Tailoring PMD to Secure Coding

Markus Aderhold Artjom Kochtchi

Technische Universität Darmstadt, Germany
Modeling and Analysis of Information Systems

Technical Report TUD-CS-2013-0245

September 2013

Abstract

In this report, we investigate how PMD can be tailored to check Java code with respect to secure coding guidelines. We chose PMD among four publicly available tools for the static analysis of Java code: FindBugs, Hammurapi, Jlint, and PMD. First, we describe our selection process, which includes an overview of these four tools with a focus on their architecture, their functionality, and their intended application areas. Second, we present an implementation of a so-called rule for PMD so that Java programs can be checked with respect to two secure coding guidelines from the CERT Oracle Secure Coding Standard for Java.

Contents

1 Introduction

Secure coding guidelines describe good programming practices that support developers in improving the quality of code so that the resulting software is more secure. Determining whether some given code actually complies with secure coding guidelines easily becomes non-trivial, especially if the code consists of thousands of lines of code. Thus tool support is desirable to check code with respect to secure coding.

In this report, we investigate how determining whether some given code complies with secure coding guidelines can be supported by a publicly available tool for the static analysis of code. Since there exist hundreds of secure coding guidelines (e. g., [1, 10, 12]), we focus our investigation by choosing among the available secure coding guidelines.

As a starting point for this report, we choose the CERT Oracle Secure Coding Standard for Java [10], which is fairly recent and actively maintained. Within this standard, we choose the first secure coding guideline that targets a specific method from the Java library:

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

Our reason for choosing a guideline that targets a specific method is that we expect the analysis to be simpler if a specific method is targeted instead of methods that are characterized more abstractly. For example, some guidelines characterize methods by their functionality such as methods for "logging", "normalization", or "canonicalization"; it seems more difficult to identify calls of such methods compared to identifying calls of `Runtime.exec()`.

In order to get an impression how an analysis can be generalized to guidelines that do not target a specific method, we in addition choose the following guideline:

> IDS03-J. Do not log unsanitized user input.

In Section 2, we describe our selection of a publicly available tool as a basis to implement an analysis with respect to these two secure coding guidelines. This includes an overview of four tools (FindBugs, Hammurapi, Jlint, and PMD) with a focus on their architecture, their functionality, and their intended application areas. In Section 3, we document our implementation of an analysis with respect to these two secure coding guidelines. Section 4 concludes with some summarizing remarks.

2 Tool Selection

In this section, we give an overview of four tools: FindBugs, Hammurapi, Jlint, and PMD. These tools are listed as tools for the static analysis of Java code among several others in [13, 15]. All of these four tools are freely available, open-source, and provide a reasonable amount of documentation to work with. Other tools mentioned in [13, 15] may as well share these properties and thus may be valuable for secure coding, but we restrict ourselves to these four tools in this report.

Two of the tools (Hammurapi and PMD) analyze Java *source code* directly, while the other two tools (FindBugs and Jlint) operate on Java *bytecode*. Although the latter tools do not analyze source code, the results of analyses can be linked to source code. Thus users of all tools have the possibility to interpret the results of analyses in terms of the source code.

All tool descriptions follow the same structure, as far as the given documentation and the offered functionality allow. First, a short overview is given. The section "Architecture" then describes the architecture of the tool. The section "Functionality" contains information on what can be done with the tool. "Usage" states how users can interact with the tool, including available interfaces, and customization options. Known limitations are listed in the "Limitations" section. The section "Project" summarizes the current state of development (as at July 2012) and potential future directions. Last, "Impression" contains a subjective summary of pros and cons for each tool.

The descriptions of the tools are based on the respective documentation, including documentation in the source code of the tools. For this report, we structured the descriptions uniformly as a basis for our selection of a tool for secure coding analyses.

All tools make use of some kind of *rules* that describe certain program behavior or characteristics that are generally associated with erroneous behavior, bad style, or bad practice. The tools then analyze programs to detect patterns described by rules and report matches to the user. The terminology varies among tools, but in this report the words *rule* and *violation* are used to refer to such descriptions of erroneous program behavior and code that matches them, respectively. Accordingly, the term *ruleset* refers to a collection of rules, grouped by theme or by some other criteria.

2.1 FindBugs

FindBugs is an open-source tool for static analysis of Java bytecode and is released under LGPL. It is written in Java and comes with a stand-alone GUI. A command line interface and plugins for common IDEs are also available. It aims for the detection of bugs "that matter" and tries to advocate the usage of static analysis in software development by making it more accessible. The rules range from general bad practices to the correctness of multithreading and vulnerabilities to attacks. In addition to bug detection, a cloud service provides collaborative prioritization and discussion of bugs.

Architecture

FindBugs uses Apache Commons BCEL[1] (Byte Code Engineering Library) to analyze Java bytecode files. Using a Visitor pattern [7], FindBugs notifies all rules of the fact that an analysis has started and then steps through bytecode instructions, passing opcodes to rules for inspection.

A plugin system allows one to extend FindBugs by placing `jar` files into a `plugin` subdirectory of the installation and including the path into the configuration file. This is also a way to include custom rules into FindBugs.

[1]`http://commons.apache.org/bcel/`

Own rules (called "bug descriptors" or "bug patterns" in FindBugs) can be developed by subclassing one of FindBugs' base classes for rules. It is also possible to use implementations that use the ASM[2] framework for analysis.

The framework also contains algorithms that calculate call graphs and data flows; however, their details seem not to be documented.

Functionality

FindBugs in its stand-alone GUI version can be run directly from the website via Java Webstart or downloaded for offline use. A plugin for Eclipse is also available directly from the developers of FindBugs. A stable version, release candidates and nightly builds of the plugin can be downloaded directly or via an update site. Plugins for Netbeans, IntelliJ IDEA, Ant, and Maven are provided from third-party development teams.

Rules are provided for a variety of bugs, including rather elaborate cases where the program is analyzed across method boundaries and across class boundaries. Discovered bugs are given a rank ranging from 1 to 20 (1 being the highest concern). The ranks are grouped into rougher buckets "scariest", "scary", "troubling", and "of concern" for easy filtering.

FindBugs analyses class files, but is capable of displaying detected bugs in the source code for ease of readability, given a source code directory is provided. Results can be directly displayed in the stand-alone GUI or in the respective IDE. Also, results can be exported to XML, HTML, Emacs, and xdocs (for Apache Maven) formats.

FindBugs is distributed with many rules that are divided into eight categories. Categories in FindBugs do not necessarily group similar rules together. Instead, the categories "Bad Practice", "Correctness", and "Dodgy Code" differ in the likeliness of a rule violation constituting an actual bug. The remaining categories organize rules by theme. Roughly 400 rules are included in FindBugs and are organized into the following categories:

Correctness. Rules in this category aim to detect actual bugs and thus try to minimize false positives.

Correctness bugs include wrong type handling like casts that are guaranteed to fail, or **`instanceof`** checks that are always true or false, respectively. Also detected is **`null`** dereferencing that seems certain to occur, should control reach the code. Likely logic errors are detected where statements do not have an effect or calculation will always have the same result.

Rules also perform correctness checks which are simply not covered by the compiler: the creation of self-containing collections, invalid regular expressions and format strings, apparent infinite loops, and loss of precision when operating on numbers.

Bad programming practices are detected with regards to Java naming conventions that constitute more than bad style, like method names that are easily confused with popular Java methods and might accidentally shade them or might have been intended to override them. Implementations of `equals()` that

[2]`http://asm.ow2.org/`

appear to be irreflexive, asymmetric, or otherwise contradicting the specification of proper implementations of `equals()` are reported.

Bad Practice. Bad Practice rules detect "violations of recommended and essential coding practice" that may result in bugs, but accuracy of detection may be lower than of Correctness rules or some developers might not care about fixing bad practices as much as others.

Rules include conventions that are not compiler-checked but can easily result in errors, like wrong implementations or usage of methods from the class `Object` (`equals()` and `hashCode()`) or from special interfaces (`Comparable` and `Serializable`), wrong handling of well-known APIs like Swing, ignoring return values from methods known to not have any side effects, and inappropriate exception handling.

Some rules of a similar nature to those in the Correctness category are included, but apparently for the rules in the Bad Practice category, FindBugs is less confident in actual bugs being present; i. e., false positives are more likely.

Dodgy Code. Dodgy code is written in a confusing way and is therefore less transparent and robust. The patterns are similar to those in the Correctness and Bad Practice categories, but more false positives are accepted.

Experimental. Experimental rules use alternative implementations to rules already present in other categories, or they are new rules that remain to be thoroughly tested before recommending their use.

Malicious Code Vulnerability. Code that is vulnerable to misuse by interacting components is reported along with some advice on which idioms should be used instead. This includes the proper protection of `ClassLoader`s, passing references of mutable objects around, and usage of proper visibility modifiers as well as the **`final`** keyword.

Multithreaded Correctness. FindBugs supports elaborate rules to detect errors in synchronization and thread safety.

Apart from the detection of incorrect idioms like double checked locking, several other cases of inconsistencies in synchronization and failed attempts of synchronization are detected. Other rules identify incorrect thread behavior, especially the correct use and interaction of methods like `wait()` and `notify()`.

Performance. Rules in the Performance category do not point out actual bugs in the strict sense, but rather program behavior that is inefficient.

Examples include unnecessary boxing, unboxing, and conversion between types, explicit garbage collection, the usage of constructors for `String` or number types like `Double`. Unused fields and methods are also pointed out.

Security. Security rules detect the use of unsanitized external (and therefore probably dangerous) data for HTTP communication or in SQL statements.

Usage

For the analysis, the user provides one or more paths that contain Java class files. The analysis of archived classes in `zip`, `jar`, `ear`, and `war` files is also possible. Additionally, auxiliary paths can be given that contain classes referred to in the code under analysis. This allows reasoning over class hierarchies.

Analysis results can be saved and loaded later, hierarchically organized, filtered using different criteria, and annotated to aid the review process. Alternatively, plugins integrate display and browsing into an IDE.

FindBugs allows the user to define custom filters. They are defined using a special XML format. Apart from the usual possibility to filter by bugs and bug ranks, FindBug filters can also define elaborate requirements concerning affected packages, classes, methods, fields, and variable names.

A few properties can be passed to FindBugs on the command line to customize the analysis. For example, one parameter tells FindBugs to take assertions into consideration when determining data ranges, while another parameter tells FindBugs to consider comments in otherwise empty blocks or switch cases as valid implementations of the functionality for this case. A full list of switches is given in the documentation.[3]

Without the implementation of specific rules, the scope of the analysis can be expanded by placing annotations, such as `@NonNull`, in front of elements under inspection. FindBugs will then attempt to uncover violations of the introduced restriction. A full list of annotations is available in the documentation.[4]

Additional rules can be implemented in Java by subclassing the respective FindBugs base classes for rules. To include them into FindBugs, a `jar` file is to be created and declared as a plugin. Understanding of Java bytecode is needed for the implementation of rules.

In addition to the analysis of code, FindBugs comes with tools that perform data mining on the results of analyses. For that, a history of results can be recorded and saved. A tool named `filterBugs` and several other command line tools can then be used to crawl such recordings for interesting data.[5]

Limitations

No limitations are apparent or explicitly documented for FindBugs. A bug tracker for current issues is available on the project's Sourceforge page.[6]

Project

FindBugs was created by Bill Pugh and David Hovemeyer and is now developed and maintained at the University of Maryland by Bill Pugh and a team of volunteers.

The current version of FindBugs is version 2 and dates to December 20, 2011. All resources are available on the tool's website.[7] There is also a written documentation, along with several recordings of talks, slides, and various

[3]`http://findbugs.sourceforge.net/manual/analysisprops.html`
[4]`http://findbugs.sourceforge.net/manual/annotations.html`
[5]`http://findbugs.sourceforge.net/manual/datamining.html`
[6]`http://sourceforge.net/tracker/?group_id=96405`
[7]`http://findbugs.sourceforge.net/`

publications, although the content does not necessarily keep up with the implementation. The documentation is concise and focuses mainly on practical aspects of installing, configuring, and running FindBugs.

The project showed regular commitment in the past and continues to be developed further. The developers report having used the tool successfully to discover and to report bugs in the Java API and in Google's codebase.

Impression

FindBugs makes a very solid impression regarding precision and usefulness. It has good user interfaces and is very flexible. Its sparse technical documentation is a downside. A very interesting aspect of the tool is its direction towards productivity and practical applicability.

2.2 Hammurapi

Hammurapi is a freely available tool written in Java and released under GPL. It can be used for the analysis of Java source code (and allegedly also of any other language by providing additional parsers; only one parser for Java is provided, though). Rules can check code for certain characteristics and can also generate metrics. The tool is developed with enterprise scale applications in mind. Hence, it offers integration into several development phases, from IDE and build integration to the distribution of analysis results over the network.

Architecture

Hammurapi is built in a modular fashion, dividing the program into the core API, language modules for parsing, inspectors (can be roughly understood as rules), reporters (for rendering), and a number of auxiliary libraries.

The tool includes two other libraries by the Hammurapi Group, namely Mesopotamia[8] for parsing Java source code, and Hammurapi Rules,[9] a JSR-94-compatible[10] engine responsible for the formulation of rules and for the inference of non-compliance.

Java source code is parsed into an abstract syntax tree (AST) that is later converted into a heterogeneous tree that represents actual Java constructs. This representation is used by inspectors to detect problems. Inspectors may also generate metrics or calculate and set intermediate results that can be accessed by other inspectors. This process is called "chaining".

Rules can implement simple checks on the AST, but they can also be used to infer facts about the program. Inferred facts can, in turn, be used by other rules for a more detailed or for a more precise analysis. Rules can be developed in Java and plugged into Hammurapi. Thorough documentation with examples is part of the Hammurapi distribution.

[8] `http://www.hammurapi.biz/hammurapi-biz/ef/xmenu/hammurapi-group/mesopotamia/index.html`

[9] `http://www.hammurapi.biz/hammurapi-biz/ef/xmenu/hammurapi-group/products/hammurapi-rules/index.html`

[10] Java Rule Engine API, see `http://www.jcp.org/en/jsr/detail?id=94`

Functionality

Hammurapi is aimed at enterprise-scale development to provide code quality baselines. While developers can use IDE plugins to have their code checked at development time, Ant Tasks also check compliance in the version control systems. Projects leads and quality assurance teams can provide rules and modify rules as requirements develop or change.

Additionally, intermediate results (e. g., parsed files or review results) are saved into databases and can be distributed easily. Hammurapi includes web services that are capable of providing results for display and download. Different components of Hammurapi can be installed on different machines.

Hammurapi comes with a set of 96 predefined rules. These rules are organized into 8 categories:

Legal. There are two rules in the Legal category, which check for the presence of copyright information in every source file and, in an outsourcing scenario, verify that the developing organization does not put its name into the source code.

Exception Handling. Rules for Exception Handling check that the code does not throw exceptions that are too general, that the **`catch`** block is never empty, that the **`throws`** clause is not too long, and that re-thrown exceptions are properly constructed. It is also possible to provide a list of approved exceptions that are permitted to appear in the **`throws`** clause.

Coding Guidelines. Coding guidelines check for compliance with coding standards such as naming conventions and conventions on the order of modifiers, or they impose hard limits on the maximum number of literals for numbers, literals for strings, parameters to a method, lines of code per file, depth of block nesting, and on cyclomatic complexity. They further check that package declarations are present and that no unnecessary imports exist. Furthermore, code is detected that can be expressed in a shorter way. This includes comparisons with boolean literals, unnecessary braces, and empty blocks.

Other rules detect bad choices of classes; e. g., `Vector` instead of other collections, or `StringBuffer` in single-threaded applications instead of `StringBuilder`. Some rules encourage to use collections instead of arrays in general.

There are some rules that might be interpreted to be controversial. For example, the use of Java's ternary operator `?:` is discouraged and **`for`** expressions need to contain all three parts (initialization, condition, and update).

Threading and Synchronization. Rules of this category check rather superficially for bad style in concurrent programming: Classes extending `Thread` should implement `run()`, `synchronize` should be used at block rather than at method level, `notifyAll()` should be preferred to `notify()`, `Thread.yield()` should not be used at all, and `wait()` should be called only inside loops.

Logging. The Logging category contains exactly one rule that states that `System.out` and `System.err` should not be used for logging.

Documentation. There is one rule in the Documentation category that checks the correctness and completeness of JavaDoc comments.

Potential Bugs. Rules from the Potential Bugs category indicate typical sources of bugs; e. g., **`switch`** statements that do not contain **`break`**s or lack a **`default`** case, comparison of objects by `==` and `!=` instead of `equals`, or the invocation of an abstract method from the constructor (which can result in subclasses working on incompletely constructed objects).

Another set of rules requires the programmer to avoid shadowing superclass fields, reassigning formal parameters (that should best be declared **`final`**), or performing assignments inside conditionals.

When overriding `equals()`, `hashCode()` should be overridden as well, and vice versa and implementations of `clone()` should invoke `super.clone()`.

Performance. Some operations in Java are possible, but discouraged due to performance reasons. Such cases include the construction of `BigDecimals` with the values 0 and 1 (they are already predefined constants in `BigDecimal`) and the explicit construction of `String`s and `Boolean`s. Also, it should be avoided to call `System.gc()` explicitly.

One rule detects operations on immutables that ignore the return value. Although this is a performance error in that such an operation is void, it is more likely a programming error.

Usage

Developers can check their projects by running Hammurapi on their projects from within Eclipse. Rulesets can be created by providing an XML configuration file. As for version 5, there is no way to ignore or set aside specific violation reports. Developers can create additional rules by implementing them in Java. In a similar fashion, any desired output format can be achieved by implementing a custom reporter.

Limitations

The documentation of version 5.6.0 documents known limitations:

- Not all cases of generic types are resolved correctly, i. e., objects are sometimes considered to be of type `Object` instead of their actual (known) type.
- There are difficulties handling *vararg* arguments.
- The API for extracting the usage of types is incomplete.

Project

Hammurapi is developed by the developed by the Hammurapi Group. The current stable version 5.7.0 is available from the old website[11], and an experimental version 6.3.0 is available from the new website.[12] Version 6, however, is in development stadium.

[11]`http://hammurapi.biz/`
[12]`http://hammurapi.com/`

All necessary components are documented to be packed into an Eclipse update site. While this works as promised for an update site included in the distributable of version 5.7.0, we could not observe any effects on the Eclipse menu or on the Eclipse functionality when installing from the 6.3.0 update site. Documentation for Hammurapi 5 can be found in the Hammurapi folder after installation.

Hammurapi Rules, responsible for validation, is in the process of being replaced by a system called Event Bus. Hammurapi Rules is still being used in Hammurapi 5, though. In the future, Hammurapi is going to be rebuilt using the Whole Brain Programming approach,[13] therefore Hammurapi 6 might have been discontinued already. Although information on future directions is available, there is no clear indication of whether the project is active or when it will receive updates.

Impression

Hammurapi is a very complex tool. Probably, this is due to the intended use in a distributed enterprise development environment. This complexity leads to a lot of overhead for extension and configuration, where extensive XML files need to be created to configure the components.

The documentation is rich and includes many technical and conceptual details. However, sometimes the documentation neglects the practical aspects of using Hammurapi.

2.3 Jlint

Jlint is a free command line tool released under GPL v2.0. It is written in C++ and analyzes Java bytecode for common programming errors. It is built upon AntiC, an analyzer for C and C++, relying in parts on Java's C heritage and extending the analysis by reasoning over Java semantics to check more complex and more specific Java rules. Debug information present in class files is used to report detected violations with source files and line numbers. Results are written into text format and can be viewed directly or using Emacs (the output follows Emacs' default format to encode file names, line numbers and messages). A third-party Eclipse plugin is available from Sourceforge.[14]

Architecture

Jlint consists of two parts: firstly, the Jlint core, a semantic analyzer for Java bytecode, and secondly, AntiC, a program that checks Java sources for C-style programming errors.[15]

While AntiC performs checks on syntax level, Jlint relies on local and global data flow analyses to determine and to reason over possible values of variable. It also builds a call graph to uncover race conditions.

[13] `http://doc.hammurapi.com/dokuwiki/doku.php?id=products:whole_brain_programming:start`

[14] `http://sourceforge.net/projects/jlinteclipse/`

[15] The term "C-style programming errors" means errors that Java shares with C/C++ and that result from certain language idiosyncracies, such as the accidental comparison of Strings with == and unintended fallthrough in `switch` blocks due to missing `break` statements.

In the project, two files (`antic.c` and `jlint.cc`) account for the majority of the program logic and mostly consist of nested `if-else`-expressions. It is therefore hard to identify what Jlint actually does. Rulesets are hardcoded into these files and hence not easily extendible or configurable apart from the on/off-switches that Jlint takes as parameters.

Functionality

Jlint can be used to check Java programs (although AntiC can also check C/C++), by providing a directory or class file archives as parameters to a command line call. A list of 52 checks performed by AntiC and Jlint can be found in the Jlint manual:

AntiC. AntiC is used to detect source code that is syntactically correct, but may have different semantics from what the programmer had in mind. All detected problems can be grouped into three categories.

Suspicious Literals. Literals are declared suspicious when they look unintended. For example, using octal representation of characters in strings is limited to certain digits: `"\127"` is correct and yields a `W`, while `"\128"` yields a new line feed and the character 8, because 128 is not a valid octal digit, but the prefix 12 is.

Other errors include the usage of unknown control sequences in strings (like `"\x"`) and multi-byte characters (like `'ab'`), which are actually compile time errors in Java. Also, the use of `l` (lowercase L) in variable names is reported, because it is easily confused with and often even indistinguishable from `1` (one).

Operator Priorities. Arithmetic expressions that omit braces are considered potentially ambiguous when operator priorities are non-intuitive. For example, `(1 << 2 - 1)`, equals `(1 << (2 - 1))`, but `(1 << 2 | 1)`, though similar, equals `((1 << 2) | 1)`.

Blocks and Statement Bodies. Regarding blocks, there are two main cases that often lead to errors. For many statements, braces around the body may be omitted when the body consists of a single expression. When indentation indicates that the programmer intended a block to span several statements, but the actual block is shorter, there are probably braces missing. The second case is unintentional fallthrough due to a missing `break` statement in `case` clauses of a `switch` statement.

Jlint Core. The Jlint core performs a semantic analysis to determine possible bugs in synchronization, inheritance, and data flow.

Synchronization. Java's multi-threading is built upon shared data and locks. Access to shared data has to be synchronized among accessing threads to ensure thread-safe execution. In practice, this has proven to be difficult and error-prone, especially because correct synchronization is not checked by the compiler and because bugs tend to be non-deterministic.

Erroneous synchronization can result in race conditions (where data is concurrently modified by different threads in an unsafe manner) and in deadlocks (where two threads block program execution, because they wait for each other to complete). Jlint builds call-graphs and tries to detect unsynchronized access to shared data and call sequences that may result in deadlock situations.

Inheritance. Subclasses can hide members of their superclasses by using the same name (shadowing) or method signature (overriding). In Java, no additional effort (e.g., no special modifier) is needed and errors may easily result by using existing names or by providing incorrect method signatures.

Jlint detects cases where local variables are accessed and where it is likely that the programmer meant shadowed members instead. For overriding, Jlint finds methods that use the same name as a method in a superclass, but that do not override the superclass member, because the method signature differs. It also makes sure that `finalize` calls include a call to `super.finalize()` as documented in the Java API Specification [6].

Data Flow. During data flow analysis, domains of possible values for variables are calculated and used to uncover likely errors. For example, cases where variables are always `null`, conditions always yield the same result (due to possible ranges of operands), and instructions may lead to overflow or type truncation,[16] are considered erroneous.

Usage

Jlint is called from the command line by passing it class file directories or archives. Some parameters can be provided to influence Jlint's behavior. It is possible to enable and to disable specific error messages or whole categories of errors by specifying error codes or category names on the command line. It is furthermore possible to specify a file with previous results that will then not be reported again.

A programmer has possibilities to influence Jlint's output concerning `switch`-statements. By placing a comment containing "break" (as in "no break") or "fall" (as in "fall through"), Jlint's messages about possibly missing `break` statements are suppressed.

Defining additional rules is not possible easily due to the lack of modularity. For development, the CVS repository contains a small set of black box tests that check Jlint's output on one simple test class.

Limitations

Users cannot easily define custom rules, because all checks are hard-wired into Jlint and it would be necessary to directly change Jlint code.

Jlint builds a call graph to detect issues with multi-threading. This is done iteratively and the number of iterations is fixed to limit calculation complexity. Synchronization problems that lie beyond this limit are not detected.

Several checks performed by AntiC and Jlint do not apply to Java or are already checked by the Java compiler.

[16]Type truncation is a form of data loss that occurs when values are cast to smaller types, see `http://cwe.mitre.org/data/definitions/197.html`.

Project

Jlint was originally developed by Konstantin Knizhnik and later extended by Cyrille Artho. Several other authors contributed to the project.

The current version 3.1.2 is available from Sourceforge.[17] There is also a Jlint website,[18] but it appears outdated (with Jlint version 3.0 and older documents than their Sourceforge counterparts). The current documentation can be found in the Jlint distributable and is dated to 2004.

The overall project activity is unclear. While the last CVS activity on 11.01.2011 appears to be fairly recent, it is preceded by many years of silence. There are no signs of future directions or of active developers.

Impression

Jlint has a rather narrow scope and is not easily extendible. Therefore, it may be of limited use for more specific requirements. While the AntiC rules are mostly covered by IDEs and the Java Compiler, Jlint's remaining rules are solid and useful. The tool is not exceptionally comfortable and not very modular. Being written in C and C++, it is not easily accessible to Java programmers for inspection or changes.

2.4 PMD

PMD is free software released under a "BSD-style license".[19] It is written in Java and performs checks on Java source code. Violations of rules are presented as plain text or written into HTML or XML files. There is a number of plugins for various IDEs that equip PMD with a graphical user interface.[20] The tool promises to check code for possible bugs and to identify dead, suboptimal, or overly complicated code as well as code repetition.

Architecture

PMD employs a JavaCC[21] parser that constructs abstract syntax trees (AST) from Java sources. The main loop of PMD then examines the AST, visiting all rules that have registered to be interested in certain AST constructs. The rules can then check the AST and report violations.

PMD offers a data flow analysis that can be used by the rules. The results of the data flow analysis are given by a graph structure with information on definition, undefinition, and redefinition of local variables.

Rules are provided as Java classes or can alternatively be given by an XPath query.[22] The documentation of PMD states that rule contributions have to be tested for correctness and performance against a large code base like the JDK source and need to pass the so-called "dogfood" ruleset provided with the PMD sources.

[17]http://sourceforge.net/projects/jlint/

[18]http://artho.com/jlint/

[19]http://pmd.sourceforge.net/license.html

[20]http://pmd.sourceforge.net/integrations.html

[21]http://javacc.java.net/

[22]For an example of an XPath query, see http://pmd.sourceforge.net/xpathruletutorial.html.

Functionality

In addition to checking code for compliance with code quality rules, PMD also includes a duplicate code detector that helps to identify code that should be extracted to methods in order to avoid copy-and-paste errors and to increase maintainability.

There are several rules that feature the detection of bugs and that check code for compliance with well-known as well as with controversial best practices, including conventions, code style and design principles (taken, for example, from the Java API Specification [6] or from books like "Effective Java" [3]).

The program distributable ships roughly 250 predefined rules that are listed together with a short description on the project's website. Reported problems are classified on a scale from 1 ("change absolutely required") to 5 ("change highly optional") in order to help users to prioritize.

The rules are organized into several categories. There are also rules specific to Android, J2EE, JSF and JSP, JUnit, Jakarta Commons Logging, and for migration to new Java versions. The more general rule categories are the following:

Basic. Basic rules detect code fragments that should generally be avoided. Examples are empty blocks, overriding `equals()` but not `hashCode()` (and vice versa), use of double checked locking, and a number of cases of unnecessary code (that can be shortened). Other rules detect operations that have no effect at all and are therefore likely to be a programming error.

There are also rules regarding **`null`**-checks (e. g., variable is used before it is checked for **`null`**) that, however, only identify obvious violations where the misplaced use of a variable is in the same line as the check for **`null`**.

Braces. Rules for braces state that blocks following **`if`**, **`else`**, **`for`**, and **`while`** should use braces.

Clone Implementation. In Java, the correct implementation of the interface `Cloneable` and the `clone()` method is defined by convention and not enforced by the compiler. Clone implementation rules check if the conventions to implement the `Cloneable` interface have been met.

Code Size. Code size rules include rules that find code of high complexity. Examples are overly long classes, methods or parameter lists, many (public) fields or methods, and code that has a high number of decision points or execution paths.

Controversial. Controversial rules are ones that are disputed within the Java community. PMD features 19 such rules, e. g., that each method should have only one **`return`**, that the **`volatile`** keyword and the **`short`** type should not be used, or that variables should not be assigned **`null`**. Usually, at least some vague reasoning behind the rules is given, although the descriptions of rules are generally rather short.

Coupling. Coupling is the degree to which components are interdependent. Generally, "loose" coupling is preferred. Rules for coupling check the number of fields contained in classes, the number of imports, and the usage of interfaces instead of concrete types (i. e., declaring variables to be of type `Set` instead of `HashSet` etc.).

Design. Design rules in PMD comprise a mixture of many rules of the form "do/don't do this" or "x should/shouldn't be y". The types of rules range from rules that encourage certain programming style to rules that detect plain programming errors.

For example, one rule instructs the user to avoid synchronization at method level. This helps synchronizing only parts of the code that actually need synchronization. This way, when adding code to that method, developers can decide to put it inside or outside the synchronized block. Other rules detect and propose fields that can be declared static. A number of rules aims to encourage keeping code simple. For example, when checking for **`null`** and then performing a check with **`instanceof`**, the code can be shortened to just the **`instanceof`** test, since it returns false when the parameter is **`null`**.

Some rules detect code that is likely to be incorrect. For example, a class with a private constructor and no static methods or fields is unusable, comparisons to `Double.NaN` are always false, and idempotent assignments have no effects.

The description of all rules in this section is beyond the scope of this report. However, it should be noted that the resemblance to the Basic rules is high.

Finalizer. Like with `clone()`, implementing `finalize()` bears some pitfalls and is subject to conventions that are not enforced by the compiler. Corresponding rules are grouped together in the this category. If implemented, the method `finalize()` should be **`protected`**, have no arguments, include a call **`super`**`.finalize()`, and otherwise never be called explicitly.

Import Statement Rules. This category contains some rules for importing that are generally also being followed by common IDEs. Imports should not be duplicated, `java.lang` should not be imported, and unused imports should be removed. Also, too many static imports should be avoided.

JavaBean. A JavaBean is a class that fulfils certain requirements for automatic instantiation. It has to provide a parameterless public constructor, getters and setters for fields and be serializable. Details can be found in the JavaBean API Specification [9]. PMD JavaBean rules check that classes are serializable. This includes the presence of a parameterless constructor and all required getters and setters.

Java Logging. Logging rules encourage users to use loggers instead of directly printing messages and dumping stack traces to the standard output. Only one logger per class should be present and be declared **`static`** and **`final`**.

Naming. Rules in this category describe naming convention listed in the the Code Conventions for the Java Programming Language [8].

Optimization. Rules in the Optimization category deal with code that can rewritten to be more efficient. This includes making variables **`final`** where it is possible, avoid object instantiation in loops, using `StringBuffer`s for String construction and some specific idioms that are to be preferred to others.

Strict Exception. Some patterns where the consensus is that they are generally to be avoided can be checked using PMD's Strict Exception rules. The rules state to avoid raw exception types (`Exception` and `Throwable`), extending `Error`, throwing exceptions within a **`finally`** block, using exceptions for control flow, and catching `NullPointerExceptions`.

String and StringBuffer. String rules detect String usage in an unnecessary or wasteful manner. Most notably, repetition of String literals and inefficient String construction is discouraged. There are also several rules that suggest idioms that are to be preferred to others for the sake of performance.

Security Code Guidelines. The rules in this category check code with respect to two security guidelines from Sun. The two rules concern arrays: Methods should not return internal arrays, and arrays passed from outside should be defensively copied before they are stored.

Unused Code. Unused code rules detect the presence of unused fields, local variables, private methods, and formal parameters.

Usage

PMD can be used from the command line or using a plugin for an IDE. In the latter case, reported errors are shown (with respective error messages) directly in the source code.

The analysis can be customized by providing rulesets in XML. It is possible to include and to exclude certain rules and whole categories of rules. For actual use, the PMD website advises to start with a small set and include additional rules according to one's needs. PMD also takes a parameter called `minimumpriority` that makes PMD exclude all rules below a given priority.

When reviewing violations, programmers may put annotations to prevent PMD from reporting on certain issues again. This is done by placing an annotation `@SuppressWarnings("PMD")` in front of blocks to be left out or `@SuppressWarnings("PMD.ClassName")` to ignore certain rules. PMD also obeys the JDK `@SuppressWarnings("unused")` annotation and skips lines that contain a *`//NOPMD`* comment (a marker that can be redefined at will).

Additional rules can be created by implementing them (in Java or XPath) and plugging them into PMD's execution loop.

Limitations

PMD can detect and report bugs found by matching AST patterns on source code. However, when analyzing source code, PMD provides only very limited type information to the rules; in fact, the types of objects often are not determined by PMD. This makes it difficult to implement rules in PMD that refer to the types of objects.

Project

PMD is developed by InfoEther Inc. The current version 4.2.5 has been released in February 2008. It is available from the tool's Sourceforge page.[23] PMD is thoroughly documented in a book named "PMD Applied" [4] ("the official manual"), which is available for purchase. A brief documentation on installation, usage, and predefined rules is available online.

According to the web page, version 5.0 is currently being worked on and will introduce better support for cross-file and for data flow analysis, have messy code parts cleaned up, and include new libraries and library versions. The version control system does indeed show regular commits, so the project appears to be active.

Impression

Overall, PMD leaves a good impression of an active, high-quality community project. The code quality of the rules meets the project's goal to pass its "dogfood" ruleset. In fact, all but one rule do pass it. The integration into several IDEs makes it seamlessly interact with the development process.

The categorization of rules is sometimes hard to comprehend; e. g., rules from "Basic", "Controversial", and "Design" categories are difficult to tell apart. Several rules from categories other than "Controversial" could as well be regarded as controversial. For example, the rule `ReturnEmptyArrayRatherThanNull` from the "Design" category states that it is a better behavior to return an empty array rather than **`null`**. It is hard to see how this can be regarded as a general rule, placed in the same category as rules that detect actual programming errors.

Running the "dogfood" ruleset on the complete PMD source produces 265 problems. Checking for compliance with the complete ruleset yields 19,152 problems, which illustrates the need of smaller rulesets. 6,759 of the reported problems alone are due to rules that require method parameters and local read-only variables to be declared **`final`**.

That being said, PMD should be thought of as a framework for rules (to be developed, if necessary) rather than a fixed set of rules to be used out-of-the box. New rules can be created by implementing Java classes, so there is no need to learn any domain specific language, although it may pose a challenge to work on Java ASTs due to their complexity.

2.5 Choosing a Tool

None of the tools from the previous sections addresses the secure coding guidelines IDS03-J and IDS07-J from [10]. Thus our goal is now to choose a tool as a basis to implement new rules for these guidelines.

For choosing a tool, we use the following criteria:

1. The architecture of the tool shall be open so that additional rules can be added easily.

2. The tool shall be actively maintained so that there is hope that it remains up to date (e. g., concerning changes in the Java programming language).

[23] `http://pmd.sourceforge.net/`

```java
// ...
String dir = System.getProperty("dir");
Runtime rt = Runtime.getRuntime();
Process proc = rt.exec(new String[] {"sh", "-c", "ls␣" + dir});
// ...
```

Listing 1: Example of non-compliance in IDS07-J (shortened) [10]

PMD satisfies these criteria. In addition, PMD's built-in data flow analysis is appealing, because both IDS03-J and IDS07-J address flows of (un)sanitized data. Note that our choice of PMD does not imply that other tools are less suited for analyses with respect to these two secure coding guidelines. Instead, our choice is intended to provide a reasonable basis for the implementation of analyses with respect to IDS03-J and IDS07-J.

3 Secure Coding with PMD

In this section, we document our implementation of a rule in PMD that checks Java code with respect to the secure coding guidelines IDS03-J and IDS07-J from [10]. Our goal is to obtain an implementation that avoids false negatives, i. e., cases where the guidelines are violated, but no violations are reported. In turn, we accept the possibility of false positives, i. e., cases where violations are reported, but which actually are not violations of the guidelines. This conservative approach shall ensure that all violations are reliably detected.

As outlined in Section 1, we address IDS07-J first before generalizing our implementation to IDS03-J.

3.1 Introduction to IDS07-J

The secure coding guideline IDS07-J is intended to avoid some command injection hazards from data that originates from untrusted sources and is not validated or sanitized before it is used as an argument to critical subsystems. The rule is part of a set of rules that are concerned with "Input Validation and Data Sanitization" and reads as follows:

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

Each Java application has access to a singleton `Runtime` object that is obtained via `Runtime.getRuntime()`. This object can be used to interface with the execution environment. In particular, it provides a method `exec()` that executes commands on the operating system.

According to the rule, no untrusted, unsanitized data should be passed to `Runtime.exec()`. That is, "[a]ny string data that originates from outside the program's trust boundary must be sanitized before being executed as a command on the current platform." [10]

The example in Listing 1 shows non-compliant code that is susceptible to command injection attacks. In this example, a value from the environment is stored in the variable `dir` and then used as an argument to a shell command

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

Listing 2: Example of compliance with IDS07-J: sanitization [10]

without prior inspection. By providing a specially formed argument, the attacker may exploit this to execute arbitrary commands. For example, using the value `dummy & echo bad` for `dir`, the command executed is

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

which is actually two commands, where the contents of the (possibly non-existent) folder `dummy` are listed and then `bad` is printed to the console. In the same manner, any command can be appended to the `ls` command.

IDS07-J suggests three possibilities for compliant solutions:

Sanitization. This possibility prevents command injection by passing only sanitized Strings to `Runtime.exec()`, for example by whitelisting allowed characters (excluding spaces, ampersands, etc.). The example in Listing 2 uses a regular expression to allow input that consists of alphanumeric characters as well as the characters `@` ("at") and `.` ("dot").

Restricted User Choice. Giving the user a choice between a fixed set of possible (trusted) arguments solves the problem of command injection. An example of how this can be achieved is shown in Listing 3. The code still obtains a value from the environment, but ensures by a call to `Integer.parseInt()` that it is an integer. Subsequently, the value is used to select between one of two possible values for `dir`: `"data1"` and `"data2"`, which are given as literals in the program.

Avoid `Runtime.exec()`. When the task performed by a system command can be accomplished by some other means, it is almost always advisable to do so. Hence, a compliant solution for the examples given above would be to use Java's `File.list()` on directories instead of calling `ls` through `Runtime.exec()` as in the example in Listing 4.

3.2 Elaboration on IDS07-J

Before implementing a PMD rule for IDS07-J we introduce some terminology and explain how we intend to apply static analysis to detect violations of the guideline.

Terminology. IDS07-J refers to "untrusted" and to "unsanitized" input or data. "Untrusted data", in the sense of IDS07-J, can be understood as strings "originating from outside the program's trust boundary", while "unsanitized data" refers to data that has not been sanitized. More specifically, we distinguish between sanitized and unsanitized *strings* in the context of IDS07-J, because `Runtime.exec()` takes only strings as parameters.

```java
// ...
String dir = null;

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

Listing 3: Example of compliance with IDS07-J: restricted user choice [10]

```java
File dir = new File(System.getProperty("dir"));
if (!dir.isDirectory()) {
  System.out.println("Not␣a␣directory");
} else {
  for (String file : dir.list()) {
    System.out.println(file);
  }
}
```

Listing 4: Example of compliance with IDS07-J: avoidance (shortened) [10]

Static analysis. We aim for an analysis on the method level. We consider literals within the method as sanitized, because literals constitute constants defined by the programmer and should therefore never be malformed or malicious. All strings originating from outside the method are treated as unsanitized unless we have information that such a string is sanitized. More precisely, we determine the sanitization state of expressions of type `String` as follows:

1. Literals are sanitized.
2. Return values of methods are unsanitized unless the method is annotated to return sanitized values; see Subsection 3.3 for details.
3. Formal parameters of the method are unsanitized.
4. Fields of objects and classes are unsanitized.
5. The sanitization states of local variables are determined by a separate analysis that is described in Subsection 3.3.
6. An expression `e1 + e2` is sanitized iff `e1` and `e2` are sanitized.

Strictly speaking, a string concatenation `e1 + e2` should be treated as unsanitized, exactly as if the string concatenation was computed by calling `concat()`; after all, the concatenation of two sanitized strings could form a malicious string. However, we deviate from this very conservative point of view in order to prevent the rule from producing too many false positives.

For each occurrence of a call of `Runtime.exec()`, the analysis needs to determine whether its arguments are sanitized or not. Building upon the sanitization state of expressions as defined above, IDS07-J can be restated to say that no unsanitized expressions should be passed to the `Runtime.exec()` method.

Note that our approach correctly handles the compliant possibilities of restricted user choice and avoidance of `Runtime.exec()` calls. In addition, we also account for possibilities to sanitize unsanitized strings (via special annotations, to be introduced in the following subsection) in order to cover all three possibilities of compliance suggested by IDS07-J.

3.3 Implementation of a Rule for IDS07-J

According to the discussion in the previous subsection, our implementation has three basic elements:

- For the programmer, a means to annotate the program to signal the sanitization of unsanitized strings.
- For the program analysis, a data sanitization analysis that determines the sanitization state of local variables.
- For the program analysis, the actual rule implementation that finds calls to `Runtime.exec()` in the program and uses the results of the data sanitization analysis to detect and to report rule violations.

```
1 @Sanitization
2 private String sanitize(String input) throws IOException {
3   if (!Pattern.matches("[0-9A-Za-z@.]+", input)) {
4     // Handle error
5     throw new IOException();
6   }
7   return input;
8 }
```
Listing 5: Implementing sanitization

```
dir = sanitize(dir);
// ...
```
Listing 6: Using sanitization

Annotations for Sanitization

IDS07-J suggests the sanitization of data from untrusted sources. The code example in Listing 2 shows one possibility by matching the input against a pattern that whitelists permitted characters in `dir`. As a result, if `dir` is unsanitized before the pattern match, it is intended to be sanitized once the program proceeds past the `if`-block.

Our implementation accounts for this possibility by introducing an annotation called `@Sanitization` that can be utilized to annotate sanitization methods. For such methods, the analysis considers the return value as sanitized.

Putting this into practice means placing input validation (such as the code from the example) into a method annotated by `@Sanitization`. The result may look as displayed in Listing 5 (filling in the parts that are left out in the example in Listing 2). Note that the lines 3–6 in Listing 2 correspond to lines 1–3 from Listing 2. Line 5 ensures that control reaches the `return` statement only if the data is valid. The annotation in line 1 signals that the return value of the method is sanitized.

In order to sanitize the value in `dir` before a call of `Runtime.exec()`, the value can now be sanitized by passing it to the sanitization method and by overwriting it with the return value of the sanitization method, see Listing 6.

Actually, the form of sanitization proposed in the compliance example from IDS07-J is rather an input validation. This manifests itself in the fact that the method `sanitize()` in Listing 5 returns either exactly the input value (if it is valid) or nothing at all. Sanitization methods as presented here are capable of such detection of malformed input as well as of actual data sanitization (when input and return value to the method are not necessarily equal). An example for this is given in Subsection 3.5.

Data Sanitization Analysis

A central element of the rule implementation is the determination of the sanitization state of expressions as described in Section 3.2. The sanitization state of expressions generally depends on the sanitization states of the variables that

```java
// ...
String dir = null;
if (Boolean.valueOf(System.getProperty("selectDir"))) {
  dir = System.getProperty("dir");
} else {
  dir = "default";
}
Runtime rt = Runtime.getRuntime();
Process proc = rt.exec(new String[] {"sh", "-c", "ls␣" + dir});
// ...
```

Listing 7: Code example for Data Sanitization Analysis

occur. We compute the sanitization states of variables by a dedicated *Data Sanitization Analysis*.

The Data Sanitization Analysis is similar to the Available Expressions Analysis in [11]. The purpose of the Available Expressions Analysis is to determine "[f]or each program point, which expressions must have already been computed, and not later modified, on all paths to the program point." For the Data Sanitization Analysis, we are instead interested in variables that have been assigned sanitized values, and not later modified, on all paths to the program point.

PMD provides a control graph representation of the program, where every statement is represented as a node and edges connect nodes when control might pass from one statement to another. As an example, consider Listing 7 and its graph representation in Figure 1.

Adopting the notation from [11], we identify every node by a unique label that we give as a superscript to the statement it represents. For every such label $\ell \in \mathbf{Lab}_\star$ (where $\mathbf{Lab}_\star$ is the set of all labels in the program), we calculate sets $DS_{entry}(\ell) \subseteq \mathbf{Var}_\star$ and $DS_{exit}(\ell) \subseteq \mathbf{Var}_\star$ (where $\mathbf{Var}_\star$ is the set of all local variables). These sets comprise those variables that definitely contain sanitized values before or after the execution of the statement identified by ℓ, respectively.

The sets $DS_{entry}(\ell)$ and $DS_{exit}(\ell)$ are computed using a forward analysis (i. e., following the edges of the control flow graph) as follows:

- The sets are initialized with $DS_{entry}(\ell) = DS_{exit}(\ell) = \mathbf{Var}_\star$ for all labels $\ell \in \mathbf{Lab}_\star$.
- When updating the sets $DS_{entry}(\ell)$ and $DS_{exit}(\ell)$ for some $\ell \in \mathbf{Lab}_\star$, $DS_{entry}(\ell)$ is set to the intersection of the exit sets of all its predecessor nodes.

  For example, consider $\ell = 5$. As can be seen in Figure 1, control might pass from nodes identified by 3 and 4. If $DS_{exit}(3) = \emptyset$ (because `dir` has been assigned an unsanitized value) and $DS_{exit}(4) = \{\texttt{dir}\}$ (because `dir` has been assigned a literal, i. e., a sanitized value), the data sanitization analysis computes $DS_{entry}(5) = DS_{exit}(3) \cap DS_{exit}(4) = \emptyset$.
- After determining the value for $DS_{entry}(\ell)$, $DS_{exit}(\ell)$ is computed based on $DS_{entry}(\ell)$ as follows: If the statement denoted by ℓ is an assignment to some variable v, then v is added to $DS_{exit}(\ell)$ if the expression on the right-hand side of the assignment is sanitized; if the expression on the right-hand

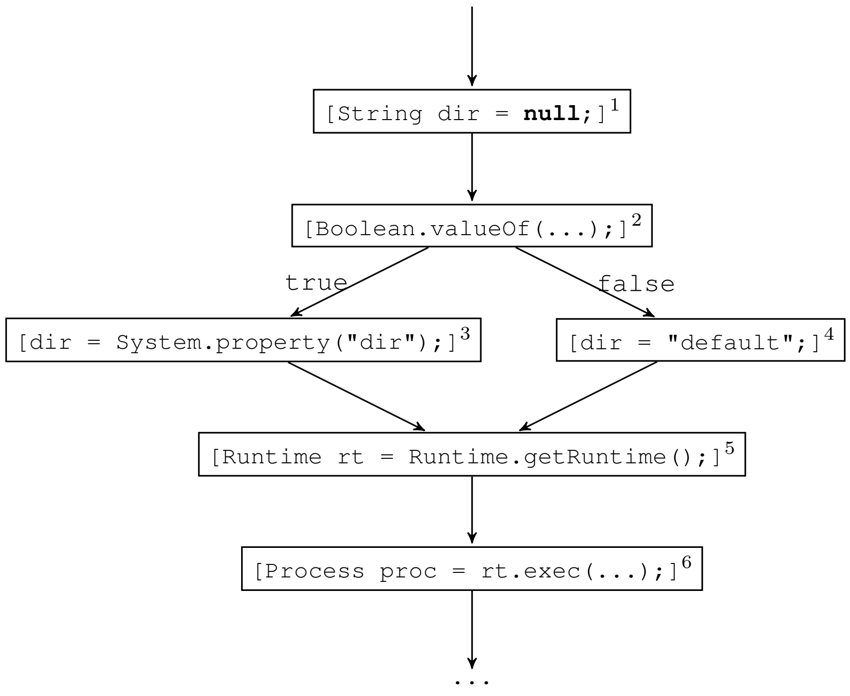
[String dir = null;]1
[Boolean.valueOf(...);]2
true
false
[dir = System.property("dir");]3
[dir = "default";]4
[Runtime rt = Runtime.getRuntime();]5
[Process proc = rt.exec(...);]6
...

Figure 1: Control flow graph for the code in Listing 7

side of the assignment is unsanitized, then v is removed from $DS_{exit}(\ell)$. If the statement denoted by ℓ is not an assignment, then $DS_{exit}(\ell)$ is equal to $DS_{entry}(\ell)$.

For example, consider $\ell = 2$ in the program represented by Figure 1. $DS_{entry}(2) = \{\texttt{dir}\}$. Because there is no assignment in the statement, $DS_{exit}(2) = DS_{entry}(2) = \{\texttt{dir}\}$, i. e., `dir` is still sanitized. On the other hand, $DS_{exit}(3) = DS_{entry}(3) \setminus \{\texttt{dir}\} = \emptyset$, because `dir` appears on the left hand side of an assignment and is assigned an unsanitized value. For $\ell = 4$, `dir` is not removed from the set, because it is assigned a sanitized value.

Note that the data sanitization considers only local variables. Other identifiers (e. g., identifiers of fields or of parameters) are never present in the sets $DS_{entry}(\ell)$ and $DS_{exit}(\ell)$ and thus are always treated as unsanitized.

A Rule for PMD

PMD parses the Java program and creates an abstract syntax tree (AST). Rules in PMD are Visitors [7] that implement a `visit()` method. The type of its first parameter identifies elements from the AST that it visits. It means that during a PMD run, all AST elements of a certain type are passed to the rule's `visit()` method. AST elements are themselves ASTs (sub-trees of the original tree) and rules can traverse them to discover violations and post them to PMD using a dedicated context object (also passed to the `visit()` method). At the end of a run, PMD reports all violations that have been posted by the rules to the user.

Our implementation works on method level, i. e., the rule visits all AST elements that represent method declarations. In the `visit()` method, a Data Sanitization Analysis is performed, all calls of `Runtime.exec()` are found, and a violation of IDS07-J is reported iff the argument of `Runtime.exec()` is unsanitized.

3.4 Generalization to IDS03-J

So far, we have focused on implementing a rule for IDS07-J. Now we generalize our implementation to IDS03-J.

In fact, our implementation is capable of addressing a broader range of cases where strings from untrusted sources are passed to critical subsystems, by generalizing from `Runtime.exec()` to user-defined sinks.

We introduce a configuration file in which the methods that are to be considered as critical subsystems are listed. For example, for IDS07-J a configuration file can be used that contains a single entry:

```
java.lang.Runtime#exec
```

For other secure coding guidelines, this file can be changed to contain other or more methods that should not receive unsanitized data. For IDS03-J, such a critical method is `Logger.severe()`.

Unfortunately, PMD's capability to infer types from Java sources files is limited, because static type inference for objects is generally a difficult task. Therefore, calls of `Logger.severe()` might be missed if for some call `o.severe()`, PMD cannot infer that object `o` is of type `Logger`. However, if methods have distinctive names (as is common practice in Java), one can more generally look for all calls of a method `severe()`, regardless of the inferred type of the object. We therefore allow users to define sinks by method name only. For example, using a configuration file with the entries

```
#warning
#info
#severe
```

is useful to catch calls to a logger, because those are common names for logging methods and are unlikely to appear anywhere else. Even if they do appear in some other class, then our analysis is conservative in identifying more potentially critical method calls.

3.5 Example of a Data Sanitization

A case where sanitization is broadly practised is the handling of HTML tags in forums and other internet services where textual user input is accepted, saved, and displayed [2]. Users that (on purpose or by accident) enter HTML tags into their texts could affect the markup structure of the website that the texts are embedded into.

Let us consider a simple example where a user comment is displayed within a `<div>` element (see Listing 8). The user can input text that contains HTML tags to create an HTML structure that was not intended. For example, inserting the input

```
<div class="userComment">
<!-- user input goes here -->
</div>
```

Listing 8: Input sanitization: HTML template for user comments

```
<div class="userComment">
<script type="text/javascript">alert("Bad!");</script>
</div>
```

Listing 9: Input sanitization: HTML with unsanitized user input

```
<script type="text/javascript">alert("Bad!");</script>
```

into Listing 8 yields Listing 9. Nothing will be displayed in the comment block. Instead, the text enclosed in the `<script>` tags ends up being executed as JavaScript. In a similar manner, it is also possible to change the document in unintended ways.

This problem can be avoided by proper input sanitization, meaning that HTML tags are removed altogether or by replacing all HTML special characters by their escape sequences.

The implementation of a possible sanitization method is given in Listing 10. It is a minimal example that replaces all occurrences of angle brackets by their HTML escape sequences. This simple treatment of user input is insufficient for real applications, but enough to prevent the insertion of JavaScript code that was possible without sanitizing the input.

Note that Listing 10 significantly differs from Listing 5: While the method in Listing 5 either returns the exact input or nothing at all (validation), the method in Listing 10 can produce result values different from the input (sanitization) and will always return unless the input is `null`. For our example input that contains HTML tags, the return value is:

```
&lt;script type="text/javascript"&gt;
alert("Bad!");
&lt;/script&gt;
```

Including the sanitized input into a comment block does no longer execute the code. Instead, it is displayed as text.

In our implementation of the rule, we do not distinguish between validation and sanitization; due to the `@Sanitization` annotation, our implementation is able to recognize the method in Listing 10 as a sanitization method similarly to the method in Listing 5.

3.6 Case Study

To test our implementation, we applied it to the source code of Kunagi.[24] Kunagi is a free and open-source project management tool. We used version 0.23.2 from August 15, 2012. The program consists of about 30,000 lines of code.

[24]http://www.kunagi.org/

```
@Sanitization
private String sanitize(String input) {
  String result = input.replaceAll("<", "&lt;");
  String result = input.replaceAll(">", "&gt;");
  return result;
}
```

Listing 10: Input sanitization: escaping special HTML characters

		True	False
Outcome	Positive	97	16
	Negative	18	20

Table 1: Evaluation results

In practice, there are hardly any calls of `Runtime.exec()` in Java code. For Kunagi, the analysis with respect to IDS07-J succeeded trivially, because there are no calls of `Runtime.exec()` in the code. Therefore, we subsequently focus on the results when checking Kunagi with respect to IDS03-J. Specifically, we detected potentially unsanitized input to the logger methods `warn()`, `info()`, `error()`, and `fatal()`.

Our implementation finds 115 critical sinks, 84 of which are reported to violate it. In total, 113 violations are reported (because there are calls with more than one unsanitized argument). From a quick glance, roughly 43 violations are true positives with a high confidence that sanitization should be performed (because the parameters contain user input of some kind). Another 35 violations report objects that originate from within the system. In practice they may or may not contain unsanitized data. 19 cases report the logging of exceptions that can, but are unlikely to contain unsanitized data. In sum, 97 of 113 reported violations constitute true positives of a different degree of confidence.

Of the remaining reported violations, 7 actually are no violations of IDS03-J. The remaining 9 reported violations are duplicates (the implementation needs to be adjusted to not report such cases). In sum, 16 of 113 reported violations are false positives.

There are 18 cases of true negatives (i. e., sinks that are correctly not reported at all) and 20 cases of false negatives. The false negatives reveal that our implementation misses some violations of the guideline: Apparently, PMD treats constructors specially and, in particular, separately from methods. Furthermore, references to **`this`** seem to be treated specially and, in particular, are not considered as references to variables. In one case, we observed that a block for exception handling was ignored; without further insights into the details of PMD's analysis, we could not find out why this happened and how this could be avoided.

The results are summarized in Table 1. From the high number of true positives, we conclude that our implementation in PMD can be used to effectively identify violations of the secure coding guideline IDS03-J. In order to be able to verify compliance with IDS03-J, some further insights into PMD's internal workings is needed so that false negatives are avoided.

```
public final class SanitizedArguments {

  public void listDir() {
    list("sanitized");
  }

  private void list(String dir) {
    Runtime rt = Runtime.getRuntime();
    Process proc = rt.exec(new String[] {"sh", "-c", "ls␣" + dir});
    // ...
  }
}
```

Listing 11: Sanitized method arguments

3.7 Future Work

Critical Sinks. Critical sinks are currently given as pairs of a class and a method or just as method names. It might be useful to extend this to point out the exact parameters that are of interest. Also, it is currently not possible to derive the type a method is called on in most cases. This is due to PMD's limited capability to infer types from the Java sources (because type information is sparse).

Constants. The implementation assumes that all return values of method calls and all fields contain unsanitized data. This is too strict, because methods that always return literals or final fields that are assigned literals on initialization are not unsanitized in the sense of the guideline. It is generally possible to statically determine the presence of such cases, which could be done to reduce the number of false positives.

String Operations. As stated in Subsection 3.2, we treat string concatenation to produce sanitized results if sanitized Strings are concatenated. We do this to prevent the proliferation of false positives, because string concatenation is very common and would return sanitized variables to an unsanitized state very often. The best example is the concatenation of two literals. There are several other cases where we could have relaxed the rule further, for example when calling `String.concat()` or using `StringBuffer` and `StringBuilder`.

Summarizing, it can be said that a more detailed analysis is needed to properly differentiate between actual rule violation and safe string operations.

Literal Arguments. If methods are known to be called only with sanitized data, the arguments are not necessarily unsanitized. Consider Listing 11 as a small example. The `final` keyword in front of the class prevents the class from being subclassed. The private method `list()` is called only from the public method `listDir()` with a string literal. Hence, `dir` is always sanitized within `list()`, but will be treated as unsanitized in our implementation, because it is a method argument. Detecting such cases would further reduce the presence of false positives.

```
1 if (!Pattern.matches("[0-9A-Za-z@.]+", dir)) {
2   throw new IOException();
3 }
4 Runtime rt = Runtime.getRuntime();
5 Process proc = rt.exec(new String[] {"sh", "-c", "ls␣" + dir});
6 // ...
```

Listing 12: Implicit sanitization

Implicit Sanitization. Implicit data sanitization is not recognized; e. g., a call to `Runtime.exec()` in some program branch that is only reached when input data is valid, but not through means discussed in Subsections 3.2 and 3.3. Consider, for example, the code from Listing 12 as an instance of the compliant example in Listing 2 from Subsection 3.3.

Here, control flow only reaches line 5 if the value of `dir` is sanitized, because it passes the validation criteria in lines 1–3 or an exception is thrown. Such cases will be reported as false positives by the Data Sanitization Analysis and need to be made explicit by moving them to sanitization methods annotated with `@Sanitization` like shown in Listings 5 and 6.

It is possible to implement some means of implicit sanitization, for example following approaches like in Perl's Taint Mode[25] [14].

Java Complexity. PMD provides Java abstract syntax trees and the rule traverses those trees to find assignments, methods calls and other relevant structures. As Java is a very complex language, it is difficult to make sure that all cases are covered. To ensure this, more work and more testing would be required.

The tests were limited to 9 test cases showing typical situations in which programs might (legally or illegally) call `Runtime.exec()`. They cover all cases shown in the IDS07-J specification and some others, but are by no means an exhaustive coverage of all possible forms of a Java programs.

Reflections. Reflections are not taken into account. They could be used to obfuscate method calls under inspection that the rule would not recognize as such.

4 Conclusion

In this report, we investigated how a tool for the static analysis of Java code can be tailored to check programs with respect to secure coding guidelines. Regarding the tool, we chose PMD as a basis to implement a so-called rule that checks Java programs with respect to the two secure coding rules IDS03-J and IDS07-J. We chose PMD because it provides an open architecture that allows one to implement and to add additional rules for the analysis. Furthermore, PMD is actively maintained and provides basic functionality that would otherwise be tedious and error-prone to implement: It parses the program and passes

[25] `http://perldoc.perl.org/perlsec.html#Laundering-and-Detecting-Tainted-Data`

abstract syntax trees of the methods of the program to the rules for analysis. Also, PMD's built-in data flow analysis proved helpful to implement our Data Sanitization Analysis for IDS03-J and IDS07-J.

Within a case study, we applied PMD to a Java program with about 30,000 lines of code so that the Java program was checked with respect to the rule we implemented. The analysis identified a large number of violations of IDS03-J. Unfortunately, some violations of IDS03-J were missed by our implementation in PMD. We believe that these false negatives can be eliminated by delving more into the internal workings of PMD and its representation of abstract syntax trees. Overall, our results are promising and indicate that the implementation of rules to check programs with respect to secure coding guidelines may help to achieve compliance with secure coding standards.

As future work, it would be interesting to investigate how other tools for the static analysis of Java code can be tailored to check programs with respect to secure coding guidelines. Depending on the infrastructure that is offered by other tools, this might significantly influence the precision of the analysis.

Acknowledgments

This work was partially funded by the DFG (German Research Foundation) within the Computer Science Action Program in the project FM-SecEng (MA 3326/1-3). We thank Heiko Mantel for helpful comments on a previous version of this report.

References

[1] Apple Inc. Secure coding guide, 2008. Online available at `http://developer.apple.com/documentation/Security/Conceptual/SecureCodingGuide/SecureCodingGuide.pdf`.

[2] Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *21st IEEE Computer Security Foundations Symposium*, 2008.

[3] Joshua Bloch. *Effective Java (2nd Edition).* Prentice Hall, 2008.

[4] Tom Copeland. *PMD Applied.* Centennial Books, 2005.

[5] Oracle Corporation. *Secure Coding Guidelines for the Java Programming Language, Version 4.0.* `http://www.oracle.com/technetwork/java/seccodeguide-139067.html`, accessed 06.08.2012.

[6] Oracle Corporation. *Java Platform, Standard Edition 6 API Specification*, 2011. `http://docs.oracle.com/javase/6/docs/api/`.

[7] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[8] Sun Microsystems Inc. *Code Conventions for the Java Programming Language*, 1997. `http://www.oracle.com/technetwork/java/codeconventions-150003.pdf`.

[9] Sun Microsystems Inc. *JavaBeans API Specification*, 1997. `http://download.oracle.com/otndocs/jcp/7224-javabeans-1.01-fr-spec-oth-JSpec/`.

[10] Fred Long, Dhruv Mohindra, Robert C. Seacord, Dean F. Sutherland, and David Svoboda. *The CERT Oracle Secure Coding Standard for Java*. Addison-Wesley, 2011. `http://www.securecoding.cert.org/confluence/display/java/`.

[11] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[12] Robert C. Seacord. *The CERT C Secure Coding Standard*. SEI Series in Software Engineering. Addison-Wesley, 2008.

[13] Carnegie Mellon University. Static Source Code Analysis Tools, 2012. `http://www.cert.org/secure-coding/tools.html`, accessed 08.08.2012.

[14] Larry Wall et al. *Perl Language Reference Manual – For Perl Version 5.12.1*. Network Theory Limited, 2010. `http://www.network-theory.co.uk/docs/perlref/`.

[15] Wikipedia. List of static code analysis — Wikipedia, The Free Encyclopedia, 2012. `http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis#Java`, accessed 08.08.2012.