Writing Secure Java Code:

A Taxonomy of Heuristics and an Evaluation of Static Analysis Tools

Michael Shawn Ware

A thesis submitted to the Graduate Faculty of

JAMES MADISION UNIVERSITY

In

Partial Fulfillment of the Requirements

for the degree of

Master of Science

Department of Computer Science

May 2008

ACKNOWLEDGMENTS

This work would not have been possible without the help of many different people. I thank my committee chair, Dr. Christopher Fox, for accepting to undertake this project, guiding me through the process of writing a master's thesis, and providing constant encouragement along the way. Dr. Fox and I had numerous thoughtful conversations that greatly improved the structure and content of my thesis. I thank the other two members of my committee, Dr. David Bernstein and Dr. Michael Norton, for their comments, advice, and time in reviewing this work. Whenever I had a question about a Java-related topic, I knew that I could count on Dr. Bernstein to provide clarification. I also thank Professor Sam Redwine for taking time to review my work and providing suggestions and ideas. Finally, I extend my thanks to the people who work at Fortify Software, Inc., especially Brian Chess and Taylor McKinley, for granting James Madison University an educational license for Fortify SCA.

TABLE OF CONTENTS

LIST OF TABLES v
LIST OF FIGURES vi
ABSTRACT vii
1 INTRODUCTION 1
1.1 Coding Standards and Static Analysis 1
1.2 A New Taxonomy Rooted in Design Principles 2
1.3 A Static Analysis Study 3
1.4 Scope and Roadmap 3
2 EXISTING DESIGN PRINCIPLES, CODING PRACTICES, AND TAXONOMIES 6
2.1 Writing Secure Code 6
2.2 Principles and Practices 8
2.3 Taxonomies 13
3 JAVA SECURITY 21
3.1 Evolution of Java Platform Security 21
3.2 Guidelines for Writing Secure Code in Java 30
3.2.1 Securing Classes 31
3.2.2 Securing Packages 35
3.2.3 Securing Inheritance 35
3.2.4 Securing Serialization 38
3.2.5 Securing Deserialization 38
3.2.6 Securing Native Methods 39
3.2.7 Securing Synchronization 40
3.2.8 Securing Privileged Code 40
3.2.9 Securing Sensitive Standard API Calls 42
3.2.10 Securing Reflection 42
3.2.11 Securing Errors and Exceptions 43
3.2.12 Securing Objects in Transit 43
3.2.13 Securing Dynamically Loaded Code 44
3.2.14 Securing Mechanisms that Provide Security 45
3.3 Summary 46
4 A NEW TAXONOMY OF DESIGN PRINCIPLES AND HEURISTICS 47
4.1 Motivation 47
4.2 Definitions 47
4.3 Methodology 48
4.4 Design Principles 50
4.5 The New Taxonomy 57
4.6 Discussion 64
5 A STATIC ANALYSIS STUDY 69
5.1 Background 69
5.2 Materials 70
5.3 Methods 71
5.4 Results 72
5.5 Discussion 77
5.6 Related Studies 85

6 CONCLUSION..88
7 BIBLIOGRAPHY 90

LIST OF TABLES

Table 1: Classifying the SQL injection vulnerability 66
Table 2: Tools included in the static analysis study 70
Table 3: Summary of static analysis study results 73
Table 4: Individual coding heuristic violations identified by each tool 76
Table 5: Violations of coding heuristics found only by one tool 84

LIST OF FIGURES

Figure 1: Taxonomy of security design principles by Benzel et al. [8] 10
Figure 2: Explicitly prevent cloning 32
Figure 3: Clone-like functionality in a factory method 32
Figure 4: Initialization block alternative 34
Figure 5: Trusted subclassing 36
Figure 6: Finalizer idiom 37
Figure 7: Using private lock objects for synchronization 40
Figure 8: Common invocation of AccessController.doPrivileged 41
Figure 9: PrivilegedAction idiom 41
Figure 10: Taxonomy levels of abstraction 48
Figure 11: Design principles and their relationships 49
Figure 12: SecurityManager check in a constructor but not in clone (CMe.1.b) 77
Figure 13: Invoking clone on an instance of a non-final class 78
Figure 14: Lack of input validation on data passed to privileged code 79
Figure 15: Storing a password in a String 80
Figure 16: Use of a public lock variable 81
Figure 17: Use of untrusted input with ProcessBuilder 82

ABSTRACT

The software security community is currently emphasizing the development of secure coding standards and their automated enforcement using static analysis techniques. Unlike languages such as C and C++, a secure coding standard for the Java programming language does not exist. In this thesis, a comprehensive collection of coding heuristics for writing secure code in Java SE 6 are organized into a taxonomy according to the design principles they help to achieve. By mapping secure coding heuristics to design principles, the goal is to help developers become more aware of the quality and security-related design problems that arise when specific coding heuristics are violated. The taxonomy's design-driven methodology also aims to make understanding, applying, and remembering both design principles and coding heuristics easier. To determine how well the collection of secure coding heuristics can be enforced using static analysis techniques, eight tools are subjected to 72 test cases that comprise a total of 115 distinct coding heuristic violations. A significant number of serious violations, some of which make attacks possible, were not identified by any tool. Even if all of the tools were combined into a single tool, more than half of the violations included in the study would not be identified.

1 Introduction

Vulnerabilities are software weaknesses that can be exploited by an attacker to compromise the security of a system. The exploitation of a vulnerability can lead to unauthorized access to information, unauthorized modification of data, unauthorized use of services, and other kinds of security breaches. Statistics from the Computer Emergency Response Team Coordination Center (CERT/CC) indicate that software is being deployed with an increasing number of vulnerabilities: 3,780 were reported in 2004, 5,990 in 2005, and 8,064 in 2006 [36].

To ameliorate the problem, the Software Engineering Institute (SEI) has created the CERT Secure Coding Initiative for developing secure coding standards [37], international standards bodies are working to provide language-independent guidance for avoiding vulnerabilities [38], and MITRE Corporation is aiming to increase communication about all kinds of software weaknesses with its Common Weaknesses Enumeration (CWE) community effort [26].

1.1 Coding Standards and Static Analysis

Coding standards play a key role in guiding the development of secure software systems [3, 41, 44]. Developers can use static analysis tools to enforce coding standards during all phases of construction. Static analysis is appealing because defects can be identified early and fixed prior to deployment. Recently, new classification schemes [26, 46, 48] for organizing security-related defects and their causes have emerged with the goal of improving the performance of static analysis tools while making developers more aware of insecure coding practices.

Recent initiatives have placed an emphasis on developing standards for languages that are highly susceptible to errors and vulnerabilities such as C and C++. Although Java is

inherently safer and more secure than C and C++, Java also has features and application programming interfaces (APIs) that can be used in an insecure manner. Yet, a secure coding standard for Java does not exist, and best practices for avoiding security-related problems in Java are often mentioned only within vulnerability taxonomies and large collections of software weaknesses. Furthermore, while numerous tools exist for statically scanning Java code, there is a lack of empirical evidence showing how well these tools detect problems [69].

1.2 A New Taxonomy Rooted in Design Principles

Previous and ongoing efforts, such as the CWE, have focused on categorizing security-related defects and the types of coding errors that cause them. While these efforts have produced useful information, they have a shortcoming: they fail to explain how insecure coding practices affect the overall design of software components. Focusing entirely on known vulnerabilities and other software weaknesses also tends to ignore important development goals for high-quality software, such as striving for simplicity and writing understandable code, which can be addressed by coding standards.

In this work, a new approach is described: coding heuristics that aim to increase the security and quality of Java code are correlated with the design principles they help to achieve. A design-driven approach can help illuminate how code quality and security degrade when coding rules are violated. In a more positive light, it can help explain why following specific coding rules increases the quality and security characteristics of code.

I argue that the new taxonomy is beneficial to developers striving to understand and construct secure software in Java for the following reasons:

- It has both theoretical and practical importance.
- Its methodology makes design principles and their associated coding rules easier to understand, apply, and remember.
- It lays the groundwork for producing a secure coding standard for the Java programming language.

1.3 A Static Analysis Study

To help contribute to other efforts that are exploring the use of static analysis techniques for security [69], eight different static analysis tools for Java are evaluated to determine how well they are able to identify violations of secure coding heuristics that are included in the new taxonomy. Each tool is subjected to a total of 115 distinct violations of coding heuristics to answer an overarching question: are static analysis tools effective at enforcing heuristics for writing secure code in Java?

Results indicate the following:

1. Even if all eight tools were combined into a single tool, over half of the violations included in the study would not have been identified.
2. A number of serious violations, some of which make attacks possible, were not identified by any tool.

To the author's knowledge, this study is one of the first to report on how well static analysis tools can enforce a wide variety of secure coding heuristics in Java.

1.4 Scope and Roadmap

This work focuses on the Java Platform, Standard Edition (Java SE) 6 release from Sun Microsystems [56]. The author assumes that the reader has a working knowledge of the Java programming language. For demonstrative purposes, secure coding heuristics for the Java Platform, Enterprise Edition (Java EE) 5 release [57] are included within the new

taxonomy. This work, however, should only be considered a comprehensive study of the core aspects of Java SE 6.

Chapter 2 discusses existing design principles, secure coding practices, and taxonomies. In Section 2.1, the phrase "writing secure code" is defined to establish the context in which secure code should be considered. To understand how techniques for writing secure code can be correlated with design principles, principles of design are outlined in Section 2.2. In Section 2.3, existing taxonomies for categorizing vulnerabilities and the coding errors that cause them are surveyed.

The goal of Chapter 3 is to cover all security-relevant issues of the Java platform that should be addressed by a secure coding standard. An overview of how Java platform security has evolved through the years is provided in Section 3.1. In Section 3.2, numerous guidelines for achieving secure code in Java are described in detail.

Chapter 4 proposes a new taxonomy of design principles and heuristics for writing secure code in Java. Section 4.1 discusses the goal of the taxonomy while Section 4.2 lays out key definitions. Section 4.3 describes its methodology, which considers three levels of abstraction. Design principles are defined and discussed in Section 4.4. The complete taxonomy is presented in Section 4.5 with a discussion of its advantages and limitations following in Section 4.6.

Chapter 5 describes a study that subjects eight static analysis tools (Checkstyle, Eclipse TPTP, FindBugs[1], Fortify[2] SCA, Jlint, Lint4j, PMD, and QJ-Pro) to 115 distinct violations of secure coding heuristics. Section 5.1 describes why static analysis is useful and important, Section 5.2 outlines the tools that are included in the study, and Section 5.3 describes how the study was performed. Results are presented in Section 5.4 and a

[1] FindBugs is a registered trademark of The University of Maryland.
[2] Fortify is a registered trademark of Fortify Software, Inc.

discussion is provided in Section 5.5. Related static analysis studies in Java are surveyed in Section 5.6.

2 Existing Design Principles, Coding Practices, and Taxonomies

2.1 Writing Secure Code

The activity of writing code occurs during many phases of any software engineering process. Defects in code that manifest as vulnerabilities can allow attackers to circumvent protection mechanisms that are provided by a system's security architecture [1]. The consequences of defects that have an impact on security can be drastic. In the most severe cases, vulnerabilities may lead to the abuse of user or system privileges that allows unauthorized access or modification to sensitive information and resources [2].

2.1.1 Definition of Secure Code

To determine how to prevent software defects that jeopardize security, the meaning of writing secure code must first be understood. Bishop [4] describes an informal process for developing programs that enforce security policies. His methodology for achieving "program security" involves the following:

1. Establishing requirements and policy to handle threats.
2. Devising a design that achieves needed security services.
3. Implementing access control on design modules.
4. Applying common management and programming rules to avoid security holes.
5. Testing and distributing securely.

Bishop's methodology contains low-assurance techniques that help to "reduce vulnerabilities and improve both the quality and the security of code" [4]. Parts (3) and (4) of Bishop's methodology relate to writing code. Part (1) corresponds to the requirements phase of the software development life cycle (SDLC), part (2) corresponds to design, and part (5) corresponds to testing.

Howard and Lipner [5] define secure code as robust code that is designed to withstand attack by malicious attackers. Like Bishop's view, it is assumed that software will be attacked, and that secure code will offer resistance to malicious actions. Howard and

Lipner acknowledge that security is a subset of quality and explicitly state that secure code is not code that implements security features. The distinction between secure code and security functionality is subtle but important: code that implements security functionality must be secure itself; that is, security functionality must meet specifications and not contain vulnerabilities.

In the *Software Assurance Common Body of Knowledge* (SwA-CBK) [3], secure code is characterized as meeting security constraints and being free of vulnerabilities while also helping to achieve quality. In the SwA-CBK, writing secure code means producing code that has the following characteristics:

1. Is correct and meets specifications.
2. Meets required security property constraints.
3. Does not contain weaknesses that could manifest as vulnerabilities.
4. Simplifies the detection and correction of not only faults but of such weaknesses.

From the definition above, parts (1) and (4) relate to traditional software quality while parts (2) and (3) relate to security. The definition of secure code in the SwA-CBK is adopted in this thesis. It is sound, clear, and subsumes the prior two definitions:

- Bishop's methodology has activities for achieving parts (2) and (3).
- Howard and Lipner's definition is covered by parts (1), (2), and (3).

2.1.2 Secure Code in the Greater Software Context

Secure software realizes "with justifiably high confidence but not guaranteeing absolutely – a substantial set of explicit security properties and functionality including all those required for its intended usage" [7]. Experience has shown that adding security to software after it is constructed is costly, difficult, and is more likely to lead to failure [25]. To achieve secure software, security must be integrated into all aspects of the process carried out to produce it [7]. Thus, writing secure code is an activity that helps to achieve secure software during the implementation phase of the SDLC.

2.2 Principles and Practices

A design is realized by writing code. If a design is deficient, then code that realizes it will also likely be deficient. Design principles exist so that designs can be created that exhibit high-quality characteristics [10]. When design principles are followed, better designs are achieved. At lower levels of abstraction, programmers have identified best practices that should be followed to create high-quality code that is secure. When secure coding practices are followed, code is less likely to contain vulnerabilities.

Because people interpret the meaning of design principles and how they should be applied differently, one of the goals of the taxonomy proposed in this work is to correlate secure coding practices in Java with design principles. In the remainder of this section, existing design principles and secure coding practices are surveyed. Chapter 4 provides definitions and descriptions of principles that are included in the new taxonomy.

2.2.1 Principles of Software Design

Adhering to software design principles helps to satisfy the requirement that writing secure code should simplify the process of removing both quality-related and security-related defects in software. As pointed out by Bloch, such principles are important when designing and implementing robust application programming interfaces (APIs) [11].

Parnas [12] discussed techniques for modularizing programs with a focus on information hiding in the early 1970s. The criteria for decomposing modules discussed by Parnas are still relevant today. Fox [10] provides a thorough discussion of software engineering design covering modularity principles, implementability principles, and aesthetic principles. Fox's taxonomy of constructive principles is as follows [10]:

Engineering Design Principles
- *Constructive*
  - *Modularity*
    - *Small Modules*
    - *Information Hiding*
    - *Least Privilege*
    - *Coupling*
    - *Cohesion*
  - *Implementability*
    - *Simplicity*
    - *Design with Reuse*
    - *Design for Reuse*
  - *Aesthetic*
    - *Beauty*

2.2.2 Principles of Secure Software Systems Design

When design principles for security are followed, designs are made more secure [8]. Seminal work by Saltzer and Schroeder proposed the following principles for designing secure systems within the context of operating systems [1]:

Economy of mechanism
Fail-safe defaults
Complete mediation
Open design
Separation of privilege
Least privilege
Least common mechanism
Psychological acceptability
Work factor
Compromise recording

Saltzer and Schroeder's principles have proven to be important in guiding the design of not only operating systems but of all secure software systems [3]. Benzel et al. [8] analyzed and refined Saltzer and Schroeder's principles for modern system contexts. In their work, they indicate two important differences in comparison to previous efforts:

1. They assume unspecified functionality may be intentionally introduced by an adversary within the development process.
2. Their analysis considers both the design of components as well as their composition.

The diagram in Figure 1 shows their taxonomy, which was reproduced from their work [8] with only the security principles relevant for writing secure code. The security principles that are categorized within the "system life cycle" group, which includes principles mainly based on procedural rigor, are not shown. Principles that correspond to Saltzer and Schroeder's work appear bolded and italicized.

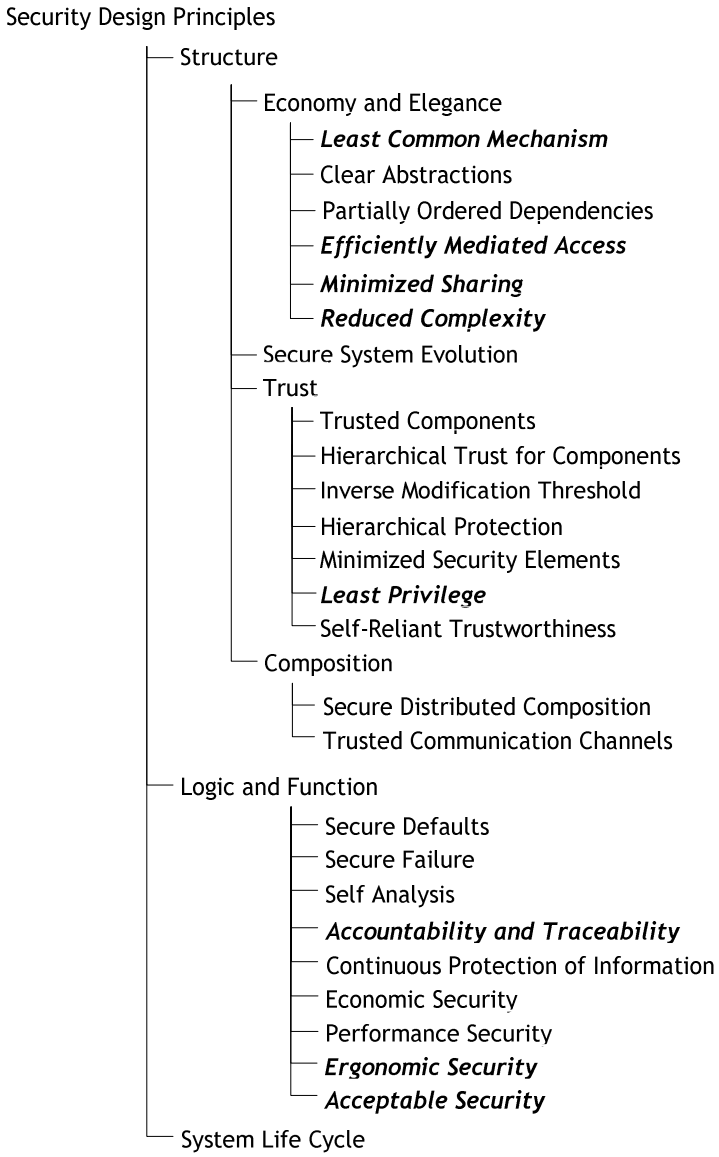

Figure 1: Taxonomy of security design principles by Benzel et al. [8]

Section 6 of the SwA-CBK covers principles for secure software design with an emphasis on "minimizing and simplifying the portion of the software that must be trusted" and employing mechanisms that "reduce the possibility for security violations" [3]. Additional guidelines and techniques included in the SwA-CBK that have not been previously mentioned in this section are listed below [3]:

Analyzability
Treat as conflict
Defense in depth
Separation of duties
Separation of roles
Separation of trust domains
Constrained dependency
Physical, logical, and domain isolation

Numerous other sources provide discussions and recommendations for how to achieve previously mentioned principles or variations of them. Informal "guiding principles for software security" have been discussed at length by Viega and McGraw [9]. Graff and van Wyk [13] outline thirty basic security principles at an architectural level of abstraction. Howard and Lipner [5] also provide a discussion of design principles and apply them to writing secure code. Finally, the Department of Homeland Security's Build Security In (BSI) [53] project has a section in its "knowledge area" dedicated to design principles and how they have been interpreted through the years.

Ongoing work by Redwine [45] aims to be the most complete and coherent organization of software system security principles to date. Redwine covers principles that span all aspects of the process used to produce secure software systems, not just design and implementation. His organization scheme is based on how principles and guidelines limit, reduce, or manage security impacts across three streams [45]:

The adverse – emphasizes violators, violator gains, and attempted violations
The system – emphasizes opportunities for violations, violations, and potential and actual losses
The environment – emphasizes environment of conflict, dependence on environment, and trust

2.2.3 Secure Coding Practices

In this section, secure coding practices that apply to all languages are outlined. While design principles have theoretical importance, coding practices do not. Coding practices are specific implementation advice about how code should be written.

There are many resources that discuss secure coding practices. Graff and van Wyk [13] and Howard and Lipner [5] discuss secure coding practices at length. Chess and West [43] also provide an in-depth analysis of secure programming techniques, with specific examples in C and Java. Finally, the SwA-CBK [3] lists many secure coding practices when discussing secure software construction.

The following list[3] covers important secure coding practices that all developers should follow, regardless of the programming language or environment being used:

- Avoid using dangerous language constructs [3].
- Minimize code size and complexity [3, 13].
- Ensure asynchronous consistency [3].
- Validate and cleanse[4] all data that is received from untrusted sources, including user input and data that is retrieved from environment sources [3, 5, 13, 43].
- Perform bounds checking on all buffers [5, 13, 43].
- Ensure variables are properly initialized; do not depend on default initialization [13].
- Explicitly check all system call or method return values [5, 13, 43].
- Ensure files cannot be opened using relative file paths or indirect file references [13].
- Ensure a file is not opened two or more times using its name in the same program [13].
- Avoid invoking less trusted programs, such as command shells, from within more trusted programs [3, 13].

---

[3] The list is representative, not exhaustive. Readers should consult the referenced works for more in-depth discussions and examples.
[4] Graff and van Wyk define cleansing data "as the process of examining the proposed input data for indications of malicious intent."

- Avoid using "pseudo-random" number generators [13]; instead, use APIs that product cryptographically secure random numbers [5, 43].
- Always fail to a secure state (i.e., fail gracefully) [13]; implement error and exception handling safely [3] and ensure sensitive information is not leaked when systems fail [5, 43].
- Protect secrets while they are stored in memory [5, 43].
- Avoid hard-coding secrets (e.g., passwords, encryption keys) in source code [5]; encrypt and store them in protected, external files instead [13].
- Ensure access control decisions are not based on untrusted data, such as environment data, or the names of entities [5, 13].
- Use parameterized statements to build database queries (i.e., to prevent SQL injection) [5, 43].
- Require permission for performing serialization and do not deserialize data from untrusted sources [5].
- Avoid using file locations that anyone can access, even temporarily [5, 13].
- Store application security related actions to a log that is only accessible to administrators [5, 43].
- Remove code that is obsolete or that is not used [3, 13].
- Adhere to coding standards [3] and coding style guidelines [13].

To produce high-quality designs, software designers should follow design principles as much as possible. Likewise, programmers should write code that does not violate secure coding practices. Adhering to design principles and secure coding practices, however, does not guarantee that software is absolutely secure. Nevertheless, following such principles and practices improves the state of software in the face of unknown attacks that may occur in the future [9].

2.3 Taxonomies

One goal of this thesis is to correlate coding heuristics for increasing the quality and security of code with principles of design. Toward that end, it is necessary to study previous attempts at classifying software characteristics or coding errors that have security implications in order to compare, improve, and build upon them.

Since the early 1970s, there has been interest in organizing security flaws, vulnerabilities, software weaknesses, and the coding errors that cause them. There has also

been debate in how these terms should be defined. Correcting or removing a flaw, vulnerability, or software weakness requires one or more changes to be made to software; if a change is made to software, a defect exists. Therefore, in this thesis, a security flaw, vulnerability, or software weakness is considered to be a software defect that degrades security.

The Research Into Secure Operating Systems (RISOS) study [30] and the Program Analysis (PA) study [31] were the first two attempts to classify security flaws. Numerous other taxonomies have stemmed from these early works. This section provides an overview of different classification schemes that have been proposed since the early 1970s.

2.3.1 Classification Schemes

The RISOS study of 1972 defines seven classes of security flaws in operating systems with the goal of making it easier to analyze the security of new systems [30]. The taxonomy is as follows:

1 Incomplete parameter validation
2 Inconsistent parameter validation
3 Implicit sharing of privileged / confidential data
4 Asynchronous-validation / Inadequate-serialization
5 Inadequate identification / authentication/ authorization
6 Violable prohibition / limit
7 Exploitable logic error

The PA study [31] was undertaken at about the same time as the RISOS study. The goal of the PA study was to identify automatic techniques for detecting operating system vulnerabilities (i.e., static analysis techniques). The PA study resulted in the proposal of a taxonomy that contains nine classes of flaws. Bishop and Bailey have reported on Neumann's presentation of the taxonomy [33]:

1 Improper protection (initialization and enforcement)
1a. Improper choice of initial protection domain
1b Improper isolation of implementation detail
1c Improper change
1d Improper naming
1e Improper deallocation or deletion
2 Improper validation
3 Improper synchronization
3a Improper indivisibility
3b Improper sequencing
4 Improper choice of operand or operation

In 1994, Landwehr et al. [32] set out to provide an understandable record of security flaws that were found in real systems. They aimed to categorize flaws by considering how flaws entered the system (genesis), when flaws entered the system (time of introduction), and where flaws manifested in the system (location).

It is interesting to note that 32 of the 50 flaws studied by Landwehr et al. fell under the "inadvertent flaws" category, which is a subcategory of genesis. Inadvertent flaws are most likely to be introduced by programmers. The elements within the taxonomy that fall under the inadvertent flaws category are similar to the categories that were produced in the RISOS and PA studies [32]:

1 Validation error (incomplete/inconsistent)
2 Domain error (including object reuse, residuals, and exposed representation errors)
3 Serialization/aliasing (including TOCTOU [5] errors)
4 Identification/authentication inadequate
5 Boundary condition violations (including resource exhaustion and constraint errors)
6 Other exploitable logic error

Claiming that previous taxonomies suffer from being too generic and ambiguous, Aslam [75] attempted to create a more precise scheme for classifying security faults in the UNIX operating system. Aslam devised a decision procedure that used selection criteria to

[5] A time-of-check-time-of-use (TOCTOU) flaw occurs in asynchronous programs when shared data is changed after it is checked but before it is used [32].

determine which category a fault was distinctly placed. Yet, in 1996, Bishop and Bailey [33] conducted a critical analysis of vulnerability taxonomies and argued that all previously proposed taxonomies have a common problem: they fail to define classification schemes that place vulnerabilities into unique categories.

In 2005, Weber et al. [34] proposed a software flaw taxonomy that aimed to be suitable for developers of static analysis tools. They correlated their taxonomy with high-priority security threats in modern systems, such as the *Open Web Application Security Project* (OWASP) [35] list of the top ten most serious web application vulnerabilities. Weber et al. disagree with Bishop and Bailey in that a flaw that has different classifications should be viewed as a problem with taxonomies. Instead, Weber et al. argue that if a flaw can be classified under multiple categories, it is a noteworthy characteristic of the flaw itself.

The *Comprehensive Lightweight Application Security Process* (CLASP) is an activity-driven, role-based process that has formal best practices for building security into an existing or new SDLC [48]. The CLASP vulnerability lexicon aims to help prevent design and coding errors that can lead to vulnerabilities. CLASP identifies 104 "problem types," which are defined to be causes of vulnerabilities. The 104 problem types are categorized into the following five top-level categories [48]:

1 Range and Type Errors
2 Environmental Problems
3 Synchronization and Timing Issues
4 Protocol Errors
5 General Logic Errors

Claiming that an intuitive, practical taxonomy will help developers better understand coding mistakes that lead to vulnerabilities, Tsipenyuk et al. [46] proposed the seven

pernicious kingdoms (SPK) taxonomy[6]. Their taxonomy consists of seven "kingdoms" of coding errors, plus one more that is not within the scope of writing code:

1 Input Validation and Representation
2 API Abuse
3 Security Features
4 Time and State
5 Error Handling
6 Code Quality
7 Encapsulation
**Environment*

The eighth kingdom is marked with an asterisk to indicate that it is not related to coding practices. A set of coding errors that are important to enterprise developers are categorized under each kingdom. The SPK taxonomy focuses on coding errors that can be checked by static analysis tools. It is also the first approach to organizing coding errors in a manner that developers without a background in security can understand. With respect to Java, the SPK contains coding errors from both the J2SE and J2EE frameworks.

Most recently, the CWE has emerged as an ongoing community effort for collecting and organizing software weaknesses in code, design, and architecture [67]. The CWE classification tree is designed to be enumerated online and lists all of the previously mentioned taxonomies as sources. In Draft 7 of the CWE, the *Location.Code.SourceCode* node seems to encompass most of the SPK taxonomy, except that "Input Validation and Representation" has been renamed "Data Handling".

The CWE draws on ideas from all previous taxonomies and attempts to be the most complete collection of software weaknesses to date while remaining applicable to all programming languages. A side effect of its comprehensiveness is that it covers a variety of programming language, domain, and technology-specific issues that make enumerating it to

[6] *Fortify Software, Inc.* [28] has created an online vulnerability catalog that contains details for each coding error in the SPK along with specific coding examples. It is worthwhile to study it online.

find specific secure coding practices difficult. The following is a glimpse of top-level nodes within Draft 7 of the CWE [26]:

Location
Configuration
Code
Source Code
Data Handling
API Abuse
Security Features
Time and State
Error Handling
Code Quality
Encapsulation
Byte/Object Code
Environment
Motivation/Intent
Intentional
Malicious
Nonmalicious
Inadvertent

2.3.2 Discussion

Through the years, classification schemes have been proposed to categorize implementation-level flaws found in operating systems [30, 31, 75], root causes of vulnerabilities [48], categories of coding errors that impact security [46], and most recently security-related software weaknesses [26]. Taxonomies that are currently emphasized in practice, such as the SPK, CLASP, and the CWE, act as a checklist of quality and security-related problems that can be utilized during development (i.e., by developers or static analysis tools), and they serve this purpose well.

However, these efforts lack a view of the problem from a design theory perspective. I believe that developers must be able to recognize and understand how decisions or mistakes made at the code-level affect the overall design of software components. Developers should realize that when a decision is made to call a method on an external class, the principle of coupling should be considered. When a decision is made to catch but ignore

an exception, developers should understand that the principle of secure failure is being violated. I believe that a design-driven approach can help developers apply a security-aware mindset when writing code.

To exemplify, while numerous security-related coding errors fall under the category of "Error Handling" [26, 46], this label does not explain why errors should be properly handled. If an error code is returned from a method but is not properly checked, what effect does ignoring the error have on this module and other modules that depend on it? A well-designed program is able to assess its internal state, detect errors, and fail in a secure manner. Thus, the deeper problem involves designing components so that they check for error conditions and fail in a manner that is secure. In this view, the act of performing error handling is a means to achieving an end (i.e., failing securely). Benzel et al. [8] refer to these design principles as "self-analysis" and "secure failure."

Another example is the CWE node labeled "Data Handling" [26]. This node contains weaknesses that are related to the improper handling of data input, which may result in buffer overflows, command injections, SQL injections, and other input-related vulnerabilities. While on the surface these are data handling problems, the deeper issue is one of trust. All data that is received from untrusted sources should be properly handled prior to its interaction with trusted components. In software design, this technique is achieved conceptually by establishing well-defined trust boundaries [3, 5, 43].

Interestingly, a lack of theory, at least from a design perspective, was not regarded as an oversight during the creation of previous taxonomies. The SPK taxonomy was intentionally devised not to incorporate theory; it is aimed to be as practical as possible [25, 46]. While placing coding errors into simple classes may help developers understand and

communicate about the kinds of coding errors that may lead to vulnerabilities, it does not paint a clear picture of the larger problem that results from having them in software.

To acknowledge the importance of a design-driven approach, a new taxonomy is proposed in Chapter 4 that aims to connect guidelines for writing secure code in Java with the design principles they help to achieve. To help build the taxonomy, the next chapter describes how Java platform security has evolved through the years. It also covers numerous guidelines for improving the security and quality of Java code that all Java developers should learn and apply.

3 Java Security

The Java platform is complex. It consists of both a programming language and runtime environment, which deploys a Java Virtual Machine (JVM). To write secure code, developers must understand how security is provided by both language features and the runtime environment. Otherwise, security that is provided by the Java platform may be vitiated by weaknesses in application code.

It is critical to understand the security mechanisms provided by the Java platform because code can be written that can affect the outcome of security decisions made at runtime. For instance, custom permissions can be created for an application and code can be written to ensure an application has certain permissions at runtime. This is both a construction and configuration issue: code must exist to issue the permission check, and a security policy must be externally configured and enabled at runtime to produce an outcome.

The goal of this chapter is to cover all aspects of the Java platform that should be addressed by a secure coding standard. The first part of this chapter provides an overview of how Java platform security has evolved through the years. In the second part, numerous coding guidelines for writing secure code in Java are described.

3.1 Evolution of Java Platform Security

Sun Microsystems officially announced Java as a technology on May 23, 1995 at *SunWorld*. At that time, Java was proclaimed to be the most secure platform for developing applications that utilized network technology. To quote a white paper:

> *The architecture-neutral and portable aspects of the Java language make it the ideal development language to meet the challenges of distributing dynamically extensible software across networks [14].*

Nine months later in February of 1996, Princeton University researchers[7] publicly described the first attack on the Java platform. A vulnerability in the applet security manager enabled an attacker to subvert the domain name system. The applet could then connect to an arbitrary host on the Internet [15]. Seven other vulnerabilities within the Java platform were found and reported in that same year [16].

While security was a design goal for the Java platform [14], the security incidents reported in 1996 clearly illuminated the fact that the platform was not impenetrable. Over the years, Java platform security has evolved from being a limited, restriction-based architecture into a flexible architecture with support for configuring and enforcing fine-grained security policies.

3.1.1 Original Java Platform Security (JDK 1.0)

The security architecture of the Java Development Kit (JDK) 1.0 release focuses on securely downloading and executing remote code, or applets. Code that resides on the local file system is called a Java application. A Java application is given full access rights, or permissions, to system resources. Code that resides on external networks that is dynamically downloaded during execution is called an applet. Applets are not trusted and are executed within an isolated area in the memory space of a web browser. Since applets are not trusted, applets are given limited access rights to system resources. Granting applications all permissions and applets only limited permissions is known as the "sandbox" model [17].

Aspects of the original security architecture exist today in the Java SE 6 release. The security of the Java platform consists of the following mechanisms [17]:

---

[7] The group consisted of Drew Dean, Ed Felton, and Dan Wallach of the Department of Computer Science at Princeton University. This work lead to the formation of an influential Java research group called the Safe Internet Programming (SIP) team.

- Safe language features
- Bytecode verification
- Runtime checks enforced by a Java Virtual Machine (JVM)

3.1.1.1 Security Provided by Safe Language Features

The Java language is strongly typed. All manipulation of a typed construct is verified to be safe, both statically by a compiler and dynamically by the Java Virtual Machine (JVM). Language type safety contributes to the correctness of a program. If a program cannot implement security functionality because it cannot be correctly executed, required security may not be provided [17]. Thus, since Java is strongly typed, incorrectly typed programs that may lead to vulnerable program states are not a concern.

The *Java Language Specification* (JLS) [18] defines access modifiers as mechanisms for providing access control to the implementation details of classes and packages. Specifically, four distinct access modifiers are defined:

- `private`
- `protected`
- `public`
- `package` (the default access)

Each modifier permits different accessibility to a class or package implementation. The JLS formally describes a number of accessibility rules that can be summarized as follows:

- A `private` entity is only accessible within its enclosing class and by all objects of that class.
- A `protected` entity is only accessible by a subclass or by other entities declared within its package.
- A `public` entity is accessible by any code that can access its enclosing class (if any).
- An entity that is not explicitly declared `private`, `protected`, or `public` is implicitly declared `package` and is only accessible to entities declared within its package.
- All members of an interface are implicitly declared `public`.

Since access modifiers are the primary constructs for hiding information within modules, effective use of them leads to safer, more robust, and more understandable APIs [11]. However, developers cannot assume that using access modifiers guarantees security [15]. The Java platform has features, such as the serialization API and inner classes, that circumvent such protection.

3.1.1.2 Security Provided by Bytecode Verification

Java bytecode, which is the result of program compilation, is analyzed by a bytecode verifier to ensure that only legitimate instructions are executed at runtime [17]. Specifically, the bytecode verifier checks for the following:

- Memory management violations
- Stack underflows and overflows
- Illegal data type casts

Java bytecode can be modified by any individual with access to the resource (e.g., file) that contains them. Hence, a bytecode verifier is essential because bytecode cannot be considered trustworthy when executed. The format of bytecode must be correct and language rules must be enforced prior to program execution [17]. The process of verifying bytecode contributes to ensuring the integrity but not the security of Java code; bytecode verification cannot detect when legitimate bytecode is maliciously modified.

3.1.1.3 Security Provided by the Java Virtual Machine

The JVM is responsible for linking, initializing, and executing bytecode that has passed verification. At runtime, the JVM performs automatic memory management, garbage collection, and range checks on array references [17].

A fundamental element of the JVM is a `ClassLoader`. With respect to security, a `ClassLoader` is responsible for ensuring that executing code cannot interfere with other executing code by defining namespaces. When classes are loaded, a `ClassLoader` binds all

classes with a `ProtectionDomain`, which associates them with a set of permissions.

Finally, the JVM employs a `SecurityManager` that is responsible for mediating access to sensitive system resources. The `SecurityManager` consults a security policy, of the `Policy` class, at runtime to restrict the operations that can be performed by executing code.

3.1.2 JDK 1.1 Security

In the JDK 1.0 "sandbox" model, applets are not and cannot be trusted. To provide support for trusting remote code so that applets known to be safe could have more extensive permissions, JDK 1.1 introduced the notion of signed applets using digital signatures. In JDK 1.1, when an applet is downloaded and its digital signature is verified, the applet is treated the same as a Java application residing on the local file system and is given full access rights to the system [17]. Unsigned applets are not trusted and execute within an isolated environment with limited access rights as defined by the original architecture [19].

3.1.3 Java 2 SE (J2SE 1.2) Security

Even with applet signing functionality, JDK 1.1 has an "all or nothing" permission-based architecture: trusted code is given full access to system resources and code that is not trusted is given limited access. The J2SE 1.2 release on December 4, 1998 aimed to improve the security model by including support for fine-grained access control. The ability to have fine-grained access control called for a security policy that could be customized [17].

3.1.3.1 Security Enhancements in J2SE 1.2

A number of improvements were made in J2SE 1.2 to achieve flexible access control. Gong et al. [17] state the following enhancements:

- The design allowed for a policy enforcement mechanism to be separate from the description of a security policy.
- A general `checkPermission` method was added to the `SecurityManager` class to handle all security checks on sensitive resources.
- The design allowed for all code, whether local, remote, signed, or unsigned, to be subjected to the same security checks.
- The designs of both the `SecurityManager` and `ClassLoader` classes were improved.

The J2SE security architecture uses a security policy to decide which access permissions are granted to executing code. If a security policy is not explicitly specified, the JDK 1.1 security model becomes the default [17]. Further details concerning how fine-grained access control is provided in J2SE are described in *Security Managers and the Java SE SDK* [19]. Gong et al. [17] explain the process of executing an applet or Java application within the security architecture in detail.

3.1.3.2 Protection Domains

A protection domain, or an instance of the `ProtectionDomain` class, is a mechanism for grouping classes and associating them with a set of permissions [20]. In order to provide separation, the JVM maintains a mapping of code to protection domains and protection domains to permissions. A class is mapped to a `ProtectionDomain` once, when the class is loaded; its `ProtectionDomain` cannot be changed during the class's lifetime within the JVM [17]. A `ClassLoader` maps a class to a `ProtectionDomain` based on the class's `CodeSource`, which constitutes the location of the code[8] and its signers[9], if any are provided.

Two distinct protection domains exist:

- *System*, which consists of core classes that are granted full permissions.
- *Application*, which consists of non-core classes that are granted limited permissions defined by a security policy.

[8] A class's location is specified as a URL, which can describe both file system and network locations.
[9] The signers of a class are specified by digital certificates associated with the class; the reader should consult Gong et al. [17] for further details.

3.1.3.3 Mediating Access to Resources

An instance of the `SecurityManager` class is deployed by the JVM and is activated whenever a decision is needed to determine whether to grant or deny a request for accessing a sensitive resource. The J2SE 1.2 release introduced two `checkPermission` methods to the `SecurityManager` class to enforce a security policy [17]. A `SecurityManager`, however, does not specify how access checks should be performed. Rather, a `SecurityManager` delegates all security decision making to the `AccessController` class [17]. Thus, a `SecurityManager` acts as a simple interface for invoking security checks.

By default, applets execute under the presence of a `SecurityManager`. Conversely, by default, when local applications are executed, a `SecurityManager` is not installed. A `SecurityManager` can be installed by including the `-Djava.security.manager` argument when executing an application at the command line [19]. An application can also programmatically install a `SecurityManager` by calling:

```
System.setSecurityManager(new SecurityManager());
```

3.1.3.4 Enforcing Permissions

During program execution, it is possible for a thread of execution to cross multiple protection domains. Gong et al. [17] explain that this can occur, for example, when an application, which is in the *application* `ProtectionDomain`, is required to interact with the *system* `ProtectionDomain` to print a message using an output stream.

Whether or not sensitive actions can be performed by code is determined at runtime by evaluating the `ProtectionDomain` of each class on the call stack (i.e., the execution context). Under usual circumstances, when access to a sensitive resource is guarded with a `SecurityManager` check, access is permitted only if every `ProtectionDomain` in the execution context is granted the required permission [17].

Because requiring every `ProtectionDomain` on the call stack to possess the needed permission can be too restrictive, a class called `AccessController` exists with a method named `doPrivileged` so that "privileged operations" can be performed by trusted code. When a class invokes `AccessController.doPrivileged`, the `SecurityManager` is told to ignore the protection domains of all previous class method invocations on the call stack. As long as the class that invokes `doPrivileged` has the required permission, the `SecurityManager` check will succeed, even if one or more previous callers on the stack do not have the required permission [17]. When this happens, a less "powerful" domain cannot gain additional permissions as a result of being called by a more "powerful" domain [20]. Power, in this context, is determined by the number of permissions that has been granted to a `ProtectionDomain`.

As mentioned in Section 3.1.3.3, the `SecurityManager` class delegates access control decision making to the `AccessController` class. If a request to access a resource is granted, `AccessController` returns silently. If a request is denied, `AccessController` throws a `SecurityException`.

3.1.3.5 Configuring Policy

An important J2SE feature is that permissions are specified in configuration files. These configuration files are loaded by a `Policy` object, which represents a specification of the permissions that are available to code that is executing [21]. There is a default system-wide policy file and a single user policy file. The user policy file is optional. In Java SE 6, the default system policy file is located at the following directory:

```
java.home/lib/security/java.policy
```

The user policy file is located at the following directory:

```
user.home/java.policy
```

The system policy file specifies system-wide permissions, such as granting all possible

permissions to standard extensions, allowing any code to listen on ports that are not privileged, and allowing any code to read properties, such as `file.separator`, that are not sensitive [21].

When a `Policy` object is initialized, the content of the user policy, if present, is added to the content of the system policy. If neither policy is present, then a default system policy is used [21]. It is possible to enforce a specific `Policy` when an application is executed at the command line using the `-Djava.security.policy` command line argument.

3.1.3.6 Creating Custom Permissions

The flexibility of the J2SE security architecture is further exemplified by giving developers the capability to create new types of permissions that are unique to applications. Custom permissions can be created by extending the `java.security.BasicPermission` class and providing an `implies` method [17, 20]. Gong et al. [17] note a few guidelines that should be followed when new permissions are created:

1. If possible, classes that represent permissions should be declared `final`.
2. If the abstract class of the custom permission or permission collection has a concrete implementation of the `implies` method, then the type of the permissions should be taken into consideration to prevent overriding by malicious subclasses.
3. A custom `PermissionCollection` should be used if the permission "has complex processing semantics for either its name or the actions it specifies" [17].
4. Ensure the `implies` method is correctly implemented.

After a custom permission class is created, appropriate entries must be added to the policy configuration files. Once policy entries have been added, the permission can be enforced by calling the `checkPermission` method of the `SecurityManager` class using an instance of the custom permission class [17]. An example of code that issues a security check is shown in Figure 5 in Section 3.2.3.

3.1.4 Mobile Code

Code that is transferred from one machine to another machine across a network and

then dynamically executed is called mobile code [47]. Distributed programs, such as Java applets and applications built using the Java Remote Method Invocation (RMI) technology, utilize mobile code.

Immobile code is code that is installed and executed locally. To a `SecurityManager` operating inside a JVM, there is no inherent difference between mobile and immobile code. The `ProtectionDomain` associated with code that is executing either has permissions granted to it in a `Policy` object that is in effect or it does not.

The granting of applet and application permissions is an issue based on trust. The key to securely executing Java code, whether it is mobile or immobile, is ensuring that a legitimate `SecurityManager` is installed and an appropriate `Policy` is in effect. At a minimum, the integrity of the policy configuration files must be guaranteed. Likewise, users must be aware of the consequences of granting Java applets and applications permissions at runtime.

3.2 Guidelines for Writing Secure Code in Java

As described in Section 3.1, Java has the goal of being a secure platform for executing both local and remote code. A secure platform, however, cannot guarantee that code being executed is secure. For instance, while the Java security architecture can place restrictions on code that is downloaded, it cannot defend against implementation bugs that occur in trusted code [2].

An initial examination of every aspect of the Java platform with a focus on security was conducted by the SIP team at Princeton University in 1997 [22]. Based on this work, McGraw and Felton published twelve rules for writing secure code in Java [23]. Their rules covered varying levels of implementation advice, from how to ensure classes are initialized prior to use to simply not storing secrets in code.

Since the publication of McGraw and Felton's twelve rules, numerous statements concerning how to write secure code in Java have been made by different people and organizations. Some of the other influential works include:

- Suggestions for writing robust APIs [11]
- Recommendations for coping with Java pitfalls [24]
- Securing objects in memory, storage, and transit [17]
- The SPK taxonomy of coding errors [25, 46]
- Secure coding guidelines from Sun Microsystems [2]
- Accounting for Java subtleties that can compromise code [51]
- The CWE community effort [26]

3.2.1 Securing Classes

To limit accessibility to class entities, it is widely agreed that all class entities, such as fields, methods, and nested classes, should be declared `private` by default [2, 11, 17, 25]. Since it is easier to employ both security and validity checks when sensitive data is hidden and isolated, accessor methods should be used to access `private` fields [2]. From an API design perspective, accessor methods also preserve the flexibility to change a class's internal representation [11]. Providing an accessor method does not necessitate providing a mutator method; mutator methods should be implemented only as they are required [11].

Since `public static` fields are visible to all code, it is best to treat them as constant. To achieve this, `public static` fields should be declared `final`, if primitive, and made immutable, if not primitive [2, 11]. This requirement is important when considering that a `SecurityManager` cannot verify the access or modification of `public static` variables [2].

The use of mutable `static` objects should be limited. When used, mutable `static` objects should be declared `private` [11] and retrieved using accessor methods [2]. The direct assignment of `public` data to `private` mutable objects, such as array-typed fields, should not occur without good reason [26: 496].

By default, a sensitive class should protect itself from being cloned by explicitly not allowing it and throwing an exception if cloning is attempted [15]. Since the Java cloning mechanism creates new object instances without calling a constructor, the risk is that an implementation of the `clone` method may neglect to perform the same validity and security checks that may be present in a constructor [2, 51]. If a class must be `Cloneable`, the `clone` method must not call non-final methods, which may be overridden in a malicious manner [11], and should make defensive copies (i.e., deep copies) of mutable fields.

To disable the cloning mechanism provided by the `Cloneable` interface, McGraw and Felton [23] recommend making the `clone` method `final` and explicitly throwing an exception, as shown in Figure 2.

```
public final void clone() throws java.lang.CloneNotSupportedException {
   throw new java.lang.CloneNotSupportedException();
}
```

Figure 2: Explicitly prevent cloning

The technique shown in Figure 2 defeats the Java cloning mechanism that is provided by the `Cloneable` interface. An alternative method for copying objects is to provide a `public` factory method, as shown in Figure 3.

```
public final A copy() {
   A a = new A();
   a.mutableField = this.mutableField;

   return a;
}
```

Figure 3: Clone-like functionality in a factory method

The advantage of using a factory method is that a class constructor is always called; thus, all security and validity checks that are present in the constructor will be performed. However, the factory method shown in Figure 3 does not properly deep-copy a mutable field, named `mutableField`, of class `A`. As a result, callers that obtain a copy of the object are able to modify the internal state of the original object. Similar to implementations of the

`Cloneable` interface, factory methods that exist for clone-like functionality should be verified to correctly handle mutable fields.

Classes should be made as immutable as possible by default [2, 11]. Immutable classes should not provide clone functionality [11]. If a `public` class cannot be made immutable, it should provide a facility for making deep copies of it [2]. As described by Bloch, a class can be made immutable by adhering to the following five rules [11]:

1. *Do not provide any methods, or mutators, that modify an instance of the class.*
2. *Ensure that no methods may be overridden, usually by declaring the class to be* `final`.
3. *Declare all fields as* `final`.
4. *Declare all fields as* `private`.
5. *Ensure exclusive access to any mutable components of the class.*

If a class is not specified to manipulate mutable input parameters, make defensive copies of them and only manipulate the copies [2, 11]. Additionally, a `clone` method should not be used to copy non-final parameters since it may return an instance of a subclass that is designed for malicious purposes [11]. Likewise, unless a method is specified to return a direct reference to a mutable `private` field, return a defensive copy of the field [2, 11, 26: 495]. When untrusted input is stored in mutable objects, input validation should always occur after objects have been defensively copied [2, 11].

A class that is not `final` and that contains sensitive data should impose self initialization checks in its `public` and `protected` methods so that it remains unusable until it is fully constructed. While considered to be inelegant to some [51], a class initialization flag may help achieve this [2, 11, 25] and is the most cautious approach to take. A sensitive class should also prevent unauthorized construction by enforcing a `SecurityManager` check at all instantiation points (i.e., in constructors, in `clone` of `Cloneable` classes, and in `readObject` and `readObjectNoData` of `Serializable` classes) [2].

Nested types were added to Java as an extension. Consequently, nested types are implemented in terms of a source code transformation applied by a compiler [39]. The transformation widens the accessibility of its enclosing class members, including `private` members, to `package` visibility by providing `static package` methods in both the nested and enclosing classes. Any nested class that is declared `private` is also converted to `package` accessibility during compilation [2]. While the compiler does not allow for these hidden methods to be called at compile-time, the resulting bytecode is susceptible to a package insertion attack at runtime [2].

An inner class is a nested class that is not explicitly or implicitly declared `static` [18]. Since an inner class widens the accessibility of its enclosing class members, including `private` members, to `package` visibility, it is better not to use them [15]. If an inner class must be used, make the inner class `static` if it does not require a reference to its enclosing instance [11]. Additionally, if its enclosing class is `private` or `final`, the inner class should be `private` [26: 492].

To avoid confusion and prevent degradation of code readability when better alternatives exist, instance initializer blocks and `static` initializer blocks should not be utilized [26: 545]. As shown in Figure 4, initialization blocks can often be replaced by `private final` methods, which have the advantage of being reusable.

```
private static int id;

// confusing use of static initializer
static {
   id = 111;
}

-------------------------------------------------------------------------------

private static int id = initializeId();

// less confusing use of static method initializer
private final static int initializeId() {
   return 111;
}
```

Figure 4: Initialization block alternative

3.2.2 Securing Packages

The scope of classes within a package should be reduced as much as possible. By default, a class should have `package` access [2]. A class should only be declared `public` if it is part of an API [2, 11].

The `java.security` properties file defines two properties for protecting packages when a `SecurityManager` is installed. These properties are named `package.definition` and `package.access`:

- By adding a package name to the `package.definition` property, code is prevented from maliciously *claiming to be part of* a package at runtime, unless the code is granted the `defineClassInPackage` permission [40].
- By adding a package name to the `package.access` property, code is prevented from maliciously *accessing* entities within a package at runtime, unless the code is granted the `accessClassInPackage` permission [40].

The `package.definition` property protects packages from attacks when malicious code declares itself to be part of the same package as other code. Such an attack is only successful when both the malicious code and the target code are loaded by the same `ClassLoader` instance. As long as a `ClassLoader` properly isolates unrelated code, such as when applets are loaded by a browser plug-in, malicious code cannot access the entities of other classes even if the malicious code declares itself to be in the same package [2].

In the `java.security` properties file in Java SE 6, it is noted that none of the class loaders supplied with the JDK enforce the `package.definition` check; thus, a custom `ClassLoader` is required. This requirement is important for applications that dynamically load code.

3.2.3 Securing Inheritance

Classes should be declared `final` to prevent them from being maliciously subclassed [2, 15]. To prevent malicious overriding of methods, methods should be `final` [2, 15].

Malicious class subclassing or method overriding can occur whether a class has `public` or `package` accessibility.

A sensitive class that is `public` and not `final` should limit subclassing to trusted implementations by employing a `SecurityManager` check at all points of instantiation [2]. This is achieved by verifying the class type of the instance being created and issuing a permission check. The code that is shown in Figure 5 models the technique as illustrated in Sun's secure coding guidelines [2].

```
public class A {
     public A() {
          Class subClass = getClass();
          if (subClass != A.class) {  // a subclass is being instantiated
               SecurityManager sm = System.getSecurityManager();
               if (sm != null) {
                    sm.checkPermission(new SubclassPermission("A"));
               }
          }
     }
}
```

Figure 5: Trusted subclassing

The technique shown in Figure 5 requires a `SecurityManager` to be installed, as described in Section 3.1.3.3, and a custom permission class, named `SubclassPermission`, to be created as described in Section 3.1.3.6. If the permission is not granted to the executing code that is attempting to subclass class `A`, a `SecurityException` is thrown.

Calling methods that are not `final` should not be made within a constructor since this potentially gives power to a malicious subclass, possibly prior to a class being fully initialized [2]. In addition, a subclass should not increase the accessibility of methods from its parent class. An example of a malicious overriding of a method is when the accessibility of a class's finalizer method is increased from `protected` to `public`. If this happens, external code can improperly invoke the `finalize` method and then proceed to call other methods on the finalized instance. The risk is that a superclass implementation may not be designed to handle such conditions. As a result, an instance may be left in an insecure state or may leak

sensitive information in thrown exceptions [2]. An exception to this guideline is when providing a `public clone` method for a `public` mutable class that implements the `Cloneable` interface [2].

Finalizers are invoked by the JVM before the storage for an object is reclaimed by the garbage collector. Since the JLS [18] does not specify when a finalizer will be invoked, finalizers should be avoided and should not perform time or security-critical operations [11, 25, 26:583]. The JLS also notes that it is "usually good practice" to invoke `super.finalize` inside finalizers.

If a `public` class that is not `final` must require a finalizer method, the "finalizer guardian idiom" [11] should be utilized to ensure the finalizer is executed, even in the face of a malicious subclass that may not call `super.finalize` [11]. The finalizer guardian idiom utilizes an anonymous inner class that exists only to finalize its enclosing instance. This has two implications: an additional object must be created for every object to be finalized [11] and the use of an inner class violates information hiding. As shown in Figure 4, the `Timer` class in Java SE 6 uses this idiom.

```
public class Timer {
    private TaskQueue queue = new TaskQueue();
    private TimerThread thread = new TimerThread(queue);

    /**
     * This object causes the timer's task execution thread to exit
     * gracefully when there are no live references to the Timer object and no
     * tasks in the timer queue.  It is used in preference to a finalizer on
     * Timer as such a finalizer would be susceptible to a subclass's
     * finalizer forgetting to call it.
     */
    private Object threadReaper = new Object() {
        protected void finalize() throws Throwable {
            synchronized(queue) {
                thread.newTasksMayBeScheduled = false;
                queue.notify(); // In case queue is empty.
            }
        }
    };

    // code suppressed
}
```

Figure 6: Finalizer idiom

3.2.4 Securing Serialization

If preserving object state is not needed, classes should not be able to be serialized [15, 26: 499] since serialization is prone to information leaks [2] and breaks information hiding [11]. If serialization is required, ensure sensitive fields are protected by implementing one of the following techniques [2]:

1. Declare sensitive fields as `transient` (if default serialization is used).
2. Implement `writeObject`, `writeReplace`, or `writeExternal` and ensure sensitive fields, including `transient` fields, are not written to the stream.
3. Utilize the `serialPersistentFields` array and only store fields that are not sensitive in it.

If a caller can retrieve the internal state of a `Serializeable` class (i.e., using an accessor method), and the retrieval is guarded with a `SecurityManager` check, the same check should be enforced in the implementation of a `writeObject` method [2].

For classes that are `Externalizable`, the `readExternal` and `writeExternal` methods are declared `public` and can therefore be invoked by any code that has access to the class. Such classes must ensure that the `readExternal` method is never invoked if the object was explicitly constructed by trusted code or is never invoked after the object has been properly deserialized (i.e., to ensure the class is initialized exactly once) [39].

3.2.5 Securing Deserialization

Even if a class is not `Serializeable`, it should not be capable of being deserialized [15]. If deserialization is possible, attackers can attempt to create a rogue sequence of bytes that deserializes into an instance of the class. To ensure a class cannot be deserialized, provide a `final readObject` and `final readObjectNoData` method that both throw an appropriate exception.

If deserialization must occur, the `readObject` method of the `Serializeable` class must not call methods that are not `final` since an attacker may override them [11], potentially in a

malicious manner. The `serialVersionUID` field should be declared `private` since it should only be applied to the declaring class [39].

Sun recommends viewing deserialization the same as invoking any constructor. Thus, the same security checks, validity checks, and default initialization of members should occur [2]. Techniques to prevent partially initialized or deserialized objects should be employed, such as a class initialization flag. If a class initialization flag is used, it should be declared `private` and `transient` and only set in `readObject` or `readObjectNoData` [2].

Sensitive mutable objects should be defensively copied prior to being assigned to class fields in the implementation of a `readObject` method [2, 11]. This prevents attempts that are made to obtain references to the internal state of an object.

As with serialization, if a method allows a caller to modify `private` internal state and the assignment is guarded with a `SecurityManager` check, the same check should be enforced in the implementation of `readObject` [2].

3.2.6 Securing Native Methods

Native methods allow for code that is written in other languages, such as C and C++, to interact with Java code. Since native methods completely bypass all security offered by the Java platform (i.e., access modifiers and memory protection) [52], security-critical applications should not use them.

When native methods are required, they should be declared `private` [2]. Native calls can then be wrapped in `public` accessor methods. The advantage of using wrappers is that `SecurityManager` and validity checks can be programmatically enforced prior to a native method call [2].

3.2.7 Securing Synchronization

Bloch suggests not invoking methods that are not `final` from within `synchronized` methods since these methods can be overridden by malicious subclasses [11]. In addition, Bloch suggests that lock objects should always be declared `private`. Otherwise, a caller can hold a lock in a malicious manner by preventing other callers from legitimately obtaining the lock (i.e., create a denial of service situation). The "private lock idiom [11]" shown in Figure 7 can be used to securely synchronize operations.

```
private Object lock = new Object();

public void foo() {
   synchronized(lock) {
      // call synchronized operations
   }
}
```

Figure 7: Using private lock objects for synchronization

Having empty `synchronized` blocks in code reflects poor style and may be indicative of error-prone, obsolete, or overly complex code that has security issues [25]. Such code should be reviewed for correctness.

3.2.8 Securing Privileged Code

As discussed in Section 3.1.3.4, a privileged code block temporarily escalates the permission set of a certain block of code. Sun recommends not invoking the `doPrivileged` method using caller provided input to avoid performing inadvertent operations that may jeopardize security on behalf of untrusted code [2, 17]. To aid readability and make auditing security-critical code easier, Gong et al. [17] recommend making privileged code as short as possible and wrapping `doPrivileged` method calls in `private` methods.

Furthermore, the semantics of the `doPrivileged` method are confusing and complicated. Most examples involve using an anonymous inner class to invoke `doPrivileged`, as shown in Figure 8 [17].

```
private final void privilegedMethod() {
   String user = (String)AccessController.doPrivileged(new PrivilegedAction() {
      public Object run() {
         return System.getProperty("user.name");
      }
   }
}
```

Figure 8: Common invocation of AccessController.doPrivileged

This idiom has several disadvantages: a dynamic cast is required on the returned value and an anonymous inner class is used thereby violating guidelines related to class design. Another disadvantage of using an anonymous inner class is that the code inside the privileged code block has access to other visible class entities, which it may not need. To hide information and avoid dynamic casting, Gong et al. [17] describe the idiom that is shown in Figure 9.

```
final class ThePrivilegedAction implements PrivilegedAction {
   private String property;
   private String result;

   public ThePrivilegedAction(String property) {
      this.property = property;
   }

   public Object run() {
      result = System.getProperty(property);
      return result;
   }

   public String getResult() {
      return result;
   }
}

private final void privilegedMethod() {
   ThePrivilegedAction pa = new ThePrivilegedAction("user.name");
   AccessController.doPrivileged(pa);
   String user = pa.getResult();
}
```

Figure 9: PrivilegedAction idiom

This idiom is advantageous: it is easier to understand, it removes the need for a dynamic cast on the return, it does not utilize an inner class, it promotes reuse, and the `ThePrivilegedAction` class hides information relevant to the sensitive operation. Whenever possible, this idiom should be used when invoking `doPrivileged`.

Finally, the `doPrivileged` method should not manipulate mutable objects that are declared `public`, including objects also declared `final` and `static`. Since a `doPrivileged` block escalates the set of permissions that are available, performing sensitive operations using `public` state is dangerous.

3.2.9 Securing Sensitive Standard API Calls

Certain standard Java APIs[10], such as `java.lang.Class.newInstance`, must be invoked carefully because they perform operations using the immediate caller's `ClassLoader`. As explained in Section 3.1.3.4, all callers in an execution context are required to have permission before a sensitive operation is performed. However, when invoking `java.lang.Class.newInstance` on a `Class` object, if the immediate caller's `ClassLoader` is an ancestor of the `Class` object's `ClassLoader`, then the `SecurityManager` check is bypassed [2]. The risk is that `java.lang.Class.newInstance` could be invoked on behalf of code that would otherwise not have permission to perform the action.

Other Java APIs, such as `java.lang.Class.forName`, perform tasks using the immediate caller's `ClassLoader` and must be invoked carefully as well. Sun recommends that these standard API calls should not be invoked on behalf of code that is not trusted. In addition, these method calls should not process user-provided input or return objects to code that is not trusted [2].

3.2.10 Securing Reflection

Similar to the standard API calls discussed in Section 3.2.9, during reflective calls, the JVM bases all language access checks on the immediate caller rather than each caller in the execution context. Thus, methods that perform reflection, such as `java.lang.reflect.Field.get`, should not be invoked by code that is not trusted [2].

---

[10] The complete list of standard API calls that can cause trouble are incorporated into the taxonomy in Chapter 4. The list can also be also found online at Sun's secure coding guidelines site [2].

Additionally, methods that call reflective methods should not be invoked with input provided by untrusted code, and objects returned from reflective methods should not be propagated to untrusted code [2].

3.2.11 Securing Errors and Exceptions

All class methods that return a value should be checked for error or status codes [25, 26: 389]. If return values are not checked, errors may not be detected. To detect all possible exceptions, exceptions should not be caught in an overly-broad manner [25, 26: 396]. Likewise, exceptions should not be thrown in an overly-broad manner [25, 26: 397]. Gong et al. [17] suggest not swallowing exceptions that have security relevance, such as `java.security.AccessControlException` or `java.lang.SecurityException` since this may ignore attempts to bypass security.

Sun warns that sensitive information should be purged from exceptions before being propagated upstream, by using one of the following techniques [2]:

- Throwing a new instance of the same exception but with a sanitized message.
- Throwing a different type of exception and message (which is not sensitive) altogether.

It is good practice to log exceptions in a secure manner so that errors and exceptional conditions can be traced and reconstructed. However, care must be taken to ensure sensitive information is not leaked when it is logged [43], which may occur by printing exceptions to an output stream such as the console [25].

3.2.12 Securing Objects in Transit

To provide for object authenticity, or data integrity, Gong et al. [17] recommend that `Serializable` objects be signed. The `SignedObject` class serves this purpose. An instance of `SignedObject` contains a deep copy of a `Serializable` object to be signed and its signature. The object is always serialized before its digital signature is generated [17].

To provide for confidentiality, Gong et al. [17] also recommend that objects be sealed. The `SealedObject` class serves this purpose. An instance of `SealedObject` contains an encrypted form of a `Serializable` object. Like `SignedObject`, a deep copy of the original object is sealed. An instance of `SealedObject` can be decrypted using an appropriate key and deserialized to retrieve the original object.

If object-level access control is needed, objects should be guarded [17]. The `GuardedObject` class serves this purpose. An instance of `GuardedObject` contains an object to be guarded and an instance of `Guard`. The instance of `Guard` provides logic for protecting access to the object being guarded. When a client attempts to retrieve the object being guarded, the `Guard` instance returns silently if access is permitted or throws a `SecurityException` if access is not permitted. This mechanism was added in J2SE 1.2 so that access control can be enforced within the execution context of a consumer when the supplier exists in a different execution context [17].

Signing and sealing objects are important for securely transmitting objects across a network. To provide both object integrity and object confidentiality, an instance of `SignedObject` can be sealed [17]. However, signing and sealing objects introduces complexity: keys that are utilized to perform code signing and sealing must be managed securely [17].

3.2.13 Securing Dynamically Loaded Code

The powerful feature of being able to dynamically load and execute code at runtime presents severe security risks. It is crucial for code that originates from two unrelated locations to be loaded by separate `ClassLoader` instances [23, 51]. If this does not occur, namespaces will not be defined properly. As a result, trusted and untrusted code may execute in the same areas of memory.

While using separate `ClassLoader` instances is important, it is just as important to grant safe permissions to external code bases. For instance, the `java.security.AllPermission` permission should never be granted to external code bases. The "fail-safe" [1] approach is to grant permissions that are explicitly needed by external code.

3.2.14 Securing Mechanisms that Provide Security

Passwords and other secret information should never be stored in source code. Storing passwords in classes that connect to databases (e.g., when using `DriverManager.getConnection`) is a common problem [28]. A more secure approach is to store secrets in an encrypted form in external configuration files that only authorized entities can access [43].

Sensitive data, such as passwords or cryptographic keys, should never be stored in an immutable class, such as `String`, since removing such data from memory is dependent on the garbage collector. A better approach is to use a character array (e.g., the `getPassword` method of the `JPasswordField` class) and explicitly clear the array (e.g., using `Arrays.fill`) when it is no longer needed [17].

It is crucial for mechanisms that perform cryptographic operations to use sound algorithms. For instance, random numbers should be generated securely using cryptographically secure APIs, such as the `SecureRandom` class [29]. Another example involves the `Cipher` class, which can transform information using many different algorithms. The DES algorithm, which is known to be weak [4], can be specified when constructing a `Cipher` instance. Algorithms that are known to be strong, such as AES, should be used with the `Cipher` class instead.

3.3 Summary

The first part of this chapter provided a brief overview of Java platform security. The second part surveyed and discussed numerous guidelines for writing secure code in Java. It is clear that there are many complex aspects of both the Java programming language and runtime environment that make developing secure Java applications difficult. The next chapter aims to make understanding and remembering these issues easier by associating concrete, implementation-level coding advice for increasing the quality and security of Java code with abstract design principles.

4 A New Taxonomy of Design Principles and Heuristics

4.1 Motivation

In the research literature, principles for creating secure, high-quality designs and guidelines for achieving secure code are discussed in isolation. Design principles tend to be philosophically described at a high-level of abstraction while coding guidelines are stated as concrete implementation advice. Developers, who are striving to write secure code while also following principles of secure design, need better guidance about how to realize design principles in code. In other words, statements are needed about how design principles can be achieved when writing code.

To fulfill this need, the taxonomy described in this chapter correlates heuristics, or rules, for writing secure code in Java with the design principles those rules help to achieve. The taxonomy's classification scheme aims to be as complete as possible at the top-most (i.e., principle) level. New rules may be added at lower-levels as new coding practices for writing secure code in Java are discovered. The ultimate goal is to show coherent relationships between design principles and their associated coding rules so that both principles and coding heuristics are easier to understand, apply, and remember.

4.2 Definitions

This section defines key terms that are found within the taxonomy.

- A *module* is a conceptually simple program unit with parts that has a well-defined interface [10].
- An entity `A` may deem an entity `B` *untrusted* if `B` resides entirely outside `A`'s control, or at least outside `A`'s control at some point in time.
- Privileged code refers to any invocation of the `doPrivileged` method of the `AccessController` class.
- Any entity that is *security-critical* is needed to enforce some aspect of a security policy[11]. In Java, for instance, any module that contains privileged code or that issues a `SecurityManager` check is security-critical.

[11] Bishop's definition of a security policy applies here: "a statement of what is, and what is not, allowed." [4]

- Any entity that is *sensitive* is protected by one or more security-critical entities.
- An entity is *fragile* if it is easily broken when changed. In the object-oriented paradigm, inheritance makes entities fragile, as changes to superclasses may inadvertently impact subclasses.
- *External code bases* refer to code that is located at external locations. Applets, for example, are downloaded from external code bases. In Java, permissions can be granted to external code bases.

4.3 Methodology

As shown in Figure 10, the new taxonomy consists of elements from three different levels of abstraction. At the highest level of abstraction are *design principles*. When design principles are followed, better overall designs are achieved. Design principles were chosen from the previous work surveyed in Chapter 2, with the addition of a few new principles and changes to names.

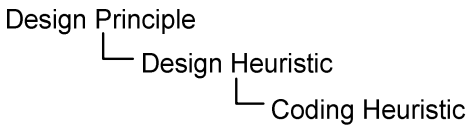

Figure 10: Taxonomy levels of abstraction

At an intermediate level of abstraction are *design heuristics*. Design heuristics are language-independent rules for creating designs that help to satisfy design principles. Design heuristics serve an important role in the taxonomy: to bridge the conceptual gap that exists between concrete coding practices and abstract design principles.

At the lowest level of abstraction are *coding heuristics*. Coding heuristics are rules, which may or may not be language-specific, for producing code that helps to achieve design heuristics. The numerous guidelines for securing Java code that were discussed in Chapter 3 are specified as coding heuristics.

The organization of design principles within the taxonomy appears in Figure 11. As shown, elements may subsume other elements that are positioned at the same level of abstraction. This may occur at design heuristic and coding heuristic levels as well.

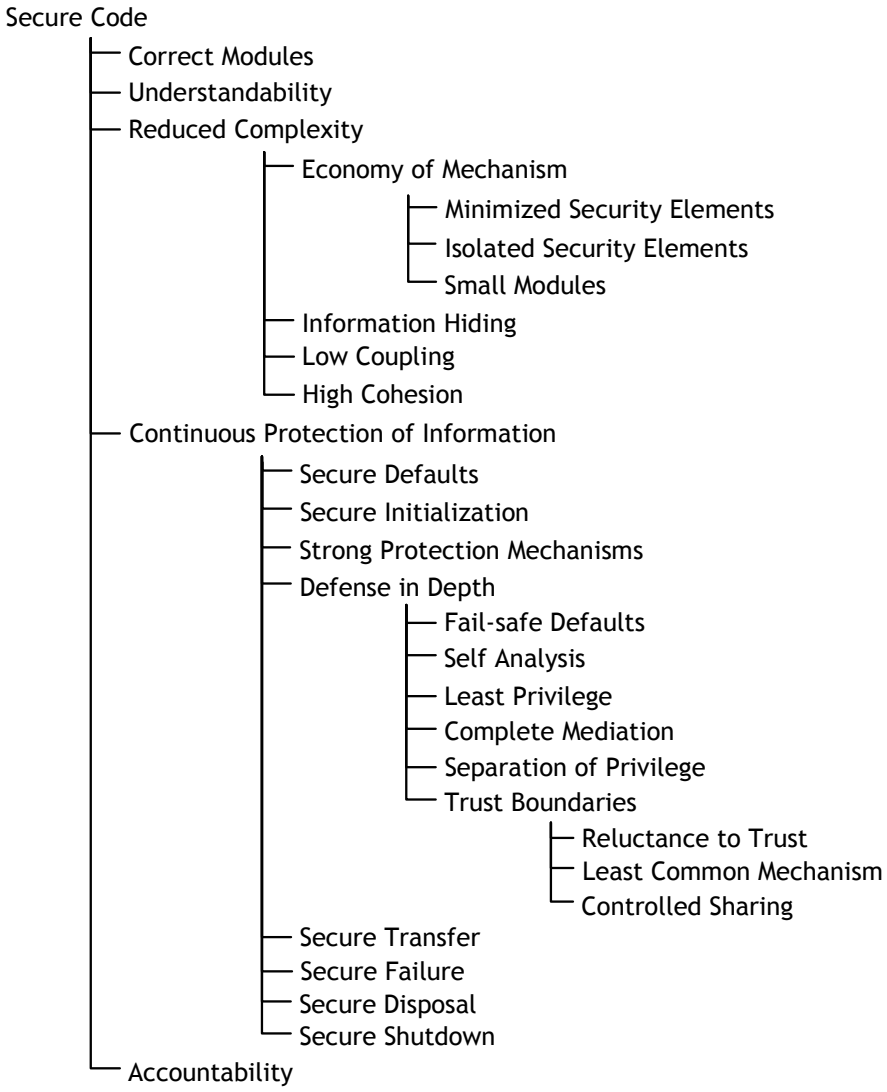

Figure 11: Design principles and their relationships

Classification Scheme

In accord with the definition of secure code in Section 2.1.1, the principles at the top-most level describe designs with modules that have the following characteristics:

- Are made of correct parts
- Are made of unambiguous parts
- Are made of structurally simple parts
- Have mechanisms for protecting information in all system contexts
- Have mechanisms for tracing actions to the entities that cause them

Achieving simplicity (i.e., having reduced complexity) is accomplished by decomposing a system into modular components through information hiding, low coupling, and high cohesion. Making modules small and simple (i.e., having economy of mechanism) plays an essential role. With respect to secure design, having economy of mechanism means there should be as few security-critical modules as possible (i.e., minimized security elements), security-critical modules should be easily identifiable and separated from less-critical modules (i.e., isolated security elements), and security-critical modules should be as structurally small as possible.

The goal of continuously protecting information means that information should be protected in accord with a security policy during the "creation, storage, processing, and communication of information as well as during system initialization, execution, failure, interruption, and shutdown" [8]. To protect information, defensive mechanisms must exist. Defense in depth is a strategy for designing modules so that multiple restriction-based mechanisms are exercised when interactions between entities occur, thereby making it harder for one or more malicious actions to completely compromise a system. When data is exchanged or untrusted entities are involved, an implicit trust boundary is crossed. In such situations, it is prudent to assume maliciousness (i.e., be reluctant to trust), minimize the mechanisms that are common to entities (i.e., have least common mechanism), and control how data is shared when it is accessed simultaneously (i.e., have controlled sharing).

4.4 Design Principles

The principle of ***correct modules*** states that designs with modules that meet specifications and that are made by following safe programming practices are better. Striving to construct correct modules does not have a major influence on shaping designs;

correctness depends on programmers being competent. However, many types of software failures that have an impact on security are caused by incorrect modules [29][12].

The principle of ***understandability*** states that designs with modules that are easier to understand are better. The ability to identify and trace functionality throughout source code depends on how easily source code can be read and understood. To provide for understandability, designs must be realized by writing code in clear and consistent ways while avoiding the use of confusing language constructs.

The principle of ***reduced complexity*** [8] states that designs with modules that are less complex are better. This principle is also referred to as simplicity [10] and relies on the claim that simpler systems have fewer vulnerabilities [8] and faults. Managing system complexity has been said to be the most important concept in software development [42].

The principle of ***economy of mechanism*** [1] states that designs with modules that are simple and small are better. While this principle applies to any module, it is critical for modules that provide security-critical functionality to be conceptually small and simple so they can be analyzed [3].

The principle of ***minimized security elements*** [8] states that designs that have a minimal number of security-critical modules are better. Reducing the amount of security-critical functionality that is employed throughout a system makes it easier to analyze and easier to verify that security-critical mechanisms are correct [3].

The principle of ***isolated security elements*** states that designs that isolate modules that provide security-critical functionality are better. This principle has structural importance:

[12] No attempt is made to include all defects that have security relevance as the list is arguably endless. Rather, a representative set of problems is provided. The reader should consult the CWE [26] for a more comprehensive list.

designs that separate security-critical modules from less-critical modules make identifying, protecting, changing, and testing such modules easier.

The principle of ***small modules*** [10] states that designs with small modules are better. While this principle applies to all aspects of a design, it is crucial for modules that provide security-critical functionality to have the least number of lines of code and interface operations as possible.

The principle of ***information hiding*** [10, 12] states that designs with modules that shield the details of their internal structure and processing from other modules are better. This principle helps to create a design with "clear abstractions" [8], which describes modules that have simple, unambiguous interfaces.

The principle of ***low coupling*** [10] states that designs that minimize the degree of connection between pairs of modules are better. A related concept is the principle of ***high cohesion*** [10], which states that designs that maximize the degree to which a module's parts are related to one another are better. It is more likely for highly cohesive modules to have low coupling.

The principle of ***continuous protection of information*** [8] states that designs that perform operations that continuously protect sensitive information in every system state are better.

The principle of ***secure defaults*** [8] states that designs with modules that have secure initial configurations are better. Although tradeoffs between security and other quality factors, such as usability and performance, are inevitable, it is prudent to err on the side of security.

The principle of ***secure initialization*** [45] states that designs with modules that perform initialization functionality in a secure manner are better. Programmers must be

aware of programming language subtleties concerning initialization mechanisms. In Java, for instance, since the serialization API does not invoke class constructors, all class initialization that occurs in constructors must also be present within implementations of `readObject` and `readObjectNoData`.

The principle of ***strong protection mechanisms*** states that designs with modules that provide protection in the most secure manner possible are better. This principle acknowledges that developers should "assume the attacker has access to all source code and all designs" [5]. Protection mechanisms should be chosen based on good engineering judgment or known strength rather than on the assumption that secrets will remain hidden [9] or that introducing complexity will add security (i.e., the security through obscurity approach) [4].

The principle of ***defense in depth*** [3] states that designs that establish protective barriers across multiple dimensions of a module are better. When each layer of defense works in concert with all other layers [9], defense in depth helps to reduce the likelihood of a single point of failure [5].

The principle of ***fail-safe defaults*** [1] states that designs with modules that deny access to objects unless entities have been granted explicit access permissions are better [4]. In other words, the default case should be to not allow a sensitive operation to take place.

The principle of ***self analysis*** [8] states that designs with modules that assess their own internal state are better. Modules that perform self analysis are able to detect and securely recover from faults by permitting access to modules only when operating in valid states. In the object-oriented paradigm, performing self analysis can ensure that classes are only used after they are fully constructed. In Java, self analysis can help to prevent a *finalizer*

attack, whereby malicious subclasses obtain a reference to a partially initialized super class during finalization and attempt to call methods on it [2, 51].

The principle of ***least privilege*** [1] states that designs with modules that do not have access to unneeded resources are better [10]. From a security standpoint, designs that produce execution contexts that operate using the least set of privileges necessary to complete the job are better.

The principle of ***complete mediation*** [1] states that designs with modules that check every access to sensitive objects for authorization are better. In Java, for instance, if a `Cloneable` class has a `SecurityManger` check in its constructor, it is important to ensure the same `SecurityManager` check is provided in its `clone` method. Because class objects can be cloned without invoking a constructor, `SecurityManager` checks that are placed in constructors can be bypassed.

The principle of ***separation of privilege*** [1] states that designs with modules that require two or more conditions to be met before an action is permitted are better. To illustrate this principle in Java, consider a malicious class named `M` that attempts to subclass a sensitive class named `SensitiveClass`. One design alternative is to require two conditions to be met before `M` is allowed to subclass `SensitiveClass` by:

- Utilizing the `package.access` system property to require `M` to have permission to access entities within the package that contains `SensitiveClass`.
- Enforcing a `SecurityManager` check in the constructor of `SensitiveClass` to ensure `M` has the necessary permission to subclass `SensitiveClass`.

The principle of ***trust boundaries*** [5] states that designs that clearly establish domains of trust between interacting modules that exchange or manipulate data are better. Violations of trust represent one of the most common types of software weaknesses [3, 5]. Chess and West [43] note that the purpose of input validation is to allow data to move from

untrusted domains to trusted domains (i.e., across a trust boundary). This fundamental concept is perhaps one of the most important design goals to achieve. In practice, however, this goal appears to be the most challenging: improperly handling input is suspected to be the most common cause of software vulnerabilities [9].

The principle of ***reluctance to trust*** [9, 50] states that it is better to have designs with modules that assume all interactions with external entities are malicious. This principle says that all external data that crosses a trust boundary should be considered "to be insecure and a source of attack [5]." While common input-related vulnerabilities result from misplaced trust, the object-oriented paradigm presents new challenges due to inheritance and polymorphism. It is not sufficient to only place restrictions on the type of information that can flow between objects. In systems where potentially malicious code is loaded dynamically, objects must not trust other objects based on type safety. For instance, calling non-final methods from within the constructor of a non-final class is dangerous since objects loaded dynamically at runtime may be able to maliciously override the method [2].

The principle of ***least common mechanism*** [1] states that designs with modules that minimize the number of shared access paths to information are better. Shared mechanisms may provide opportunities for two or more entities to communicate or interact in malicious or inadvertent ways [4]. At the code level, shared mechanisms can be prevented by creating new instances of objects instead of allowing entities to use a shared instance [3]. When different instances are used, if one instance is attacked or becomes corrupt, the other instances are less likely to be affected.

The principle of ***controlled sharing*** states that designs whose modules share information in a controlled manner are better. When two or more modules can access shared information potentially at the same time (e.g., in multithreaded environments), there is

potential for timing and sequencing errors to result in race conditions or deadlocks. When race conditions occur, the integrity of information may be compromised or complete failures may occur. Deadlocks lead to failure. If a security-critical module fails in an insecure state, attackers may be able to bypass protection mechanisms [44]. Viega and McGraw note that file system race conditions, or TOCTOU flaws, are the most common type of race conditions that pose a threat to system security [9].

The principle of ***secure transfer*** states that designs with modules that protect information that is transferred outside complete control of a software system are better. This includes the transfer of information to persistent storage, such as disk, and over network channels, such as sockets.

The principle of ***secure failure*** [8] states that designs with modules that do not jeopardize security when failures occur are better. When this principle is followed, modules are designed to make a transition to a secure state that denies rather than permits access when failures are detected [8]. Additionally, this principle says that modules should not reveal sensitive information during failures that may aid attackers.

The principle of ***secure disposal*** [45] states that designs with modules that dispose of sensitive information when it is no longer needed are better. When this principle is followed, sensitive objects, such as passwords and cryptographic keys, are completely removed from memory after they are used.

As discussed above, while it is important to ensure modules are initialized securely, it is equally important to ensure modules are terminated in a secure manner: the principle of ***secure shutdown*** [45] states that designs with modules that do not jeopardize security when performing shutdown or termination functionality are better.

The principle of ***accountability*** [8] states that designs that record security-relevant sequences of actions and trace them to the entity (e.g., module or user) that caused the actions to occur are better. This principle has also been called "compromise recording" [1] and "recording of compromises" [3].

4.5 The New Taxonomy

This section contains the complete taxonomy[13]. Design principles are bolded and given an acronym, design heuristics are identified by a principle identifier combined with a trailing number, and coding heuristics are identified by a design heuristic identifier combined with a trailing letter.

1. Secure Code

1.1 Correct Modules (CMo)

CMo.1 Adhere to API specifications.

CMo.1.a Avoid calling `Thread.run` [26:543].

CMo.1.b Ensure a `clone` method calls `super.clone` [26:580].

CMo.1.c Ensure implementations of `readObject` and `writeObject` of `Serializable` classes are `private`.

CMo.1.d In EJB, avoid using `Socket` connections [26:577].

CMo.1.e In EJB, avoid performing `ClassLoader` operations [26:578].

CMo.1.f In EJB, avoid performing `SecurityManager` operations [26:578].

CMo.1.g In J2EE, use web application container mechanisms to obtain connections to resources [26:245].

CMo.1.h In J2EE, avoid using `Socket` connections [26:246].

CMo.1.i In J2EE, ensure classes that extend validation forms override the inherited `validate` method and call `super.validate` [26:103].

CMo.1.j In J2EE, ensure form beans extend an `ActionForm` subclass of the validator framework [26:104].

CMo.1.k Ensure a `finalize` method calls `super.finalize` [26:568].

CMo.1.l Ensure the `serialPersistentFields` member of `Serializable` classes is `private`, `static`, and `final`.

CMo.2 Follow safe programming practices.

CMo.2.a Avoid using deprecated APIs.

CMo.2.b Avoid omitting `break` statements in `switch` statements [26:484].

CMo.2.c Always account for the `default` case in `switch` statements [26:478].

CMo.2.d Avoid comparing classes by name.

CMo.2.e Check preconditions of `public` methods and throw exceptions.

1.2 Understandability (U)

U.1 Remove unused or obsolete functionality from modules.

U.1.a Remove unused classes, imports, interfaces, methods, parameters, and fields.

---

[13] The taxonomy can be viewed interactively online at the following URL: http://peregrin.jmu.edu/~warems

U.1.b In J2EE, remove fields from validation forms that do not map to fields in action forms [26:110].

U.1.c In J2EE, remove unused validation forms [26:107].

U.2 Avoid using confusing language constructs when more understandable alternatives exist.

U.2.a Avoid using `static` initializer blocks to initialize class variables - consider using `private final static` methods instead.

U.2.b Avoid using instance initializer blocks to initialize instance variables - consider using `private` or `protected final` methods instead.

U.3 Avoid using confusing, inconsistent, or duplicate names for entities.

U.3.a Class names should not include the names of other classes -- unless it is a subclass of them.

U.3.b In J2EE, ensure validation forms have unique names [26:102].

1.3 Reduced Complexity (RC)

1.3.1 Economy of Mechanism (EoM)

1.3.1.1 Minimized Security Elements (MSE)

MSE.1 Employ as few security-critical modules as possible.

MSE.1.a Invoke `AcessController.doPrivileged` minimally.

MSE.1.b Minimize the number of distinct classes with privileged code.

1.3.1.2 Isolated Security Elements (ISE)

ISE.1 Separate security-critical modules from other modules.

ISE.1.a Consider making all `Permission` classes belong to the same package.

ISE.1.b Consider making all `PrivilegedAction` and `PrivilegedActionException` classes belong to the same package.

ISE.1.c Consider making all privileged code belong to the same package.

ISE.1.d Avoid utilizing anonymous `PrivilegedAction` classes when invoking `AccessController.doPrivileged` - create classes for distinct `PrivilegedActions` instead.

1.3.1.3 Small Modules (SM)

SM.1 Keep all modules, especially security-critical modules, small.

SM.1.a Limit the number of lines of code within privileged code blocks to 10 lines.

SM.1.b Limit the number of lines of code within all methods to 60 lines.

1.3.2 Information Hiding (IH)

IH.1 Limit entity accessibility.

IH.1.a Declare class entities (fields, methods, nested classes, nested interfaces) `private`.

IH.1.a.a Declare mutable `static` fields `private`.

IH.1.a.b Declare lock variables `private`.

IH.1.a.c Wrap privileged code blocks in `private` methods.

IH.1.a.d Avoid `native` methods; if they are required, declare them `private`.

IH.1.a.e Declare class initialization fields `private` and `transient` (for `Serializable` classes).

IH.1.a.f Declare the `serialVersionUID` field of `Serializable` classes private.

IH.1.a.g Declare `transient` fields `private`.

IH.1.b Declare classes and interfaces `package` - unless they are documented in a public API.

IH.1.c Avoid making subclass constructors more accessible than superclass constructors.

IH.1.d Avoid increasing the accessibility of inherited methods.

IH.1.e Avoid inner classes; when used, ensure an inner class is no more accessible than its enclosing class.

IH.2 Limit entity extensibility.

IH.2.a Declare classes `final`.

IH.2.a.a Declare `Permission` and `BasicPermission` subclasses `final`.

IH.2.a.b Declare `PrivilegedAction` and `PrivilegedActionException` subclasses `final`.

IH.2.b Declare methods `final`.

IH.2.b.a Declare methods that contain privileged code `final` -- especially in non-final classes.

IH.2.b.b Declare methods that enforce `SecurityManager` checks `final` -- especially in non-final classes.

IH.3 Limit internal object state exposure.

IH.3.1 Limit exposure when accessing object entities.

IH.3.1.a Avoid providing multiple accessor methods for the same field.

IH.3.2 Limit exposure when making objects persistent.

IH.3.2.a If default serialization is used, declare sensitive fields `transient`.

IH.3.2.b Ensure sensitive fields, including `transient` fields, are not written to `writeObject`, `writeReplace`, or `writeExternal` streams.

IH.3.2.c Ensure sensitive fields, including `transient` fields, are not stored in the `serialPersistentFields` array.

IH.3.2.d Make sensitive classes prevent deserialization by providing a `final readObject` method and throwing exceptions.

IH.3.2.e Make sensitive classes prevent serialization by providing a final `writeObject` method and throwing exceptions.

IH.3.3 Limit exposure when copying objects.

IH.3.3.a Consider providing `clone` functionality for `public` mutable classes only.

IH.3.3.b Make sensitive classes prevent cloning by providing a `final clone` method and throwing exceptions.

1.3.3 Low Coupling (LC)

LC.1 Minimize module dependencies.

LC.1.a Minimize references to global entities (e.g., `public` class fields, external files, etc.)

LC.1.b Minimize calling methods on other class instances.

LC.1.c Require as few class method parameters as possible.

LC.1.d Consider making class method parameters have `interface` types.

1.3.4 High Cohesion (HC)

HC.1 Keep related data and behavior in the same module.

HC.1.a Maximize references to `private` or `protected` entities (e.g., fields and methods) from `private` or `protected` methods within the same module.

1.4 Continuous Protection of Information (CPoI)

1.4.1 Secure Defaults (SDe)

SDe.1 Ensure the default configurations of modules are secure.

SDe.1.a In J2EE, ensure session identifiers are configured to be at least 128 bits in length [26:6].

1.4.2 Strong Protection Mechanisms (SPM)

SPM.1 Utilize secrets securely.

SPM.1.1 Avoid hard-coding passwords in source code -- consider storing encrypted passwords in external configuration files instead [43].

SPM.1.1.a Avoid using hard-coded passwords in the `getConnection` method of the `DriverManager` class.

SPM.2 Use cryptographically strong algorithms.

SPM.2.a Use an AES `Cipher` when using `SealedObject`.

SPM.2.b Use a SHA-256, SHA-384, or SHA-512 algorithm when using `MessageDigest.getInstance` to perform cryptographic hashing.

SPM.2.c Use the RSA algorithm when using `KeyPairGenerator.getInstance` and specify (i.e., calling `KeyPairGenerator.initialize`) a key size of at least 1024 bits [43].

SPM.3 Generate truly random numbers.

SPM.3.a Use the `SecureRandom` class to generate random numbers.

1.4.3 Secure Initialization (SI)

SI.1 Ensure modules are securely initialized.

SI.1.a All default class initialization that occurs in constructors for `Serializable` classes should also occur within `readObject` and `readObjectNoData`.

SI.1.b Avoid throwing exceptions in constructors, especially in `public`, non-final classes.

1.4.4 Defense in Depth (DiD)

1.4.4.1 Fail-safe Defaults (FsD)

FsD.1 Deny access to objects unless entities have explicit access permission.

FsD.1.a Install a `SecurityManager` by invoking `System.setSecurityManager` – do not rely on the `-Djava.security.manager` command line execution argument.

FsD.1.b Enforce a default-deny security policy when granting permissions (i.e., `Permission` subclasses) to external code bases.

1.4.4.2 Self Analysis (SA)

SA.1 Ensure objects are used only when operating in valid, secure states.

SA.1.a Consider enforcing class initialization checks (e.g., using an initialized flag) in `public` and `protected` methods -- especially in sensitive non-final classes and `Serializable` classes.

SA.1.b Ensure the `readExternal` method of `Externalizable` classes is invoked once during deserialization (e.g., using an initialized flag).

SA.1.c Check all method return values.

1.4.4.3 Least Privilege (LP)

LP.1 Ensure external code bases have minimal privileges.

LP.1.a Grant explict permissions (i.e., `Permission` subclasses) to external code bases only as needed and enforce them using `SecurityManager` checks.

LP.1.b Never grant `java.security.AllPermission` to external code bases.

LP.2 Ensure module entities have minimal privileges.

LP.2.a Remove unnecessary classes, imports, interfaces, methods, parameters, and fields.

LP.2.b Ensure privileged code blocks only have access to code that is necessary to perform sensitive operations.

LP.2.c Consider making parameters of `public` constructors and methods `final` – especially when handling critical data values.

LP.2.d Consider making class fields `final` – especially when handling critical data values.

1.4.4.4 Complete Mediation (CMe)

CMe.1 Authorize all instantiations of sensitive objects.

CMe.1.a Consider enforcing `SecurityManager` checks within `public` and `protected` constructors (or `public` factory methods) of sensitive classes -- especially when `SecurityManager` checks are present within `clone`, `readObject`, or `readObjectNoData`.

CMe.1.b Ensure all `SecurityManager` checks that occur in constructors of `Cloneable` classes also occur within `clone` (or methods that provide clone-like functionality).

CMe.1.c Ensure all `SecurityManager` checks that occur in constructors of `Serializable` classes also occur within `readObject` and `readObjectNoData`.

CMe.2 Authorize all accesses to the internal state of sensitive objects.

CMe.2.a Consider enforcing `SecurityManager` checks before setting internal state in all `public` and `protected` mutator methods of sensitive classes.

CMe.2.b If a `Serializable` class invokes a `SecurityManager` check before internal state can be modified (e.g., in `public` mutator methods), enforce the same check within `readObject`.

CMe.2.c Consider enforcing `SecurityManager` checks before retrieving internal state in all `public` and `protected` accessor methods of sensitive classes.

CMe.2.d If a `Serializable` class invokes a `SecurityManager` check before internal state can be retrieved (e.g., in `public` accessor methods), enforce the same check within `writeObject`.

CMe.2.e Consider enforcing `SecurityManager` checks during the retrieval of sensitive objects using `GuardedObjects`.

CMe.3 Authorize all accesses to sensitive communication channels.

CMe.3.a Consider enforcing `SecurityManager` checks prior to transferring or retrieving sensitive information over `Sockets`.

1.4.4.5 Separation of Privilege (SoP)

SoP.1 Require multiple permissions before granting access to modules.

SoP.1.a Consider requiring permission for untrusted code to access sensitive packages using the `package.access` security property.

SoP.1.b Consider requiring permission for untrusted code to join sensitive packages using the `package.definition` security property.

SoP.1.c Consider requiring permission for untrusted code to subclass sensitive `public`, non-final classes by enforcing `SecurityManager` checks in constructors.

1.4.4.6 Trust Boundaries (TB)

1.4.4.6.1 Reluctance to Trust (RtT)

RtT.1 Assume input data is maliciously crafted.

RtT.1.1 Validate all input data (i.e., parameters).

RtT.1.1.a Validate untrusted input before passing it to privileged code blocks.

RtT.1.1.b Perform validation checks on defensive copies rather than on original mutable objects.

RtT.1.1.c Ensure all input validation checks that occur in constructors of `Serializable` classes also occur within `readObject` and `readObjectNoData`.

RtT.1.1.d In J2EE, ensure action forms have associated validation forms [26:108].

RtT.1.1.e In J2EE, ensure all form fields, whether populated with data or not, are validated [26:105].

RtT.1.2 Securely interact with external modules using input data.

RtT.1.2.a Validate and encode untrusted input before using it as command parameters when invoking `Runtime.exec` and `ProcessBuilder.start`.

RtT.1.2.b Use absolute file paths when opening files with untrusted input.

RtT.1.2.c Validate and encode untrusted input before saving it to log files.

RtT.1.3 Keep input data and control information separate.

RtT.1.3.a Make use of data value placeholders in parameterized statements when executing database queries using `java.sql.PreparedStatement`.

RtT.1.3.b Use XPath variables when evaluating XML queries using `javax.xml.xpath.XPathVariableResolver` [43].

RtT.2 Assume interactions with fragile module entities are malicious.

RtT.2.a Only call `final` methods within privileged code blocks of non-final classes.

RtT.2.b Only call `final` methods within synchronized blocks of non-final classes.

RtT.2.c Only call `final` methods within `readObject` and `readObjectNoData` of non-final `Serializable` classes.

RtT.2.d Only call `final` methods within constructors of non-final classes.

RtT.2.e Only call `final` methods within clone of non-final classes.

RtT.2.f Only invoke `clone` on instances of `final` classes.

RtT.3 Assume requests from external sources are malicious.

RtT.3.a Avoid invoking methods that bypass `SecurityManager` checks depending on the immediate caller's `ClassLoader` on behalf of untrusted code -- especially with untrusted input [2].

- RtT.3.a.a Avoid invoking `Class.newInstance` with untrusted input.
- RtT.3.a.b Avoid invoking `Class.getClassLoader` with untrusted input.
- RtT.3.a.c Avoid invoking `Class.getClasses` with untrusted input.
- RtT.3.a.d Avoid invoking `Class.getField(s)` with untrusted input.
- RtT.3.a.e Avoid invoking `Class.getMethod(s)` with untrusted input.
- RtT.3.a.f Avoid invoking `Class.getConstructor(s)` with untrusted input.
- RtT.3.a.g Avoid invoking `Class.getDeclaredClasses` with untrusted input.
- RtT.3.a.h Avoid invoking `Class.getDeclaredField(s)` with untrusted input.
- RtT.3.a.i Avoid invoking `Class.getDeclaredMethod(s)` with untrusted input.
- RtT.3.a.j Avoid invoking `Class.getDeclaredConstructor(s)` with untrusted input.
- RtT.3.a.k Avoid invoking `ClassLoader.getParent` with untrusted input.
- RtT.3.a.l Avoid invoking `ClassLoader.getSystemClassLoader` with untrusted input.
- RtT.3.a.m Avoid invoking `Thread.getContextClassLoader` with untrusted input.

RtT.3.b Avoid invoking methods that use the immediate caller's `ClassLoader` to perform operations on behalf of untrusted code -- especially with untrusted input [2].

- RtT.3.b.a Avoid invoking `Class.forName` with untrusted input.
- RtT.3.b.b Avoid invoking `Package.getPackage(s)` with untrusted input.
- RtT.3.b.c Avoid invoking `Runtime.load` with untrusted input.
- RtT.3.b.d Avoid invoking `Runtime.loadLibrary` with untrusted input.
- RtT.3.b.e Avoid invoking `System.load` with untrusted input.
- RtT.3.b.f Avoid invoking `System.loadLibrary` with untrusted input.
- RtT.3.b.g Avoid invoking `DriverManager.getConnection` with untrusted input.
- RtT.3.b.h Avoid invoking `DriverManager.getDriver(s)` with untrusted input.
- RtT.3.b.i Avoid invoking `DriverManager.deregisterDriver` with untrusted input.
- RtT.3.b.j Avoid invoking `ResourceBundle.getBundle` with untrusted input.

RtT.3.c Avoid invoking methods that perform accessibility checks using the immediate caller's `ClassLoader` on behalf of untrusted code -- especially with untrusted input [2].

- RtT.3.c.a Avoid invoking `Constructor.newInstance` with untrusted input.
- RtT.3.c.b Avoid invoking `Field.set*` with untrusted input.
- RtT.3.c.c Avoid invoking `Field.get*` with untrusted input.
- RtT.3.c.d Avoid invoking `Method.invoke` with untrusted input.
- RtT.3.c.e Avoid invoking `AtomicIntegerFieldUpdater.newUpdater` with untrusted input.
- RtT.3.c.f Avoid invoking `AtomicLongFieldUpdater.newUpdater` with untrusted input.
- RtT.3.c.g Avoid invoking `AtomicReferenceFieldUpdater.newUpdater` with untrusted input.

RtT.4 Avoid invoking less trusted modules from within more trusted modules.

- RtT.4.a Avoid invoking methods on dynamically loaded code within the scope of privileged code blocks.

RtT.4.b Avoid invoking `native` methods – especially with untrusted input.

1.4.4.6.2 Controlled Sharing (CS)

CS.1 Perform atomic operations on shared data.

CS.1.a Use the `synchronized` keyword on private lock variables to provide mutual exclusion in multithreaded modules.

CS.1.b Open file streams using file handles (e.g., `File` objects) instead of file names, especially within the same module.

1.4.4.6.3 Least Common Mechanism (LCM)

LCM.1 Define separate namespaces for dynamically loaded code.

LCM.1.a Ensure code from different code bases is dynamically loaded using separate `ClassLoader` instances.

LCM.2 Minimize information sharing through module interfaces.

LCM.2.a Pass defensive copies of `private` mutable objects as parameters.

LCM.2.b Return defensive copies of `private` mutable objects from methods.

LCM.2.b.a Ensure implementations of `clone` (i.e., `Cloneable`) return defensive copies.

LCM.2.b.b Ensure `public` factory methods that implement clone-like functionality return defensive copies.

LCM.2.c Defensively copy `public` mutable input parameters - especially when assigning to `private` fields.

LCM.2.d Avoid referencing `public` mutable fields, even those `final` and `static` -- if required, make defensive copies.

LCM.2.e Declare `public static` fields that are objects `final` and make them immutable.

LCM.2.f Declare `public static` fields that contain primitive values `final`.

LCM.2.g Create defensive copies of mutable fields in `readObject` of `Serializable` classes during deserialization.

1.4.5 Secure Transfer (ST)

ST.1 Protect objects that are transferred over communication channels or to persistent storage.

ST.1.a Consider sealing sensitive objects that require mobility or persistence using `SealedObject`.

ST.1.b Consider signing sensitive objects that require mobility or persistence using `SignedObject`.

ST.1.c Consider using `SSLSockets` instead of normal `Sockets`.

ST.1.d Consider using `CipherInputStream` and `CipherOutputStream` when persisting objects.

1.4.6 Secure Failure (SF)

SF.1 Handle all errors securely.

SF.1.a Avoid catching overly-broad exceptions, such as `java.lang.Exception` [26:396].

SF.1.b Avoid throwing overly-broad exceptions, such as `java.lang.Exception` [26:397].

SF.1.c Avoid catching unchecked exceptions, such as `java.lang.NullPointerException` -- unless they are caught at the top-most level [43].

SF.1.d Avoid returning (i.e., using the `return` keyword) inside `finally` blocks.

SF.1.e In J2EE, catch `java.lang.Throwable` at the top-most level [43].

SF.1.f In J2EE, ensure default error pages are enabled [26:7].

SF.2 Always release allocated resources.

SF.2.a Always invoke `close` on objects that open resource streams (e.g., `FileReader`, `FileInputStream`, `PreparedStatement`, `Statement`, etc.) in `finally` blocks.

SF.3 Prevent leaking sensitive information during failures.

SF.3.a Consider throwing new exceptions of the same type but with a sanitized message.

SF.3.b Consider throwing different types of exceptions (i.e., user-defined exceptions) entirely.

1.4.7 Secure Disposal (SDi)

SDi.1 Minimize the lifetime of secrets that are stored in memory.

SDi.1.a Store sensitive data in mutable objects, such as character arrays rather than immutable objects, such as `Strings`, so that they can be explicitly cleared (e.g., using `Arrays.fill`) when no longer needed.

SDi.1.a.a Use `JTextField.getPassword` when processing passwords in Swing code.

SDi.1.a.b Use `Console.readPassword` when requesting passwords from console input streams.

SDi.2 Employ secure object disposal mechanisms.

SDi.2.a Avoid explicitly calling `finalize` [26:486].

SDi.2.b Consider making sensitive non-final classes prevent finalizer attacks by providing an empty finalizer method and declaring it `final`.

SDi.2.c Avoid using finalizers to perform time or security-critical operations -- if required for classes that are `public` and non-final, utilize the finalizer idiom.

1.4.8 Secure Shutdown (SS)

SS.1 Employ secure application shutdown mechanisms.

SS.1.a Invoke, `System.exit()` minimally.

SS.1.b In J2EE, avoid calling `System.exit()` [26:382].

1.5 Accountability (Ac)

Ac.1 Employ a logging scheme.

Ac.1.a Consider logging every caught exception -- especially security-relevant exceptions, such as `SecurityException`, `AccessControlException`, and `SSLException`.

Ac.1.b Avoid utilizing output streams for logging purposes -- use a well-known logging API instead, such as `java.util.logging` or Apache log4j [55].

Ac.1.b.a Avoid writing log events using `System.out`.

Ac.1.b.b Avoid writing log events using `Console.printf`.

Ac.1.b.c Avoid writing log events using `System.err`.

4.6 Discussion

Relationships between Design Principles

Even though each design principle is distinctly classified in the taxonomy, relationships exist between them. For instance, having low coupling between modules is an overarching goal. When coupling occurs on shared data, such as a `public static final` array in Java, modules may be able to simultaneously access the array (i.e., assuming atomicity is not guaranteed). Because simultaneous access to shared data is possible, this type of coupling is a controlled sharing design issue as well. The principle of controlled sharing says that accessing shared data should be atomic. However, since a `public static final` array is a mutable object that can be modified by untrusted code, this example is also a trust boundary

design problem, which says that data should be considered malicious and validated accordingly before it is used.

Relationships also exist between design principles that exist on the same level within the taxonomy. For example, the principle of defense in depth calls for different types of restriction to be enforced, such as least privilege and complete mediation, when modules interact. When failures occur because entities are restricted from performing some action, the system should fail in a secure manner. Thus, a secure failure occurs after defense in depth mechanisms have done their job. Because failing securely is a secure state transition that does not enforce restriction, secure failure is not subordinate to defense in depth in the taxonomy; restriction-based mechanisms do not cause all system failures.

Taxonomy Advantages

When compared to previous work, the taxonomy discussed here is advantageous for the following three reasons:

1. It has both theoretical and practical importance.
2. Its methodology makes understanding, applying, and remembering secure coding heuristics easier.
3. It lays the groundwork for producing a secure coding standard in Java.

The taxonomy maintains a solid theoretical basis that is applicable to the design of all software systems. Many of the design principles within the taxonomy are derived from lessons learned through years of experience. Concepts, such as Saltzer and Schroeder's "complete mediation" [1], or the concept of "modularization" [10] conceived by Parnas that led to information hiding, are not likely to change.

Although rooted in design theory, the taxonomy also remains practical by offering guidance that is specific to the Java language. As an example, consider the coding heuristic that says *invoke* `AccessController.doPrivileged` *minimally (MSE.1.a)*, which falls under the principle of minimized security elements. According to the taxonomy, following this coding

heuristic helps to minimize security elements through economy of mechanism and thereby reduce module complexity. In this way, coding heuristics are practical, language-specific statements about how theoretical design principles can be achieved. This specific coding heuristic also shows that the taxonomy acknowledges that reducing software complexity is just as important as not introducing vulnerabilities in code.

Whereas the focus of previous taxonomies has been to classify different types of security flaws [30, 31, 32, 75] or the kinds of coding errors that lead to them [48, 46, 26], the taxonomy discussed here focuses on classifying coding heuristics according to the design principles they help to achieve. The difference between the taxonomy's design-driven approach and previous ones can be shown by considering the SQL injection vulnerability. A SQL injection vulnerability allows untrusted user input to be improperly mixed with commands in a SQL statement. The result is that a SQL statement can be maliciously modified before it is executed. Both the CLASP and SPK taxonomies provide details for how to prevent SQL injection[14]; however, they differ in how the SQL injection problem is fundamentally classified. At the highest level of abstraction, the differences between the taxonomy discussed here and the CLASP and SPK schemes are shown in Table 1.

Taxonomy	Classification
CLASP	Range and Type Errors
SPK	Input Validation and Representation
New Design-driven Taxonomy	Continuous Protection of Information -> Defense in Depth -> Trust Boundaries -> Reluctance to Trust

Table 1: Classifying the SQL injection vulnerability

CLASP classifies the SQL injection problem according to the type of problem it is. The SPK taxonomy classifies the SQL injection problem according to the coding error that

[14] While not shown here, proper use of parameterized statements prevents SQL injection; see coding heuristic RtT.1.3.a.

allows the attack to occur. The design-driven approach, however, classifies SQL injection according to how software should be designed to prevent the attack from occurring. Developers should recognize that the SQL query will execute based on untrusted input that crosses a trust boundary. Thus, defensive measures should be implemented so that information is continuously protected. By mapping coding heuristics to design principles, the design-driven scheme allows developers to gain a deeper understanding of the fundamental problem being addressed by writing code in specific ways. For this same reason, the design-driven approach makes understanding, applying, and remembering coding heuristics easier.

Finally, to the author's knowledge, the taxonomy contains the most concise and comprehensive collection of secure coding heuristics for Java to date. As such, it represents an effort towards contributing to a secure coding standard in Java.

Taxonomy Limitations

Mapping language-specific rules for writing secure code to design principles makes coding rules easier to remember and design principles easier to understand. However, not all security concerns can be solved by only considering the internal characteristics of software. For instance, if the user interface of a product is designed poorly, users may utilize features in insecure ways or find ways to bypass them because they are annoying. Saltzer and Schroeder refer to this design issue as the principle of "psychological acceptability" [1]. Design issues that relate to external characteristics of software are valid and should be considered, but it is not clear how to provide specific guidance at the code-level to help solve such problems.

Many architectural flaws that impact security also cannot be addressed by providing guidance at the code-level. In a secure instant messaging application, if messages that are sent from a client to a server can be intercepted and replayed at a later time by a third party,

an architectural flaw exists. Two other types of architectural-level issues are flawed password aging and single-factor authentication mechanisms. It is difficult to imagine a coding standard that can specifically address all of the possible ways these mechanisms can be implemented in code.

Lastly, it is important to realize that both design heuristics and coding heuristics provide guidance, but make no guarantee, for achieving some end [10]. By providing guidance, the assumption is that adhering to design and coding heuristics will help increase the quality and security of code. Thus, even if every design and coding heuristic is strictly adhered to, the resultant design and code may still have vulnerabilities.

5 A Static Analysis Study

5.1 Background

Static analysis is a process for analyzing code[15] without executing it. Static analysis is appealing because problems can be automatically identified as soon as they are introduced into code [59]. The advantages of utilizing static analysis tools to aid a code review process are becoming well-known. Scanning source code with a tool is included as an activity in three mainstream, lightweight secure software development methodologies: McGraw's seven "touchpoints" [25], CLASP [46], and Microsoft's Security Development Lifecycle (SDL) [58]. In their recent book on the subject, Chess and West [43] explain that existing applications of static analysis have the following capabilities:

- Enforce type checking
- Ensure that source code adheres to style guidelines
- Identify specific software components to aid program understanding
- Verify program correctness based on formal specifications
- Identify defects, or bugs, in programs
- Identify security vulnerabilities in programs

Despite the wide range of capabilities listed above, a small number of studies have been carried out regarding the use of static analysis tools for Java. Finding studies that focus on the use of static analysis tools to enforce secure coding standards or secure coding practices is difficult; the study described in this chapter may be the first. The NIST Software Assurance Metrics and Tool Evaluation (SAMATE) Project [69] is one effort aiming to bridge current gaps in static analysis research.

Because the coding heuristics included in the new taxonomy cross many dimensions of module design and tools likely have a single capability from the seven that are listed

[15] Static analysis techniques exist to analyze source code, bytecode, and object code in various languages.

above, it is not clear what kinds of coding heuristics or how many heuristics can be enforced by tools. For instance, imposing a limit on the number of the lines of code that is within the scope of privileged code (SM.1.a) is an entirely different problem than ensuring input that is received from untrusted sources is validated prior to being passed as a parameter to privileged code (RtT.1.1.a). While both guidelines are important to writing secure code, an algorithm that counts the number of lines code within a block of code tackles a less complex problem than algorithms that identify, trace, and evaluate the flow of information across modules.

How effectively can static analysis tools enforce a wide variety of secure coding heuristics in Java? The study that follows aims to answer this question by evaluating eight popular static analysis tools for Java.

5.2 Materials

The names and versions of the eight tools included in the study are provided in Table 1. The "Run Method" column in Table 1 notes how each tool was executed during the study.

Tool	Version	Run Method	Availability
Checkstyle	4.4.0	Eclipse SDK 3.2.2 plug-in	free
Eclipse TPTP Analysis	4.4.0.3	Eclipse SDK 3.3.1 Test and Performance Tools Platform (TPTP) all-in-one package	free
FindBugs	1.3.2	GUI	free
Fortify SCA	5.0	GUI	commercial
Jlint	3.1	Command line	free
Lint4j	0.9.1	Eclipse SDK 3.2.2 plug-in	free
PMD	4.1	Command line	free
QJ-Pro	2.2.0	GUI	free

Table 2: Tools included in the static analysis study

To better understand their goals and objectives, the following provides a brief description of each tool:

- **Checkstyle**: a tool that analyzes source code to find layout issues, class design problems, duplicate code, and bugs [60].
- **Eclipse TPTP Analysis**: a static analysis framework that allows third parties to integrate different types of static analysis using a consistent interface [61]; a set of rules for common J2SE issues are included by default and were used in the study.
- **FindBugs**: a tool that analyzes Java bytecode to find occurrences of "bug patterns", which are code idioms that are likely to be errors [62].
- **Fortify SCA**: a commercial tool that uses a patented dataflow analysis to identify software vulnerabilities in twelve different languages [63].
- **Jlint:** a tool that analyzes Java source code and bytecode to detect bugs, inconsistencies, and problems with synchronization by performing "data flow analysis and building the lock graph" [70].
- **Lint4j**: a tool that analyzes Java source and bytecode to detect defects related to threading, performance, complex contracts, and security vulnerabilities by performing "type, data flow, and lock graph analysis" [64].
- **PMD**: a tool that scans source code to detect potential bugs, dead code, suboptimal code, overcomplicated expressions, and duplicate code [65].
- **QJ-Pro**[16]: an inspection tool that checks for conformance to coding standards, language misuse, conformance to best practices, code structure issues, and potential bugs [66].

5.3 Methods

Test cases[17] were developed in the current version of Java, which is Java SE 6. A total of 72 test cases were created. Each test case consists of one or more simple violations of coding heuristics from the taxonomy laid out in Chapter 4. In total, there are 133 violations. Multiple instances of some violations appear in a few different test cases; of the 133 violations, 115 violations are distinct.

Secure coding heuristics that are related to configuration, J2EE, or Enterprise Java Beans (EJB) were not included in the study. An example of a configuration coding heuristic is LP.1.b: *Never grant* `java.security.AllPermission` *to external code bases.* An example of a J2EE

[16] The version of QJ-Pro included in the study was only able to scan Java 1.4 compatible code.
[17] The test cases included in the study can be obtained at the following URL: http://peregrin.jmu.edu/~warems

coding heuristic is CMo.1.h: *In J2EE, avoid using Socket connections [26:246]*. The focus of the study is on violations of Java SE 6 secure coding heuristics.

Each tool was configured to perform all possible checks[18] using their default settings against all 72 test cases. Source code files for all test cases were placed in Java packages and given a unique test case identifier (e.g., `test1`, `test2`, `test3`, etc.). A package named `util.common` is also included that contains classes to help carry out certain violations. For instance, a class having a fully qualified name of `util.common.UntrustedClient` is provided with a method named `getInput`, which returns a `String` that is read from an arbitrary `Socket` connection. All warnings that were generated by tools based on classes in the `util.common` package and its subpackages were not counted or evaluated.

5.4 Results

Each tool was subjected to all 72 test cases and the output produced from each tool was manually reviewed. Table 3 summarizes the results by showing the total number of warnings generated by each tool, the total number of violations correctly identified, the total number of violations identified that were not also identified by other tools, and the number of false positives as related to the coding heuristics. The number in parentheses represents the number of distinct violations identified out of the corresponding count.

[18] Some tools had specific rules for J2EE; these rules were not included.

Tool	# of warnings	# of violations identified	# of violations identified (and not by other tools)	# of false positives
Fortify SCA	56	30 (27)	9 (8)	2
PMD	339	23 (20)	3 (3)	3
QJ-Pro	2761	20 (17)	5 (4)	5
Checkstyle	4111	17 (17)	2 (2)	2
FindBugs	25	14 (12)	3 (3)	0
Eclipse TPTP	109	9 (8)	0	3
Lint4j	50	8 (8)	3 (3)	0
Jlint	13	3 (3)	0	2
Combined		60 (50)	n/a	n/a

Table 3: Summary of static analysis study results

Some warnings that were produced by the tools were not relevant to any coding heuristic being tested; no attempt was made to determine their validity. Table 4 shows individual results for each tool against each coding heuristic violation that is included in every test case. An "x" indicates that a tool correctly identified a violation of a specific coding heuristic in a certain test case. Results are sorted in alphabetical order by the identifiers given to coding heuristics.

Test Case ID	Heuristic ID	Eclipse TPTP	Lint4j	Jlint	Find-Bugs	Check-style	PMD	QJ-Pro	Fortify SCA
16	Ac.1.a		x			x	x	x	x
3	Ac.1.b.a						x	x	x
1	Ac.1.b.b								
2	Ac.1.b.c						x	x	x
40	CMe.1.a								
37	CMe.1.b								
34	CMe.1.c								
22	CMe.2.a								
34	CMe.2.b								
22	CMe.2.c								
34	CMe.2.d								
71	CMe.3.a								
7	CMo.1.a				x				x
5	CMo.1.b	x				x	x		
17	CMo.1.c	x	x		x				
15	CMo.1.k			x		x	x		x
19	CMo.1.l		x						

Test Case ID	Heuristic ID	Eclipse TPTP	Lint4j	Jlint	Find-Bugs	Check-style	PMD	QJ-Pro	Fortify SCA
72	CMo.2.a								x
6	CMo.2.b	x		x	x	x	x	x	x
6	CMo.2.c	x	x			x	x	x	
4	CMo.2.d	x							x
70	CS.1.a			x				x	
68	CS.1.b								
46	IH.1.a							x	x
40						x		x	
12	IH.1.a.a								
11	IH.1.a.b								
25	IH.1.a.c								
43	IH.1.a.d							x	
34	IH.1.a.e					x		x	
20	IH.1.a.f		x						
19	IH.1.a.g					x		x	
23	IH.1.b								
21	IH.1.c								
21	IH.1.d								
10	IH.1.e								
46								x	x
23	IH.2.a								
14	IH.2.a.a								
26	IH.2.a.b								
23	IH.2.b								
9									
25	IH.2.b.a					x[19]			
22	IH.2.b.b					x[20]			
34	IH.3.2.b								
19	IH.3.2.c								
22	IH.3.2.d								
22	IH.3.2.e								
22	IH.3.3.b								
24	ISE.1.a								
	ISE.1.b								
	ISE.1.c								
25	ISE.1.d								
58	LCM.1.a[21]								
63	LCM.2.a								
46	LCM.2.b				x	x	x		x
63					x		x		

[19] Checkstyle warned that the method should be final, but the warning was not influenced by the presence of privileged code in the method.
[20] Checkstyle warned that the method should be final, but the warning was not influenced by the presence of a SecurityManager check in the method.
[21] FindBugs warned that the ClassLoader should be used inside a privileged block of code, which is not relevant to the coding heuristic.

Test Case ID	Heuristic ID	Eclipse TPTP	Lint4j	Jlint	Find-Bugs	Check-style	PMD	QJ-Pro	Fortify SCA
59	LCM.2.b.a								
61	LCM.2.b.b								
63	LCM.2.c				x		x		
48									
25	LCM.2.d								
46	LCM.2.e[22]					x			x
65					x				
66	LCM.2.f							x	
60	LCM.2.g								
27	LP.2.a				x	x	x	x	x
60							x	x	
25	LP.2.b								
29	LP.2.c					x	x		
36	LP.2.d						x		
49	RtT.1.1.a								
47	RtT.1.1.b								
34	RtT.1.1.c								
50	RtT.1.2.a[23]								
52									x
68	RtT.1.2.b								x
52	RtT.1.2.c								
54	RtT.1.3.a				x				x
55					x				x
57	RtT.1.3.b								
25	RtT.2.a								
56	RtT.2.b								
34	RtT.2.c								
41	RtT.2.d	x					x		
23							x		
40	RtT.2.e								
41	RtT.2.f								
45	RtT.3.a.a								x
43	RtT.3.b.f[24]								x
42	RtT.3.c.c								
25	RtT.4.a								
44	RtT.4.b								x
34	SA.1.a								
33	SA.1.b								
64	SA.1.c							x	
44								x	
70					x				

[22] QJ-Pro warned that non-final, static fields should not be declared; however, it did not encourage immutability.
[23] QJ-Pro warned that Runtime.exec should be avoided; however, it did not recognize its use with untrusted input.
[24] PMD warned that System.loadLibrary is dangerous but did not warn that it is invoked with untrusted input.

Test Case ID	Heuristic ID	Eclipse TPTP	Lint4j	Jlint	Find-Bugs	Check-style	PMD	QJ-Pro	Fortify SCA
46	SDi.1.a[25]								
18									
36									
54									
55									
38	SDi.2.a		x		x		x		x
34	SDi.2.b								
15	SDi.2.c							x	
16	SF.1.a	x				x			x
46		x							x
8	SF.1.b	x					x		x
53	SF.1.c						x		x
67	SF.1.d		x				x	x	x
72	SF.2.a				x				x
13	SF.3.a								
13	SF.3.b[26]								
34	SI.1.a								
35	SI.1.b								
25	SM.1.a[27]					x	x	x	
25	SM.1.b[28]					x	x		
39	SoP.1.c								
28	SPM.1.1.a[29]								x
32									x
31	SPM.2.a								
29	SPM.2.b								
29	SPM.2.c								
30	SPM.3.a								x
62	SS.1.a		x						
71	ST.1.a								
	ST.1.b								
	ST.1.c								
72	ST.1.d[30]								
69	U.2.a								
51	U.2.b						x		
35	U.3.a				x				
TOTAL		**9**	**8**	**3**	**14**	**17**	**23**	**20**	**30**

Table 4: Individual coding heuristic violations identified by each tool

[25] Fortify SCA correctly warned that the password used was not encrypted; however, it did not warn that the password was stored in a String object.
[26] PMD warned that exceptions should not be caught only to be re-thrown; however, it did not warn that the exception message was not sanitized.
[27] Checkstyle warned that the anonymous inner class is longer than 20 lines; PMD and QJ-Pro warned that the method is excessively long.
[28] Checkstyle warned that the method is longer than 150 lines; PMD warned the method is excessively long.
[29] Eclipse TPTP warned that a non-constant, String literal was used; however, it did not warn that the String literal was a password.
[30] Fortify SCA warned that reading from the FileInputStream could result in a denial of service attack.

5.5 Discussion

After analyzing the totals presented in Table 3 and the details of each test case in Table 4, it is apparent that most of the tools were able to identify violations that other tools did not. Fortify SCA, which identified 27 distinct violations, found nine times as many distinct violations as Jlint, which found the least number of violations. Thus, for the test cases included in this study, there is a considerable difference in the apparent effectiveness of the tools.

The combined effort from all of the tools is superior to any single tool performance. Yet, if all of the tools were combined into a single tool, only 50 out of a possible 115 distinct violations would have been identified. Clearly, a significant number of coding heuristic violations were not identified by any tool. The most serious kinds of violations that were not identified by any tool are briefly discussed next, with example code from the test cases.

```
package test37;

import util.common.permissions.ConstructPermission;

/**
 * This class violates the following coding heuristics:
 *
 * <ul>
 * <li>CMe.1.b Ensure all SecurityManager checks that occur in
 * constructors of Cloneable classes also occur within
 * clone (or methods that provide clone-like functionality).
 * </ul>
 *
 * @author M. S. Ware
 */
public final class BadCloneSecurityCheck implements Cloneable {
   public BadCloneSecurityCheck() {
      /*
       * Authorize the caller
       */
      final SecurityManager sm = System.getSecurityManager();
      if (sm != null) {
         sm.checkPermission(new ConstructPermission("BadCloneSecurityCheck"));
      }
   }

   public Object clone() throws CloneNotSupportedException {
      final BadCloneSecurityCheck theClone = (BadCloneSecurityCheck)super.clone();

      return theClone;
   }
}
```

Figure 12: SecurityManager check in a constructor but not in clone (CMe.1.b)

In Figure 12, a `SecurityManager` check is enforced in the constructor of a `Cloneable` class, but the same `SecurityManager` check is not provided in the `clone` method. This violates the principle of complete mediation, since a caller can bypass the `SecurityManager` check when cloning an existing object. A similar subtlety involves serialization (CMe.1.c): when `readObject` and `readObjectNoData` are called during deserialization, a class constructor is not invoked. Tools were also not able to identify violations of this coding heuristic.

```
package test41;

import java.util.Date;

/**
 * This class violates the following coding heuristics:
 *
 * <ul>
 * <li>RtT.2.d Only call final methods within constructors of non-final classes.
 * <li>RtT.2.f Only invoke clone on instances of final classes.
 * </ul>
 *
 * @author M. S. Ware
 */
public class BadConstructor {
   private final Date startDate;

   public BadConstructor(final Date startDate) {
      /*
       * Date is a non-final class (RtT.2.f)
       */
      this.startDate = (Date)startDate.clone();

       /*
        * This method is overridable (RtT.2.d)
        */
       init();
   }

   public final Date getDate() {
      return new Date(this.startDate.getTime());
   }

   public void init() {

   }
}
```

Figure 13: Invoking clone on an instance of a non-final class

In Figure 13, the `clone` method is invoked on a `Date` object that is a parameter of the constructor. Because the `Date` class is not `final`, the risk here is that an untrusted caller may provide a `Date` instance that has been overridden in a malicious manner. This is an issue of trust, and the most conservative approach is to not invoke `clone` on an instance of a non-

final class. A better approach is to create a new `Date` instance based on the state of the parameter, as in:

```
this.startDate = new Date(startDate.getTime());
```

```
package test49;

// imports omitted for brevity

/**
 * This class violates the following coding heuristics:
 *
 * <ul>
 * <li>RtT.1.1.a Validate untrusted input before passing it to privileged
 * blocks (i.e., invocations of AccessController.doPrivileged).
 * </ul>
 *
 * @author M. S. Ware
 */
final class BadPrivBlockValidation {
   public static final void openFile() throws FileNotFoundException {
      final String filePath = UntrustedClient.getInstance().getInput();

      doPrivilegedOpenFile(filePath);
   }

   private static final void doPrivilegedOpenFile(final String filePath) {
      final BadFileNamePrivilegedAction pa =
         new BadFileNamePrivilegedAction(filePath);

         FileInputStream fis = null;
         try {
            try {
               fis = (FileInputStream)AccessController.doPrivileged(pa);

               /*
                * Continue processing the FileInputStream
                */

            } finally {
               /*
                * Close the stream (if it exists)
                */
               if (fis != null) {
                  fis.close();
               }
            }
         } catch(PrivilegedActionException caught) {
            LogHandler.log(Level.SEVERE, "An error occurred.", caught);
         } catch(IOException caught) {
            LogHandler.log(Level.SEVERE, "An error occurred.", caught);
         }
   }
}
```

Figure 14: Lack of input validation on data passed to privileged code

In Figure 14, a file path is obtained by invoking `UntrustedClient.getInstance().getInput()`, which reads a line of data from a `Socket`. The untrusted file path is then stored in a `BadFileNamePrivilegedAction`, which is subsequently passed to `AccessController.doPrivileged`. Although not shown, the

`BadFileNamePrivilegedAction` class does not validate the file path. This is dangerous because the file path is never validated before being sent as a parameter to privileged code, which executes with escalated permissions. The result is that a caller may successfully open a sensitive file, such as "`/etc/passwd`", which it may not have permission to access otherwise.

```
package test18;

// imports omitted for brevity

/**
 * This class violates the following coding heuristics:
 *
 * <ul>
 * <li>SDi.1.a Store sensitive data in mutable objects, such as character
 * arrays rather than immutable objects, such as Strings, so
 * that they can be explicitly cleared (e.g., using Arrays.fill)
 * when no longer needed.
 * </ul>
 *
 * @author M. S. Ware
 */
final class BadComplexGetConnection {
   public final static Connection getConnection() throws SQLException {
      /*
       * Get the username
       */
      final String user = UntrustedClient.getInstance().getUser();

      /*
       * Get the password
       */
      final String password = new
         String(UntrustedClient.getInstance().getPassword());

       /*
        * Create and return the Connection
        */
      return DriverManager.getConnection("jdbc:mysql:mydb", user, password);
   }
}
```

Figure 15: Storing a password in a String

In Figure 15, a username and password are retrieved from an untrusted client. Here, `UntrustedClient.getInstance().getPassword` returns a character array representation of the password (by invoking `JPasswordField.getPassword`). The problem is that a `String` object is immediately constructed from the character array. Now, this class cannot remove the password from memory in a secure manner after it is no longer needed; disposal of the contents of the `String` is dependent upon the garbage collector.

While being able to identify passwords that are stored in `String` objects seems complex, being able to identify `public` lock variables seems trivial. Yet, no tool identified the violation that is shown in Figure 16.

```
package test11;

/**
 * This class violates the following coding heuristics:
 *
 * <ul>
 * <li>IH.1.a.b Declare lock variables <code>private</code>.
 * </ul>
 *
 * @author M. S. Ware
 */
final class BadLock {
   public static final Object lock = new Object();

   public static final void merge() {
      synchronized(lock) {
         doSomething();
      }
   }

   private static final void doSomething() {
      // doSomething
   }
}
```

Figure 16: Use of a public lock variable

Finally, as shown in Figure 17, no tool identified the direct use of untrusted input with the `ProcessBuilder` class, which allows Java code to invoke operating system processes[31]. This vulnerability is called "command injection" [43].

[31] Fortify SCA did, however, identify the same violation using Runtime.exec (see test52).

```
package test50;

/**
 * This class violates the following coding heuristics:
 *
 * <ul>
 * <li>RtT.1.2.a Validate and encode untrusted input before using it as
 * command parameters when invoking Runtime.exec and ProcessBuilder.start.
 * </ul>
 *
 * @author M. S. Ware
 */
final class BadProcessBuilder {

   /**
    * Execute a command and return the Process
    *
    * @return  The Process
    */
   public static final Process execCommand() {
   Process cmd = null;

      try {
         final String parameters = UntrustedClient.getInstance().getInput();

         final ProcessBuilder procBuilder =
             new ProcessBuilder(new String[] { "cmd", "/c dir " + parameters });

         ls = procBuilder.start();

         /*
          * Finish working with the process
          */

      } catch(IOException caught) {
         LogHandler.log(Level.SEVERE, "An error occurred.", caught);
      }

      return cmd;
   }
}
```

Figure 17: Use of untrusted input with ProcessBuilder

Other interesting and serious violations of coding heuristics that were not identified by any tool are as follows:

- IH.3.2.b: a `private transient` field that is guarded with `SecurityManager` checks is serialized in the `writeObject` method of a `Serializable` class (test34).
- SPM.2.a: the weak DES algorithm is used instead of the AES algorithm when using the `Cipher` class (test31).
- SPM.2.b: the weak MD5 algorithm is used instead of one of the SHA-256, SHA-384, or SHA-512 algorithms when using the `MessageDigest.getInstance` (test29).
- SPM.2.c: a key size of 512 bits is specified instead of at least 1024 bits when using the `RSA` algorithm with `KeyPairGenerator.getInstance` (test29).
- RtT.1.3.b: untrusted input from a `Socket` is injected into an `XPath` query allowing an injection attack to occur (test57).
- LCM.1.a: a single `URLClassLoader` instance is constructed with three different URLs (test58).

- LCM.2.a: a reference to a `private` mutable array is passed as a parameter to outside code instead of a defensive copy (test63).
- LCM.2.b.a: a `clone` method in a `Cloneable` class returns an object with a reference to a mutable field thereby exposing the object's state (test59).
- RtT.1.1.b: a validation check is performed on a mutable input parameter instead of a defensive copy that was made (test47).
- IH.3.2.c: a sensitive field that is `transient` is improperly stored in the `serialPersistentFields` array class member in a `Serializable` class (test19).
- ST.1.c: a normal `Socket` is used rather than an `SSLSocket` (test71).

Of the 60 total violations that were identified, no single violation was identified by all eight tools, 8 violations were identified by four or more tools, 15 violations were identified by three or more tools, and 35 violations were identified by two or more tools. Table 5 lists the 25 specific violations that only one tool was able to identify in the study. These observations suggest that tools identify different types of coding heuristic violations, and that some tools cannot identify a violation in all circumstances. For instance, QJ-Pro identified a violation of SA.1.c in `test44` and `test64` but did not in `test70`, which was identified by FindBugs.

Tool	Secure Coding Heuristics
Fortify SCA	CMo.2.a RtT.1.2.a RtT.1.2.b RtT.3.a.a RtT.3.b.f RtT.4.b SPM.1.1.a (test28 and test32) SPM.3.a
PMD	LP.2.d RtT.2.d (test23) U.2.b
QJ-Pro	IH.1.a.d LCM.2.f SA.1.c (test44 and test64) SDi.2.c
Checkstyle	IH.2.b.a IH.2.b.b
FindBugs	LCM.2.e (test65) SA.1.c (test70) U.3.a
Lint4j	CMo.1.l IH.1.a.f SS.1.a

Table 5: Violations of coding heuristics found only by one tool

From a design perspective, it is interesting to note that no tool was able to identify any violation that falls under the principle of complete mediation, principle of isolated security elements, principle of secure initialization, and principle of secure transfer. On the other hand, every violation that falls under the principle of correct modules, principle of small modules, and principle of secure shutdown was identified by at least one tool.

We can conclude that when default checks and settings are used by the eight tools in this study, there is still a need for manual code review. Of course, this conclusion is based on the assumption that the individuals performing a manual code review would be able to find violations of coding heuristics that the tools were not able to identify.

5.6 Related Studies

Few studies have been performed that compare the effectiveness of static analysis tools for Java, especially with respect to finding security-related flaws and defects. I believe that the study described above is one of the first to report on the ability of tools to identify specific violations of secure coding heuristics. Our field would benefit greatly from additional studies that compare tools with a focus on security.

Rutar et al. [71] compared the results of applying PMD (ver. 1.9), FindBugs (ver. 0.8.2), Jlint (ver. 3.0), ESC/Java (ver. 2.0a7), and Bandera (ver. 0.3b2) to five large open source Java 1.4 applications. They concluded that no tool performs better than all others, and that there is little correlation of warnings between pairs of tools. Although the study reported above subjected tools to a much smaller code base, it supports these findings. Since their analysis suggested that tools find different kinds of bugs, they proposed the creation and use of a meta-tool that could combine the output produced by tools.

Hovemeyer and Pugh [62] described the results from executing FindBugs (ver. 0.8.4) on a test suite of seven Java applications and libraries. All of the bug detectors that were included in the study found at least one instance of a bug. Their results show that FindBugs was able to identify a surprisingly large number of real bugs while maintaining a false positive rate near fifty percent. They compared the number of warnings produced by FindBugs to PMD (ver. 1.9) and showed that FindBugs produced a much lower warning count. However, they did not compare the details of the warnings produced by PMD with FindBugs.

Wagner et al. [72] applied FindBugs (ver. 0.8.1), PMD (ver. 1.8), and QJ-Pro (ver. 2.1) to five J2EE web applications. They aimed to compare the defects found by static analysis tools with defects found by performing human code reviews and traditional black box and white box testing. They observed that human code reviews found more defects

than static analysis tools, but that static analysis tools identified defects that were not found during reviews. They also observed that dynamic testing techniques found completely different defects than static analysis techniques. They concluded that dynamic testing was better at finding logical errors that resulted in failures during actual use while tools were better at finding defects related to the maintainability of code.

In later work, Wagner et al. [73] aimed to determine whether static analysis tools can be used to improve the efficiency of software defect-detection. They applied FindBugs and PMD to two business-oriented J2EE web applications[32]. They found that when considering successive versions of an application, few of the warnings produced by tools correspond to actual failures that are found post-release. They suspected that most failures post-release are caused by logic errors, which tools have a difficult time finding. They also note that by combining the output produced by both FindBugs and PMD, fault-prone classes in the selected tools could be identified.

Ayewah et al. [74] report on results that were obtained from applying FindBugs on production software, including Sun's JDK 1.6.0 codebase, Sun's Glassfish J2EE server codebase, and portions of Google's codebase. On Sun's JDK 1.6.0 codebase, they manually evaluated all of the medium and high priority warnings that were related to correctness issues. Of 379 warnings, they decided 5 were due to poor analysis by FindBugs, 160 were not possible, 176 seemed to have functional impact, and 38 seemed to have substantial functional impact to the application. In general, they concluded that many FindBugs warnings relate to obvious defects that would not result in serious problems. They show that sometimes these defects seem to be intentionally introduced by programmers and claim that static analysis tools warn about such issues because they simply "don't know what the code

[32] It is not clear which versions of FindBugs and PMD were used in this work.

is supposed to do." By looking at build history, they also show that FindBugs produces warnings that developers seem to be willing to address and in most cases fix prior to new builds.

6 Conclusion

Future Work

While I believe that the new taxonomy of design principles proposed in this work has a solid theoretical basis, the taxonomy will have to be updated to reflect new findings, as the Java language evolves and new secure coding rules, techniques, and practices are discovered. At the moment, a comprehensive study of the J2EE platform is needed so that a more complete set of secure coding heuristics for J2EE can be added to the taxonomy.

The static analysis study reported in this work evaluated tools using their default checks and settings. Some of the checks provided by the tools can be customized. When properly configured, tools may be able to identify violations that they would not find, or attempt to find, by default. Additionally, some tools can be extended by allowing custom checks to be written and incorporated into their rulesets. Indeed, if a tool did not identify a violation with its default checks and settings, it may be able to be extended or configured to do so. Investigating how many violations the tools may ultimately be able to identify is left for future research.

Finally, while this thesis focuses on the Java language, the methodology of the taxonomy discussed here is applicable to other object-oriented languages and platforms. For example, secure coding heuristics for the C# programming language (of the .NET framework) can be classified according to the scheme as well. Such work could also help contribute to a C# secure coding standard, which currently does not exist.

Towards More Secure Software

Software that is poorly constructed, operated, and maintained is likely to contain vulnerabilities. Developers who write code using insecure practices are central to the

problem. Organizations that are striving to combat the problem share a common goal: to increase awareness of secure development practices and provide guidance for preventing software vulnerabilities. The *Secure Programming Skills Assessment* effort by the SANS Software Security Institute [68] further exemplifies the industry's current emphasis on requiring developers to learn and apply secure coding skills.

This thesis hopes to contribute in three ways:

1. Catalog secure coding heuristics to help create a secure coding standard for Java.
2. Propose a new taxonomy of design principles and coding heuristics that makes understanding, applying, and remembering heuristics easier.
3. Report on the effectiveness of eight different static analysis tools at enforcing the wide variety of secure coding heuristics that are included in the new taxonomy.

As CERT/CC vulnerability statistics indicate [36], the importance of writing secure code has never been more urgent. As technology evolves and new languages are introduced, there is a need for developers to become more security-aware when constructing software [44]. By mapping coding heuristics to design principles, the taxonomy proposed in this work supports a coding process that is driven by secure design. It may help developers adopt and apply a security-conscious mindset when writing code. Such fundamental knowledge is crucial for helping individuals understand how to construct secure software, regardless of the language paradigm or system environment used.

7 Bibliography

[1] M. D. Schroeder, and J. H. Saltzer, "The Protection of Information in Computer Systems," in *Proceedings of the IEEE*, vol. 63, no. 9, 1975, pp. 1278-1308. Available: http://web.mit.edu/Saltzer/www/publications/protection.

[2] Sun Microsystems, Inc., "Secure Coding Guidelines for the Java Programming Language, version 2.0," *Sun Microsystems, Inc.* [Online]. Available: http://java.sun.com/security/seccodeguide.html. [Accessed: Aug. 30, 2007].

[3] S. Redwine, Jr., Ed., *Software Assurance: A Guide to the Common Body of Knowledge to Produce, Acquire, and Sustain Secure Software*, Workforce Education and Training Working Group, U.S. Department of Homeland Security, Draft Version 1.1, September 2006.

[4] M. Bishop, *Introduction to Computer Security*. Boston, MA: Addison-Wesley, 2005.

[5] M. Howard and D. Lipner, *Writing Secure Code*, 2nd ed. Redmond, Washington: Microsoft Press, 2003.

[6] G. McGraw, "Software Security," *IEEE Software and Privacy*, vol. 2, no. 2, pp. 80-83, March/April 2004.

[7] S. Redwine, Jr. and N. Davis, Eds., *Processes to Produce Secure Software: Towards more Secure Software*, National Cyber Security Summit, Software Process Subgroup of the Task Force on Security across the Software Development Lifecycle, vol. 1, March 2004.

[8] T. V. Benzel, C. E. Irvine, T. E. Levin, G. Bhaskara, and P. C. Nguyen, "Design Principles for Security," Naval Postgraduate School : Monterey, California, Tech Rep. NPS-CS-05-010, September 2005. Available: http://handle.dtic.mil/100.2/ADA437854. [Accessed: Sept. 5, 2007].

[9] J. Viega and G. McGraw, *Building Secure Software: How to Avoid Security Problems the Right Way*. Indianapolis, IN: Addison-Wesley, 2002.

[10] C. Fox, *Introduction to Software Engineering Design: Processes, Principles, and Patterns with UML2*. Boston, MA: Addison-Wesley, 2006.

[11] J. Bloch, *Effective Java Programming Language Guide*. Prentice Hall, 2001.

[12] D. L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," in *Communications of the ACM*, vol. 15, no. 12, 1972, pp. 1053-1058.

[13] M. Graff and K. van Wyk, *Secure Coding: Principles and Practices*. Sebastopol, CA: O'Reilly, 2003.

[14] Sun Microsystems, Inc., "White Paper: The Java Language Environment," *Sun Microsystems, Inc.,* 1997. [Online]. Available: http://java.sun.com/docs/white/langenv/Security.doc5.html. [Accessed: October 1, 2006].

[15] G. McGraw and E. Felton, *Java Security: Hostile Applets, Holes, and Antidotes*. Canada: Wiley Computer Publishing, 1997.

[16] Sun Microsystems, Inc. "News and Updates: Chronology of security-related bugs and issues," *Sun Microsystems, Inc.*, 2002. [Online]. Available: http://java.sun.com/sfaq/chronology.html. [Accessed: October 4, 2006].

[17] L. Gong, G. Ellison, and M. Dageforde, *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*, 2nd ed. Boston, MA: Addison-Wesley, 2003.

[18] J. Gosling, B. Joy, G. Steele, and G. Bracha, "The Java Language Specification," 3rd ed. *Sun Microsystems, Inc.* [Online]. Available: http://java.sun.com/docs/books/jls. [Accessed: December 11, 2007].

[19] Sun Microsystems, Inc. "Security Managers and the Java SE SDK," Rev. 1.7, *Sun Microsystems, Inc.*. [Online]. Available: http://java.sun.com/javase/6/docs/technotes/guides/security/smPortGuide.html. [Accessed: March 13, 2008].

[20] L. Gong, "Java 2 Platform Security Architecture," Ver. 1.2, *Sun Microsystems, Inc.* 1997-2002. [Online]. Available: http://java.sun.com/javase/6/docs/technotes/guides/security/spec/security-spec.doc.html. [Accessed: March 13, 2008].

[21] Sun Microsystems, Inc. "Default Policy Implementation and Policy File Syntax," Rev. 1.6, *Sun Microsystems, Inc.* [Online]. Available: http://java.sun.com/javase/6/docs/technotes/guides/security/PolicyFiles.html. [Accessed: March 11, 2008].

[22] D. Dean, E. Felton, D. Wallach, "Java Security: Web Browsers and Beyond," in Proceedings of the 1996 IEEE Symposium on Security and Privacy (SP '96), 1996.

[23] G. McGraw and E. Felton, *Securing Java: Getting Down to Business with Mobile Code*. John Wiley & Sons, Inc., 1998. [Online]. Available: http://www.securingjava.com. [Accessed: March 13, 2008].

[24] R. Alexander, J. Bieman, and J. Viega, "Coping with Java Programming Stress," *IEEE Computer*, vol. 33, no. 4, pp. 30-38, April 2000.

[25] G. McGraw, *Software Security: Building Security In*. Boston, MA; Addison-Wesley, 2006.

[26] The MITRE Corporation, "Common Weaknesses Enumeration: A Community-Developed Dictionary of Software Weakness Types," Draft 7, *The MITRE Corporation*, 2007. [Online]. Available: http://cwe.mitre.org. [Accessed: October 28, 2007].

[27] Sun Microsystems, Inc. "API for Privileged Blocks," Rev. 1.6, *Sun Microsystems, Inc.* [Online]. Available: http://java.sun.com/javase/6/docs/technotes/guides/security/doprivileged.html. [Accessed: October 5, 2007].

[28] Fortify Software Inc., "Fortify Taxonomy: Software Security Errors," *Fortify Software Inc.*, 2006. [Online]. Available: http://www.fortifysoftware.com/vulncat. [Accessed October 4, 2006].

[29] M. Howard, D. LeBlanc, and J. Viega, *19 Deadly Sins of Software Security: Programming Flaws and How to Fix Them*. Emeryville, CA: McGraw-Hill/Osborne, 2005.

[30] R. P. Abbott, J. S. Chin, J. E. Donnelley, W. L. Konigsford, S. Tokubo, and D. A. Webb, D. "Security Analysis and Enhancements of Computer Operating Systems," Institute for Computer Sciences and Technology, National Bureau of Standards, Tech. Rep. NBSIR 76-1041, April 1976. Available: http://cwe.mitre.org/about/sources.html.

[31] R. Bisbey II and D. Hollingsworth, "Protection Analysis: Final Report," CA: University of Southern California Information Sciences Institute, Tech. Rep. ISI/RR-78-13, 1978.

[32] C. Landwehr, A. Bull, J. McDermott, and W. Choi, "A Taxonomy of Computer Program Security Flaws, with Examples," in *ACM Computing Surveys (CSUR)*, vol. 26, no. 3, September 1994, pp. 211-254.

[33] M. Bishop and D. Bailey, "A Critical Analysis of Vulnerability Taxonomies," University of California at Davis, Tech. Rep. CSE-96-11, September 1996.

[34] S. Weber, P. Karger, and A. Paradkar, "A Software Flaw Taxonomy: Aiming Tools At Security," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, July 2005.

[35] Open Web Application Security Project, "The free and open application security community," [Online]. Available: http://www.owasp.org. [Accessed: March 1, 2008].

[36] CERT, "CERT Statistics," *Software Engineering Institute: Carnegie Mellon.* [Online]. Available: http://www.cert.org/stats. [Accessed: October 3, 2007].

[37] CERT, "Secure Coding," *Software Engineering Institute: Carnegie Mellon.* [Online]. Available: http://www.cert.org/secure-coding. [Accessed: December 12, 2006].

[38] ISO/IEC JTC 1/SC 22/OWG: Vulnerabilities, "Guidance for Avoiding Vulnerabilities through Language Selection and Use," [Online]. Available: http://www.aitcnet.org/isai. [Accessed: October 24, 2007].

[39] K. Arnold, J. Gosling, D. Holmes, *The Java Programming Language*, 4th ed., Upper Saddle River, NJ: Addison-Wesley, 2005. [Online]. Available: http://proquest.safaribooksonline.com/0321349806. [Accessed: September 25, 2007].

[40] D. Piliptchouk, "Java vs. .NET Security – Part 3," *O'Reilly ONJava.com*, [Online]. Available: http://www.onjava.com/pub/a/onjava/2004/01/28/javavsdotnet.html?page=2. [Accessed: October 15, 2007].

[41] R. Seacord, "Secure Coding Standards," in *Proceedings of the Static Analysis Summit*, NIST Special Publication 500-262, July 2006. Available: http://samate.nist.gov/docs/NIST_Special_Publication_500-262.pdf.

[42] S. McConnell, *Code Complete*, 2nd ed. Redmond, Washington: Microsoft Press, 2004.

[43] B. Chess and J. West, *Secure Programming with Static Analysis*. Boston, MA: Addison-Wesley, 2007.

[44] K. Goertzel Ed., T. Winograd, H. McKinley, P. Holley, *Security in the Software Lifecycle: Making Software Development Processes – and Software Produced by Them – More Secure*, US Department of Homeland Security, Draft Version 1.2, August 2006.

[45] S. Redwine Jr., "Towards an Organization for Software System Security Principles and Guidelines," Institute for Infrastructure and Information Assurance, James Madison University: Harrisonburg, VA, Tech. Rep. 08-01, Version 1.0, February 2008.

[46] K. Tsipenyuk, B. Chess, G. McGraw, "Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors," in *NIST Workshop on Software Security Assurance Tools, Techniques, and Metrics*, Long Beach, CA, November 2005.

[47] A. Carzaniga, G. Picco, G. Vigna, "Is Code Still Moving Around? Looking Back at a Decade of Code Mobility," in *Companion to the proceedings of the 29th International Conference on Software Engineering*, 2007, pp. 9-20.

[48] CLASP, "Comprehensive Lightweight Application Security Process," Secure Software, Inc., Version 2.0, 2006. [Online]. Available: http://searchsoftwarequality.techtarget.com/searchAppSecurity/downloads/clasp_v20.pdf. [Accessed December 5, 2007].

[49] A. J. Riel, *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.

[50] S. Barnum and M Gegick, "Reluctance to Trust," *Build Security In: Setting a Higher Standard for Software Assurance*, Cigital Inc., 2005. [Online]. Available: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/principles/355.html. [Accessed: January 17, 2008].

[51] C. Lai, "Java Insecurity: Accounting for Subtleties That Can Compromise Code," *IEEE Software*, vol. 25, no. 1, pp. 13-19, January/February 2008.

[52] S. Liang, *The Java Native Interface: Programmer's Guide and Specification*, Palo Alto, CA: Sun Microsystems Inc., 2002. [Online]. Available: http://java.sun.com/docs/books/jni/html/titlepage.html.

[53] S. Barnum and M Gegick, "Design Principles," *Build Security In: Setting a Higher Standard for Software Assurance*, Cigital Inc., 2005. [Online]. Available: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/principles/358.html?branch=1&language=1. [Accessed: January 17, 2008].

[54] CERT, "Top 10 Secure Coding Practices," *Software Engineering Institute: Carnegie Mellon.* [Online]. Available: https://www.securecoding.cert.org/confluence/display/seccode/Top+10+Secure+Coding+Practices. [Accessed: February, 16, 2008].

[55] Apache Software Foundation, "Logging Services: log4j," *Apache Software Foundation*, [Online]. Available: http://logging.apache.org/log4j/1.2/index.html. [Accessed: February 18, 2008].

[56] Sun Microsystems, Inc. "Java SE 6," *Sun Microsystems, Inc.* [Online]. Available: http://java.sun.com/javase/6.

[57] Sun Microsystems, Inc. "Java EE at a Glance," *Sun Microsystems, Inc.* [Online]. Available: http://java.sun.com/javaee.

[58] Howard and Lipner, *The Security Development Lifecycle: SDL: A Process for Developing Demonstrably More Secure Software*, Microsoft Press, June 2006.[Online]. Available: http://proquest.safaribooksonline.com/0735622140. [Accessed: January 28, 2008].

[59] B. Chess and G. McGraw, "Static Analysis for Security," *IEEE Security and Privacy*, pp. 32-35, vol. 2, no. 6, pp. 76-79, November/December 2004.

[60] O. Burn, "Checkstyle 4.4," [Online]. Available: http://checkstyle.sourceforge.net. [Accessed: March 1, 2008].

[61] S. Gutz and O. Marquez, "TPTP Static Analysis Tutorial Part 1: A Consistent Analysis Interface," [Online]. Available: http://www.eclipse.org/tptp/home/documents/process/development/static_analysis/TPTP_static_analysis_tutorial_part1.html. [Accessed: February 10, 2008].

[62] D. Hovemeyer and W. Pugh, "Finding Bugs is Easy," in *SIGPLAN Notices*, vol. 39, no. 12, December 2004, pp. 92-206.

[63] Fortify Software Inc., "Fortify Source Code Analysis (SCA)," *Fortify Software Inc.* [Online]. Available: http://www.fortify.com/products/sca.

[64] utils.com, "Lint4j," [Online]. Available: http://www.jutils.com.

[65] InfoEther, "PMD," [Online]. Available: http://pmd.sourceforge.net.

[66] QJ-Pro, "Code Analyzer for Java," [Online]. Available: http://qjpro.sourceforge.net.

[67] R. Martin, S. Barnum, S. Christey, "Being Explicit about Security Weaknesses," presented at Black Hat DC 2007, 2007. Available: http://cwe.mitre.org/about/documents.html.

[68] The SANS Institute, "SANS Software Security Institute," *The SANS Institute*. [Online]. Available: http://www.sans-ssi.org. [Accessed: March 7, 2008].

[69] National Institute of Standards and Technology, "SAMATE: Software Assurance Metrics and Tool Evaluation," *National Institute of Standards and Technology*. [Online]. Available: http://samate.nist.gov. [Accessed: February 5, 2008].

[70] Jlint, "About Jlint," [Online]. Available: http://jlint.sourceforge.net.

[71] N. Rutar, C. Almazan, and J. Foster, "A Comparison of Bug Finding Tools for Java," in *Proceedings of the 15th IEEE International Symposium on Software Reliability Engineering*, France, November 2004.

[72] S. Wagner, J. Jurjens, C. Koller, P. Trischberger, "Comparing Bug Finding Tools with Reviews and Tests," in *Proceedings of the 17th International Conference on Testing of Communication Systems*, 2005, pp. 40-55.

[73] S. Wagner, F. Deissenboeck, J. Wimmer, M. Aichner, M. Schwab, "An Evaluation of Two Bug Pattern Tools for Java," to appear in *Proceedings of the 1st IEEE International Conference on Software Testing, Verification and Validation*, 2008.

[74] N. Ayewah, W. Pugh, J. Morgenthaler, J. Penix, Y. Zhou, "Evaluating Static Analysis Defect Warnings On Production Software," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 2007, pp. 1-8.

[75] T. Aslam, "A taxonomy of security faults in the unix operating system," M.S. Thesis, Purdue University, 1995.