td>ConfigurationProxy.java</td>
<td>19</td>
<td>0.0380</td>
<td>2.000</td>
</tr>
</tbody>
</table>

6.2 Accuracy

6.2.1 CERT Validation

The CERT website lists several code samples for each secure coding rule along with the rule definition [10]. The samples are presented in pairs, first is an example of a violation of the rule as shown in Figure 8 where user input is fed directly into the query string. After showing the example of the violation, an example of proper coding is given as shown in Figure 9 where the Java PreparedStatement.setString() method is used instead. The initial validation of the Secure Coding Assistant was done using the CERT code segments. Rule logic was not considered to be complete until it adequately detected all secure coding violation examples shown on the CERT website [10] for that particular rule.
Figure 8. Sample code segment violating IDS00-J [16].

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

Figure 9. Sample code segment showing proper usage of executing a SQL command as shown in IDS00-J [16].

```java
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();
```

6.2.2 False Positive Study

The Stanford SecuriBench [18] was used for the false positive study. It consists of applications that have various types of documented vulnerabilities. The Stanford group identified 30 vulnerabilities in 2005 when SecuriBench was first made public. After running seven of the eight programs through the Secure Coding Assistant several thousand potential CERT violations were detected.

The Secure Coding Assistant generated 4,172 secure coding alerts, but the overall distribution shown in Table 5 is quite interesting. Only 8 out of the 21 implemented rules detected violations. Of those 8 rules, 77% of the violations detected were all in one rule, EXP00-J, which states that a programmer should never ignore a value returned by a method [19]. The reason for this is that method return values are often indicators of
whether or not the call was successful or they contain some other output that is beneficial to the caller of the method. According to CERT, “Ignoring method return values can lead to unexpected behavior” [19]. Upon further investigation, it is not always clear what is to be done with the return value. For instance, a large number of alerts were generated on calls to the StringBuffer.append() method. This method returns a reference to the modified buffer but the buffer that calls it is also modified. It is not clear in the Java documentation the reason for the return value but it mentions that developers should be using the StringBuilder class now instead. There were other circumstances where ignoring the return value seemed appropriate. The Properties.setProperty() method returns a reference to the previous value before it overwrites it. Though there are many cases where capturing the previous value is useful, ignoring it is certainly not always a security risk. The high count of EXP00-J violations, even though they are correctly categorized according to CERT, are very likely due to the CERT rule itself needing some additional restrictions. Many other CERT rules have exceptions and this one would benefit from excluding methods that return a reference to the calling object or certain types of return values that are for convenience for the programmer like capturing a value before overwriting it.

The next highest rule violation detected was ERR08-J which cautions developers against catching a NullPointerException or any of its ancestors [20]. This type of exception is thrown when an application is running and attempts to dereference a pointer that has not been initialized to a value. According to CERT, when this type of runtime
error is ignored the application becomes unstable. Rather than catching the exceptions, CERT advises that the application terminate immediately. The rule also states that the ancestors `RuntimeException`, `Exception` and `Throwable` should never be caught since catching one of these could implicitly catch the `NullPointerException`. The Secure Coding Assistant identified 740 violations of this rule which accounted for 17.7% of the rule violations detected, but a majority of those were an ancestor and not the `NullPointerException`. Quite frequently developers will catch an exception as a generic `Exception` rather than the more specific type of exception that is being thrown (e.g. `IOException`). Unfortunately, this means that the tool cannot accurately detect this violation when developers catch generic exceptions. Of the remaining rules, 1.2% fell into the IDS category which accounts for data sanitization and helps prevent SQL injection attacks and 4.1% were in other categories.

Since EXP00-J and ERR08-J both contain a large number of exceptional cases, they have been excluded from the false positive study. To identify the false positive results, each secure coding alert was visually inspected only categorized as a “true positive” if the code segment was a true reflection of the secure coding rule outlined by CERT; all others are classified as a “false positive.” The results of this study in Table 6 reflect an overall false positive rate of 8.6% which are isolated to the IDS00-J, IDS11-J and MSC02-J CERT rules.
<table>
<thead>
<tr>
<th>Level</th>
<th>Full Name</th>
<th>Total</th>
<th>Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>L2</td>
<td>EXP00-J. Do not ignore values returned by methods</td>
<td>3,211</td>
<td>75.6%</td>
</tr>
<tr>
<td>L1</td>
<td>ERR08-J. Do not catch NullPointerException or any of its ancestors</td>
<td>740</td>
<td>17.4%</td>
</tr>
<tr>
<td>L2</td>
<td>MET04-J. Do not increase the accessibility of overridden or hidden methods</td>
<td>138</td>
<td>3.2%</td>
</tr>
<tr>
<td>L1</td>
<td>IDS00-J. Prevent SQL injection</td>
<td>116</td>
<td>2.7%</td>
</tr>
<tr>
<td>L1</td>
<td>MET06-J. Do not invoke overridable methods in clone()</td>
<td>25</td>
<td>0.6%</td>
</tr>
<tr>
<td>L1</td>
<td>IDS11-J. Perform any string modifications before validation</td>
<td>7</td>
<td>0.2%</td>
</tr>
<tr>
<td>L1</td>
<td>MSC02-J. Generate strong random numbers</td>
<td>7</td>
<td>0.2%</td>
</tr>
<tr>
<td>L1</td>
<td>IDS07-J. Sanitize untrusted data passed to the Runtime.exec() method</td>
<td>2</td>
<td>0.0%</td>
</tr>
<tr>
<td>L1</td>
<td>IDS01-J. Normalize strings before validating them</td>
<td>0</td>
<td>0.0%</td>
</tr>
<tr>
<td>L1</td>
<td>FIO08-J. Distinguish between characters or bytes read from a stream and -1</td>
<td>0</td>
<td>0.0%</td>
</tr>
<tr>
<td>L1</td>
<td>SEC07-J. Call the superclass's getPermissions() method when writing a custom class loader</td>
<td>0</td>
<td>0.0%</td>
</tr>
<tr>
<td>L1</td>
<td>SER01-J. Do not deviate from the proper signatures of serialization methods</td>
<td>0</td>
<td>0.0%</td>
</tr>
<tr>
<td>L1</td>
<td>STR00-J. Don't form strings containing partial characters from variable-width encodings</td>
<td>0</td>
<td>0.0%</td>
</tr>
<tr>
<td>L2</td>
<td>ENV02-J. Do not trust the values of environment variables</td>
<td>0</td>
<td>0.0%</td>
</tr>
<tr>
<td>L2</td>
<td>EXP02-J. Do not use the Object.equals() method to compare two arrays</td>
<td>0</td>
<td>0.0%</td>
</tr>
<tr>
<td>L2</td>
<td>NUM09-J. Do not use floating-point variables as loop counters</td>
<td>0</td>
<td>0.0%</td>
</tr>
<tr>
<td>L2</td>
<td>OBJ09-J. Compare classes and not class names</td>
<td>0</td>
<td>0.0%</td>
</tr>
<tr>
<td>L3</td>
<td>DCL02-J. Do not modify the collection's elements during an enhanced for statement</td>
<td>0</td>
<td>0.0%</td>
</tr>
<tr>
<td>L3</td>
<td>LCK09-J. Do not perform operations that can block while holding a lock</td>
<td>0</td>
<td>0.0%</td>
</tr>
<tr>
<td>L3</td>
<td>NUM07-J. Do not attempt comparisons with NaN</td>
<td>0</td>
<td>0.0%</td>
</tr>
<tr>
<td>L3</td>
<td>THI05-J. Do not use Thread.stop() to terminate threads</td>
<td>0</td>
<td>0.0%</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>4,206</td>
<td></td>
</tr>
</tbody>
</table>
Table 6. False positive analysis.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Total Count</th>
<th>True Positive Count</th>
<th>True Positive Percent</th>
<th>False Positive Count</th>
<th>False Positive Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>MET04-J</td>
<td>138</td>
<td>138</td>
<td>100.0%</td>
<td>0</td>
<td>0.0%</td>
</tr>
<tr>
<td>IDS00-J</td>
<td>42</td>
<td>29</td>
<td>69.0%</td>
<td>13</td>
<td>31.0%</td>
</tr>
<tr>
<td>MET06-J</td>
<td>25</td>
<td>25</td>
<td>100.0%</td>
<td>0</td>
<td>0.0%</td>
</tr>
<tr>
<td>IDS11-J</td>
<td>7</td>
<td>5</td>
<td>71.4%</td>
<td>2</td>
<td>28.6%</td>
</tr>
<tr>
<td>MSC02-J</td>
<td>7</td>
<td>3</td>
<td>42.9%</td>
<td>4</td>
<td>57.1%</td>
</tr>
<tr>
<td>IDS07-J</td>
<td>2</td>
<td>2</td>
<td>100.0%</td>
<td>0</td>
<td>0.0%</td>
</tr>
<tr>
<td>Total</td>
<td>221</td>
<td>202</td>
<td>91.4%</td>
<td>19</td>
<td>8.6%</td>
</tr>
</tbody>
</table>

The largest false positive result was 57.1% and found in detecting the MSC02-J rule. This rule states that a cryptographically secure random number generator should always be used in applications where security is important [21]. The false positive results logged were instances where the random number was being used for purposes besides encryption or security. Visual inspection showed the numbers were used for a randomly sorted list which not related to application security so they could not be counted as a true positive result. Fixing this issue with MSC02-J would be difficult since it requires knowledge of how the random number would be used. Adding a set of meta tags to the tool to allow programmers to disable security warnings for a line of code would solve this issue. For example, putting @SuppressSecurity:MSC02 before the line that generates the alert would cause that rule to be ignored when evaluating the following line for potential vulnerabilities.

The next highest false positive rate is seen in the IDS00-J rule detection which checks for correct usage of the PreparedStatement.setString() method [16]. All of the false positive results stemmed from query strings that did not require user
input. In these cases, the value being inserted into the query string was a constant value. Additional analysis on how the query string is built would be required to reduce the false positive rate for IDS00-J. This would include parsing the expression into subcomponents and tracing their origin in the source code. In cases where the input is coming from other services or modules this type of a trace would not be feasible.

The last notable result was the 28.6% false positive rate on the IDS11-J rule which recommends to “Perform any string modifications before validation” [22]. The method of detection for IDS11-J is to look for strings that are modified after they have been validated (using the Pattern.matcher() method), but in two instances in SecuriBench the strings were validated a second time after modification thus fixing the potential security vulnerability. A modification to the rule implementation that looks for additional validation after a string is modified would eliminate the IDS11-J false positives in SecuriBench.

6.2.3 False Negative Study

A false negative analysis of the Secure Coding Assistant requires segments of Java source code with known vulnerabilities. The SecuriBench programs have a large number of known vulnerabilities but a detailed listing of where they are in the source code does not exist. For this reason, a limited false negative analysis of the Secure Coding Assistant was performed by looking for examples of insecure Java code from organizations that document vulnerabilities like the Open Web Application Security
Project (OWASP) and Common Weakness Enumeration (CWE). The first test is the example shown on the OWASP website [23] for preventing SQL injection attacks in Java in Figure 10. The structure of the code is almost identical to that of the CERT examples so it was not a surprise that the tool detected the vulnerability without an issue, Figure 11.

```java
conn = pool.getConnection();
String sql = "select * from user where username='" + username +"' and password='" + password + "]";
stmt = conn.createStatement();
rs = stmt.executeQuery(sql);
if (rs.next()) {
    loggedIn = true;
    out.println("Successfully logged in");
} else {
    out.println("Username and/or password not recognized");
}
```

**Figure 10. SQL injection example from OWASP website [23].**

```
conn = pool.getConnection();
String sql = "select * from user where username='" + username +"' and password='" + password + "";
stmt = conn.createStatement();
rs = stmt.executeQuery(sql);
if (rs.next()) {
    loggedIn = true;
    System.out.println("Successfully logged in");
} else {
    System.out.println("Username and/or password not recognized");
}
```

**Figure 11. Output of SQL injection detection.**

Next, the CWE library was searched for code that would relate to the IDS01-J rule to normalize strings before validation. Figure 12 from the CWE Dictionary [24] is in the “Validate Before Canonicalize” section but is similar to the IDS01-J rule to validate
before normalizing a string [25]. In this example the path variable is being tested to see if it begins with /save_dir/ but there is no guarantee that the path name is in canonical form. To correct this code, the path string needs to be converted to canonical form before the comparison. Unfortunately, the violation went undetected by the Secure Coding Assistant. The key difference between the IDS01-J rule on the CERT website and the CWE example is that the CWE example includes canonicalization in the category of normalization but the CERT rule only gives the example of the normalize method. With a small modification to the rule detection logic canonicalization could be detected as well.

```java
String path = getInputPath();
if (path.startsWith("/safe_dir/"))
{
    File f = new File(path);
    return f.getCanonicalPath();
}
```

**Figure 12. Validate before canonicalize example from CWE [24].**

Another code segment from CWE is shown in Figure 12 which illustrates a vulnerability that should be detected under the CERT OBJ09-J rule. OBJ09-J states that class comparison should be done using the == operator on the class objects themselves and not the class names [26]. In the example given, changing the comparison line to obj.getClass() == this.getClass() would rectify the problem. In this example the Secure Coding Assistant successfully detected the vulnerability.
public class TrustedClass {
    @Override
    public boolean equals(Object obj) {
        boolean isEquals = false;
        // first check to see if the object is of the same class
        if (obj.getClass().getName().equals(
                this.getClass().getName())) {
            // then compare object fields
            if (...) { isEquals = true; }
        }
        return isEquals;
    }
}
Chapter 7
LIMITATIONS, CONCLUSION AND FUTURE WORK

The Secure Coding Assistant is a highly extensible early-detection static analysis tool that, unlike similar tools, is open source and will help developers learn and adhere to the CERT secure coding rules. Even though many of the CERT rules cannot be fully automated, about half of the Java rules can and about a quarter of those have been built into the Secure Coding Assistant. Future development work will focus on fine-tuning the existing rule detection logic, building logic for additional rule detection, expanding the tool to support additional programming languages and adding additional features.

The SecuriBench testing showed that some rules like EXP00-J need additional documentation on exception cases. The Secure Coding Assistant can help detect such cases and aid in fine-tuning the CERT rule library. The false positive and false negative study showed that there are several little adjustments that could be made to the rule logic to improve performance. There are also several rules that cannot be automated because the rule itself is context-specific. For instance, whether or not an application is running in a multi-threaded environment and requires thread safety or whether or not the Java application is interoperating with programs developed in other programming languages. These types of things cannot be identified through code inspection but a system of meta tags could be developed to indicate whether or not a block of code requires a certain type of specialized security.
The tool could also benefit from a few small improvements to the functional design. The markers, for instance, contain information from the CERT website [10] but they do not have a hyperlink back to the website itself. A precursory review of the marker structure found that customizing the marker text to have hyperlinks would be possible but requires a fair amount of additional design and implementation work. Also some additional controls to the user interface to control the scanning of the source code files would be useful. For instance, the plugin is always scanning the open file in the background and a programmer may want to scan an entire workspace at once, pause it and monitor the status. Lastly, there are instances where a programmer may see an alert and not agree that it is a security concern as was the case in the many EXP00-J and MSC02-J alerts in the SecuriBench test. In these cases, it would be very helpful for the programmer to have a way to indicate that they would like to ignore a particular rule in a block of code.

There are many static analysis tools that are available to the programming community. Several of these are Eclipse plugins, a few of them provide early-detection techniques but none of them are open-source learning tools for the CERT secure coding rules. The Secure Coding Assistant provides the development community with an educational tool in secure coding practices. It is open source, extensible and will be maintained. For more detailed information, please visit the project website at http://benw408701.github.io/SecureCodingAssistant/.
## CERT Rule Selection

<table>
<thead>
<tr>
<th>Section</th>
<th>Full Name</th>
<th>Can Automate</th>
<th>Automated</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule 00. Input Validation and Data Sanitization (IDS)</td>
<td>IDS00-J. Prevent SQL injection</td>
<td>Yes</td>
<td>Yes</td>
<td>L1</td>
</tr>
<tr>
<td>Rule 00. Input Validation and Data Sanitization (IDS)</td>
<td>IDS01-J. Normalize strings before validating them</td>
<td>Yes</td>
<td>Yes</td>
<td>L1</td>
</tr>
<tr>
<td>Rule 00. Input Validation and Data Sanitization (IDS)</td>
<td>IDS02-J. Canonicalize path names before validating them</td>
<td>Depricated: Moved to FIO16-J.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rule 00. Input Validation and Data Sanitization (IDS)</td>
<td>IDS03-J. Do not log unsanitized user input</td>
<td>Yes</td>
<td>No</td>
<td>L2</td>
</tr>
<tr>
<td>Rule 00. Input Validation and Data Sanitization (IDS)</td>
<td>IDS04-J. Safely extract files from ZipInputStream</td>
<td>Unknown</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 00. Input Validation and Data Sanitization (IDS)</td>
<td>IDS05-J. Use a safe subset of ASCII for file and path names</td>
<td>Depricated: Moved to FIO99-J.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rule 00. Input Validation and Data Sanitization (IDS)</td>
<td>IDS06-J. Exclude unsanitized user input from format strings</td>
<td>Yes</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 00. Input Validation and Data Sanitization (IDS)</td>
<td>IDS07-J. Sanitize untrusted data passed to the Runtime.exec() method</td>
<td>Yes</td>
<td>Yes</td>
<td>L1</td>
</tr>
<tr>
<td>Rule 00. Input Validation and Data Sanitization (IDS)</td>
<td>IDS08-J. Sanitize untrusted data included in a regular expression</td>
<td>Yes</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 00. Input Validation and Data Sanitization (IDS)</td>
<td>IDS09-J. Specify an appropriate locale when comparing locale-dependent data</td>
<td>Depricated: Moved to STR02-J.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rule 00. Input Validation and Data Sanitization (IDS)</td>
<td>IDS10-J. Don’t form strings containing partial characters</td>
<td>Depricated: Moved to STR01-J.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rule 00. Input Validation and Data Sanitization (IDS)</td>
<td>IDS11-J. Perform any string modifications before validation</td>
<td>Yes</td>
<td>Yes</td>
<td>L1</td>
</tr>
<tr>
<td>Rule 00. Input Validation and Data Sanitization (IDS)</td>
<td></td>
<td>Removed from CERT Website</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rule 00. Input Validation and Data Sanitization (IDS)</td>
<td>IDS13-J. Use compatible character encodings on both sides of file or network IO</td>
<td>Depricated: Moved to STR04-J.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------------------------------------------------</td>
<td>--------------------------------------------------------------------------------</td>
<td>-------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rule 00. Input Validation and Data Sanitization (IDS)</td>
<td>IDS14-J. Do not trust the contents of hidden form fields</td>
<td>Yes</td>
<td>No</td>
<td>L2</td>
</tr>
<tr>
<td>Rule 00. Input Validation and Data Sanitization (IDS)</td>
<td>IDS15-J. Do not allow sensitive information to leak outside a trust boundary</td>
<td></td>
<td>Under Construction</td>
<td></td>
</tr>
<tr>
<td>Rule 00. Input Validation and Data Sanitization (IDS)</td>
<td>IDS16-J. Prevent XML Injection</td>
<td>Yes</td>
<td>No</td>
<td>L1</td>
</tr>
<tr>
<td>Rule 00. Input Validation and Data Sanitization (IDS)</td>
<td>IDS17-J. Prevent XML External Entity Attacks</td>
<td>Yes</td>
<td>No</td>
<td>L2</td>
</tr>
<tr>
<td>Rule 01. Declarations and Initialization (DCL)</td>
<td>DCL00-J. Prevent class initialization cycles</td>
<td>Unknown</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 01. Declarations and Initialization (DCL)</td>
<td>DCL01-J. Do not use public identifiers from the Java Standard Library</td>
<td>Yes</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 01. Declarations and Initialization (DCL)</td>
<td>DCL02-J. Do not modify the collection’s elements during an enhanced for statement</td>
<td>Yes</td>
<td>Yes</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 02. Expressions (EXP)</td>
<td>EXP00-J. Do not ignore values returned by methods</td>
<td>Yes</td>
<td>Yes</td>
<td>L2</td>
</tr>
<tr>
<td>Rule 02. Expressions (EXP)</td>
<td>EXP01-J. Do not use a null in a case where an object is required</td>
<td>No</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 02. Expressions (EXP)</td>
<td>EXP02-J. Do not use the Object.equals() method to compare two arrays</td>
<td>Yes</td>
<td>Yes</td>
<td>L2</td>
</tr>
<tr>
<td>Rule 02. Expressions (EXP)</td>
<td>EXP03-J. Do not use the equality operators when comparing values of boxed primitives</td>
<td>Yes</td>
<td>No</td>
<td>L2</td>
</tr>
<tr>
<td>Rule 02. Expressions (EXP)</td>
<td>EXP04-J. Do not pass arguments to certain Java Collections Framework methods that are a different type than the collection parameter type</td>
<td>Yes</td>
<td>No</td>
<td>L2</td>
</tr>
<tr>
<td>Rule 02. Expressions (EXP)</td>
<td>EXP05-J. Do not follow a write by a subsequent write or read of the same object within an expression</td>
<td>Yes</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 02. Expressions (EXP)</td>
<td>EXP06-J. Expressions used in assertions must not produce side effects</td>
<td>Yes</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 02. Expressions (EXP)</td>
<td>EXP07-J. Prevent loss of useful data due to weak references</td>
<td>Under Construction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rule 03. Numeric Types and Operations (NUM)</td>
<td>NUM00-J. Detect or prevent integer overflow</td>
<td>Yes</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 03. Numeric Types and Operations (NUM)</td>
<td>NUM01-J. Do not perform bitwise and arithmetic operations on the same data</td>
<td>Unknown</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 03. Numeric Types and Operations (NUM)</td>
<td>NUM02-J. Ensure that division and remainder operations do not result in divide-by-zero errors</td>
<td>Yes</td>
<td>No</td>
<td>L2</td>
</tr>
<tr>
<td>Rule 03. Numeric Types and Operations (NUM)</td>
<td>NUM03-J. Use integer types that can fully represent the possible range of unsigned data</td>
<td>No</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 03. Numeric Types and Operations (NUM)</td>
<td>NUM04-J. Do not use floating-point numbers if precise computation is required</td>
<td>No</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 03. Numeric Types and Operations (NUM)</td>
<td>-</td>
<td>Removed from CERT Website</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rule 03. Numeric Types and Operations (NUM)</td>
<td>-</td>
<td>Removed from CERT Website</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rule 03. Numeric Types and Operations (NUM)</td>
<td>NUM07-J. Do not attempt comparisons with NaN</td>
<td>Yes</td>
<td>Yes</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 03. Numeric Types and Operations (NUM)</td>
<td>NUM08-J. Check floating-point inputs for exceptional values</td>
<td>No</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 03. Numeric Types and Operations (NUM)</td>
<td>NUM09-J. Do not use floating-point variables as loop counters</td>
<td>Yes</td>
<td>Yes</td>
<td>L2</td>
</tr>
<tr>
<td>Rule 03. Numeric Types and Operations (NUM)</td>
<td>NUM10-J. Do not construct BigDecimal objects from floating-point literals</td>
<td>Yes</td>
<td>No</td>
<td>L2</td>
</tr>
<tr>
<td>Rule 03. Numeric Types and Operations (NUM)</td>
<td>NUM11-J. Do not compare or inspect the string representation of floating-point values</td>
<td>Unknown</td>
<td>No</td>
<td>L2</td>
</tr>
<tr>
<td>Rule 03. Numeric Types and Operations (NUM)</td>
<td>NUM12. Ensure conversions of numeric types to narrower types do not result in lost or misinterpreted data</td>
<td>Yes</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 03. Numeric Types and Operations (NUM)</td>
<td>NUM13. Avoid loss of precision when converting primitive integers to floating-point</td>
<td>Yes</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 03. Numeric Types and Operations (NUM)</td>
<td>NUM14. Use shift operators correctly</td>
<td>Yes</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 04. Characters and Strings (STR)</td>
<td>STR00. Don't form strings containing partial characters from variable-width encodings</td>
<td>Yes</td>
<td>Yes</td>
<td>L1</td>
</tr>
<tr>
<td>Rule 04. Characters and Strings (STR)</td>
<td>STR01. Do not assume that a Java char fully represents a Unicode code point</td>
<td>Yes</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 04. Characters and Strings (STR)</td>
<td>STR02. Specify an appropriate locale when comparing locale-dependent data</td>
<td>Yes</td>
<td>No</td>
<td>L2</td>
</tr>
<tr>
<td>Rule 04. Characters and Strings (STR)</td>
<td>STR03. Do not encode noncharacter data as a string</td>
<td>Unknown</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 04. Characters and Strings (STR)</td>
<td>STR04. Use compatible character encodings when communicating string data between JVMs</td>
<td>No</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 05. Object Orientation (OBJ)</td>
<td>-</td>
<td>Removed from CERT Website</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rule 05. Object Orientation (OBJ)</td>
<td>OBJ01. Limit accessibility of fields</td>
<td>No</td>
<td>No</td>
<td>L1</td>
</tr>
<tr>
<td>Rule 05. Object Orientation (OBJ)</td>
<td>OBJ02. Preserve dependencies in subclasses when changing superclasses</td>
<td>No</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 05. Object Orientation (OBJ)</td>
<td>OBJ03. Prevent heap pollution</td>
<td>Unknown</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 05. Object Orientation (OBJ)</td>
<td>OBJ04. Provide mutable classes with copy functionality to safely allow passing instances to untrusted code</td>
<td>Yes</td>
<td>No</td>
<td>L2</td>
</tr>
<tr>
<td>Rule 05. Object Orientation (OBJ)</td>
<td>OBJ05. Do not return references to private mutable class members</td>
<td>No</td>
<td>No</td>
<td>L1</td>
</tr>
<tr>
<td>Rule 05. Object Orientation (OBJ)</td>
<td>OBJ06-J. Defensively copy mutable inputs and mutable internal components</td>
<td>Unknown</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 05. Object Orientation (OBJ)</td>
<td>OBJ07-J. Sensitive classes must not let themselves be copied</td>
<td>Unknown</td>
<td>No</td>
<td>L2</td>
</tr>
<tr>
<td>Rule 05. Object Orientation (OBJ)</td>
<td>OBJ08-J. Do not expose private members of an outer class from within a nested class</td>
<td>Yes</td>
<td>No</td>
<td>L2</td>
</tr>
<tr>
<td>Rule 05. Object Orientation (OBJ)</td>
<td>OBJ09-J. Compare classes and not class names</td>
<td>Yes</td>
<td>Yes</td>
<td>L2</td>
</tr>
<tr>
<td>Rule 05. Object Orientation (OBJ)</td>
<td>OBJ10-J. Do not use public static nonfinal fields</td>
<td>Yes</td>
<td>No</td>
<td>L2</td>
</tr>
<tr>
<td>Rule 05. Object Orientation (OBJ)</td>
<td>OBJ11-J. Be wary of letting constructors throw exceptions</td>
<td>No</td>
<td>No</td>
<td>L1</td>
</tr>
<tr>
<td>Rule 05. Object Orientation (OBJ)</td>
<td>OBJ12-J. Respect object-based annotations</td>
<td>Under Construction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rule 05. Object Orientation (OBJ)</td>
<td>OBJ13-J. Ensure that references to mutable objects are not exposed</td>
<td>Unknown</td>
<td>No</td>
<td>L1</td>
</tr>
<tr>
<td>Rule 06. Methods (MET)</td>
<td>MET00-J. Validate method arguments</td>
<td>Unknown</td>
<td>No</td>
<td>L2</td>
</tr>
<tr>
<td>Rule 06. Methods (MET)</td>
<td>MET01-J. Never use assertions to validate method arguments</td>
<td>Unknown</td>
<td>No</td>
<td>L2</td>
</tr>
<tr>
<td>Rule 06. Methods (MET)</td>
<td>MET02-J. Do not use deprecated or obsolete classes or methods</td>
<td>Yes</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 06. Methods (MET)</td>
<td>MET03-J. Methods that perform a security check must be declared private or final</td>
<td>Yes</td>
<td>No</td>
<td>L2</td>
</tr>
<tr>
<td>Rule 06. Methods (MET)</td>
<td>MET04-J. Do not increase the accessibility of overridden or hidden methods</td>
<td>Yes</td>
<td>Yes</td>
<td>L2</td>
</tr>
<tr>
<td>Rule 06. Methods (MET)</td>
<td>MET05-J. Ensure that constructors do not call overridable methods</td>
<td>Yes</td>
<td>No</td>
<td>L2</td>
</tr>
<tr>
<td>Rule 06. Methods (MET)</td>
<td>MET06-J. Do not invoke overridable methods in clone()</td>
<td>Yes</td>
<td>Yes</td>
<td>L1</td>
</tr>
<tr>
<td>Rule 06. Methods (MET)</td>
<td>MET07: J. Never declare a class method that hides a method declared in a superclass or superinterface</td>
<td>Yes</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>------------------------</td>
<td>--------------------------------------------------------------------------------------------------</td>
<td>-----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>Rule 06. Methods (MET)</td>
<td>MET08: J. Preserve the equality contract when overriding the equals() method</td>
<td>Unknown</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 06. Methods (MET)</td>
<td>MET09: J. Classes that define an equals() method must also define a hashCode() method</td>
<td>Yes</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 06. Methods (MET)</td>
<td>MET10: J. Follow the general contract when implementing the compareTo() method</td>
<td>Unknown</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 06. Methods (MET)</td>
<td>MET11: J. Ensure that keys used in comparison operations are immutable</td>
<td>Yes</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 06. Methods (MET)</td>
<td>MET12: J. Do not use finalizers</td>
<td>Yes</td>
<td>No</td>
<td>L2</td>
</tr>
<tr>
<td>Rule 07. Exceptional Behavior (ERR)</td>
<td>ERR00: J. Do not suppress or ignore checked exceptions</td>
<td>Unknown</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 07. Exceptional Behavior (ERR)</td>
<td>ERR01: J. Do not allow exceptions to expose sensitive information</td>
<td>Unknown</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 07. Exceptional Behavior (ERR)</td>
<td>ERR02: J. Prevent exceptions while logging data</td>
<td>Yes</td>
<td>No</td>
<td>L2</td>
</tr>
<tr>
<td>Rule 07. Exceptional Behavior (ERR)</td>
<td>ERR03: J. Restore prior object state on method failure</td>
<td>Unknown</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 07. Exceptional Behavior (ERR)</td>
<td>ERR04: J. Do not complete abruptly from a finally block</td>
<td>Yes</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 07. Exceptional Behavior (ERR)</td>
<td>ERR05: J. Do not let checked exceptions escape from a finally block</td>
<td>Yes</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 07. Exceptional Behavior (ERR)</td>
<td>ERR06: J. Do not throw undeclared checked exceptions</td>
<td>Unknown</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 07. Exceptional Behavior (ERR)</td>
<td>ERR07: J. Do not throw RuntimeException, Exception, or Throwable</td>
<td>Yes</td>
<td>No</td>
<td>L2</td>
</tr>
<tr>
<td>Rule 07. Exceptional Behavior (ERR)</td>
<td>ERR08: J. Do not catch NullPointerException or any of its ancestors</td>
<td>Yes</td>
<td>Yes</td>
<td>L1</td>
</tr>
<tr>
<td>Rule 07. Exceptional Behavior (ERR)</td>
<td>ERR09-J. Do not allow untrusted code to terminate the JVM</td>
<td>Yes</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 08. Visibility and Atomicity (VNA)</td>
<td>VNA00-J. Ensure visibility when accessing shared primitive variables</td>
<td>Yes</td>
<td>No</td>
<td>L2</td>
</tr>
<tr>
<td>Rule 08. Visibility and Atomicity (VNA)</td>
<td>VNA01-J. Ensure visibility of shared references to immutable objects</td>
<td>Yes</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 08. Visibility and Atomicity (VNA)</td>
<td>VNA02-J. Ensure that compound operations on shared variables are atomic</td>
<td>Yes</td>
<td>No</td>
<td>L2</td>
</tr>
<tr>
<td>Rule 08. Visibility and Atomicity (VNA)</td>
<td>VNA03-J. Do not assume that a group of calls to independently atomic methods is atomic</td>
<td>Yes</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 08. Visibility and Atomicity (VNA)</td>
<td>VNA04-J. Ensure that calls to chained methods are atomic</td>
<td>Unknown</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 08. Visibility and Atomicity (VNA)</td>
<td>VNA05-J. Ensure atomicity when reading and writing 64-bit values</td>
<td>Yes</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 08. Visibility and Atomicity (VNA)</td>
<td>Removed from CERT Website</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rule 09. Locking (LCK)</td>
<td>LCK00-J. Use private final lock objects to synchronize classes that may interact with untrusted code</td>
<td>Unknown</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 09. Locking (LCK)</td>
<td>LCK01-J. Do not synchronize on objects that may be reused</td>
<td>Yes</td>
<td>No</td>
<td>L2</td>
</tr>
<tr>
<td>Rule 09. Locking (LCK)</td>
<td>LCK02-J. Do not synchronize on the class object returned by getClass()</td>
<td>Unknown</td>
<td>No</td>
<td>L2</td>
</tr>
<tr>
<td>Rule 09. Locking (LCK)</td>
<td>LCK03-J. Do not synchronize on the intrinsic locks of high-level concurrency objects</td>
<td>Unknown</td>
<td>No</td>
<td>L2</td>
</tr>
<tr>
<td>Rule 09. Locking (LCK)</td>
<td>LCK04-J. Do not synchronize on a collection view if the backing collection is accessible</td>
<td>Yes</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 09. Locking (LCK)</td>
<td>LCK05-J. Synchronize access to static fields that can be modified by untrusted code</td>
<td>Unknown</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 09. Locking (LCK)</td>
<td>LCK06-J. Do not use an instance lock to protect shared static data</td>
<td>Yes</td>
<td>No</td>
<td>L2</td>
</tr>
<tr>
<td>Rule 09. Locking (LCK)</td>
<td>LCK07-J. Avoid deadlock by requesting and releasing locks in the same order</td>
<td>Yes</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 09. Locking (LCK)</td>
<td>LCK08-J. Ensure actively held locks are released on exceptional conditions</td>
<td>Yes</td>
<td>No</td>
<td>L2</td>
</tr>
<tr>
<td>Rule 09. Locking (LCK)</td>
<td>LCK09-J. Do not perform operations that can block while holding a lock</td>
<td>Yes</td>
<td>Yes</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 09. Locking (LCK)</td>
<td>LCK10-J. Use a correct form of the double-checked locking idiom</td>
<td>Yes</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 09. Locking (LCK)</td>
<td>LCK11-J. Avoid client-side locking when using classes that do not commit to their locking strategy</td>
<td>Unknown</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 10. Thread APIs (THI)</td>
<td>THI00-J. Do not invoke Thread.run()</td>
<td>Yes</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 10. Thread APIs (THI)</td>
<td>THI01-J. Do not invoke ThreadGroup methods</td>
<td>Unknown</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 10. Thread APIs (THI)</td>
<td>THI02-J. Notify all waiting threads rather than a single thread</td>
<td>Unknown</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 10. Thread APIs (THI)</td>
<td>THI03-J. Always invoke wait() and await() methods inside a loop</td>
<td>Yes</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 10. Thread APIs (THI)</td>
<td>THI04-J. Ensure that threads performing blocking operations can be terminated</td>
<td>Unknown</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 10. Thread APIs (THI)</td>
<td>THI05-J. Do not use Thread.stop() to terminate threads</td>
<td>Yes</td>
<td>Yes</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 11. Thread Pools (TPS)</td>
<td>TPS00-J. Use thread pools to enable graceful degradation of service during traffic bursts</td>
<td>Unknown</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 11. Thread Pools (TPS)</td>
<td>TPS01-J. Do not execute interdependent tasks in a bounded thread pool</td>
<td>Unknown</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 11. Thread Pools (TPS)</td>
<td>TPS02-J. Ensure that tasks submitted to a thread pool are interruptible</td>
<td>Unknown</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 11. Thread Pools (TPS)</td>
<td>TPS03-J. Ensure that tasks executing in a thread pool do not fail silently</td>
<td>Unknown</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 11. Thread Pools (TPS)</td>
<td>TPS04-J. Ensure ThreadLocal variables are reinitialized when using thread pools</td>
<td>Unknown</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 12. Thread-Safety Miscellaneous (TSM)</td>
<td>TSM00-J. Do not override thread-safe methods with methods that are not thread-safe</td>
<td>Unknown</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 12. Thread-Safety Miscellaneous (TSM)</td>
<td>TSM01-J. Do not let the this reference escape during object construction</td>
<td>Unknown</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 12. Thread-Safety Miscellaneous (TSM)</td>
<td>TSM02-J. Do not use background threads during class initialization</td>
<td>Unknown</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 12. Thread-Safety Miscellaneous (TSM)</td>
<td>TSM03-J. Do not publish partially initialized objects</td>
<td>Unknown</td>
<td>No</td>
<td>L2</td>
</tr>
<tr>
<td>Rule 13. Input Output (FIO)</td>
<td>FIO00-J. Do not operate on files in shared directories</td>
<td>Unknown</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 13. Input Output (FIO)</td>
<td>FIO01-J. Create files with appropriate access permissions</td>
<td>Unknown</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 13. Input Output (FIO)</td>
<td>FIO02-J. Detect and handle file-related errors</td>
<td>Unknown</td>
<td>No</td>
<td>L2</td>
</tr>
<tr>
<td>Rule 13. Input Output (FIO)</td>
<td>FIO03-J. Remove temporary files before termination</td>
<td>Unknown</td>
<td>No</td>
<td>L2</td>
</tr>
<tr>
<td>Rule 13. Input Output (FIO)</td>
<td>FIO04-J. Release resources when they are no longer needed</td>
<td>Yes</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 13. Input Output (FIO)</td>
<td>FIO05-J. Do not expose buffers created using the wrap() or duplicate() methods to untrusted code</td>
<td>No</td>
<td>No</td>
<td>L1</td>
</tr>
<tr>
<td>Rule 13. Input Output (FIO)</td>
<td>FIO06-J. Do not create multiple buffered wrappers on a single byte or character stream</td>
<td>No</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 13. Input Output (FIO)</td>
<td>FIO07-J. Do not let external processes block on IO buffers</td>
<td>Unknown</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 13. Input Output (FIO)</td>
<td>FIO08-J. Distinguish between characters or bytes read from a stream and -1</td>
<td>Yes</td>
<td>Yes</td>
<td>L1</td>
</tr>
<tr>
<td>Rule 13. Input Output (FIO)</td>
<td>FIO09-J. Do not rely on the write() method to output integers outside the range 0 to 255</td>
<td>Yes</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 13. Input Output (FIO)</td>
<td>FIO10-J. Ensure the array is filled when using read() to fill an array</td>
<td>Unknown</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>Rule 13. Input Output (FIO)</td>
<td>FIO11-J. Do not convert between strings and bytes without specifying a valid character encoding</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rule 13. Input Output (FIO)</td>
<td>Deprecation: Moved to STR03-J.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rule 13. Input Output (FIO)</td>
<td>FIO12-J. Provide methods to read and write little-endian data</td>
<td>No</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 13. Input Output (FIO)</td>
<td>FIO13-J. Do not log sensitive information outside a trust boundary</td>
<td>Unknown</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 13. Input Output (FIO)</td>
<td>FIO14-J. Perform proper cleanup at program termination</td>
<td>Unknown</td>
<td>No</td>
<td>L1</td>
</tr>
<tr>
<td>Rule 13. Input Output (FIO)</td>
<td>FIO15-J. Do not reset a servlet's output stream after committing it</td>
<td>Yes</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 13. Input Output (FIO)</td>
<td>FIO16-J. Canonicalize path names before validating them</td>
<td>Yes</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 14. Serialization (SER)</td>
<td>SER00-J. Enable serialization compatibility during class evolution</td>
<td>Yes</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 14. Serialization (SER)</td>
<td>SER01-J. Do not deviate from the proper signatures of serialization methods</td>
<td>Yes</td>
<td>Yes</td>
<td>L1</td>
</tr>
<tr>
<td>Rule 14. Serialization (SER)</td>
<td>SER02-J. Sign then seal sensitive objects before sending them outside a trust boundary</td>
<td>No</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 14. Serialization (SER)</td>
<td>SER03-J. Do not serialize unencrypted sensitive data</td>
<td>Yes</td>
<td>No</td>
<td>L2</td>
</tr>
<tr>
<td>Rule 14. Serialization (SER)</td>
<td>SER04-J. Do not allow serialization and deserialization to bypass the security manager</td>
<td>Unknown</td>
<td>No</td>
<td>L2</td>
</tr>
<tr>
<td>Rule 14. Serialization (SER)</td>
<td>SER05-J. Do not serialize instances of inner classes</td>
<td>Yes</td>
<td>No</td>
<td>L1</td>
</tr>
<tr>
<td>Rule 14. Serialization (SER)</td>
<td>SER06-J. Make defensive copies of private mutable components during deserialization</td>
<td>Yes</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 14. Serialization (SER)</td>
<td>SER07-J. Do not use the default serialized form for classes with implementation-defined invariants</td>
<td>Yes</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>--------------------------------------------</td>
<td>---------------------------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SER08-J Minimize privileges before deserializing from a privileged context</td>
<td>SEC00-J Do not allow privileged blocks to leak sensitive information across a trust boundary</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unknown No L1</td>
<td>No No L2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SER09-J Do not invoke overridable methods from the readObject() method</td>
<td>SEC01-J Do not allow tainted variables in privileged blocks</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unknown No L3</td>
<td>No No L1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SER10-J Avoid memory and resource leaks during serialization</td>
<td>SEC02-J Do not base security checks on untrusted sources</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unknown No L3</td>
<td>Yes No L1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SER11-J Prevent overwriting of externalizable objects</td>
<td>SEC03-J Do not load trusted classes after allowing untrusted code to load arbitrary classes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unknown No L2</td>
<td>Unknown No L1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SEC04-J Protect sensitive operations with security manager checks</td>
<td>SEC04-J Protect sensitive operations with security manager checks</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unknown No L1</td>
<td>No No L1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SEC05-J Do not use reflection to increase accessibility of classes, methods, or fields</td>
<td>SEC05-J Do not use reflection to increase accessibility of classes, methods, or fields</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unknown No L1</td>
<td>Unknown No L1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SEC06-J Do not rely on the default automatic signature verification provided by URLClassLoader and java.util.jar</td>
<td>SEC06-J Do not rely on the default automatic signature verification provided by URLClassLoader and java.util.jar</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No No L1</td>
<td>No No L1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SEC07-J Call the superclass's getPermissions() method when writing a custom class loader</td>
<td>SEC07-J Call the superclass's getPermissions() method when writing a custom class loader</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Yes Yes L1</td>
<td>Yes Yes L1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SEC08-J Trusted code must discard or clean any arguments provided by untrusted code</td>
<td>SEC08-J Trusted code must discard or clean any arguments provided by untrusted code</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Under Construction</td>
<td>Under Construction</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rule 15. Platform Security (SEC)</td>
<td>SEC09-J Never leak the results of certain standard API methods from trusted code to untrusted code</td>
<td>Under Construction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rule 15. Platform Security (SEC)</td>
<td>SEC10-J Never permit untrusted code to invoke any API that may (possibly transitively) invoke the reflection APIs</td>
<td>Under Construction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rule 16. Runtime Environment (ENV)</td>
<td>ENV00-J Do not sign code that performs only unprivileged operations</td>
<td>No</td>
<td>No</td>
<td>L1</td>
</tr>
<tr>
<td>Rule 16. Runtime Environment (ENV)</td>
<td>ENV01-J Place all security-sensitive code in a single JAR and sign and seal it</td>
<td>No</td>
<td>No</td>
<td>L1</td>
</tr>
<tr>
<td>Rule 16. Runtime Environment (ENV)</td>
<td>ENV02-J Do not trust the values of environment variables</td>
<td>Yes</td>
<td>Yes</td>
<td>L2</td>
</tr>
<tr>
<td>Rule 16. Runtime Environment (ENV)</td>
<td>ENV03-J Do not grant dangerous combinations of permissions</td>
<td>Yes</td>
<td>No</td>
<td>L1</td>
</tr>
<tr>
<td>Rule 16. Runtime Environment (ENV)</td>
<td>ENV04-J Do not disable bytecode verification</td>
<td>No</td>
<td>No</td>
<td>L1</td>
</tr>
<tr>
<td>Rule 16. Runtime Environment (ENV)</td>
<td>ENV05-J Do not deploy an application that can be remotely monitored</td>
<td>No</td>
<td>No</td>
<td>L1</td>
</tr>
<tr>
<td>Rule 16. Runtime Environment (ENV)</td>
<td>ENV06-J Production code must not contain debugging entry points</td>
<td>No</td>
<td>No</td>
<td>L1</td>
</tr>
<tr>
<td>Rule 17. Java Native Environment (JNI)</td>
<td>JNI00-J Define wrappers around native methods</td>
<td>No</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 17. Java Native Environment (JNI)</td>
<td>JNI01-J Safely invoke standard APIs that perform tasks using the immediate caller’s class loader instance (loadLibrary)</td>
<td>Under Construction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rule 17. Java Native Environment (JNI)</td>
<td>JNI02-J Do not assume object references are constant or unique</td>
<td>Under Construction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rule 17. Java Native Environment (JNI)</td>
<td>JNI03-J Do not use direct pointers to Java objects in JNI code</td>
<td>Under Construction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rule 17. Java Native Environment (JNI)</td>
<td>JNI04-J Do not assume that Java strings are null-terminated</td>
<td>Under Construction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rule 49. Miscellaneous (MSC)</td>
<td>MSC00-J Use SSLSocket rather than Socket for secure data exchange</td>
<td>No</td>
<td>No</td>
<td>L2</td>
</tr>
<tr>
<td>Rule 49. Miscellaneous (MSC)</td>
<td>MSC01-J. Do not use an empty infinite loop</td>
<td>Yes</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 49. Miscellaneous (MSC)</td>
<td>MSC02-J. Generate strong random numbers</td>
<td>Yes</td>
<td>Yes</td>
<td>L1</td>
</tr>
<tr>
<td>Rule 49. Miscellaneous (MSC)</td>
<td>MSC03-J. Never hard code sensitive information</td>
<td>Yes</td>
<td>No</td>
<td>L1</td>
</tr>
<tr>
<td>Rule 49. Miscellaneous (MSC)</td>
<td>MSC04-J. Do not leak memory</td>
<td>Unknown</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 49. Miscellaneous (MSC)</td>
<td>MSC05-J. Do not exhaust heap space</td>
<td>Unknown</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 49. Miscellaneous (MSC)</td>
<td>MSC06-J. Do not modify the underlying collection when an iteration is in progress</td>
<td>Yes</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 49. Miscellaneous (MSC)</td>
<td>MSC07-J. Prevent multiple instantiations of singleton objects</td>
<td>Yes</td>
<td>No</td>
<td>L3</td>
</tr>
<tr>
<td>Rule 49. Miscellaneous (MSC)</td>
<td>MSC08-J. Do not store nonserializable objects as attributes in an HTTP session</td>
<td>Under Construction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rule 49. Miscellaneous (MSC)</td>
<td>MSC09-J. For OAuth, ensure (a) [relying party receiving user’s ID in last step] is same as (b) [relying party the access token was granted to]</td>
<td>Under Construction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rule 49. Miscellaneous (MSC)</td>
<td>MSC10-J. Do not use OAuth 2.0 implicit grant (unmodified) for authentication</td>
<td>Under Construction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rule 49. Miscellaneous (MSC)</td>
<td>MSC11-J. Do not let session information leak within a servlet</td>
<td>Yes</td>
<td>No</td>
<td>L2</td>
</tr>
</tbody>
</table>
APPENDIX B

Application Source Code

Plugin.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<plugin>
  <extension
    id="edu.csus.plugin.securecodingassistant.SecureCompilationParticipants"
    name="Secure Compilation Participants"
    point="org.eclipse.jdt.core.compilationParticipant">
    <compilationParticipant
      class="edu.csus.plugin.securecodingassistant.compilation.SecureCompilationParticipant"
      createsProblems="false"
      id="edu.csus.plugin.securecodingassistant.compilationParticipant"
      modifiesEnvironment="false">
    </compilationParticipant>
  </extension>
  <extension
    id="edu.csus.plugin.securecodingassistant.securecodingmarker"
    name="Secure Coding Marker"
    point="org.eclipse.core.resources.markers">
    <super
      type="org.eclipse.core.resources.problemmarker">
    </super>
    <persistent
      value="false">
    </persistent>
    <attribute
      name="edu.csus.plugin.securecodingassistant.securecodingmarker.violatedRule">
    </attribute>
    <attribute
      name="edu.csus.plugin.securecodingassistant.securecodingmarker.securityLevel">
    </attribute>
  </extension>
</plugin>
```

Globals.java

```java
package edu.csus.plugin.securecodingassistant;

/**
 * Global variables used in Secure Coding Assistant
 * @author Ben White
 */
public class Globals {

  /**
   * Marker-related global variables
   * @author Ben White
   */
```
public static class Markers {
    /**
     * Custom IMarker type
     */
    public static final String SECURE_MARKER = 
    "edu.csus.plugin.securecodingassistant.securecodingmarker";

    /**
     * Custom string attribute, the rule that was violated
     */
    public static final String VIOLATED_RULE = SECURE_MARKER + ".violatedRule";

    /**
     * Custom string attribute, the security level of the violated rule.
     * Possible values: {@link Globals.Markers#SECURITY_LEVEL_LOW},
     * {@link Globals.Markers#SECURITY_LEVEL_MEDIUM} or {@link 
     * Globals.Markers#SECURITY_LEVEL_HIGH}
     */
    public static final String SECURITY_LEVEL = SECURE_MARKER + ".securityLevel";

    /**
     * L3 according to CERT website
     */
    public static final int SECURITY_LEVEL_LOW = 1;

    /**
     * L2 according to CERT website
     */
    public static final int SECURITY_LEVEL_MEDIUM = 2;

    /**
     * L1 according to CERT website
     */
    public static final int SECURITY_LEVEL_HIGH = 3;
}

InsecureCodeSegment.java

package edu.csus.plugin.securecodingassistant.compilation;

import org.eclipse.core.resources.IMarker;
import org.eclipse.core.resources.IResource;
import org.eclipse.core.runtime.CoreException;
import org.eclipse.jdt.core.compiler.ReconcileContext;
import org.eclipse.jdt.core.dom.ASTNode;
import edu.csus.plugin.securecodingassistant.Globals;
import edu.csus.plugin.securecodingassistant.rules.IRule;

/**
 * Represents a segment of source code that is potentially insecure. Insecure code
 * segments are identified in {@link SecureCompilationParticipant} and added to a collection for
 * tracking. An <code>InsecureCodeSegment</code> keeps track of the rule that was
 * violated and also handles adding an {@link IMarker} in the development environment to alert the user.
 * @author Ben White
 * @see SecureCompilationParticipant
 * @see IRule
class InsecureCodeSegment {

/**
 * A marker in the development environment that alerts the programmer
 * that they have an insecure segment of code
 */
private IMarker m_marker;

/**
 * The resource where the marker is stored
 */
private IResource m_resource;

/**
 * Create new insecure code segment at given node that violates an {@link IRule}
 * @param node The AST node where the {@link IRule} was violated
 * @param rule The {@link IRule} that was violated
 * @param context The <code>ReconcileContext</code>, this is needed to create an {@link IMarker}
 * at the code location
 */
public InsecureCodeSegment(ASTNode node, IRule rule, ReconcileContext context) {
    int start, end, line;
    start = node.getStartPosition();
    end = start + node.getLength();
    String severity = "";
    switch (rule.securityLevel()) {
        case Globals.Markers.SECURITY_LEVEL_HIGH:
            severity = "High";
            break;
        case Globals.Markers.SECURITY_LEVEL_MEDIUM:
            severity = "Medium";
            break;
        case Globals.Markers.SECURITY_LEVEL_LOW:
            severity = "Low";
            break;
    }

    try {
        IResource resource = context.getWorkingCopy().getResource();
        m_resource = resource;
        line = context.getAST8().getLineNumber(start - 1);

        m_marker = resource.createMarker(Globals.Markers.SECURE_MARKER);
        m_marker.setAttribute(IMarker.MESSAGE,
            String.format("Rule violated: %s\nSeverity: %s\n" + "NOTE: The text and/or code below is from the " + "CERT website https://www.securecoding.cert.org\n" + "Rule description: %s\n" + "Rule Solution: %s\n", rule.getRuleName(), severity,
            rule.getRuleText(), rule.getRuleRecommendation()));

        m_marker.setAttribute(IMarker.SEVERITY, IMarker.SEVERITY_WARNING);
        m_marker.setAttribute(Globals.Markers.SECURITY_LEVEL, rule.securityLevel());
        m_marker.setAttribute(IMarker.LINE_NUMBER, line);
        m_marker.setAttribute(IMarker.CHAR_START, start);
        m_marker.setAttribute(IMarker.CHAR_END, end);
        m_marker.setAttribute(IMarker.LOCATION, String.format("line %d", line));
        m_marker.setAttribute(Globals.Markers.VIOLATED_RULE, rule.getRuleName());

        // Show high security levels as errors, lower as warnings
        if (rule.securityLevel() == Globals.Markers.SECURITY_LEVEL_HIGH)
### SecureCompilationParticipant.java

```java
package edu.csus.plugin.securecodingassistant.compilation;

import java.util.ArrayList;
import java.util.Iterator;
import org.eclipse.core.resources.IResource;
import org.eclipse.core.runtime.CoreException;
import org.eclipse.jdt.core.IJavaElementDelta;
import org.eclipse.jdt.core.IJavaProject;
import org.eclipse.jdt.core.JavaModelException;
import org.eclipse.jdt.core.compiler.CompilationParticipant;
import org.eclipse.jdt.core.compiler.ReconcileContext;
import org.eclipse.jdt.core.dom.ASTNode;
import edu.csus.plugin.securecodingassistant.rules.IRule;
import edu.csus.plugin.securecodingassistant.rules.RuleFactory;

public class SecureCompilationParticipant extends CompilationParticipant {

    /**
     * Collection of rules to be tested when reconcile(ReconcileContext) is called
     */
    private ArrayList<IRule> m_rules;
```
/**
 * Collection of insecure code segments that have been detected
 */
private ArrayList<InsecureCodeSegment> m_insecureCodeSegments;

/**
 * Creates new <code>SecureCompilationParticipant</code>
 */
public SecureCompilationParticipant() {
    super();
    m_rules = RuleFactory.getAllRules();
    m_insecureCodeSegments = new ArrayList<InsecureCodeSegment>();
}

/**
 * Always returns true
 * @param project the <code>IJavaProject</code> that is being compiled
 * @return Always returns <code>true</code>
 */
public boolean isActive(IJavaProject project) {
    return true;
}

/**
 * Overridden <code>reconcile()</code> method that creates the <code>SecureNodeAnalyzer</code> to scan
 * through the abstract syntax tree and look for secure code rule violations
 * @param context The <code>ReconcileContext</code> that is being reconciled
 */
@SuppressWarnings("deprecation")
@Override
public void reconcile(ReconcileContext context) {
    super.reconcile(context);

    // Check to see if AST has changed
    IJavaElementDelta elementDelta = context.getDelta();
    if (elementDelta != null &&
       (elementDelta.getFlags() & IJavaElementDelta.F_AST_AFFECTED) != 0) {
        CompilationUnit compilation = null;
        IResource resource = context.getWorkingCopy().getResource();
        try {
            compilation = context.getAST4();
        } catch (JavaModelException e) {
            // From context.getAST4()
            e.printStackTrace();
        }
    }

    // Clear all existing markers
    clearMarkers(resource);

    if (compilation != null) {
        // Create a new NodeVisitor to go through the AST and look for violated rules
        SecureNodeAnalyzer visitor = new SecureNodeAnalyzer(m_rules, context);
        compilation.accept(visitor);
    }

    // Update insecure code segment list
    if (visitor != null) {
        ArrayList<InsecureCodeSegment> newInsecureCodeSegments = visitor.getInsecureCodeSegments();
        m_insecureCodeSegments.addAll(newInsecureCodeSegments);
private void clearMarkers(IResource resource) {
Iterator<InsecureCodeSegment> csItr = m_insecureCodeSegments.iterator();
while (csItr.hasNext()) {
InsecureCodeSegment cs = csItr.next();
try {
if (cs.getResource().equals(resource)) {
    cs.deleteMarker();
    csItr.remove();
}
} catch (CoreException e) {
    // Couldn't delete marker
    e.printStackTrace();
}
}
}

SecureNodeAnalyzer.java

package edu.csus.plugin.securecodingassistant.compilation;
import java.util.ArrayList;
import org.eclipse.jdt.core.compiler.ReconcileContext;
import org.eclipse.jdt.core.dom.ASTNode;
import org.eclipse.jdt.core.dom.ASTVisitor;
import edu.csus.plugin.securecodingassistant.rules.IRule;
/**
 * A custom <code>ASTVisitor</code> for the Secure Coding Assistant that checks each
 * AST (Abstract Syntax Tree) to see if there has been a rule violation. This node
 * visitor
 * must be given a list of <link IRule> rules to check for as well as the existing list
 * of the
 * <link InsecureCodeSegment> objects to prevent the creation of duplicate violations.
 * @author Ben White
 * @see IRule
 * @see SecureCompilationParticipant
 * @see InsecureCodeSegment
 */
class SecureNodeAnalyzer extends ASTVisitor {
    /**
     * A collection of rules to check for
     */
    private ArrayList<IRule> m_rules;
    /**
     * A collection of new <code>InsecureCodeSegment</code> objects. This collection is
     * added to each time a new insecure section of code is identified.
     */
    private ArrayList<InsecureCodeSegment> m_insecureCodeSegments;
    /**
     * The <code>ReconcileContext</code> where new violations will be created.
     */
private ReconcileContext m_context;

/**< *
 * Creates a new <code>NodeVisitor</code>. You must call this constructor and not the
default constructor. The <code>rules</code>, <code>insecureCodeSegments</code>, and
<code>context</code> are required to scan for new insecure code.
 * @param context The <code>ReconcileContext</code> where new violations will be
created.
 */
public SecureNodeAnalyzer(ArrayList<IRule> rules,
    ReconcileContext context) {
    super();

    m_rules = rules;
    m_insecureCodeSegments = new ArrayList<InsecureCodeSegment>();
    m_context = context;
}

/**< *
 * Visits each node in the abstract syntax tree and
 * adds to the <code>InsecureCodeSegment</code> collection that is returned by
 * {@link SecureNodeAnalyzer#getInsecureCodeSegments()} */
@Override
public void preVisit(ASTNode node) {
    // Iterate through rules
    for (IRule rule : m_rules)
        if (rule.violated(node))
            m_insecureCodeSegments.add(new InsecureCodeSegment(node, rule, m_context));
}

/**< *
 * Returns a list of all of the insecure code segments that were detected after visiting
 * all of the nodes in the abstract syntax tree.
 * @return Collection of <code>InsecureCodeSegment</code> objects
 */
public ArrayList<InsecureCodeSegment> getInsecureCodeSegments() {
    return m_insecureCodeSegments;
}

IRule.java

package edu.csus.plugin.securecodingassistant.rules;

import org.eclipse.jdt.core.dom.ASTNode;
import edu.csus.plugin.securecodingassistant.Globals;

/**< *
 * Interface for a secure coding rule
 * @author Ben White
 */
public interface IRule {

/**< *
 * Checks to see if the rule has been violated in a given node
 * @param node The node to be evaluated
 * @return true if the rule was violated, false otherwise
 */
public boolean violated(ASTNode node);

/**
 * The description of the rule that was violated
 * @return The description of the rule that was violated
 */
public String getRuleText();

/**
 * The name of the rule violated
 * @return The name of the rule violated
 */
public String getRuleName();

/**
 * The recommended action that will satisfy the rule
 * @return The recommended action that will satisfy the rule
 */
public String getRuleRecommendation();

/**
 * The security level of the violated rule
 * @return A {@link Globals.Markers} security level
 * @see Globals.Markers
 */
public int securityLevel();

}

RuleFactory.java

package edu.csus.plugin.securecodingassistant.rules;

import java.util.ArrayList;

/**
 * Creates a set of rules that implement the <code>IRule</code> interface
 * @author Ben White
 * @see IRule
 */
public final class RuleFactory {

/**
 * Cannot instantiate
 */
private RuleFactory() {
}

/**
 * Creates a collection of all possible rules to test
 * @return A collection of all possible rules to test
 */
public static ArrayList<IRule> getAllRules() {
    ArrayList<IRule> rules = new ArrayList<IRule>();

    // 00. Input Validation and Data Sanitization
    rules.add(new IDS00J_PreventSQLInjection());
    rules.add(new IDS01JNormalizeStringsBeforeValidating());
    rules.add(new IDS07J_RuntimeExecMethod());
    rules.add(new IDS11J_ModifyStringsBeforeValidation());

    // 01. Declarations and Initialization
    rules.add(new DCL02J_DoNotModifyElements());
// 02. Expressions
rules.add(new EXP00J_DoNotIgnoreValuesReturnedByMethods());
rules.add(new EXP02J_DoNotUseObjectEqualsToCompareArrays());

// 03. Numeric Types and Operations
rules.add(new NUM07J_DoNotAttemptComparisonsWithNaN());
rules.add(new NUM09J_DoNotUseFloatingPointAsLoopCounters());

// 04. Characters and Strings
rules.add(new STR00J_PartialCharFromVarWidthEnc());

// 05. Object Orientation
rules.add(new OBJ09J_CompareClassesAndNotClassNames());

// 06. Methods
rules.add(new MET04J_DoNotIncreaseTheAccessibilityOfOveriddenMethods());
rules.add(new MET06J_DoNotInvokeOverridableMethodsInClone());

// 07. Exceptional Behavior
rules.add(new ERR08J_DoNotCatchNullPointerException());

// 09. Locking
rules.add(new LCK09J_DoNotPerformOperationsThatCanBlockWhileHoldingLock());

// 10. Thread APIs
rules.add(new THI05J_DoNotUseThreadStopToTerminateThreads());

// 13. Input Output
rules.add(new FIO08J_DistinguishBetweenCharactersOrBytes());

// 14. Serialization
rules.add(new SER01J_DoNotDeviateFromTheProperSignaturesOfSerializationMethods());

// 15. Platform Security
rules.add(new SEC07J_CallTheSuperclassGetPermissionsMethod());

// 16. Runtime Environment
rules.add(new ENV02J_DoNotTrustTheValuesOfEnvironmentVariables());

// 49. Miscellaneous
rules.add(new MSC02J_GenerateStrongRandomNumbers());

return rules;
}

NodeNumPair.java
package edu.csus.plugin.securecodingassistant.rules;
import org.eclipse.jdt.core.dom.ASTNode;

/**
 * This pair of an ASTNode and an int is to be used to keep track of
 * the location of a node in a syntax tree in relationship to other nodes.
 * @author Ben White
 * @see ASTNode
 * @see ASTNodeProcessor
 */
class NodeNumPair {

/**
 * The stored node
*/
private ASTNode m_node;

/**
 * The number associated with the node
 */
private int m_num;

/**
 * Construct a new pair
 * @param node The <code>ASTNode</code>
 * @param num The number to associate with the node
 */
public NodeNumPair (ASTNode node, int num) {
    m_node = node;
    m_num = num;
}

/**
 * Retrieve the node
 * @return the node
 */
public ASTNode getNode() {
    return m_node;
}

/**
 * Retrieve the number
 * @return the number
 */
public int getNum() {
    return m_num;
}

ASTNodeProcessor.java

package edu.csus.plugin.securecodingassistant.rules;

import java.util.ArrayList;
import org.eclipse.jdt.core.dom.ASTVisitor;
import org.eclipse.jdt.core.dom.Assignment;
import org.eclipse.jdt.core.dom.ClassInstanceCreation;
import org.eclipse.jdt.core.dom.EnhancedForStatement;
import org.eclipse.jdt.core.dom.MethodInvocation;
import org.eclipse.jdt.core.dom.SuperMethodInvocation;
import org.eclipse.jdt.core.dom.VariableDeclarationFragment;

/**
 * A custom <code>ASTVisitor</code> that is used by rules to parse an abstract syntax tree.
 * This can be used to get a list of methods that are in the same block as another for instance.
 * @author Ben White
 * @see org.eclipse.jdt.core.dom.ASTNode
 * @see IRule
 */
class ASTNodeProcessor extends ASTVisitor {
    private ArrayList<NodeNumPair> m_methods;
/**
 * A list of assignments that are in the syntax tree
 */
private ArrayList<NodeNumPair> m_assignments;

/**
 * A list of instantiations
 */
private ArrayList<NodeNumPair> m_instantiations;

/**
 * A list of enhanced for statements
 */
private ArrayList<NodeNumPair> m_enhancedForStatements;

/**
 * A list of super method invocations
 */
private ArrayList<NodeNumPair> m_superMethods;

/**
 * A list of variable declaration fragments
 */
private ArrayList<NodeNumPair> m_variableDeclarations;

/**
 * Counts the number of nodes visited
 */
private int m_nodeCounter;

/**
 * Create new node processor, no arguments required
 */
public ASTNodeProcessor() {
    super();
    m_methods = new ArrayList<NodeNumPair>();
    m_assignments = new ArrayList<NodeNumPair>();
    m_instantiations = new ArrayList<NodeNumPair>();
    m_enhancedForStatements = new ArrayList<NodeNumPair>();
    m_superMethods = new ArrayList<NodeNumPair>();
    m_variableDeclarations = new ArrayList<NodeNumPair>();
    m_nodeCounter = 0;
}

/**
 * Will build a list of <code>MethodInvocation</code> objects that are in the
 * syntax tree
 */
@Override
public boolean visit(MethodInvocation methodInvocation) {
    m_methods.add(new NodeNumPair(methodInvocation, ++m_nodeCounter));
    return super.visit(methodInvocation);
}

/**
 * Will build a list of <code>Assignment</code> objects that are in the
 * syntax tree
 */
@Override
public boolean visit(Assignment assignment) {
    m_assignments.add(new NodeNumPair(assignment, ++m_nodeCounter));
    return super.visit(assignment);
}
/**
 * Will build a list of `ClassInstanceCreation` objects that are in the syntax tree
 */
@Override
public boolean visit(ClassInstanceCreation instantiation) {
    m_instantiations.add(new NodeNumPair(instantiation, ++m_nodeCounter));
    return super.visit(instantiation);
}

/**
 * Will build a list of `EnhancedForStatement` objects that are in the syntax tree
 */
@Override
public boolean visit(EnhancedForStatement statement) {
    m_enhancedForStatements.add(new NodeNumPair(statement, ++m_nodeCounter));
    return super.visit(statement);
}

/**
 * Will build a list of `SuperMethodInvocation` objects that are in the syntax tree
 */
@Override
public boolean visit(SuperMethodInvocation method) {
    m_superMethods.add(new NodeNumPair(method, ++m_nodeCounter));
    return super.visit(method);
}

/**
 * Will build a list of `VariableDeclarationFragment` objects that are in the syntax tree
 */
@Override
public boolean visit(VariableDeclarationFragment declaration) {
    m_variableDeclarations.add(new NodeNumPair(declaration, ++m_nodeCounter));
    return super.visit(declaration);
}

/**
 * Retrieve the list of `MethodInvocation` objects that are in the syntax tree
 */
public ArrayList<NodeNumPair> getMethods() {
    return m_methods;
}

/**
 * Retrieve the list of `Assignment` objects that are in the syntax tree
 */
public ArrayList<NodeNumPair> getAssignments() {
    return m_assignments;
}
public ArrayList<NodeNumPair> getInstanceCreations() {
    return m_instantiations;
}

public ArrayList<NodeNumPair> getEnhancedForStatements() {
    return m_enhancedForStatements;
}

public ArrayList<NodeNumPair> getSuperMethodInvocations() {
    return m_superMethods;
}

public ArrayList<NodeNumPair> getVariableDeclarationFragments() {
    return m_variableDeclarations;
}

package edu.csus.plugin.securecodingassistant.rules;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import org.eclipse.jdt.core.dom.ASTMatcher;
import org.eclipse.jdt.core.dom.ASTNode;
import org.eclipse.jdt.core.dom.Assignment;
import org.eclipse.jdt.core.dom.Block;
import org.eclipse.jdt.core.dom.ClassInstanceCreation;
import org.eclipse.jdt.core.dom.Expression;
import org.eclipse.jdt.core.dom.IMethodBinding;
import org.eclipse.jdt.core.dom.ITypeBinding;
import org.eclipse.jdt.core.dom.MethodInvocation;
import org.eclipse.jdt.core.domSimpleName;

final class Utility {

    // Collection of utility methods used by the Secure Coding Assistant Rules
    final class Utility {
/**
 * Cannot instantiate
 */

private Utility() {
}

// TODO: Provide a mechanism that the programmer can use to ignore false positives

/**
 * Use to check to see if a <code>MethodInvocation</code> node of an abstract syntax tree
 * is calling a particular method from a class.
 * @param method The method invocation from the <code>ASTNode</code>
 * @param className The qualified name of the class where the method is defined
 * @param methodName The name of the method without the parameters. For instance, if the method
 * that is being searched for is <code>System.out.println()</code>, then pass
 * true to <code>Object</code> when looking to see if a method was called. For instance, if
 * method() is defined in class A but called from subclass B. Use this flag if
 * passing class A as the class name.
 * @return True if the <code>MethodInvocation</code> is <code>className.methodName()</code>
 */
public static boolean calledMethod(MethodInvocation method, String className, String methodName, boolean searchClassHierarchy) {
    return calledMethod(method, className, methodName, null, searchClassHierarchy);
}

/**
 * Use to check to see if a <code>MethodInvocation</code> node of an abstract syntax tree
 * is calling a particular method from a class with a given argument.
 * @param method The method invocation from the <code>ASTNode</code>
 * @param className The qualified name of the class where the method is defined
 * @param methodName The name of the method without the parameters. For instance, if the method
 * that is being searched for is <code>System.out.println()</code>, then pass
 * true to <code>Object</code> when looking to see if a method was called. For instance, if
 * method() is defined in class A but called from subclass B. Use this flag if
 * passing class A as the class name.
 * @return True if the <code>MethodInvocation</code> is <code>className.methodName()</code>
 */
public static boolean calledMethod(MethodInvocation method, String className, String methodName) {
    return calledMethod(method, className, methodName, null, false);
}
* that is being searched for is `<code>System.out.println()</code>`, then pass
* `<code>println</code>` as the argument. If desired search behavior is to search super class
  hierarchy up
* to `<code>Object</code>` when looking to see if a method was called. For instance, if
  `<code>method()</code>` is defined in class A but called from subclass B. Use this flag
  if
* passing class A as the class name.

* @return `<code>true</code>` if the `MethodInvocation` is

  `className.methodName()`

* @see ASTNode
  * @see MethodInvocation

```java
public static boolean calledMethod(MethodInvocation method, String className,
        String methodName, SimpleName argument, boolean searchClassHierarchy) {
    boolean nameMatch = false, withArgument = false;
    if (method.getExpression() != null && method.getExpression().resolveTypeBinding() != null) {
        String miClassName = method.getExpression().resolveTypeBinding().getBinaryName();
        String miMethodName = method.getName().getFullyQualifiedName();
        withArgument = argument == null; // Default to true if no argument required
        nameMatch = miClassName.equals(className) && miMethodName.equals(methodName);
        // If searching hierarchy then search declaring class then parent classes
        if (searchClassHierarchy && !nameMatch) {
            ITypeBinding superClass = method.resolveMethodBinding().getDeclaringClass();
            nameMatch = superClass.getBinaryName().equals(className);
            while (!nameMatch &&
                  !superClass.getBinaryName().equals(Object.class.getCanonicalName())) {
                superClass = superClass.getSuperclass();
                nameMatch = superClass.getBinaryName().equals(className);
            }
        }
        // Do argument search if required
        if (argument != null && nameMatch) {
            withArgument = argumentMatch(method.arguments(), argument);
        }
        return nameMatch & withArgument;
    }
```

/**
 * Use to check to see if a method is being called prior to a given method
 * @param method The method that was called
 * @param className The qualified name of the class of the method to be tested
 * @param methodName The name of the method to be tested to see if it occurs prior to
 * the given `{@link MethodInvocation}`
 * @return `<code>true</code>` if the `<code>className.methodName()</code>` is called prior
 * to the
 * `{@link MethodInvocation}`
 * @see ASTNode
 * @see MethodInvocation
 */
```java
public static boolean calledPrior(MethodInvocation method, String className, String
        methodName) {
    return calledPrior(method, className, methodName, null);
}
```

/**
* Use to check to see if a method is being called prior to a given method with a given argument
* @param method The method that was called
* @param className The qualified name of the class of the method to be tested
* @param methodName The name of the method to be tested to see if it occurs prior to the
* given {@link MethodInvocation}
* @param argument The argument that needs to occur in the prior method for it to be considered
* @return <code>true</code> if the <code>className.methodName()</code> is called prior to the
* {@link MethodInvocation}
* @see ASTNode
* @see MethodInvocation
*
public static boolean calledPrior(MethodInvocation method, String className, String
methodName, SimpleName argument) {
  boolean foundMethod = false;
  int methodPosition; // the location of the method in the AST where searching should stop
  boolean continueSearch = true; // false when searching shouldn't continue
  ASTNode node = getEnclosingNode(method, Block.class);

  if (node != null && node instanceof Block) {
    Block block = (Block)node;
    ASTNodeProcessor processor = new ASTNodeProcessor();
    block.accept(processor);
    ArrayList<NodeNumPair> nodes = processor.getMethods();
    methodPosition = searchNodeNumList(nodes, method);

    // Go through all blockMethods prior to method and look for a match
    Iterator<NodeNumPair> nodeItr = nodes.iterator();
    while (nodeItr.hasNext() && continueSearch && !foundMethod) {
      NodeNumPair n = nodeItr.next();
      assert n.getNode() instanceof MethodInvocation;
      MethodInvocation blockMethod = (MethodInvocation)n.getNode();
      foundMethod = calledMethod(blockMethod, className, methodName, argument, false);
      continueSearch = searchNodeNumList(nodes, blockMethod) < methodPosition;
    }
  }

  return foundMethod;
}

/**
 * Use to see if a variable has been modified after a <code>MethodInvocation</code> in the syntax tree
 * @param method The <code>MethodInvocation</code> in the syntax tree to start the search
 * @param identifier The identifier to search for
 * @return <code>true</code> if the identifier was found being modified after the node
 */
public static boolean modifiedAfter(MethodInvocation method, SimpleName identifier) {
  boolean isModified = false;
  int methodPosition; // The location of the method in the AST where searching should start
  ASTNode node = getEnclosingNode(method, Block.class);

  if (node != null && node instanceof Block) {
    Block block = (Block)node;
    ASTNodeProcessor processor = new ASTNodeProcessor();
    block.accept(processor);
    methodPosition = searchNodeNumList(processor.getMethods(), method);
// Go through all assignments
ArrayList<NodeNumPair> nodes = processor.getAssignments();
for (NodeNumPair n : nodes) {
    assert n.getNode() instanceof Assignment;
    Assignment assignment = (Assignment)n.getNode();
    if (searchNodeNumList(nodes, assignment) > methodPosition && !isModified) {
        Expression lhs = assignment.getLeftHandSide();
        isModified = lhs.subtreeMatch(new ASTMatcher(), identifier);
    }
}
return isModified;
/**
 * Returns <code>true</code> if the given node contains an instantiation for the given class
 * with the given argument
 * @param node The node to search through
 * @param className The qualified name of the class to look for a constructor for
 * @param argument Only return <code>true</code> if the constructor contains this argument
 * @return <code>true</code> if the given node contains the given constructor with the given argument
 */
public static boolean containsInstanceCreation(ASTNode node, String className, SimpleName argument) {
    boolean containsInstanceCreation = false;
    ASTNodeProcessor processor = new ASTNodeProcessor();
    node.accept(processor);
    ArrayList<NodeNumPair> nodes = processor.getInstanceCreations();
    for (NodeNumPair n : nodes) {
        assert n.getNode() instanceof ClassInstanceCreation;
        ClassInstanceCreation instanceCreation = (ClassInstanceCreation)n.getNode();
        if (instanceCreation.getType() != null &&
                instanceCreation.getType().resolveBinding() != null &&
                instanceCreation.getType().resolveBinding().getBinaryName().equals(className)) {
            containsInstanceCreation = argument == null; // true if none required
            if (!containsInstanceCreation)
                containsInstanceCreation = argumentMatch(instanceCreation.arguments(), argument);
        }
    }
    return containsInstanceCreation;
}
/**
 * Returns an enclosing node for a given node
 * @param node The node to look for in the statement
 * @param nodeType The type of node to look for (e.g. Block, WhileStatement, etc.)
 * @return The node if found, <code>null</code> otherwise
 */
public static ASTNode getEnclosingNode(ASTNode node, Class<? extends ASTNode> nodeType) {
    ASTNode parent = node.getParent();
    while (parent != null && !nodeType.equals(parent.getClass()))
        parent = parent.getParent();
return parent == null ? null : parent;
}

/**
 * Returns <code>true</code> if the given argument occurs in the list of arguments
 * @param arguments A <code>List</code> of <code>Expression</code> type arguments
 * @param argument A <code>SimpleName</code> argument to look for in the list
 * @return <code>true</code> if the given argument occurs in the list of arguments
 * @see Expression
 * @see SimpleName
 * @see ClassInstanceCreation#arguments()
 * @see MethodInvocation#arguments()
 */
public static boolean argumentMatch(List<?> arguments, SimpleName argument) {
    for (Object o : arguments)
        if (o instanceof Expression) {
            Expression e = (Expression)o;
            if (e.subtreeMatch(new ASTMatcher(), argument))
                return true;
        }
    return false;
}

/**
 * When a method is overriding another method this retrieves the
 * <code>IMethodBinding</code> object from the super class.
 * @param methodDec The <code>IMethodBinding</code> that overrides a method declaration from
 * a parent class.
 * @return The <code>IMethodBinding</code> from the parent class that was overridden or
 * <code>null</code> if none found.
 */
public static IMethodBinding getSuperClassDeclaration(IMethodBinding method) {
    // Get super class
    ITypeBinding superClass = method.getDeclaringClass().getSuperclass();
    while (superClass != null) {
        // Loop through all declared methods in the super class
        for (IMethodBinding superMethod : superClass.getDeclaredMethods()) {
            if (method.overrides(superMethod)) {
                return superMethod;
            }
        }
        // Get next super class
        superClass = superClass.getSuperclass();
    }
    return null;
}

/**
 * Searches through an array list of <code>NodeNumPair</code> objects for a particular
 * <code>ASTNode</code> and returns the number associated with it
 * @param nodeNumList A list of <code>NodeNumPair</code> objects to search through
 * @param node The <code>ASTNode</code> to search for
 * @return The number associated with the node found in the list or -1 if not found
 */
private static int searchNodeNumList(ArrayList<NodeNumPair> nodeNumList, ASTNode node) {
    for (NodeNumPair pair : nodeNumList)
        if (pair.getNode().equals(node))
            return pair.getNum();
    return -1;
}
DCL02J_DoNotModifyElements.java

```java
package edu.csus.plugin.securecodingassistant.rules;
import org.eclipse.jdt.core.dom.ASTNode;
import org.eclipse.jdt.core.dom.EnhancedForStatement;
import org.eclipse.jdt.core.dom.Modifier;
import org.eclipse.jdt.core.dom.SingleVariableDeclaration;
import edu.csus.plugin.securecodingassistant.Globals;

/**
 * The text and/or code below is from the CERT website:
 * Java Secure Coding Rule: DCL02-J. Do not modify the collection's elements during an enhanced for statement.
 * CERT Website: Unlike the basic for statement, assignments to the loop variable fail to affect the loop's iteration order over the underlying set of objects. Consequently, an assignment to the loop variable is equivalent to modifying a variable local to the loop body whose initial value is the object referenced by the loop iterator. This modification is not necessarily erroneous but can obscure the loop functionality or indicate a misunderstanding of the underlying implementation of the enhanced for statement.
 * Declare all enhanced for statement loop variables final. The final declaration causes Java compilers to flag and reject any assignments made to the loop variable.
 * @author Ben White (Plugin Logic), CERT (Rule Definition)
 * @see Java Secure Coding Rule defined by CERT: https://www.securecoding.cert.org/confluence/display/java/DCL02-J+Do+not+modify+the+collection%27s+elements+during+an+enhanced+for+statement
 */
class DCL02J_DoNotModifyElements implements IRule {

@Override
public boolean violated(ASTNode node) {
    boolean ruleViolated = false;
    if (node instanceof EnhancedForStatement) {
        EnhancedForStatement statement = (EnhancedForStatement) node;
        SingleVariableDeclaration dec = statement.getParameter();
        // Look to see if they declared as final
        for (Object o : dec.modifiers()) {
            assert o instanceof Modifier;
            Modifier m = (Modifier) o;
            if (!m.isFinal()) {
                ruleViolated = false;
                break;
            }
        }
    }
    return ruleViolated;
}
```
@Override
public String getRuleText() {
    return "CERT Website-Unlike the basic for statement, assignments to the loop "
    + "variable fail to affect the loop's iteration order over the underlying "
    + "set of objects. Consequently, an assignment to the loop variable is "
    + "equivalent to modifying a variable local to the loop body whose initial"
    + " value is the object referenced by the loop iterator. This modification"
    + " is not necessarily erroneous but can obscure the loop functionality or*
    + " indicate a misunderstanding of the underlying implementation of the "
    + "enhanced for statement.";
}

@Override
public String getRuleName() {
    return "DCL02-J. Do not modify the collection's elements during an enhanced for"
    + " statement";
}

@Override
public String getRuleRecommendation() {
    return "Declare all enhanced for statement loop variables final. The final "
    + "declaration causes Java compilers to flag and reject any assignments"
    + " made to the loop variable.";
}

@Override
public int securityLevel() {
    return Globals.Markers.SECURITY_LEVEL_LOW;
}
}

ENV02J_DoNotTrustTheValuesOfEnvironmentVariables.java

package edu.csus.plugin.securecodingassistant.rules;

import org.eclipse.jdt.core.dom.ASTNode;
import org.eclipse.jdt.core.dom.MethodInvocation;
import edu.csus.plugin.securecodingassistant.Globals;

/**
 * "<b><i>The text and/or code below is from the CERT website: <a target="_blank"
href="https://www.securecoding.cert.org">https://www.securecoding.cert.org</a></i></b>
 * Java Secure Coding Rule: ENV02-J. Do not trust the values of environment variables
 * CERT Website: Programs that execute in a more trusted domain than their environment
 * must assume that the values of environment variables are untrusted and must sanitize and
 * validate any environment variable values before use.
 * The default values of system properties are set by the Java Virtual Machine (JVM) upon
 * startup and can be considered trusted. However, they may be overridden by properties
 * from untrusted sources, such as a configuration file. System properties from untrusted
 * sources must be sanitized and validated before use.
 * Actually, relying on environment variables is more than a portability issue. An
 * attacker
can essentially control all environment variables that enter a program using a mechanism such as the `java.lang.ProcessBuilder` class.

Consequently, when an environment variable contains information that is available by other means, including system properties, that environment variable must not be used. Finally, environment variables must not be used without appropriate validation.

@see Java Secure Coding Rule defined by CERT: <a target="_blank" href="https://www.securecoding.cert.org/confluence/display/java/ENV02-J.+Do+not+trust+the+values+of+environment+variables">ENV02-J</a>

class ENV02J_DoNotTrustTheValuesOfEnvironmentVariables implements IRule {

    @Override
    public boolean violated(ASTNode node) {
        boolean ruleViolated = false;
        // Is node a method invocation?
        if (node instanceof MethodInvocation)
            // Was System.getenv() called?
            ruleViolated = Utility.calledMethod((MethodInvocation) node,
                System.class.getCanonicalName(), "getenv");
        return ruleViolated;
    }

    @Override
    public String getRuleText() {
        return "CERT Website-Programs that execute in a more trusted domain than their " + "environment must assume that the values of environment variables are " + "untrusted and must sanitize and validate any environment variable values " + "before use."
    }

    @Override
    public String getRuleName() {
        return "ENV02-J. Do not trust the values of environment variables";
    }

    @Override
    public String getRuleRecommendation() {
        return "Avoid calls to the System.getenv() command";
    }

    @Override
    public int securityLevel() {
        return Globals.Markers.SECURITY_LEVEL_MEDIUM;
    }
}

ERR08J_DoNotCatchNullPointerException.java

package edu.csus.plugin.securecodingassistant.rules;

import org.eclipse.jdt.core.dom.ASTNode;
import org.eclipse.jdt.core.dom.CatchClause;
import org.eclipse.jdt.core.dom.SingleVariableDeclaration;
import edu.csus.plugin.securecodingassistant.Globals;
Java Secure Coding Rule: ERR08-J. Do not catch NullPointerException or any of its ancestors.

CERT Website: Programs must not catch <code>java.lang.NullPointerException</code>. A <code>NullPointerException</code> exception thrown at runtime indicates the existence of an underlying null pointer dereference that must be fixed in the application code (see EXP01-J. Do not use a null in a case where an object is required for more information). Handling the underlying null pointer dereference by catching the <code>NullPointerException</code> rather than fixing the underlying problem is inappropriate for several reasons. First, catching <code>NullPointerException</code> adds significantly more performance overhead than simply adding the necessary null checks [Bloch 2008]. Second, when multiple expressions in a try block are capable of throwing a <code>NullPointerException</code>, it is difficult or impossible to determine which expression is responsible for the exception because the <code>NullPointerException</code> catch block handles any <code>NullPointerException</code> thrown from any location in the try block. Third, programs rarely remain in an expected and usable state after a <code>NullPointerException</code> has been thrown. Attempts to continue execution after first catching and logging (or worse, suppressing) the exception rarely succeed.

The text and/or code below is from the CERT website: [https://www.securecoding.cert.org](https://www.securecoding.cert.org)

```java
import java.lang.RuntimeException;
import java.lang.Exception;
import java.lang.Throwable;

public class ER08J_DoNotCatchNullPointerException implements IRule {
    @Override
    public boolean violated(ASTNode node) {
        boolean ruleViolated = false;
        if (node instanceof CatchClause) {
            String exceptionType = ((CatchClause) node).getException().resolveBinding().getType().getQualifiedName();
            ruleViolated = exceptionType.equals(NullPointerException.class.getCanonicalName()) ||
                        exceptionType.equals(RuntimeException.class.getCanonicalName()) ||
                        exceptionType.equals(Exception.class.getCanonicalName()) ||
                        exceptionType.equals(Throwable.class.getCanonicalName());
        }
        return ruleViolated;
    }

    @Override
    public String getRuleText() {
        return "Catching a null pointer exception (or any of its ancestors) increases performance overhead, are difficult to troubleshoot and programs rarely remain in a usable state after they are thrown."
    }

    @Override
    public String getRuleName() {
        return "ERR08-J. Do not catch NullPointerException or any of its ancestors";
    }
}
```
@Override
public String getRuleRecommendation() {
    return "Test for null before dereferencing and permit the exception to be " + "thrown."
}

@Override
public int securityLevel() {
    return Globals.Markers.SECURITY_LEVEL_HIGH;
}
}

EXP00J_DoNotIgnoreValuesReturnedByMethods.java

package edu.csus.plugin.securecodingassistant.rules;
import org.eclipse.jdt.core.dom.ASTNode;
import org.eclipse.jdt.core.dom.ExpressionStatement;
import org.eclipse.jdt.core.dom.ITypeBinding;
import org.eclipse.jdt.core.dom.MethodInvocation;
import edu.csus.plugin.securecodingassistant.Globals;

/**
 * The text and/or code below is from the CERT website:
 * https://www.securecoding.cert.org
 * CERT Website: Methods can return values to communicate failure or success
 * or to update local objects or fields. Security risks can arise when method
 * return values are ignored or when the invoking method fails to take suitable
 * action. Consequently, programs must not ignore method return values.
 * @author Ben White (Plugin Logic), CERT (Rule Definition)
 * @see Java Secure Coding Rule defined by CERT: EXP00-J
 */
class EXP00J_DoNotIgnoreValuesReturnedByMethods implements IRule {

    @Override
    public boolean violated(ASTNode node) {
        boolean ruleViolated = false;

        // Is the node a method invocation?
        if (node instanceof MethodInvocation) {
            MethodInvocation method = (MethodInvocation)node;
            ITypeBinding returnType = method.resolveMethodBinding() == null ? null : method.resolveMethodBinding().getReturnType();
            // Does it have a return type?
            if (returnType != null && returnType.getQualifiedName().equals("void")) {

            }
        }
    }
}

ASTNode parent = method.getParent();
    // Was the return type used?
    // (If ExpressionStatement then it was not used
    ruleViolated = parent instanceof ExpressionStatement;
}
}
return ruleViolated;
}

@Override
public String getRuleText() {
    return "CERT Website-Methods can return values to communicate failure or " + "success or to update local objects or fields. Security risks can " + "arise when method return values are ignored or when the invoking " + "method fails to take suitable action. Consequently, programs must " + "not ignore method return values."
;
}

@Override
public String getRuleName() {
    return "EXP02-J. Do not ignore values returned by methods";
}

@Override
public String getRuleRecommendation() {
    return "Capture the return value of the method call";
}

@Override
public int securityLevel() {
    return Globals.Markers.SECURITY_LEVEL_MEDIUM;
}

EXP02J_DoNotUseObjectEqualsToCompareArrays.java

package edu.csus.plugin.securecodingassistant.rules;

import org.eclipse.jdt.core.dom.ASTNode;
import org.eclipse.jdt.core.dom.MethodInvocation;
import edu.csus.plugin.securecodingassistant.Globals;

/**
 * Certification: In Java, arrays are objects and support object methods such
 * as Object.equals() in addition to methods specifically designed for
 * array manipulation. This rule is to prevent using the Object.equals() method
 * to compare two arrays. 
 * 
 * CERT Website: In Java, arrays are objects and support object methods such
 * as Object.equals(). However, arrays do not support any methods
 * besides those provided by Object. Consequently, using
 * Object.equals() on any array compares only array references, not
 * their contents. Programmers who wish to compare the contents of two arrays
 * must use the static two-argument Arrays.equals() method.
 * This method considers two arrays equivalent if both arrays contain the same
 * number of elements, and all corresponding pairs of elements in the two arrays
 * are equivalent, according to Object.equals(). In other words,
 * two arrays are equal if they contain equivalent elements in the same order.
 * To test for reference equality, use the reference equality operators, == and
 */
Because the effect of using <code>Object.equals()</code> to compare two arrays is often misconstrued as content equality, and because a better alternative exists in the use of reference equality operators, the use of the <code>Object.equals()</code> method to compare two arrays is disallowed.

@author Ben White (Plugin Logic), CERT (Rule Definition)  
@see Java Secure Coding Rule defined by CERT: <a target="_blank" href="https://www.securecoding.cert.org/confluence/display/java/EXP02-J+Do+not+use+the+Object.equals%28%29+method+to+compare+two+arrays">EXP02-J</a>

```java
class EXP02J_DoNotUseObjectEqualsToCompareArrays implements IRule {

    @Override
    public boolean violated(ASTNode node) {
        boolean ruleViolated = false;

        // Is node a method invocation?
        if (node instanceof MethodInvocation) {
            MethodInvocation method = (MethodInvocation) node;

            // Was equals called from an array?
            if (method.getExpression() != null &&
                method.getExpression().resolveTypeBinding() != null)
                ruleViolated = method.getExpression().resolveTypeBinding().isArray() &&
                    method.getName().getFullyQualifiedName().equals("equals");

        }

        return ruleViolated;
    }

    @Override
    public String getRuleText() {
        return "CERT Website-In Java, arrays are objects and support object " +
            "methods such as Object.equals(). However, arrays do not " +
            "support any methods besides those provided by Object. " +
            "Consequently, using Object.equals() on any array compares only " +
            "array references, not their contents."
    }

    @Override
    public String getRuleName() {
        return "EXP02-J. Do not use the Object.equals() method to compare" +
            " two arrays";
    }

    @Override
    public String getRuleRecommendation() {
        return "CERT Website-Programmers who wish to " +
            "compare the contents of two arrays must use the static two-" +
            "argument Arrays.equals() method. This method considers two " +
            "arrays equivalent if both arrays contain the same number of " +
            "elements, and all corresponding pairs of elements in the two " +
            "arrays are equivalent, according to Object.equals(). In other " +
            "words, two arrays are equal if they contain equivalent elements " +
            "in the same order. To test for reference equality, use the " +
            "reference equality operators, == and !=";
    }

    @Override
    public int securityLevel() {
        return Globals.Markers.SECURITY_LEVEL_MEDIUM;
    }
}
```
FIO08J_DistinguishBetweenCharactersOrBytes.java

package edu.csus.plugin.securecodingassistant.rules;
import java.io.InputStream;
import java.io.Reader;
import org.eclipse.jdt.core.dom.ASTNode;
import org.eclipse.jdt.core.dom.CastExpression;
import org.eclipse.jdt.core.dom.MethodInvocation;
import edu.csus.plugin.securecodingassistant.Globals;

/**
 * The text and/or code below is from the CERT website: <a target="blank"
 href="https://www.securecoding.cert.org">https://www.securecoding.cert.org</a>
 */

class FIO08J_DistinguishBetweenCharactersOrBytes implements IRule {

@Override
public boolean violated(ASTNode node) {
    boolean ruleViolated = false;

    // Check to see if in CastExpression
    if (node instanceof CastExpression) {
        CastExpression cast = (CastExpression)node;
        // ...
// Check to see if expression portion is a method invocation
if (cast.getExpression() instanceof MethodInvocation) {
    MethodInvocation method = (MethodInvocation) cast.getExpression();
    String castType = cast.getType().resolveBinding().getQualifiedName();
    // Rule is violated if InputStream.read() or ancestor is called and cast to byte
    // or if Reader.read() or ancestor is called and cast to char
    ruleViolated = (castType.equals(byte.class.getCanonicalName()))
        && Utility.calledMethod(method, InputStream.class.getCanonicalName(),
            "read", true)
    || (castType.equals(char.class.getCanonicalName()))
        && Utility.calledMethod(method, Reader.class.getCanonicalName(), "read", true));
} }

return ruleViolated;

@Override
public String getRuleText() {
    return "The result of InputStream.read() or Reader.read() must never be cast "
        + "to a byte or a character since the return type is an integer and -1 "
        + "(0xffffffff) is used to indicate end of stream. If InputStream.read() "
        + "returns 0xff or Reader.read returns 0xffff then casting to an int "
        + "will make it impossible to distinguish between end of stream and the "
        + "next byte or character."
    }

@Override
public String getRuleName() {
    return "FIO08-J. Distinguish between characters or bytes read from a stream and -1";
}

@Override
public String getRuleRecommendation() {
    return "Do not cast the result of InputStream.read() to a byte or Reader.read() to a byte."
    }

@Override
public int securityLevel() {
    return Globals.Markers.SECURITY_LEVEL_HIGH;
}

}
The text and/or code below is from the CERT website: https://www.securecoding.cert.org

Java Secure Coding Rule: IDS00-J. Prevent SQL injection

CERT Website: SQL injection vulnerabilities arise in applications where elements of a SQL query originate from an untrusted source. Without precautions, the untrusted data may maliciously alter the query, resulting in information leaks or data modification. The primary means of preventing SQL injection are sanitization and validation, which are typically implemented as parameterized queries and stored procedures.

@author Ben White (Plugin Logic), CERT (Rule Definition)
@see Java Secure Coding Rule defined by CERT: https://www.securecoding.cert.org/confluence/display/java/IDS00-J+Prevent+SQL+injection

```java
public boolean violated(ASTNode node) {
    boolean ruleViolated = false;
    MethodInvocation method;

    // Detect use of Statement.ExecuteQuery or PreparedStatement.ExecuteQuery
    if (node instanceof MethodInvocation) {
        method = (MethodInvocation) node;
        // If PreparedStatement was used then make sure that setString() was called at least once, if not then the rule is violated
        if (Utility.calledMethod(method, PreparedStatement.class.getCanonicalName(), "executeQuery")
            ruleViolated = !Utility.calledPrior(method, PreparedStatement.class.getCanonicalName(), "setString");
        // If PreparedStatement was not used then see if Statement was used
        else if (Utility.calledMethod(method, Statement.class.getCanonicalName(), "executeQuery")
            // Rule is violate if the argument was created using a compound expression
            // (anything except for a literal string)
            // In this first case there is a variable being passed to executeQuery()
            if (method.arguments().size() > 0 && method.arguments().get(0) instanceof SimpleName)
                // The argument in this case would be the query string being passed to the database
                SimpleName argument = (SimpleName) method.arguments().get(0);
                ASTNode parent = node.getParent();

                // Look for assignments to the query string in parent nodes
                while (parent != null && !ruleViolated) {
                    ASTNodeProcessor processor = new ASTNodeProcessor();
                    parent.accept(processor);
                    // Check all of the assignments
                    for(NodeNumPair assignmentPair : processor.getAssignments()) {
                        Assignment assignment = (Assignment)assignmentPair.getNode();
                        // If the assignment is for the query string and the right hand side
                        // is not a literal string then the rule has been violated
                        if (assignment.getRightHandSide().subtreeMatch(new ASTMatcher(), argument)
                            && !(assignment.getRightHandSide() instanceof StringLiteral))
                            ruleViolated = true;
                    break;
                }
            }
        }
    }
}
```
if (!ruleViolated) {
    for (NodeNumPair declarationPair : processor.getVariableDeclarationFragments()) {
        VariableDeclarationFragment declaration = (VariableDeclarationFragment) declarationPair.getNode();
        // If the variable being declared is the query string and the right hand side is not a literal string then the rule has been violated
        if (declaration.getName().subtreeMatch(new ASTMatcher(), argument) && !declaration.getInitializer().instanceof StringLiteral) {
            ruleViolated = true;
            break;
        }
    }
    parent = parent.getParent();
}
// In this case there is another type of argument being sent, the rule is violated if that argument is anything except a string literal
else
    // In this case the argument could have been a simple name
    ruleViolated = !(method.arguments().size() > 0 && method.arguments().get(0).instanceof StringLiteral);
}
return ruleViolated;

@Override
public String getRuleText() {
    return CERT Website- SQL injection vulnerabilities arise in applications where elements of a SQL query
    "originate from an untrusted source. Without precautions, the untrusted data may maliciously alter the query, resulting in information leaks or data modification.
    "The primary means of preventing SQL injection are sanitization and validation, which are typically implemented as parameterized queries and stored procedures."
}

@Override
public String getRuleName() {
    return IDS00-J. Prevent SQL injection;
}

@Override
public String getRuleRecommendation() {
    return "Do not execute SQL queries with parameters directly, use a PreparedStatement and the " + "setString() method to insert the parameters";
}

@Override
public int securityLevel() {
    returnGlobals.Markers.SECURITY_LEVEL_HIGH;
}
package edu.csus.plugin.securecodingassistant.rules;
import java.text.Normalizer;
import java.util.regex.Pattern;
import org.eclipse.jdt.core.dom.ASTNode;
import org.eclipse.jdt.core.dom.MethodInvocation;
import org.eclipse.jdt.core.dom.SimpleName;
import edu.csus.plugin.securecodingassistant.Globals;

/**
 * The text and/or code below is from the CERT website:
 * <a target="_blank" href="https://www.securecoding.cert.org">https://www.securecoding.cert.org</a>
 *
 * Java Secure Coding Rule: IDS01-J. Normalize strings before validating them
 * CERT Website: 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 <code>&lt;script&gt;</code> 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.
 * @author Ben White (Plugin Logic), CERT (Rule Definition)
 * @see Java Secure Coding Rule defined by CERT: <a target="_blank" href="https://www.securecoding.cert.org/confluence/display/java/IDS01-J+Normalize+strings+before+validating+them">IDS01-J</a>
 */
class IDS01J_NormalizeStringsBeforeValidating implements IRule {

    @Override
    public boolean violated(ASTNode node) {
        boolean ruleViolated = false;
        // Check to see if Normalizer.normalize was called. Rule is violated if Pattern.matcher was called beforehand with the same argument
        if (node instanceof MethodInvocation) {
            MethodInvocation method = (MethodInvocation) node;
            if (Utility.calledMethod(method, Normalizer.class.getCanonicalName(), "normalize") {
                SimpleName argument = null;
                if (method.arguments().get(0) instanceof SimpleName) argument = (SimpleName) method.arguments().get(0);
                ruleViolated = Utility.calledPrior(method, Pattern.class.getCanonicalName(), "matcher", argument);
            }
        }
        return ruleViolated;
    }

    @Override
    public String getRuleText() {
        return "CERT Website-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.";

@Override
public String getRuleName() {
    return "IDS01-J. Normalize strings before validating them";
}

@Override
public String getRuleRecommendation() {
    return "Using Pattern.matcher() is a great way detect harmful things like angle brackets " + "which could indicate script tags, but the normalize method MUST be called " + "before running Pattern.matcher()";
}

@Override
public int securityLevel() {
    return Globals.Markers.SECURITY_LEVEL_HIGH;
}

 IDS07J_RuntimeExecMethod.java

package edu.csus.plugin.securecodingassistant.rules;

import org.eclipse.jdt.core.dom.ASTNode;
import org.eclipse.jdt.core.dom.MethodInvocation;
import edu.csus.plugin.securecodingassistant.Globals;

/**
 * The text and/or code below is from the CERT website: <a target="_blank" href="https://www.securecoding.cert.org">https://www.securecoding.cert.org</a>
 * Java Secure Coding Rule: IDS07-J. Sanitize untrusted data passed to the <code>Runtime.exec()</code> method
 * Any command that is sent to <code>Runtime.exec()</code> must be sanitized. Rather than using <code>Runtime.exec()</code>, try some other alternatives.
 * @author Ben White (Plugin Logic), CERT (Rule Definition)
 * @see Java Secure Coding Rule defined by CERT: <a target="_blank" href="https://www.securecoding.cert.org/confluence/display/java/IDS07-J.+Sanitize+untrusted+data+passed+to+the+Runtime.exec%28%29+method">IDS07-J</a>
 */

class IDS07J_RuntimeExecMethod implements IRule {

    @Override
    public boolean violated(ASTNode node) {
        boolean ruleViolated = false;
        // Runtime.exec() would be a MethodInvocation
        if (node instanceof MethodInvocation) {
            MethodInvocation method = (MethodInvocation)node;
            ruleViolated = Utility.calledMethod(method, Runtime.class.getCanonicalName(), "exec");
        }
        return ruleViolated;
    }
}
@Override
public String getRuleText() {
    return "CERT Website-External programs are commonly invoked to perform a function " + "required by the overall system. This practice 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.";
}

@Override
public String getRuleName() {
    return "IDS07-J. Sanitize untrusted data passed to the Runtime.exec() method";
}

@Override
public String getRuleRecommendation() {
    return "Avoid using Runtime.exec()";
}

@Override
public int securityLevel() {
    return Globals.Markers.SECURITY_LEVEL_HIGH;
}
}

IDS11J_ModifyStringsBeforeValidation.java

package edu.csus.plugin.securecodingassistant.rules;
import java.util.regex.Pattern;
import org.eclipse.jdt.core.dom.ASTNode;
import org.eclipse.jdt.core.dom.MethodInvocation;
import org.eclipse.jdt.core.dom.SimpleName;
import edu.csus.plugin.securecodingassistant.Globals;

/**
 * The text and/or code below is from the CERT website: <a target="_blank" href="https://www.securecoding.cert.org">https://www.securecoding.cert.org</a>
 * CERT Website: It is important that a string not be modified after validation has occurred because doing so may allow an attacker to bypass validation. For example, a program may filter out the tags from HTML input to avoid cross-site scripting (XSS) and other vulnerabilities. If exclamation marks (!) are deleted from the input following validation, an attacker may pass the string "{@code <script>}" so that the validation check fails to detect the tag, but the subsequent removal of the exclamation mark creates a tag in the input.
 */
public class IDS11J_ModifyStringsBeforeValidation implements IRule {

    @Override
    public String getRuleText() {
        return "CERT Website-External programs are commonly invoked to perform a function " + "required by the overall system. This practice 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.";
    }

    @Override
    public String getRuleName() {
        return "IDS07-J. Sanitize untrusted data passed to the Runtime.exec() method";
    }

    @Override
    public String getRuleRecommendation() {
        return "Avoid using Runtime.exec()";
    }

    @Override
    public int securityLevel() {
        return Globals.Markers.SECURITY_LEVEL_HIGH;
    }
}
@Override
public boolean violated(ASTNode node) {
    boolean ruleViolated = false;

    // Check to see if Pattern.matcher was used and if it was make sure the string
    // wasn't modified afterwards
    if (node instanceof MethodInvocation &&
        Utility.calledMethod((MethodInvocation) node, Pattern.class.getCanonicalName(),
        "matcher") ) {
        MethodInvocation method = (MethodInvocation) node;
        SimpleName str = (SimpleName) method.arguments().get(0);

        ruleViolated = Utility.modifiedAfter(method, str);
    }

    return ruleViolated;
}

@override
public String getRuleText() {
    return "CERT Website–It is important that a string not be modified after validation
has occurred because "
    + "doing so may allow an attacker to bypass validation. For example, a program may
" + "filter out the <script> tags from HTML input to avoid cross-site scripting
(XSS) " + "and other vulnerabilities. If exclamation marks (!) are deleted from the input
" + "following validation, an attacker may pass the string "<script>" so that the
" + "validation check fails to detect the <script> tag, but the subsequent removal
of " + "the exclamation mark creates a <script> tag in the input.";
}

@override
public String getRuleName() {
    return "IDS11-J. Perform any string modifications before validation";
}

@override
public String getRuleRecommendation() {
    return "The Pattern.matcher() method should be called after all modifications to the
string."
    + " Move any string modifications so that they occur prior to Pattern.matcher().";
}

@override
public int securityLevel() {
    return Globals.Markers.SECURITY_LEVEL_HIGH;
}

LCK09J_DoNotPerformOperationsThatCanBlockWhileHoldingLock.java

package edu.csus.plugin.securecodingassistant.rules;

import java.io.Console;
import java.net.Socket;
import java.util.List;
import java.util.List;
import org.eclipse.jdt.core.dom.ASTNode;
import org.eclipse.jdt.core.dom.MethodDeclaration;
import org.eclipse.jdt.core.dom.MethodInvocation;
import org.eclipse.jdt.core.dom.Modifier;
import edu.csus.plugin.securecodingassistant.Globals;

/**
 * <p>
 * Java Secure Coding Rule: LCK09-J. Do not perform operations that can block while holding a lock.
 * </p>
 * CERT Website: Holding locks while performing time-consuming or blocking operations can severely degrade system performance and can result in starvation. Furthermore, deadlock can result if interdependent threads block indefinitely. Blocking operations include network, file, and console I/O (for example, <code>Console.readLine()</code>) and object serialization. Deferring a thread indefinitely also constitutes a blocking operation. Consequently, programs must not perform blocking operations while holding a lock.
 * </p>
 * When the Java Virtual Machine (JVM) interacts with a file system that operates over an unreliable network, avoid file I/O over the network while holding a lock. File operations (such as logging) that could block while waiting for the output stream lock or for I/O to complete could be performed in a dedicated thread to speed up task processing. Logging requests can be added to a queue, assuming that the queue's <code>put()</code> operation incurs little overhead as compared to file I/O [Goetz 2006].
 * </p>
 * @author Ben White (Plugin Logic), CERT (Rule Definition)
 * @see Java Secure Coding Rule defined by CERT: <a target="_blank" href="https://www.securecoding.cert.org/confluence/display/java/LCK09-J+Do+not+perform+operations+that+can+block+while+holding+a+lock">LCK09-J</a>
 */

class LCK09J_DoNotPerformOperationsThatCanBlockWhileHoldingLock implements IRule {

@Override
public boolean violated(ASTNode node) {
  boolean ruleViolated = false;

  // Is node a method invocation for Thread.sleep(), Socket.getOutputStream(),
  // Socket.getInputStream() or Console.readLine()?
  if (node instanceof MethodInvocation) {
    MethodInvocation method = (MethodInvocation) node;
    if (Utility.calledMethod(method, Thread.class.getCanonicalName(), "sleep")
        || Utility.calledMethod(method, Socket.class.getCanonicalName(),
        "getOutputStream")
        || Utility.calledMethod(method, Socket.class.getCanonicalName(),
        "getInputStream")
        || Utility.calledMethod(method, Console.class.getCanonicalName(), "readLine")) {

      // Is the method invocation in a method definition that is declared to be synchronized?
      ASTNode encNode = Utility.getEnclosingNode(method, MethodDeclaration.class);
      if (encNode != null && encNode instanceof MethodDeclaration) {
        MethodDeclaration methodDec = (MethodDeclaration) encNode;
        List<? extends Modifier> modList = methodDec.modifiers();
        for (Object o : modList) {
          if (o instanceof Modifier) {
            break;
          }
        }
      }
    }
  } else {
    return false;
  }

  return ruleViolated;
}
Modifier mod = (Modifier)0;
ruleViolated = (mod.getKeyword().toFlagValue() & Modifier.SYNCHRONIZED) != 0;
}
}

return ruleViolated;

@Override
public String getRuleText() {
    return "Holding locks while performing time-consuming or blocking operations can "
        + "severely degrade system performance and can result in starvation. "
        + "Furthermore, deadlock can result if interdependent threads block "
        + "indefinitely. Blocking operations include network, file, and console "
        + "I/O (for example, Console.readLine()) and object serialization. "
        + "Deferring a thread indefinitely also constitutes a blocking operation. "
        + "Consequently, programs must not perform blocking operations while "
        + "holding a lock."
;
}

@Override
public String getRuleName() {
    return "LCK09-J. Do not perform operations that can block while holding a lock"
;
}

@Override
public String getRuleRecommendation() {
    return "Do not call Thread.sleep(), Socket.getOutputStream(), Socket.getInputStream() "
        + "Stream(), or Console.readLine() from a synchronized method. Instead of "
        + "Thread.sleep(), try calling wait which immediately releases current monitor.";
}

@Override
public int securityLevel() {
    return Globals.Markers.SECURITY_LEVEL_LOW;
}
}

MET04J_DoNotIncreaseTheAccessibilityOfOverriddenMethods.java

package edu.csus.plugin.securecodingassistant.rules;
import org.eclipse.jdt.core.dom.ASTNode;
import org.eclipse.jdt.core.dom.IMethodBinding;
import org.eclipse.jdt.core.dom.MethodDeclaration;
import org.eclipse.jdt.core.dom.Modifier;
import org.eclipse.jdt.core.dom.MethodDeclaration;
import org.eclipse.jdt.core.dom.Modifier;
import org.eclipse.jdt.core.dom.Modifier;
import edu.csus.plugin.securecodingassistant.Globals;

/**
 * <b><i>The text and/or code below is from the CERT website: <a target="_blank" href="https://www.securecoding.cert.org">https://www.securecoding.cert.org</a></i></b>
 * Java Secure Coding Rule: MET04-J. Do not increase the accessibility of overridden or hidden methods
 * CERT Website: Increasing the accessibility of overridden or hidden methods permits a
* malicious subclass to offer wider access to the restricted method than was originally
* intended. Consequently, programs must override methods only when necessary and must
* declare methods final whenever possible to prevent malicious subclassing. When methods
* cannot be declared final, programs must refrain from increasing the accessibility of
* overridden methods.
* </p>
* @author Ben White (Plugin Logic), CERT (Rule Definition)
* @see Java Secure Coding Rule defined by CERT: <a target="_blank" href="https://www.securecoding.cert.org/confluence/display/java/MET04-J+Do+not+increase+the+accessibility+of+overridden+or+hidden+methods">MET04-J</a>
* /

```java
class MET04J_DoNotIncreaseTheAccessibilityOfOveriddenMethods implements IRule {
  @Override
  public boolean violated(ASTNode node) {
    boolean ruleViolated = false;
    // Is this a method declaration?
    if (node instanceof MethodDeclaration) {
      MethodDeclaration methodDec = (MethodDeclaration) node;
      // Is it overriding a method declared in the parent class?
      IMethodBinding parentMethod = Utility.getSuperClassDeclaration(methodDec.resolveBinding());
      // If there is a parent method and it isn't clone (the access level of clone is
      // supposed to be increased from protected to public)
      if (parentMethod != null && !parentMethod.getName().equals("clone")) {
        // Has the accessibility been increased?
        int parentAccessModifier = parentMethod.getModifiers() &
                        (Modifier.PROTECTED | Modifier.PUBLIC);
        int accessModifier = methodDec.getModifiers() &
                        (Modifier.PROTECTED | Modifier.PUBLIC);
        switch (parentAccessModifier) {
          case 0: // Package Private which is no modifier
            // Cannot increase to protected or public
            ruleViolated = (accessModifier & Modifier.PROTECTED) != 0
            || (accessModifier & Modifier.PUBLIC) != 0;
            break;
          case Modifier.PROTECTED:
            // Cannot increase to public
            ruleViolated = (accessModifier & Modifier.PUBLIC) != 0;
            break;
          case Modifier.PUBLIC:
            // super class is already public, cannot increase visibility
            break;
        }
      }
      return ruleViolated;
    }
  }

  @Override
  public String getRuleText() {
    return "CERT Website-Increasing the accessibility of overridden or hidden "
         + "methods permits a malicious subclass to offer wider access to "
         + "the restricted method than was originally intended. Consequently, "
         + "programs must override methods only when necessary and must declare "
         + "methods final whenever possible to prevent malicious subclassing. "
         + "When methods cannot be declared final, programs must refrain from "
         + "increasing the accessibility of overridden methods.";
  }
```
@Override
public String getRuleName() {
    return "MET04-J. Do not increase the accessibility of overridden or hidden methods";
}

@Override
public String getRuleRecommendation() {
    return "Do not increase the accessibility of overridden methods.";
}

@Override
public int securityLevel() {
    return Globals.Markers.SECURITY_LEVEL_MEDIUM;
}

MET06J_DoNotInvokeOverridableMethodsInClone.java

package edu.csus.plugin.securecodingassistant.rules;

import org.eclipse.jdt.core.dom.ASTNode;
import org.eclipse.jdt.core.dom.ITypeBinding;
import org.eclipse.jdt.core.dom.MethodDeclaration;
import org.eclipse.jdt.core.dom.MethodInvocation;
import org.eclipse.jdt.core.dom.Modifier;
import edu.csus.plugin.securecodingassistant.Globals;

/**
 * <b><i>The text and/or code below is from the CERT website: <a target="_blank" href="https://www.securecoding.cert.org"https://www.securecoding.cert.org</a></i></b>
 * <p>CERT Website: Calling <b>overridable</b> methods from the <code>clone()</code> method is insecure. First, a malicious subclass could override the method and affect the behavior of the <code>clone()</code> method. Second, a trusted subclass could observe (and potentially modify) the cloned object in a partially initialized state before its construction has concluded. In either case, the subclass could leave the clone, the object being cloned, or both in an inconsistent state. Consequently, <code>clone()</code> methods may invoke only methods that are <code>final</code> or <code>private</code>.<p>*
 * @author Ben White (Plugin Logic), CERT (Rule Definition)
 * @see Java Secure Coding Rule defined by CERT: <a target="_blank" href="https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=34668550">MET06-J</a>
 */
class MET06J_DoNotInvokeOverridableMethodsInClone implements IRule {

@Override
public boolean violated(ASTNode node) {
    boolean ruleViolated = false;

    // Is this a MethodInvocation
    if (node instanceof MethodInvocation) {
        MethodInvocation methodDec = (MethodInvocation) node;
        // Is this in a clone() method?
        ASTNode encNode = Utility.getEnclosingNode(node, MethodDeclaration.class);
        if (encNode != null && encNode instanceof MethodDeclaration) {
            MethodDeclaration methodDec = (MethodDeclaration) encNode;
            ...
        }
    }
    ...
}
if (methodDec.resolveBinding().getName().equals("clone")) {
    // Verify that the class implements cloneable
    boolean implementsCloneable = false;
    if (methodDec.resolveBinding() != null) {
        for (ITypeBinding typeBinding :
          methodDec.resolveBinding().getDeclaringClass().getInterfaces()) {
            if (typeBinding.getQualifiedName().equals(Cloneable.class.getCanonicalName())) {
                implementsCloneable = true;
                break;
            }
        }
    }
    // This is a method invocation that is in an overridden clone()
    ruleViolated = implementsCloneable &&
    (method.resolveMethodBinding().getModifiers() & Modifier.FINAL) == 0 &&
    (method.resolveMethodBinding().getModifiers() & Modifier.PRIVATE) == 0;
}
return ruleViolated;
}

@Override
public String getRuleText() {
    return "Calling overridable methods from the clone() method is insecure. "
    + "First, a malicious subclass could override the method and affect "
    + "the behavior of the clone() method. Second, a trusted subclass "
    + "could observe (and potentially modify) the cloned object in a "
    + "partially initialized state before its construction has concluded. "
    + "In either case, the subclass could leave the clone, the object "
    + "being cloned, or both in an inconsistent state. Consequently, "
    + "clone() methods may invoke only methods that are final or private.";
}

@Override
public String getRuleName() {
    return "MET06-J. Do not invoke overridable methods in clone()";
}

@Override
public String getRuleRecommendation() {
    return "When overridding the clone() method make sure that none of the methods "
    + "that are called are also overriddable.";
}

@Override
public int securityLevel() {
    return Globals.Markers SECURITY_LEVEL_HIGH;
}

MSC02J_GenerateStrongRandomNumbers.java

package edu.csus.plugin.securecodingassistant.rules;
import java.util.Random;
import org.eclipse.jdt.core.dom.ASTNode;
import org.eclipse.jdt.core.dom.ClassInstanceCreation;
import org.eclipse.jdt.core.dom.IMethodBinding;
import org.eclipse.jdt.core.dom.ITypeBinding;
import org.eclipse.jdt.core.dom.IVariableBinding;
import org.eclipse.jdt.core.dom.Type;
import org.eclipse.jdt.core.dom.TypeParameter;
import org.eclipse.jdt.core.dom.VariableDeclaration;
import org.eclipse.jdt.core.dom.IExpression;
import org.eclipse.jdt.core.dom.ILiteral;
import org.eclipse.jdt.core.dom.IFieldDeclaration;
import org.eclipse.jdt.core.dom.IFormalParameter;
import org.eclipse.jdt.core.dom.ITypeDeclaration;
import org.eclipse.jdt.core.dom.IArrayType;
import org.eclipse.jdt.core.dom.IStringLiteral;
import org.eclipse.jdt.core.dom.IMemberLookup;
The text and/or code below is from the CERT website: [https://www.securecoding.cert.org](https://www.securecoding.cert.org)

Java Secure Coding Rule: MSC02-J. Generate strong random numbers

CERT Website: Pseudorandom number generators (PRNGs) use deterministic mathematical algorithms to produce a sequence of numbers with good statistical properties. However, the sequences of numbers produced fail to achieve true randomness. PRNGs usually start with an arithmetic seed value. The algorithm uses this seed to generate an output value and a new seed, which is used to generate the next value, and so on.

The Java API provides a PRNG, the java.util.Random class. This PRNG is portable and repeatable. Consequently, two instances of the java.util.Random class that are created using the same seed will generate identical sequences of numbers in all Java implementations. Seed values are often reused on application initialization or after every system reboot. In other cases, the seed is derived from the current time obtained from the system clock. An attacker can learn the value of the seed by performing some reconnaissance on the vulnerable target and can then build a lookup table for estimating future seed values.

@author Ben White (Plugin Logic), CERT (Rule Definition)
@see [Java Secure Coding Rule defined by CERT: MSC02-J. Generate strong random numbers](https://www.securecoding.cert.org/confluence/display/java/MSC02-J+Generate+strong+random+numbers)

```java
class MSC02J_GenerateStrongRandomNumbers implements IRule {

@Override
public boolean violated(ASTNode node) {
    boolean ruleViolated = false;
    // Check to see if a Random object is being created
    if (node instanceof ClassInstanceCreation) {
        ClassInstanceCreation instance = (ClassInstanceCreation) node;
        ruleViolated = instance.getType() != null &&
                        instance.getType().resolveBinding() != null &&
                        instance.getType().resolveBinding().getQualifiedName().equals(Random.class.getCanonicalName());
    }
    return ruleViolated;
}

@Override
public String getRuleText() {
    return "CERT Website-The Java API provides a PRNG, the java.util.Random class. This PRNG is "
        + "portable and repeatable. Consequently, two instances of the java.util."
        + "Random class that are created using the same seed will generate "
        + "identical sequences of numbers in all Java implementations. Seed "
        + "values are often reused on application initialization or after every system "
        + "reboot. In other cases, the seed is derived from the current time obtained"
        + " from the system clock. An attacker can learn the value of the seed by "
        + "performing some reconnaissance on the vulnerable target and can then "
        + "build a lookup table for estimating future seed values.";
}

@Override
public String getRuleName() {
    return "MSC02-J. Generate strong random numbers";
}
```

```
public String getRuleRecommendation() {
    return "java.util.Random is not a secure random number generator, use "
    + "java.security.SecureRandom instead";
}

@Override
public int securityLevel() {
    return Globals.Markers.SECURITY_LEVEL_HIGH;
}

NUM07J_DoNotAttemptComparisonsWithNaN.java

package edu.csus.plugin.securecodingassistant.rules;

import org.eclipse.jdt.core.dom.ASTNode;
import org.eclipse.jdt.core.dom.Expression;
import org.eclipse.jdt.core.dom.InfixExpression;
import org.eclipse.jdt.core.dom.QualifiedName;
import edu.csus.plugin.securecodingassistant.Globals;

/**
 * The text and/or code below is from the CERT website: <a target="_blank"
href="https://www.securecoding.cert.org">https://www.securecoding.cert.org</a>
* CERT Website: NaN (not-a-number) is unordered, so the numerical comparison
* operators <, <=, >, and >= return false if either or both operands are NaN.
* The equality operator == returns false if either operand is NaN, and the
* inequality operator != returns true if either operand is NaN.
*/

public class NUM07J_DoNotAttemptComparisonsWithNaN implements IRule {

    @Override
    public boolean violated(ASTNode node) {
        boolean ruleViolated = false;

        // Is the node an infix boolean expression?
        if (node instanceof InfixExpression) {
            InfixExpression expression = (InfixExpression)node;
            if (expression.resolveTypeBinding() != null &
            expression.resolveTypeBinding().getName().equals("boolean")) {
                // Get left-hand-side and right-hand-side
                Expression lhs = expression.getLeftOperand(), rhs = expression.getRightOperand();
                if (lhs instanceof QualifiedName)
                    ruleViolated = isNaN((QualifiedName)lhs);
                if (rhs instanceof QualifiedName)
                    ruleViolated = ruleViolated || isNaN((QualifiedName)rhs);
            }
        }

        return ruleViolated;
    }
}
public String getRuleText() {
    return "NaN (not-a-number) is unordered, so the numerical comparison operators "
    + "<, <=, >, and >= return false if either or both operands are NaN. "
    + "The equality operator == returns false if either operand is NaN, and "
    + "the inequality operator != returns true if either operand is NaN."
}

public String getRuleName() {
    return "NUM07-J. Do not attempt comparisons with NaN";
}

public String getRuleRecommendation() {
    return "Use Double.isNaN() or Float.isNaN() instead";
}

public int securityLevel() {
    return Globals.Markers.SECURITY_LEVEL_LOW;
}

private boolean isNaN(QualifiedName name) {
    return (name.getQualifier().getFullyQualifiedName().equals("Float")
            || name.getQualifier().getFullyQualifiedName().equals("Double"))
    && name.getName().getFullyQualifiedName().equals("NaN");
}

NUM09J_DoNotUseFloatingPointAsLoopCounters.java
package edu.csus.plugin.securecodingassistant.rules;
import org.eclipse.jdt.core.dom.ASTNode;
import org.eclipse.jdt.core.dom.ForStatement;
import org.eclipse.jdt.core.dom.VariableDeclarationExpression;
import edu.csus.plugin.securecodingassistant.Globals;
/**
 * The text and/or code below is from the CERT website: <a target="_blank"
 href="https://www.securecoding.cert.org">https://www.securecoding.cert.org</a>
 * Java Secure Coding Rule: NUM09-J. Do not use floating-point variables as loop counters
 * Floating point numbers are not an appropriate loop counter since floating point
 * arithmetic does not precisely represent decimal values
 * @author Ben White (Plugin Logic), CERT (Rule Definition)
 * @see Java Secure Coding Rule defined by CERT: <a target="_blank"
 href="https://www.securecoding.cert.org/confluence/display/java[NUM09-J.Do+not+use+floating-point+variables+as+loop+counters">NUM09-J</a>
 */
class NUM09J_DoNotUseFloatingPointAsLoopCounters implements IRule {

    @Override
    public boolean violated(ASTNode node) {
        boolean ruleViolated = false;
        // Is node a for loop?
if (node instanceof ForStatement) {
    ForStatement statement = (ForStatement) node;
    for (Object o : statement.initializers()) {
        // Is the initializer a declaration?
        if (o instanceof VariableDeclarationExpression) {
            VariableDeclarationExpression dec = (VariableDeclarationExpression) o;
            if (dec.getType().resolveBinding().getName().equals("float")
                || dec.getType().resolveBinding().getName().equals("Float")
                || dec.getType().resolveBinding().getName().equals("double")
                || dec.getType().resolveBinding().getName().equals("Double")) {
                ruleViolated = true;
                break;
            }
        }
    }
    return ruleViolated;
}

@Override
public String getRuleText() {
    return "Floating point numbers are not an appropriate loop counter since floating point arithmetic does not precisely represent decimal values";
}

@Override
public String getRuleName() {
    return "NUM09-J. Do not use floating-point variables as loop counters";
}

@Override
public String getRuleRecommendation() {
    return "Implement loop using an integer counter instead.";
}

@Override
public int securityLevel() {
    return Globals.Markers.SECURITY_LEVEL_MEDIUM;
}

OBJ09J_CompareClassesAndNotClassNames.java

package edu.csus.plugin.securecodingassistant.rules;

import org.eclipse.jdt.core.dom.ASTNode;
import org.eclipse.jdt.core.dom.Expression;
import org.eclipse.jdt.core.dom.MethodInvocation;
import org.eclipse.jdt.core.dom.ConstructorInvocation;
import org.eclipse.jdt.core.dom.MethodInvocation;
import edu.csus.plugin.securecodingassistant.Globals;

/**
 * The text and/or code below is from the CERT website: <a target="blank" href="https://www.securecoding.cert.org">https://www.securecoding.cert.org</a>:
 * Java Secure Coding Rule: OBJ09-J. Compare classes and not class names
 */

CERT Website: In a Java Virtual Machine (JVM), "Two classes are the same class (and consequently the same type) if they are loaded by the same class loader and they have the same fully qualified name" [JVMSpec 1999]. Two classes with the same name but different package names are distinct, as are two classes with the same fully qualified name loaded by different class loaders.
It could be necessary to check whether a given object has a specific class type or whether two objects have the same class type associated with them, for example, when implementing the `<code>equals()</code>` method. If the comparison is performed incorrectly, the code could assume that the two objects are of the same class when they are not. As a result, class names must not be compared.

Depending on the function that the insecure code performs, it could be vulnerable to a mix-and-match attack. An attacker could supply a malicious class with the same fully qualified name as the target class. If access to a protected resource is granted based on the comparison of class names alone, the unprivileged class could gain unwarranted access to the resource.

Conversely, the assumption that two classes deriving from the same codebase are the same is error prone. Although this assumption is commonly observed to be true in desktop applications, it is typically not the case with J2EE `servlet` containers. The containers can use different class loader instances to deploy and recall applications at runtime without having to restart the JVM. In such situations, two objects whose classes come from the same codebase could appear to the JVM to be two different classes. Also note that the `<code>equals()</code>` method might not return true when comparing objects originating from the same codebase.

@see Java Secure Coding Rule defined by CERT: [OBJ09-J+Compare+classes+and+not+class+names](https://www.securecoding.cert.org/confluence/display/java/OBJ09-J+Compare+classes+and+not+class+names)

```java
@Override
public boolean violated(ASTNode node) {
    boolean ruleViolated = false;
    // Is node a method invocation
    if (node instanceof MethodInvocation) {
        MethodInvocation method = (MethodInvocation) node;
        // Check to see if String.equals() is called
        if (Utility.calledMethod(method, String.class.getCanonicalName(), "equals")) {
            // Check to see if the expression is another method invocation
            Expression exp = method.getExpression();
            if (exp != null && exp instanceof MethodInvocation) {
                MethodInvocation outerMethod = (MethodInvocation) exp;
                // Rule is violated if the outer method was Class.getName
                ruleViolated = Utility.calledMethod(outerMethod, Class.class.getCanonicalName(), "getName");
            }
        }
    }
    int.class.getName().equals(Integer.class.getName());
    return ruleViolated;
}
```

```java
@Override
public String getRuleText() {
    return "When comparing two classes for equality make sure that the classes " + "themselves are being compared and not the class names."
}
```
@Override

public String getRuleName() {
    return "OBJ09-J. Compare classes and not class names";
}

@Override

public String getRuleRecommendation() {
    return "Rather than obj.getClass().getName().equals(...), use "
    + "obj.getClass() == obj2.getClass()";
}

@Override

public int securityLevel() {
    return Globals.Markers.SECURITY_LEVEL_MEDIUM;
}

}
Does the declaring class extend SecureClassLoader?

```java
if (methodDec.resolveBinding() != null &&
    methodDec.resolveBinding().getDeclaringClass() != null &&
    methodDec.resolveBinding().getDeclaringClass().getSuperclass() != null &&
    methodDec.resolveBinding().getDeclaringClass().getSuperclass().getQualifiedName().equals(SecureClassLoader.class.getCanonicalName())) {
    ruleViolated = true; // Rule is violated if method exists without call to super

    // Is there a call to super.getPermissions within the method declaration?
    ASTNodeProcessor processor = new ASTNodeProcessor();
    node.accept(processor);
    for (NodeNumPair methodNode : processor.getSuperMethodInvocations()) {
        assert methodNode.getNode() instanceof SuperMethodInvocation;
        SuperMethodInvocation method = (SuperMethodInvocation) methodNode.getNode();
        if (method.getName().getFullyQualifiedName().equals("getPermissions")) {
            ruleViolated = false;
            break;
        }
    }

    return ruleViolated;
}

@Override
public String getRuleText() {
    return "Any custom class loader must call the parent's getPermissions() method to consult the default system policy."
};

@Override
public String getRuleName() {
    return "SEC07-J. Call the superclass's getPermissions() method when writing a custom class loader.";
}

@Override
public String getRuleRecommendation() {
    return "Call super.getPermissions() whenever overriding getPermissions."
};

@Override
public int securityLevel() {
    return Globals.Markers.SECURITY_LEVEL_HIGH;
}
```

**SER01J_DoNotDeviateFromTheProperSignaturesOfSerializationMethods.java**

```java
package edu.csus.plugin.securecodingassistant.rules;

import java.io.Serializable;
import org.eclipse.jdt.core.dom.ASTNode;
import org.eclipse.jdt.core.dom.IMethodBinding;
import org.eclipse.jdt.core.dom.ITypeBinding;
import org.eclipse.jdt.core.dom.MethodDeclaration;
import org.eclipse.jdt.core.dom.Modifier;
import edu.csus.plugin.securecodingassistant.Globals;
/**
```
* Java Secure Coding Rule: SER01-J. Do not deviate from the proper signatures of serialization methods
* CERT Website: Unlike most interfaces, <code>Serializable</code> does not define the method signatures it requires. Interfaces allow only public fields and methods, whereas <code>readObject()</code>, <code>readObjectNoData()</code>, and <code>writeObject()</code> must be declared private. Similarly, the <code>Serializable</code> interface does not prevent <code>readResolve()</code> and <code>writeReplace()</code> methods from being declared <code>static</code>, <code>public</code>, or <code>private</code>. Consequently, the Java serialization mechanism fails to let the compiler identify an incorrect method signature for any of these methods.
* The <code>writeObject()</code>, <code>readObject()</code> and <code>readObjectNoData()</code> methods must be implemented with specific signatures. They must be declared private so that extending classes cannot invoke or override them.
* @author Ben White (Plugin Logic), CERT (Rule Definition)
* @see Java Secure Coding Rule defined by CERT: <a href="https://www.securecoding.cert.org/confluence/display/java/SER01-J.+Do+not+deviate+from+the+proper+signatures+of+serialization+methods">SER01-J</a>

```java
class SER01J_DoNotDeviateFromTheProperSignaturesOfSerializationMethods implements IRule {

    @Override
    public boolean violated(ASTNode node) {
        boolean ruleViolated = false;

        // Is this a method declaration?
        if (node instanceof MethodDeclaration) {
            // Is the method declared in a class that implements Serializable?
            MethodDeclaration methodDec = (MethodDeclaration) node;
            if (methodDec.resolveBinding() != null) {
                boolean implementsSerializable = false;
                IMethodBinding methodBinding = methodDec.resolveBinding();
                for (ITypeBinding typeBinding : methodBinding.getDeclaringClass().getInterfaces()) {
                    if (typeBinding.getQualifiedName().equals(Serializable.class.getCanonicalName())) {
                        implementsSerializable = true;
                        break;
                    }
                }

                // If the method name is writeObject, readObject or readObjectNoData then it must be marked private
                if (implementsSerializable && (methodBinding.getName().equals("writeObject") || methodBinding.getName().equals("readObject") || methodBinding.getName().equals("readObjectNoData"))) {
                    ruleViolated = (methodBinding.getModifiers() & Modifier.PRIVATE) != Modifier.PRIVATE;
                }

                // If the method name is readResolve or writeReplace then it must be marked protected and cannot be marked static
                if (implementsSerializable && (methodBinding.getName().equals("readResolve") || methodBinding.getName().equals("writeReplace"))) {
                    ruleViolated = true;
                }
            }
        }
        return ruleViolated;
    }
}
```
ruleViolated = (methodBinding.getModifiers() & (Modifier.PROTECTED | Modifier.STATIC)) != Modifier.PROTECTED;

return ruleViolated;

@Override
public String getRuleText() {
    return "CERT Website-Unlike most interfaces, Serializable does not define the "
    + "method signatures it requires. Interfaces allow only public fields "
    + "and methods, whereas readObject(), readObjectNoData, and writeObject() "
    + "must be declared private. Similarly, the Serializable interface does "
    + "not prevent readResolve() and writeReplace() methods from being "
    + "declared static, public, or private. Consequently, the Java "
    + "serialization mechanism fails to let the compiler identify an "
    + "incorrect method signature for any of these methods."
}

@Override
public String getRuleName() {
    return "SER01-J. Do not deviate from the proper signatures of serialization methods";
}

@Override
public String getRuleRecommendation() {
    return "writeObject, readObject, and readObjectNoData must be marked as private. "
    + "Also, readResolve and writeReplace cannot be static and must be protected.";
}

@Override
public int securityLevel() {
    return Globals.Markers.SECURITY_LEVEL_HIGH;
}

STR00J_PartialCharFromVarWidthEnc.java

package edu.csus.plugin.securecodingassistant.rules;

import java.io.InputStream;
import org.eclipse.jdt.core.dom.ASTNode;
import org.eclipse.jdt.core.dom.MethodInvocation;
import org.eclipse.jdt.core.dom.SimpleName;
import org.eclipse.jdt.core.dom.WhileStatement;
import edu.csus.plugin.securecodingassistant.Globals;

/**
 * The text and/or code below is from the CERT website: <a target="blank" href="https://www.securecoding.cert.org">https://www.securecoding.cert.org</a>
 * Java Secure Coding Rule: STR00-J. Don't form strings containing partial characters from variable-width encodings.
 * Programmers must not form strings containing partial characters, for example, when converting variable-width encoded character data to strings. Avoid this by not building a text string until confirmed that all data has been read from the buffer.
@see Java Secure Coding Rule defined by CERT: <a target="_blank" href="https://www.securecoding.cert.org/confluence/display/java/STR00-J.+Don%27t+form+strings+containing+partial+characters+from+variable+width+encodings">STR00-J</a>

```java
class STR00J_PartialCharFromVarWidthEnc implements IRule {

    @Override
    public boolean violated(ASTNode node) {
        boolean ruleViolated = false;

        // Is data being read from an input stream? Class = InputStream, method = read.
        if (node instanceof MethodInvocation
            && Utility.calledMethod((MethodInvocation)node, InputStream.class.getCanonicalName(), "read")) {
            // TODO: What if in another type of loop?
            // check to see if in while loop
            ASTNode encNode = Utility.getEnclosingNode(node, WhileStatement.class);
            if (encNode instanceof WhileStatement) {
                WhileStatement wStmt = (WhileStatement)encNode;
                MethodInvocation method = (MethodInvocation)node;

                // Capture identifier of byte buffer
                SimpleName idBuffer = method.arguments().size() < 1 ? null : (SimpleName)method.arguments().get(0);

                // If string is being constructed in the while loop then rule violated. Calling string constructor
                // with byte buffer as the parameter
                ruleViolated = Utility.containsInstanceCreation(wStmt.getBody(), String.class.getCanonicalName(), idBuffer);
            }

            return ruleViolated;
        }

    }

    @Override
    public String getRuleText() {
        return "Programmers must not form strings containing partial characters, for example, when " + "converting variable-width encoded character data to strings."
    }

    @Override
    public String getRuleName() {
        return "STR00-J. Don't form strings containing partial characters" + " from variable-width encodings";
    }

    @Override
    public String getRuleRecommendation() {
        return "Defer building text string until all data has been read by the buffer";
    }

    @Override
    public int securityLevel() {
        return GlobalsMarkers.SECURITY_LEVEL_HIGH;
    }
}
```
The text and/or code below is from the CERT website: 

Java Secure Coding Rule: THI05-J. Do not use Thread.stop() to terminate threads.

CERT Website: Certain thread APIs were introduced to facilitate thread suspension, resumption, and termination but were later deprecated because of inherent design weaknesses. For example, the Thread.stop() method causes the thread to immediately throw a ThreadDeath exception, which usually stops the thread. More information about deprecated methods is available in MET02-J. Do not use deprecated or obsolete classes or methods.

Invoking Thread.stop() results in the release of all locks a thread has acquired, potentially exposing the objects protected by those locks when those objects are in an inconsistent state. The thread might catch the ThreadDeath exception and use a finally block in an attempt to repair the inconsistent object or objects. However, doing so requires careful inspection of all synchronized methods and blocks because a ThreadDeath exception can be thrown at any point during the thread's execution. Furthermore, code must be protected from ThreadDeath exceptions that might occur while executing catch or finally blocks [Sun 1999]. Consequently, programs must not invoke Thread.stop().

@author Ben White (Plugin Logic), CERT (Rule Definition)
@see Java Secure Coding Rule defined by CERT: THI05-J Do not use Thread.stop() to terminate threads

class THI05J_DoNotUseThreadStopToTerminateThreads implements IRule {

@Override
public boolean violated(ASTNode node) {
    boolean ruleViolated = false;
    // Check to see if Thread.stop() was called
    if (node instanceof MethodInvocation) {
        MethodInvocation method = (MethodInvocation) node;
        MethodInvocatio
"repair the inconsistent object or objects. However, doing so requires "
"Careful inspection of all synchronized methods and blocks because a "
"ThreadDeath exception can be thrown at any point during the thread's execution.
"Furthermore, code must be protected from ThreadDeath exceptions that might "
"occur while executing catch or finally blocks [Sun 1999]. Consequently, "
"programs must not invoke Thread.stop()";

```java
@Override
public String getRuleName() {
    return "THI05-J. Do not use Thread.stop() to terminate threads";
}

@Override
public String getRuleRecommendation() {
    return "Design the method to be interruptible or use a volatile flag to indicate"
    + "termination requested.";
}

@Override
public int securityLevel() {
    return Globals.Markers.SECURITY_LEVEL_LOW;
}
```
CERT Validation Source Code

Test00IDS.java

```java
package testProject;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.Statement;
import java.text.Normalizer;
import java.text.Normalizer.Form;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class Test00IDS {
    @SuppressWarnings("unused")
    public void test() throws Throwable {
        // Test IDS00J
        Connection connection = DriverManager.getConnection("testing");
        Statement stmt = connection.createStatement();
        PreparedStatement pStmt = connection.prepareStatement(null);
        pStmt.setString(1, "");

        // Uncomment rs2 or comment pstmt to test rule
        ResultSet rs = pStmt.executeQuery("testing");
        ResultSet rs2 = stmt.executeQuery("testing2");

        // Test IDS01J
        String s = "\uFE64" + "script" + "\uFE65";

        // Normalize (comment out to generate warning)
        //s = Normalizer.normalize(s, Form.NFKC);

        // Validate
        Pattern pattern = Pattern.compile("[<>]");
        Matcher matcher = pattern.matcher(s);

        // Test IDS07J
        String dir = System.getProperty("dir");
        Runtime rt = Runtime.getRuntime();

        // Comment out to remove warning
        Process proc = rt.exec("cmd.exe /C dir " + dir);
    }
}
```
// Test IDS11J
String s2 = "<script>";
s2 = Normalizer.normalize(s2, Form.NFKC);

// Look for script tag
Pattern pattern = Pattern.compile("<script>");
matcher = pattern.matcher(s2);
if (matcher.find())
    throw new IllegalArgumentException("Invalid Input");

// Delete non-character code points (move to end to generate warning)
s2 = s2.replaceAll("[\p{Cn}]", "");
}

Test01DCL.java
package testProject;
import java.util.ArrayList;
public class Test01DCL {
    @SuppressWarnings("unused")
    public void test() {
        // Test DCL02J
        ArrayList<Object> list = new ArrayList<Object>();

        // Make o "final"
        for (Object o : list) {
            ...
        }
    }
}

Test02EXP.java
package testProject;
import java.util.Arrays;
public class Test02EXP {
    @SuppressWarnings("unused")
    public void test() {
        // Test EXP00J
        String s;
        int i = 3;

        // Remove assignment to generate warning
        s = String.valueOf(i);
// Test EXP02J
int[] test = new int[20];
int[] test2 = new int[30];

boolean result = Arrays.equals(test, test2);
test.equals(test2);
}
}

Test03NUM.java

package testProject;

public class Test03NUM {

    public void test() {
        // Test NUM07J
        double x = 0.0;
        double result = Math.cos(1/x); // Returns NaN if input is infinity
        if (result == Double.NaN) { // Comparison is always false!
            System.out.println("result is NaN");
        }

        // Test NUM09J
        for (float x2 = 0.1f; x2 <= 1.0f; x2 += 0.1f) {
            System.out.println(x2);
        }
    }
}

Test04STR.java

package testProject;

import java.io.IOException;
import java.io.InputStream;
import java.net.Socket;

public class Test04STR {
    @SuppressWarnings("unused")
    public void test() throws Throwable {

        // Test STR00J
        final int MAX_SIZE = 1024;
        @SuppressWarnings("resource")
        Socket socket = new Socket();
        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;
            // Uncomment this line to generate warning
            str += new String(data, offset, data.length - offset, "UTF-8");
    }
}
if (offset >= data.length) {
    throw new IOException("Too much input");
}

// This is the proper way to do it
String str2 = new String(data, "UTF-8");
in.close();

Test05OBJ.java

package testProject;

public class Test05OBJ {

    @SuppressWarnings({"unused", "null"})
    public void test() {
        // Test OBJ09J
        Integer x = 1,
        y = 2;
        Object auth = null;
        // The following will generate a warning
        boolean r1 = x.getClass().getName().equals(y.getClass().getName());
        // The following will NOT generate a warning
        boolean r2 = x.getClass() == y.getClass();
        // The following will generate a warning
        boolean r3 =
        auth.getClass().getName().equals("com.application.auth.DefaultAuthenticationHandler");
        // The following will NOT generate a warning
        boolean r4 = auth.getClass() == Object.class;
    }
}

Super.java

package testProject;

public class Super {

    protected void doLogic() {
        System.out.println("Super invoked");
    }
}

Test06MET.java

package testProject;

public class Test06MET extends Super implements Cloneable {

    // Test MET04J
    // Make this method protected to avoid error
    public void doLogic() {
    }

    // Test MET06J
// Make the doLogic() method private or final to make the error go away

public Object clone() throws CloneNotSupportedException {
    final Test06MET clone = (Test06MET) super.clone();
    clone.doLogic();
    return clone;
}

Test07ERR.java

package testProject;

public class Test07ERR {
    public void test() {
        // Test ERR08J
        String s = "test";
        try {
            String names[] = s.split(" ");
            if (names.length != 2) {
            }
        } catch (NullPointerException e) {
        }
    }
}

Test09LCK.java

package testProject;

import java.io.Console;
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.net.Socket;

public class Test09LCK {
    public synchronized void test(Socket socket, long time) throws IOException, InterruptedException {
        // Test LCK09J
        // Make method synchronized to generate warning
        @SuppressWarnings("unused")
        ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());

        Thread.sleep(time);
        Console c = System.console();
        c.readLine();
    }
}

Test10THI.java

package testProject;
public class Test10THI implements Runnable {

@SuppressWarnings("deprecation")
public void test() throws InterruptedException {
    // Test THI05J
    Thread thread = new Thread(new Test10THI());
    thread.start();
    Thread.sleep(5000);
    thread.stop(); // This generates warning
}

@Override
public synchronized void run() {
    // TODO Auto-generated method stub
}
}

Test13FIO.java

package testProject;

import java.io.FileInputStream;
import java.io.FileReader;
import java.io.IOException;

public class Test13FIO {

@SuppressWarnings({"null", "unused"})
public void test() throws IOException {
    // Test FIO08J
    FileInputStream in = null;
    // Initialize stream
    byte data;
    while (((data = (byte) in.read()) != -1) {
        // ...
    }

    FileReader in2 = null;
    // Initialize stream
    char data2;
    while (((data2 = (char) in2.read()) != -1) {
        // ...
    }
}
}

Test14SER.java

package testProject;

import java.io.IOException;
import java.io.ObjectStreamException;
import java.io.Serializable;

public class Test14SER implements Serializable {
private static final long serialVersionUID = 1L;

// Test SER01J
// Make the following three methods anything except private
// to generate an error
private void writeObject(java.io.ObjectOutputStream out) throws IOException {}

private void readObject(java.io.ObjectInputStream in) throws IOException, ClassNotFoundException {}

@SuppressWarnings("unused")
private void readObjectNoData() throws ObjectStreamException {}

// Make the following two methods static or anything
// except protected to generate an error
protected Object readResolve() { return null; }

protected Object writeReplace() { return null; }

Test16ENV.java

package testProject;

public class Test16ENV {

@SuppressWarnings("unused")
public void test() {
    // Test ENV02J
    String username = System.getenv("USER");
}

Test49MSC.java

package testProject;

import java.security.SecureRandom;
import java.util.Random;

public class Test49MSC {

@SuppressWarnings("unused")
public void test() {
    // Test MSC02J
}
// Uncomment next line to test
Random rnd = new Random();
rnd.nextInt();
SecureRandom sRnd = new SecureRandom();
}
REFERENCES


