Journal of Cybersecurity Education, Research and Practice

Volume 2020 | Number 1 Article 3

June 2020

Evaluating and Securing Text-Based Java Code through Static Code Analysis

Jeong Yang
Texas A&M University-San Antonio, jeong.yang@tamusa.edu

Young Lee
Texas A&M University-San Antonio, young.lee@tamusa.edu

Amanda Fernandez
The University of Texas at San Antonio, amanda.fernandez@utsa.edu

Joshua Sanchez
Texas A&M University-San Antonio, jsanchez@tamusa.edu

Evaluating and Securing Text-Based Java Code through Static Code Analysis

Abstract

As the cyber security landscape dynamically evolves and security professionals work to keep apace, modern-day educators face the issue of equipping a new generation for this dynamic landscape. With cyber-attacks and vulnerabilities substantially increased over the past years in frequency and severity, it is important to design and build secure software applications from the group up. Therefore, defensive secure coding techniques covering security concepts must be taught from beginning computer science programming courses to exercise building secure applications. Using static analysis, this study thoroughly analyzed Java source code in two textbooks used at a collegiate level, with the goal of guiding educators to make a reference of the resources in teaching programming concepts from a security perspective. The resources include the methods of source code analysis and relevant tools, categorized bugs detected in the code, and compliant code examples with fixing the bugs. Overall, the first text revealed a relatively moderate bug rate of approximately 44% of files analyzed contained either regular or security bugs. About 13% of the total bugs found were security bugs and the most common security bug was related to the Pseudo Random security vulnerability. The second text produced a slightly larger bug rate of 53.80% with approximately 8% of security bugs. After combining the texts for an average rate, the total number of security bugs that were likely to appear was roughly 10% percent. This encompasses security bugs such as malicious code vulnerabilities and security vulnerabilities related to exposing or manipulating data in these programs.

Keywords

Source Code Analysis, Static Analysis, Secure Coding, Defensive Programming, Java, Software Security

Cover Page Footnote

Partial support for this work was provided by Texas A&M University-San Antonio’s Initiative Seed Grant project entitled “Building a Secure Code Analyzer in a Cloud-Based Object-Oriented Programming Environment, JaguarCode.”

INTRODUCTION

Cyber security, as a discipline, is continuously evolving to uncover, understand, and predict similarly growing cyber threats. These attacks have been substantially increased over the past years in terms of frequency, complexity, and severity. Markettos et al discuss that we are facing crises with intensive security vulnerabilities in the systems design of hardware, operating systems, and applications (Markettos, 2019). They suggest that security must be ideally considered from the ground up in order to build and manage complex hardware/software systems constructed for new types of vulnerabilities. Saydjari also advocates that engineers are responsible for designing and building safe and secure systems, and encourage them to do so in partnership with system risk analysis and management (Saydjari, 2019; Stamat, 2009**)**. Yang et al have pointed out that careless software design and implementations can cause a large number of vulnerabilities and attacks on the application itself. Therefore, it is important to stress that security is considered throughout the software development process. Toward secure software assurance, programming concepts must be taught to beginning programmers from a security perspective (Yang, 2018; Yang, 2019). This could be exercised through defensive secure programming, secure coding, and secure software development practices (Yuan 2016, Yang 2019).

Applications from secure coding practices can lead to quality software systems that are safe, secure, and reliable. While there have been efforts to provide secure coding guidelines and standards (Long, 2010; Long 2014; Seacord 2013; Yu 2011), not many colleges and universities practice secure coding in their fundamental programming courses. The ultimate goal of this study is to guide the fundamental concepts of security and defensive programming from the freshman year. Moreover, it aims to ensure that secure programming concepts are taught to beginning programmers in order to build a strong cybersecurity foundation from the ground up. The concepts learned in the foundation courses are applied to build reliable software applications, which can be further enhanced and integrated with secure software paradigms.

RELATED WORK AND BACKGROUND

Static code analysis is the process by which software developers review and examine their code for problems and inconsistencies. Static code analysis tests source code through scanning without executing, but after compiling. The source code review is critical to enhancing software security through structured design, code inspection, and peer review of the code. It can be integrated into the software development process to help developers detect potential vulnerabilities at the early stage of the development, reducing risks prior to a production environment.

Through static code analysis, the quality of the code is increased. In order to examine the code and identify security vulnerabilities, developers can either manually analyze the code or leverage analysis tools. However, manually examining the code to find security and performance bugs may have a relatively high cost in both the time of the developers and complexity of vulnerabilities. The code analysis can be done using static code analyzers - tools to assist in identification of security vulnerabilities, which developers can use in examining and analyzing their source code. Such example tools are FindBugs, Find Security Bugs, Fortify, PMD, Lapse+, and SCALe. These tools examine the code and automatically detect potential errors and bugs that pass through a compiler. While proven effective, no tool to date is capable of perfect identification, and therefore some errors may persist. Problems detected by these tools include unconditional branches into loops, undeclared or uninitialized variables, parameter type mismatches, uncalled functions and procedures, non-usage of function results, possible array bound errors, and may others. These are logical errors, vulnerabilities, and security issues that the compiler does not detect. Some tools generate reports with graphical analysis results and recommend possible solutions and suggestions. The use of the analysis tools will help developers mitigate common coding errors that could negatively affect the efficiency of applications. It can also speed up examining tasks with automation.

One critical point in recent literature is the integration of object-oriented paradigms in the instruction of introductory programming. The work of Nordström et al. discussed the flaws of common textbook examples and how to improve the quality of examples. Their study revealed that the object-oriented quality of examples is low (Nordström, 2011). A number of scholars noted that introductory programming courses using the object-oriented paradigm are more complicated, compared to the imperative/procedural paradigm (Sajaniemi, 2008; Bois, 2006; Caspersen, 2007). When the object-oriented concepts are discussed**,** examples are important for learning (Westfall, 2001; CACM, 2002; Dodani, 2003; CACM, 2005). Because, in the educational context, examples must be easy to understand for learners, but still exemplary to act as role-models for the paradigm.

Textbooks can be a major source for examples of common programming problems in introductory programming courses. Many textbook examples have been evaluated through a large-scale study to capture technical, didactical and object-oriented qualities (Börstler2, 2008). The particular needs of a novice being introduced to object orientation were taken into account and some heuristics for the design of object-oriented examples for novices were developed from those. The discussion for teaching object orientation with examples is also initiated in Börstler, 2008, and the design of examples is specifically discussed in Nordström, 2010 and Nordström, 2011.

RESEARCH GOALS

The goals of this research are 1) to detect regular and security-related bugs using static code analysis tools from Java code in the textbooks, 2) to determine whether currently taught programming practices are keeping pace with the dynamic security landscape, and 3) to eliminate insecure coding practices and suggest secure coding guidelines.

To achieve the goals and to promote effective learning with textbook examples, this study uses open-source static analysis tools on the source code from two Java textbooks used in colleges and universities for their freshman and sophomore level programming courses: Text 1 - *Starting Out with Java From Control Structures through Objects, 7th Edition by Tony Gaddis* (Gaddis, 2019) and Text 2 - *Introduction to Java Programming and Data Structures, Comprehensive Version, 11th Edition, by Y. Daniel Liang* (Liang, 2019). The textbook selection is not based on market adoption rates, but these books are used at the authors' institution for CS1/CS2 and Application Programming courses. There are also many public universities in Texas that use Text 1 with the C++ or Java programming language. Two analysis tools, FindBugs and Find Security Bugs are used to identify bugs and vulnerabilities that are present in the text code. The scope of the analysis includes common programming bugs that student developers most likely encounter with security-related bugs being the highest priority for analysis.

CODE ANALYZER TOOLS

This research studies several static code analyzer tools by comparing the most prevalent tools with their ease of use as well as their capability to detect real problems in code. Choosing an easy-to-use analysis tool is essential to introduce to beginner programmers, as it develops security knowledge in a way that does not require the tool operator to have the same level of security expertise (Chess, 2002). Both FindBugs and its extension, Find Security Bugs, were selected for having the best qualifications related to the scope of this study due to the straightforward instructions, easy to use functionalities, and their capability to check real problems.

FindBugs Comparisons with Other Tools

FindBugs is an established, highly configurable tool that allows loading custom rulesets. The customizable rulesets can detect typical errors including security-related checks (Static Code Analysis Tools, 2019). In a recent study, Oskouei et al used three well-known open-source bug-finding tools, PMD, FindBugs, and Checkstyle, to run and compare results on a variety of open-source Java programs. They found that FindBugs uses data flow and syntactic analysis to detect bugs categorized by a list of bug patterns (Oskouei, 2018) and it is expandable to allow users to add new bug patterns. Like PMD, it can be easily integrated with well-known development environments such as Eclipse and IntelliJ IDEA. In a comparison study of four Java static analysis tools (two commercials: Converity Prevent and Jtest and two open-source: FindBugs and Jlint), FindBugs was found the most discussed tool in the literature (Mamun, 2010). Probably because it is open source. Among 27 Source Code Security Analyzers for Java programming language, only seven of them are free open-source tools: FindBugs, Find Security Bugs, Jlint, LAPSE, PMD, SpotBugs, and Yasca (Source Code Analyzers, 2019).

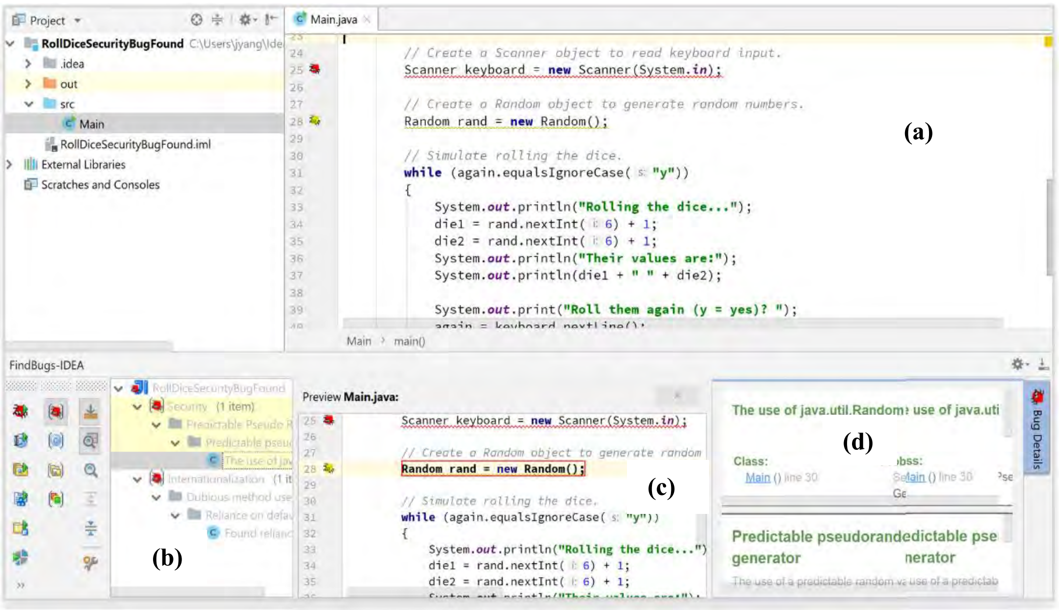
In another comparison of bug-finding tools for Java, Rutar et al applied five bug-finding tools, Bandera, ESC/Java 2, FindBugs, JLint, and PMD, to a variety of Java programs. They studied that FindBugs and JLint include dataflow components to detect syntactic bug patterns while ESC/Java uses theorem proving, and Bandera uses model checking. FindBugs also includes customizable rule sets and dataflow components for security code analysis. Unlike other tools that focus on style and formatting, FindBugs was created to find real bugs or potential performance problems in code in reducing the number of false positives (Grindstaff, 2004). According to an evaluation report, FindBugs is a good way to learn good coding practices for Java, especially for the novice software engineer, which can help them find common pieces of bad code and avoid them in the future (Analysis, 2009).

FindBugs gives correct results by uncovering potential vulnerabilities such as null pointer dereferences, redundant comparisons to null, dead store to local variables, synchronization errors, vulnerabilities to malicious code, and deadlock situations present in the code **(**Charhar, 2012; Source Code Analyzers, 2019). Its analysis outputs not only include the types of bugs found, but also describes what the bug is in depth. It provides advice on how bugs can be fixed. While other static analysis tools such as PMD and Lapse+ have similar capabilities, they do not provide the full scope of our project requirements. Although PMD can successfully detect common poor coding techniques and dead code such as null pointer dereferencing, it does not catch injection vulnerabilities and unsafe development practices (Mahmood, 2018; Charhar, 2012**,** Source Code Analyzers, 2019**)**.

Due to the comprehensiveness of its analytical properties, the detailed output of its analysis results, the easy to use and plugin functionalities, and good practice for Java for novice programmers, FindBugs is selected in this study. Furthermore, no previous study has evaluated the effectiveness of the Java static analysis tools for textbook sample codes.

FindBugs and Find Security Bugs

FindBugs is a static code analysis tool for Java programs (FindBugs, 2018). It can be a stand-alone and plugin program, which was originally designed to find occurrences of similar bugs (Hovemeyer, 2005). FindBugs requires compiled code first to detect bugs, which can contribute to obtain low false positives and detect critical security-related bugs. FindBugs itself relates to the bugs of performance or syntax. Whatever the interpreter or compiler won't catch, FindBugs will usually pick up. FindBugs-IDEA is a plugin for the IntelliJ IDEA IDE (IntelliJ, 2018). This plugin allows for seamless integration of the tool into the IntelliJ environment, providing various methods of analyzing Java programs. As a part of the plugin, certain expansions can be added to increase functionalities. One such extension is Find Security Bugs (Security Bugs, 2018). Find Security Bugs focuses on finding security vulnerabilities of Java programs such as insecure usages of variables, SQL injection vulnerabilities, pseudo-random number generators, and potential path traversal (Bugs, 2018).



```
    // Create a Scanner object to read keyboard input.
    Scanner keyboard = new Scanner(System.in);

    // Create a Random object to generate random numbers.
    Random rand = new Random();

    // Simulate rolling the dice.
    while (again.equalsIgnoreCase( s: "y"))
    {
        System.out.println("Rolling the dice...");
        die1 = rand.nextInt( i: 6) + 1;
        die2 = rand.nextInt( i: 6) + 1;
        System.out.println("Their values are:");
        System.out.println(die1 + " " + die2);

        System.out.print("Roll them again (y = yes)? ");
```

```
    Scanner keyboard = new Scanner(System.in);

    // Create a Random object to generate random
    Random rand = new Random();

    // Simulate rolling the dice.
    while (again.equalsIgnoreCase( s: "y"))
    {
        System.out.println("Rolling the dice...")
        die1 = rand.nextInt( i: 6) + 1;
        die2 = rand.nextInt( i: 6) + 1;
```

Figure 1. Find Security Bugs in Action in IntelliJ IDEA

Figure 1 shows a screenshot of Find Security Bugs in action in the IntelliJ IDEA environment showing bug detection for the sample code. From this, a user can easily identify the main issues in the code: an internationalization regular bug and a predictable random security bug as shown in areas (b) and (d) and their corresponding code lines from areas (a) and (c) of the FindBugs-IDEA at the bottom.

FindBugs Bug Patterns

When FindBugs is in action in analyzing source code, its reporting categorizes bug patterns to Bad Practice, Correctness, Malicious Code Vulnerability, Performance, Security, Dodgy Code, Multithreaded Correctness, Experimental, and Internalization (FindBugs, 2018).

Bad Practice (B) code violates recommended and essential coding practices. The examples of bad practice include equals problems, improperly formatted strings, dropped exceptions, serializable problems, and misuse of finalizing. **Dodgy Code (D)** is a confusing code that is anomalous or written in a way that can lead to errors. Examples include dead local stores, unconfirmed casts, division overflows, useless object creation, switch fall through, unconfirmed casts, and redundant null check of value known to be null. **Correctness Bugs (C)** are probable bugs with apparent coding mistakes that are probably not what developers intended. They can produce unwanted results.

Performance (P) related inefficient code can cause performance degradation and resource wasted. This could be software defects that lead to reduced throughput, increased latency, and wasted resources. For example, when a class contains an instance final field that is initialized to a compile-time static value, it should be considered to be a static field. Unread fields that are never read can be removed from the class. The examples of inefficient code include String concatenation using + in a loop, inefficient new String() constructor invoked, and inefficient number constructor invoke. This kind of code can be written differently to improve performance.

Experimental (E) code can miss cleanup of streams, database objects, or other objects that require a cleanup operation. For example, a method may fail to clean up (close, dispose of) a stream, database object, or other resource requiring an explicit cleanup operation. In general, if a method opens a stream or other resource, the method should use a try/finally block to ensure that the stream or resource is cleaned up before the method returns. **Internationalization (I)** code can inhibit the use of international characters. Using a default encoding can lead to incompatibility on systems with certain defaults. For example, when the default encoding is used for the scanner input, the use of utf-8 can resolve the issue since the presence of "utf-8" explicitly declares the encoding of the scanner.

While these bug patterns do not directly relate to security issues, it is still important for students to know these bug patterns and practice the fundamental concepts of defensive programming with them. For example, Dodgy code with the improper use of division operations can lead to integer overflows. Detecting this type of code will ensure programmers to use data types and their operations safely to prevent integer errors and buffer overflows.

Find Security Bugs Bug Patterns

Find Security Bugs' report categorizes its security bug patterns into Predictable Random, Potential Path Traversal, Malicious Code Vulnerability, and SQL vulnerability **(Security Bugs, 2018)**.

When a **Predictable Random (PR)** value is used in a certain security-critical context, it can lead to vulnerabilities. For example, when the value is used as a) a Cross-Site Request Forgery (CSRF) token as a predictable token can lead to a CSRF attack as an attacker may know the value of the token; b) a password reset token sent by email - a predictable password token can lead to an account takeover since an attacker can guess the URL of the password form: c) any other secret value.

Path Traversal (PT) is known as a directory traversal that can access files and directories that are outside the system**.** This path traversal issue can be alerted from/to reading/writing a file whose location is specified by user input with the filename comes from an input parameter. With **Malicious Code Vulnerability (M),** code can be maliciously changed by other code. Malicious code can cause undesired effects, security breaches or damages to a system. For example, a method may expose internal representation by storing an externally mutable object.

SQL queries can lead to **SQL Injections (S),** in which input values in the queries can be unsafely passed. A vulnerability occurs when the original SQL query can be altered to make a different query and the execution of the altered query result in data leaks or modification.

For instance, a database contains user names and passwords that have a string size limit of 8 and 20 respectively. A SQL command to authenticate a user takes the form SELECT * FROM users WHERE username = '<USERNAME>' AND password = '<PASSWORD>' returns records where the user names and passwords are valid.

If attackers can substitute arbitrary strings of <USERNAME> and <PASSWORD>, they can perform a SQL injection for <USERNAME> when injected into the command with: SELECT * FROM users WHERE username = 'validuser' OR '1' = '1' AND password = '<PASSWORD>'.

If 'validuser' is a valid user name, this statement selects all validuser records in the database table without checking their passwords. The attackers can log in without a correct username and password as the '1' ='1' tautology can disable both username and password validation. Therefore, sanitizing and validating untrusted input and parameterizing queries are very important to prevent SQL injection vulnerabilities.

The Texts

Text 1 - *Starting out with Java: From Control Structures through Objects* (Gaddis, 2019), was used to analyze the source code throughout all 12 chapters, as it represents the most modern example of beginner Java concepts being taught across colleges and universities. Text 2 - *Introduction to Java Programming and Data Structures, Comprehensive Version* (Liang, 2019) was also used for the analysis: 16 chapters for beginner Java concepts classified into six groups as shown in Table 1, and 14 later chapters for advanced Java concepts classified into additional seven groups in Table 5. Java source code from Chapters 12, 13 and 14 in Text 1 and Chapters 14, 15, and 16 in Text 2 were excluded from the data analysis as they cover JavaFX with controls, graphics, effects, and media for GUI programming.

RESEARCH METHODS AND RESULTS

To determine whether the currently taught programming practices keep pace with the dynamic security landscape, this section reports the results of the static code analysis of Java code from two textbooks. The analysis used a certain amount of classification of datum with Java code segments examined and the type of bugs found by the code analyzer. Certain cases had to also be considered, such as programs with a regular bug and a security bug, the scope of bug finding for Find Bugs and Find Security Bugs, the precedence of bugs in the final analysis, and the definition of the categories for data gathering.

Grouping and Data Collection for Beginner Java Concepts

The groups were composed based on similar topics of the source code in the consecutive chapters (Table 1). Most topics in Groups 1 through 5 are being taught in CS1 and CS2 courses. As the goal was to look for the presence of bugs, when analyzing the code, the data was classified and collected into two different categories: Regular Bugs and Security Bugs. Considering the purpose of the analysis, security Bugs have precedence over Regular Bugs due to the larger implications of potential security-related information/data leaks in the code.

The analysis was conducted with the Java code examples throughout the two textbooks. As the title of the books indicate, while Text 1 covers fundamental Java concepts from control structures to objects, and beyond, Text 2 covers Data Structures concepts as well as the fundamentals. It should be noted that there were intentionally incomplete code examples. Therefore, the analysis results have considered this and adjusted themselves accordingly. When these cases were encountered, the Java program was considered to be bug-free due to developer intent.

Group	Text 1		Text 2	
	Ch.	Content	Ch.	Content
1. Fundamentals, Control Structures, Methods	2	Java Fundamentals	2	Elementary Programming
	3	Decision Structures	3	Selections
	4	Loops and Files	4	Mathematical Functions, Characters, and Strings
	5	Methods	5	Loops
			6	Methods
2. Arrays	7	Arrays and ArrayList Class	7	Single Dimensional Arrays
			8	Multidimensional Arrays
3. OOP	6	A first Look at Classes	9	Objects and Classes
	8	A Second Look at Classes and Objects	10	Object-Oriented Thinking
	9	Text Processing and More about Wrapper Classes	11	Inheritance and Polymorphism
	10	Inheritance	13	Abstract Classes and Interface
4. File I/O	11	Exceptions and Advanced File I/O	12	Exception Handling and Text I/O
			17	Binary I/O
5. Recursion	15	Recursion	18	Recursion
6. Databases	16	Databases	34	Java Database programming
			35	Advanced Database programming

Table 1: Texts Group Composition

Discussions

A total of 227 files were scanned throughout 12 Chapters of Text 1 and 184 files were scanned throughout 18 Chapters of Text 2. The books use modern examples of Java concepts being taught across colleges and universities. The source code data was classified into two categories based on their bug patterns - Regular Bugs and Security Bugs. Tables 2 and 3 show the number of each bug pattern in each of these categories for each chapter of Texts 1 and 2 respectively.

The analysis results of Text 1 indicate that 46.56% (61 out of 131) of the bugs found are internalization bugs in the regular category and 42.1% (8 out of 19) of the security bugs found are related to predictable random. Chapter 16 Databases has the most bugs (25 out of 131) in the regular category (19.09%). The analysis results of Text 2 show that 48.59% (69 out of 142) of the bugs found are also internalization bugs in the regular category and 76.92% (10 out of 13) of the security bugs found are related to SQL Injections from Chapters 34 and 35 for Database Programming. Chapter 12 Exception Handling and Text I/O has the most bugs (30 out of 142) in the regular category (21.13%).

Gr.	**Ch.**	**No. Files**	**Regular Bugs**							**Security Bugs**				
			I	D	B	C	P	E	Total	PR	PT	M	S	Total
1	2	33	5	0	0	0	0	0	5	0	0	0	0	0
	3	24	11	1	6	0	0	0	**18**	0	0	0	0	0
	4	25	14	1	3	0	0	0	**18**	2	4	0	0	**6**
	5	16	1	0	1	0	0	0	2	0	1	0	0	1
2	7	33	9	2	2	0	0	0	13	0	0	1	0	1
3	6	18	1	1	1	0	0	0	3	4	0	0	0	4
	8	9	1	0	2	2	0	0	5	0	0	0	0	0
	9	13	5	0	1	0	0	0	6	0	1	0	0	1
	10	17	6	2	1	2	8	0	**19**	0	0	0	0	0
4	11	17	6	4	1	2	1	0	14	2	0	0	0	2
5	15	12	1	2	0	0	0	0	3	0	0	0	0	0
6	16	10	1	0	4	0	2	18	**25**	0	0	0	4	4
Total		**227**	**61**	13	22	6	11	18	**131**	**8**	6	1	4	**19**

Table 2: Regular and Security Bugs in Chapters for Text 1

Gr.	**Ch.**	**No. Files**	**Regular Bugs**							**Security Bugs**				
			I	D	B	C	P	E	Total	PR	PT	M	S	Total
1	2	10	8	0	3	0	0	0	11	0	0	0	0	0
	3	9	9	1	0	0	2	0	12	0	0	0	0	0
	4	7	5	0	1	0	1	0	7	0	0	0	0	0
	5	15	9	0	1	0	5	0	15	0	0	0	0	0
	6	12	3	1	1	0	0	0	5	0	0	0	0	0
2	7	10	1	2	0	0	0	0	3	0	0	0	0	0
	8	6	5	0	0	0	0	0	5	0	0	0	0	0
3	9	11	1	2	1	0	0	0	4	0	0	0	0	0
	10	10	2	0	1	0	0	0	3	0	0	0	0	0
	13	16	0	0	8	0	2	0	10	0	0	2	0	2
	11	10	1	0	0	1	0	0	2	0	0	1	0	1
4	12	22	18	4	8	0	0	0	**30**	0	0	0	0	0
	17	8	0	0	0	0	1	0	1	0	0	0	0	0
5	18	9	4	2	0	0	0	0	6	0	0	0	0	0
	19	14	0	1	0	0	2	0	3	0	0	0	0	0
6	34	7	1	0	6	0	0	6	13	0	0	0	8	8
	35	8	2	0	3	0	1	6	12	0	0	0	2	2
Total		**184**	**69**	13	33	1	14	12	**142**	0	0	3	**10**	**13**

Table 3: Regular and Security Bugs in Chapters for Text 2

The first categorized group, Fundamentals, Control Structures, and Methods had a significantly larger bug rate in Text 2 (77%) compared to Text 1 (38%) even though the number of files scanned in Text 1 was greater than Text 2. However, there were a few security bugs found (7%) with regular bugs being the most common issues (93%), such as internationalization, dodgy code, and performance errors.

Group 2 (Arrays) had a relatively small number of bugs found with bugs rates at 33% and 25% for Texts 1 and 2 respectively. There was one security bug found in this group that posed as a possible vulnerability that could reveal internal information from the method used by storing an externally mutable object. The latter of the bugs were common-type bugs related to improper programming practices. For instance, there was dodgy code where a switch case fell through due to not implementing the default case. Upon examining Group 3 (Object-Oriented Programming), the number of bugs found within Text 1 was far more surmountable than the bugs found in Text 2. This is reflected when examining the bug rates, 54%, and 34%, for Texts 1 and 2, respectively. Despite having more or fewer bugs, the texts shared a commonality with the types of bugs discovered. The most notable bug found was a security bug related to a malicious code vulnerability where the method called was returning an array that may expose internal representation.

In Group 4 (File I/O), there were a few security bugs identified, but the large portion of the bugs in this group were regular common-type bugs. These varied from simple bugs such as bad practice or correctness, which ranged from using default encoding like the Scanner class to implementation issues of methods being used. The regular bugs discovered here attributed to most of the weight when computing the bug rate, which resulted in the second-highest rate (62.5%). Subsequently, Group 5 (Recursion) reported that no security bugs were discovered.

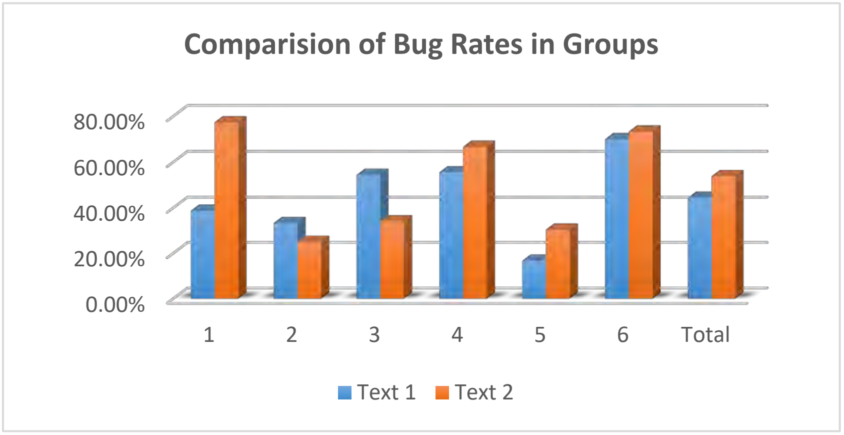


Figure 2: Textbooks Comparisons of Bug Rates in Groups

Figure 2 and Table 4 provide an overview of regular bugs and security bugs found in the combined texts by grouping the chapters into 6 groups. After the grouping process, a bug rate was calculated to determine the percentage of bugs occurring in each group. Bug rate was computed by dividing the number of bugged files with the number of files scanned for each group. In this manner, the likeliness of similar bugs occurring in other previous or future texts can be generally estimated. This group also had the lowest percentage when comparing bug rates across all groups with only ~26% overall. The few bugs that were found included simple bugs such as extraneous objects being stored into variables and possible null pointer dereferencing due to the return values when calling the implemented method.

Gr.	Text	# of Regular Bugs	# of Security Bugs	Total # of Bugs	# of Files with Bugs	# of Files Scanned	Bug Rate
1	1	43	7	50	38	98	38.78%
	2	50	0	50	41	53	**77.36%**
	Total	93	7	100	79	151	52.31%
2	1	13	1	14	11	33	33.33%
	2	8	0	8	4	16	25.00%
	Total	21	1	22	15	49	30.61%
3	1	33	5	38	31	57	54.39%
	2	19	3	22	16	47	34.04%
	Total	52	8	60	47	104	45.19%
4	1	14	2	16	10	18	55.56%
	2	31	0	31	20	30	**66.67%**
	Total	45	2	47	30	48	62.5%
5	1	3	0	3	2	12	16.7%
	2	9	0	0	7	23	30.43%
	Total	12	0	3	9	35	25.71%
6	1	25	4	29	7	10	**70.00%**
	2	25	10	35	11	15	73.33%
	Total	50	14	64	18	25	72%
Total	1	131	19	150	99	228	44.42%
	2	142	13	155	99	184	53.80%
	Total	273	32	305	198	412	48.06%

Table 4: Bug Rates in Groups

Group 6 (Databases) was the most peculiar concerning security bugs with a high probability of these types of bugs occurring throughout both Texts. These bugs were produced when non-constant strings were being passed to execute methods involving SQL statements, which were discovered to be at risk due to an SQL injection vulnerability. SQL injection is one of the most common threats to be considered since it is the most used language to communicate with databases.

While this section had the highest bug rates (72%), the latter number of bugs were regular type bugs that comprised mostly of cleaning up and closing database resources after being used which could present another opportunity for vulnerabilities. Overall, the first text revealed a relatively moderate bug rate of 44.42% with thirteen percent of security bugs. The second text produced a slightly larger bug rate of 53.80% with approximately eight percent of security bugs. After combining the texts for an average rate, the total number of security bugs that were likely to appear was roughly ten percent. This encompasses security bugs such as malicious code vulnerabilities and security vulnerabilities related to exposing or manipulating data in these programs.

The most interest in the results is that 77.36% (41 out of 53) of the elementary programming source code in Group 1 of Text 2 contains 100 bugs (93 regular bugs and 7 security bugs). The second most interested group is that 70% (7 out of 10) of the database-related source code in Group 6 of Text 1 contains 25 bugs. It was observed that a CoffeeDBManager.java class with six methods implemented in it contains 11 regular experimental and 4 security bugs. That class performs operations on the coffee database using 'select' and 'insert' SQL queries and ArrayList. 48.06% of the files from both Texts contain bugs while many have multiple different types of bug patterns. In summary, 305 bugs were found from 198 Java files out of 412 files scanned.

The pie chart in Figure 3 represents the percentage of bugs found in both Text 1 and Text 2's groups that were combined to get an average bug rate to obtain an overview of how many bugs were discovered for each. By classifying the groups with similar chapters involved, the distribution of bugs throughout the texts was analyzed further. For instance, when comparing Groups 5 and 6, (Recursion and Database), a drastic difference in bug rates occurred. While they have a similar number of files scanned, the Database grouping is much more likely to produce bugs whether they are regular or security bugs. This high percentage indicates that there should be a review of the chapters in this group to prevent future programmers from replicating these errors.

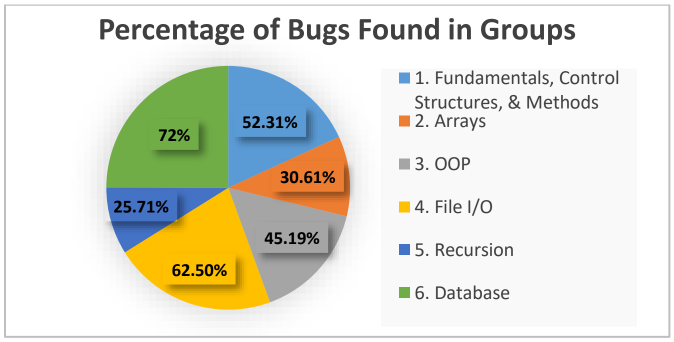


Figure 3: Percentage of Bugs Found in Groups

Grouping and Data Collection for Advanced Java Concepts

The advanced topics covered in Text 2 were classified into a multitude of groupings with corresponding chapters as shown in Table 5. Several subgroups were formed due to the text having multiple chapters covering an extensive amount of Data Structures and Algorithms such as trees, graphs, and sorting algorithms. Most of these topics are appropriate to be taught in CS2 and/or Data Structures courses. Again, the code analysis looked for the presence of bugs with classified categories: Regular Bugs and Security Bugs.

Text 2 Group	Ch.	Content
7. Fundamentals Data Structures	20	Lists, Stacks, Queues, and Priority Queues
	21	Sets and Maps
	24	Implementing Lists, Stacks, Queues, and Priority Queues
8. Algorithms	22	Developing Efficient Algorithms
	23	Sorting
	27	Hashing
9. Trees	25	Binary Search Trees
	26	AVL Trees
10. Graphs	28	Graphs and Applications
	29	Weighted Graphs and Applications
11. Collection Streams	30	Aggregate Operations for Collection Streams
12. Networking & Parall Programming	32	Multithreading and Parallel Programming
	33	Networking
13. Internationalization	36	Internationalization

Table 5: Group Composition for Advanced Topics in Text 2

Discussions

A total of 119 files were scanned throughout 14 Chapters of Text 2. These book chapters use modern examples of advanced Java concepts being taught for CS2 and Data structures courses. The source code data was also classified into two categories: Regular Bugs and Security Bugs. Table 6 displays the number of each bug pattern found in each of these categories for each chapter for advanced topics in Text 2. The analysis results indicate that 46.57% (34 out of 73) of the bugs found are performance and inefficient code related bugs in the regular category and only 3 malicious code vulnerability bugs were found from the group 10 Graphs and Applications in the security category. While these trends are somehow different from the bug findings for the beginner Java concepts, Bad Practice codes consistently present throughout the chapters in both Texts (Text 1: 22/131, Text 2: 33/142 for beginner topics and 22/72 for advanced topics).

Gr.	Ch.	No.	Regular Bugs							Security Bugs				
		Files	I	D	B	C	P	E	Total	PR	PT	M	S	Total
7	20	11	0	0	2	0	3	0	5	0	0	0	0	0
	21	9	2	1	0	0	0	0	3	0	0	0	0	0
	24	9	0	0	3	0	0	0	3	0	0	0	0	0
8	22	8	6	0	2	0	0	0	8	0	0	0	0	0
	23	8	0	0	2	0	1	0	3	0	0	0	0	0
	27	5	0	0	0	0	1	0	1	0	0	0	0	0
9	25	8	1	0	3	0	1	0	5	0	0	0	0	0
	26	2	0	1	0	0	0	0	1	0	0	0	0	0
10	28	13	1	0	6	0	6	0	13	0	0	2	0	2
	29	6	1	0	2	0	2	0	5	0	0	1	0	1
11	30	12	4	0	2	0	0	0	6	0	0	0	0	0
12	32	10	0	0	0	0	9	0	9	0	0	0	0	0
	33	10	0	0	0	0	7	0	7	0	0	0	0	0
13	36	8	0	0	0	0	4	0	4	0	0	0	0	0
	Total	**119**	15	2	22	0	**34**	0	**73**	0	0	**3**	0	**3**

Table 6: Regular and Security Bugs for Advanced Topics in Text 2

With the seven groups established, the bug rate was computed across each and the results identified the distribution of bugs throughout this section. While the groups can be examined individually, they were analyzed as an overall group for advanced topics. This resulted that the advanced topic groups had a total bug rate of about 37% with approximately 4% of security bugs detected. Figure 4 displays the distribution of bugs found in the advanced topics for Text 2. Upon examination, Group 11 Collection Streams has a comparatively high bug rate (50%), but the types of errors discovered in this group did not pose a serious threat regarding security. However, after reviewing Group 10 Graphs, the second-highest bug rate (47.37%), a series of security bugs were revealed that implicates a risk of vulnerabilities such as exposing internal information being stored within the files.

Gr.	# of Regular Bugs	# of Security Bugs	Total # of Bugs	# of Files with Bugs	# of Files Scanned	Bug Rate
7	11	0	11	9	29	31.03%
8	12	0	12	9	21	42.86%
9	6	0	6	2	10	20.00%
10	18	3	**21**	9	19	**47.37%**
11	6	0	6	6	12	**50.00%**
12	16	0	16	7	20	35.00%
13	4	0	4	2	8	25.00%
Total	73	3	76	44	119	36.97%

Table 7: Bug Rates in Groups for Advanced Topics in Text 2

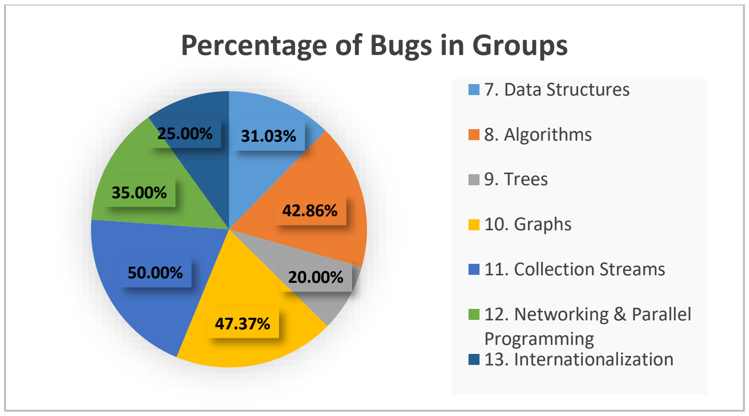


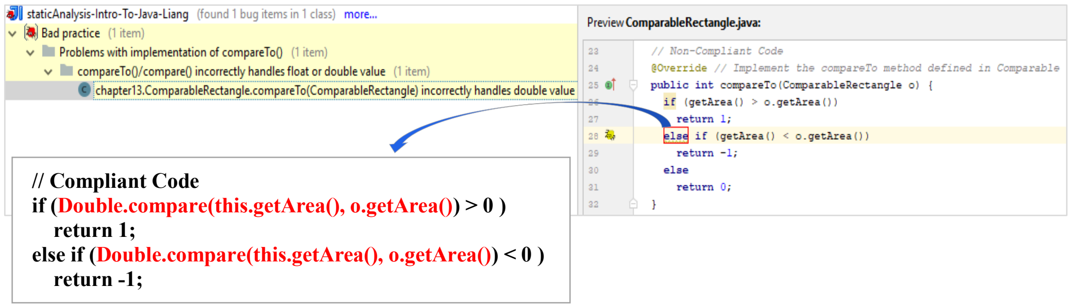
Figure 4: Percentage of Bugs Found in Groups for Advanced Topics in Text 2

CASE STUDY WITH COMPLIANT-CODE EXAMPLES

This section presents several case studies in converting non-compliant code with bugs found by the analysis to compliant code without bugs. These case studies with the compliant code can guide instructors and students to better equip teaching/learning secure programming toward reliable software development. Each case study is also interpreted in terms of the 2019 common 25 bug patterns suggested by CWE (CWE, 2019). APPENDIX A provides more examples of each category of bugs identified.

Case Study 1: Bad Practice in Regular Category

This case study explains the example of regular bugs. The code snippet in Figure 5 has a type of Bad Practice that has been identified as a problem with the implementation of the compareTo method in ComparableRectangle.java in Chapter 13 of Text 2. The issue with the compareTo method is that the comparison statements do not handle the special cases for double or float values correctly at lines 26 and 28. For example, if the values -0.0 or NaN were passed into this method, then an incorrect result may be displayed.



```
staticAnalysis-Intro-To-Java-Liang (found 1 bug items in 1 class) more...
  Bad practice (1 item)
    Problems with implementation of compareTo() (1 item)
      compareTo()/compare() incorrectly handles float or double value (1 item)
        chapter13.ComparableRectangle.compareTo(ComparableRectangle) incorrectly handles double value
```

Preview ComparableRectangle.java:

```
23  // Non-Compliant Code
24  @Override // Implement the compareTo method defined in Comparable
25  public int compareTo(ComparableRectangle o) {
26    if (getArea() > o.getArea())
27      return 1;
28    else if (getArea() < o.getArea())
29      return -1;
30    else
31      return 0;
32  }
```

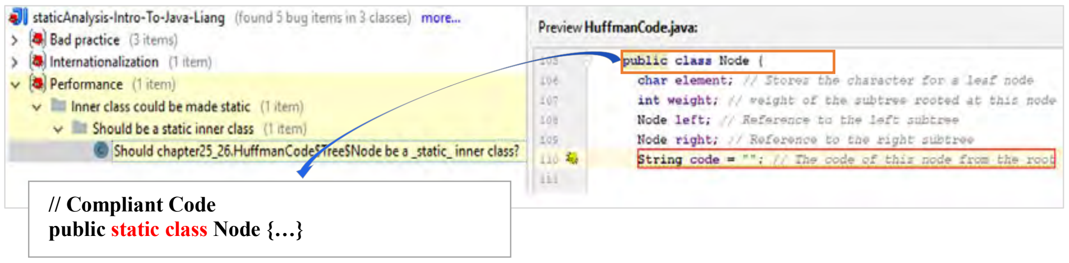
```
// Compliant Code
if (Double.compare(this.getArea(), o.getArea()) > 0 )
   return 1;
else if (Double.compare(this.getArea(), o.getArea()) < 0 )
   return -1;
```

Figure 5: Bad Practice Example

To resolve the issue with the compareTo method, the Double.compare method was used to replace the condition checks and handle the special cases correctly such as -0.0 and NaN. If these cases were left unhandled, then the return value of the area would return a negative value, which implies that an overflow also occurred in the stack. This would allow someone with malicious intent to exploit this vulnerability by causing additional overflow errors to retrieve verbose error messages. This bug is related to CWE-20: Improper Input Validation and CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer (CWE, 2019).

Case Study 2: Performance in Regular Category

This case study examines an example of non-compliant code that has a Performance related bug detected in HuffmanCode.java in Chapter 25 from Text 2. The Tree class in Figure 6 defines a Huffman coding tree. The issue represented from this code is about an inner public class, ***Node***, which is nested inside the public static ***Tree*** class. While it demonstrates the proper structure for a class, it does not use its embedded references, ***left and right***, to the object, which it defined. This reference makes the instances of the inner class, ***Node***, and may keep the reference to the creator object alive longer than necessary. Thus, the more node objects that the inner class instantiates, the more of an effect it will have on performance. This bug is related to CWE-200: Information Exposure (CWE, 2019).


staticAnalysis-Intro-To-Java-Liang (found 5 bug items in 3 classes) more...
Bad practice (3 items)
Internationalization (1 item)
Performance (1 item)
Inner class could be made static (1 item)
Should be a static inner class (1 item)
Should chapter25_26.HuffmanCode$Tree$Node be a _static_ inner class?
Preview HuffmanCode.java:

```
105 public class Node {
106     char element; // Stores the character for a leaf node
107     int weight; // weight of the subtree rooted at this node
108     Node left; // Reference to the left subtree
109     Node right; // Reference to the right subtree
110     String code = ""; // The code of this node from the root
111
```

```
// Compliant Code
public static class Node {…}
```

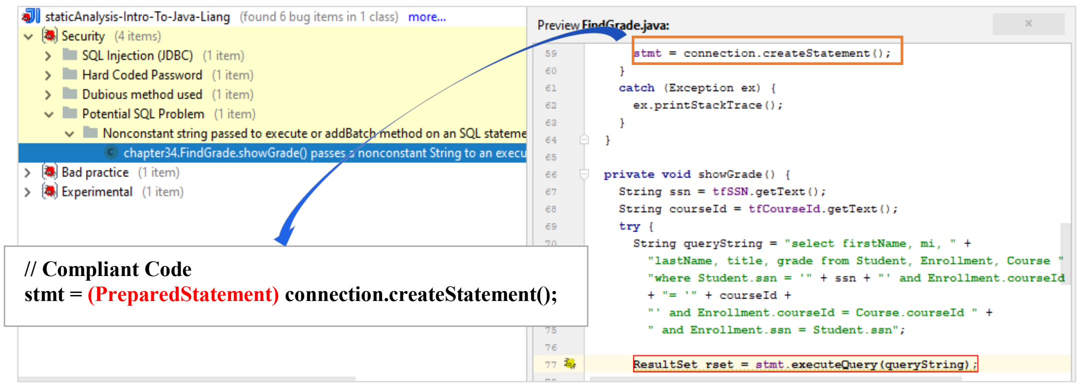
Figure 6: Performance Example

The solution to this performance issue was to refactor the inner class to be a static class (***public static class Node*** at line 105). This allows the inner class, ***Node***, to still be accessed by the outer class, ***Tree***, but limits the usage of the class's data members and methods, which increases its efficiency.

Case Study 3: SQL Injection in Security Category

This case study describes an example of non-compliant code that has a security bug detected in Text 2 in FindGrade.java in Chapter 34. The code results in a potential SQL injection problem as the SQL statement is being passed with a string that is being generated dynamically at line 77 in Figure 7.

This bug is related to CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection') (CWE, 2019).



```
staticAnalysis-Intro-To-Java-Liang   (found 6 bug items in 1 class)   more...
  Security   (4 items)
    SQL Injection (JDBC)   (1 item)
    Hard Coded Password   (1 item)
    Dubious method used   (1 item)
    Potential SQL Problem   (1 item)
      Nonconstant string passed to execute or addBatch method on an SQL stateme
        chapter34.FindGrade.showGrade() passes a nonconstant String to an execu
  Bad practice   (1 item)
  Experimental   (1 item)
```

```
// Compliant Code
stmt = (PreparedStatement) connection.createStatement();
```

Preview FindGrade.java:

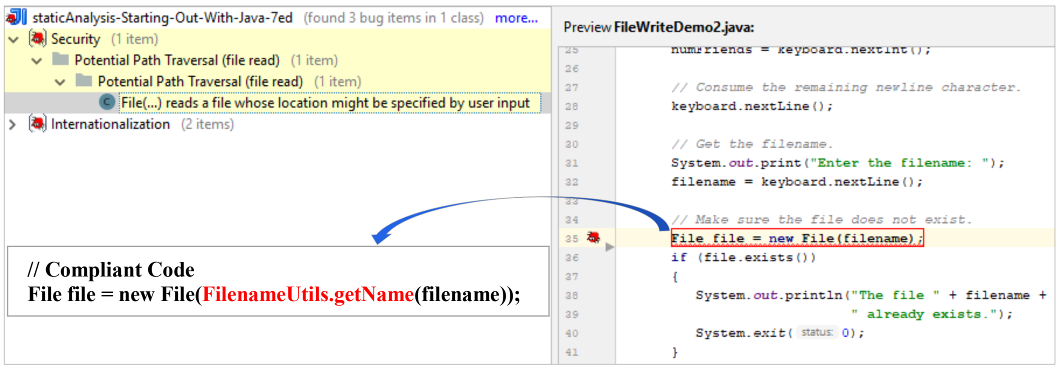
```
      stmt = connection.createStatement();
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }
  }

  private void showGrade() {
    String ssn = tfSSN.getText();
    String courseId = tfCourseId.getText();
    try {
      String queryString = "select firstName, mi, " +
        "lastName, title, grade from Student, Enrollment, Course "
        "where Student.ssn = '" + ssn + "' and Enrollment.courseId
        + "= '" + courseId +
        "' and Enrollment.courseId = Course.courseId " +
        " and Enrollment.ssn = Student.ssn";

      ResultSet rset = stmt.executeQuery(queryString);
```

Figure 7: SQL Injection Example

To resolve this issue, a Prepared Statement is recommended in the original statement's place to make it less vulnerable to SQL injection attacks at line 39. The advantage of using a Prepared Statement is that when it is executed, the DBMS can just run the prepared SQL statement without having to compile first. In the original code, the object being used to create a statement was derived from the Statement class as a global variable. This object was changed to a Prepared Statement object that inherits the Statement class and then type-casted when the statement is being created. This approach resolved the security bug found in this program and made it less vulnerable to SQL injection attacks.

Case Study 4: Potential Path Traversal in Security Category

This case study examines an example of a non-compliant code that has a security-related bug detected in FileWriteDemo2.java in Chapter 4 from Text 1. It depicts the issue involving the creating of a File object and passing the filename into the input parameter, which is demonstrated at line 35 in Figure 8. If an unfiltered parameter is passed to this file API, then files in an arbitrary file system location could be altered or modified. The vulnerability discovered, a potential path traversal attack, aims to access files and directories that are stored outside the web root folder and this vulnerability can manipulate variables that reference files with "dot-dot-slash" sequences with the purpose of targeting application source code or configuration and critical system files. This bug is related to CWE-22: Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal') (CWE, 2019).



```
staticAnalysis-Starting-Out-With-Java-7ed (found 3 bug items in 1 class) more...
  Security (1 item)
    Potential Path Traversal (file read) (1 item)
      Potential Path Traversal (file read) (1 item)
        File(...) reads a file whose location might be specified by user input
  Internationalization (2 items)
```

```
// Compliant Code
File file = new File(FilenameUtils.getName(filename));
```

Preview FileWriteDemo2.java:

```
25          numFriends = keyboard.nextInt();
26
27          // Consume the remaining newline character.
28          keyboard.nextLine();
29
30          // Get the filename.
31          System.out.print("Enter the filename: ");
32          filename = keyboard.nextLine();
33
34          // Make sure the file does not exist.
35          File file = new File(filename);
36          if (file.exists())
37          {
38             System.out.println("The file " + filename +
39                                " already exists.");
40             System.exit( status: 0);
41          }
```

Figure 8: Potential Path Traversal Example

To resolve this issue of file path traversal, line 35 was refactored to utilize the FilenameUtils class from the Apache Commons IO library (version 2.6) and pass the filename through the getName() method so that it only returns the filename minus the path from the full filename, keeping the access file relative to the current working directory. This would prevent the manipulation of web root folders and the files that hold the configurations or critical system files. This solution also helps when moving from a Windows-based development machine to a Unix based production machine.

LIMITITATIONS

This study has limitations because the code analysis has only been conducted for simple Java code from two textbooks at a small scale. Only two tools were used, which can reduce the potential breadth of the analysis. FindBugs software has not been updated since 2013. This limits the scope of analysis to JDK 1.8. Moreover, the results of the analysis rely on only those bug categories supported by FindBugs and Find Security Bugs. Just relying on one form of static code analysis may result in a large number of false positives or negatives and it is difficult to verify all of the results. In both Texts, a large number of Internalization bugs (14 in chapter 4 for Text 1 and 18 in chapter 22 for Text 2) were found.

Most of them relate to a Scanner method with input streams. Whereas the internationalization bug in these cases refers to the Scanner.in using a default encoding method that has not been stated explicitly, leading to possible problems later down the line. Therefore, the code should be imported to a machine using a different character-encoding standard. Furthermore, by declaring UTF-8 explicitly to specify an explicit charset, the issue can be resolved. Most of the Bad Practices bugs relate to format strings using ‘\n’. These can be simply fixed by using ‘%n’.

This study can be improved by conducting more analysis using SpotBugs, which is a successor of FindBugs with updated bug patterns, as well as many other static and dynamic tools. Additional research could also incorporate the latest JDK versions and their respective new features. In addition, the size of the dataset analyzed is not thoroughly comprehensive to make a concrete conclusion about education about secure Java programming practices at large.

CONCLUSIONS AND FUTURE WORK

This study, through the static code analysis, examined regular and security bugs in Java code from the two textbooks used for fundamental programming courses at a college level. Overall, the first text revealed a relatively moderate bug rate of approximately 44% of files analyzed contained either regular or security bugs. Some have both. About 13% of the total bugs found were security bugs and the most common security bug was related to the Pseudo Random security vulnerability. 87% of the total bugs found were regular bugs with the most common bugs related to Internalization. The second text produced a slightly larger bug rate of 53.80% with approximately 8% of security bugs. After combining the texts for an average rate, the total of security bugs that were likely to appear was roughly 10% percent. This encompasses security bugs such as malicious code vulnerabilities and security vulnerabilities related to exposing or manipulating data in these programs.

Remarkably, 77.36% (41 out of 53) of the elementary programming source code in Group 1 of Text 2 contains bugs (93 regular and 7 security bugs) and 70% (7 out of 10) of the database-related source code in Group 6 of Text 1 contains bugs (25 regular ad 4 security bugs). In summary, 305 bugs were found from 198 Java files out of 412 files scanned from both textbooks for beginning Java concepts. This analysis takes into account edge cases as well as removal of false positives from the final analysis results of the big groups. While the code analysis is a good start, there is significant research remaining on secure coding practices and common coding practices in general.

The advanced topic groups from Text 2 had a total bug rate of about 37% with approximately 4% of security bugs detected. The Collection Streams group has a comparatively high bug rate (50%), but the types of errors discovered in this group posed no serious security threat. However, the Graphs group demonstrated the second-highest bug rate (47.37%), in which a series of security bugs were revealed. This implies a risk of vulnerabilities, such as exposure of internal information stored within the files.

Future work includes reducing and resolving all of the detected errors, finding which bugs types have more serious issues in potential data leaks, and determining how to resolve the issues. Source code analysis will be expanded to broader domains and tools for advanced courses such as Database Systems, Computer Networks, Computer Security, and Software Engineering. A fixed code can be shared as a guideline for instructors of courses covering the foundational and advanced CS concepts with security concepts incorporated. Experimental studies will also be conducted to gather some quantitative and qualitative feedback from students to apply the tools while practicing the code in classrooms. More importantly, as Cheridari et al found false positives in 9 out of 21 systems using FindBugs in their study (Cheridari, 2018), verification studies will be conducted to identify false positives or negatives in the results from this analysis.

REFERENCES

Analysis of Software Artifacts, An Evaluation of FindBugs, Collection of unsound bug detectors for Java. (2009). Retrieved from http://www.cs.cmu.edu/~aldrich/courses/654/tools/Sandcastle-FindBugs-2009.pdf.

Barrientes, C., Yang, J., Sanchez, J., & Kim, Y. (2018). Source Code Analysis for Secure Programming Practices. *IEEE International Conference on Computational Science and Computational Intelligence*, 2018.

Bois, D. B., Demeyer, S., Verelst, J., and Temmerman, T. M. M. (2006). Does god class decomposition affect comprehensibility? In Kokol, P., editor, SE 2006 International Multi-Conference on Software Engineering, pages 346–355. IASTED.

Börstler J., Nordström M., KallinWestin L., Moström J-E., and Eliasson J., "Transitioning to OOP/Java – A Never Ending Story", In Reflections on the Teaching of Programming, M. Kölling, J. Bennedsen, and M. Caspersen, Eds. Lecture Notes in Computer Science, vol. LNCS 4821. Springer, 86-106.

Börstler, J., Christensen, H. B., Bennedsen, J., Nordström, M., Kallin Westin, L., Moström, J.-E., and Caspersen, M. E. Evaluating OO example programs for CS1. In ITiCSE '08: Proceedings of the 13th annual conference on Innovation and technology in computer science education, pages 47–52, New York, NY, USA. ACM.

Bugs Patterns. (2018). Retrieved in November 6, 2018. Retrieved from https://find-sec-bugs.github.io/bugs.htm.

CACM. 2002. Hello, world gets mixed greetings. Communications of the ACM 45, 2, 11–15.

CACM Forum. 2005. For programmers, objects are not the only tools. Communications of the ACM 48, 4, 11–12.

Caspersen, M. E. (2007). Educating Novices in The Skills of Programming. PhD thesis, University of Aarhus, Denmark.

Chahar, C., Chauhan, V. S., & Das, M. L. (2012). Code Analysis for Software and System Security Using Open Source Tools. *Information Security Journal: a Global Perspective, 21,* 6, 346-352.

Cheirdari, F. and Karabatis, G. (2018). Analyzing False Positive Source Code Vulnerabilities Using Static Analysis Tools. *2018 IEEE International Conference on Big Data (Big Data)*, Seattle, WA, USA, 2018 pp. 4782-4788. doi: 10.1109/BigData.2018.8622456.

Chess, B., & McGraw, G. (2004). Static analysis for security. *Ieee Security &amp; Privacy Magazine, 2,* 6, 76-79.

CWE Top 25 Most Dangerous Software Errors. (2019). Retrieved from http://cwe.mitre.org/top25/.

Dodani, M. H. 2003. Hello world! goodbye skills! Journal of Object Technology 2, 1, 23–28.

FindBugs™ - Find Bugs in Java Programs. (2018). Retrieved from http://findbugs.sourceforge.net.

Find Security Bugs™. Retrieved from https://find-sec-bugs.github.io/.

Hovemeyer, D. H. (2005). Simple and Effective Static Analysis to Find Bugs, Ph.D. Dissertation. University of Maryland.

IntelliJ IDEA: The Java IDE for Professional Developers by JetBrains. (n.d.). (2019). Retrieved from https://www.jetbrains.com/idea/download/#section=windows.

Gaddis, T. (2019). Starting out with Java: From control structures through objects (7th ed.). New York, NY: Pearson Education.

Grindstaff, C. (2004). Improve the Quality of Your Code. Received from https://www.ibm.com/developerworks/java/library/j-findbug1/.

Liang, Y. D. (2019). Introduction to Java Programming and Data Structures, Comprehensive Version, 11th Edition, New York, NY: Pearson Education.

Long, F., Mohindra, D., Seaccrd, R. C., Sutherland, D. F., & Svoboda, D. (2012). The CERT Oracle Secure Coding Standard for Java. Addison-Wesley.

Long, F., Mohindra, D., Seaccrd, R. C., Sutherland, D. F., & Svoboda, D. (2014). Java Coding Guidelines. Addison-Wesley.

Mahmood, R., & Mahmoud, Q. H. (2018). Evaluation of Static Analysis Tools for Finding Vulnerabilities in Java and C/C Source Code. Retrieved October 09, 2018. Retrieved from https://arxiv.org/abs/1805.09040v2.

Mamun, Md Abdullah & Khanam, Aklima & Grahn, Håkan & Feldt, Robert. (2010). Comparing Four Static Analysis Tools for Java Concurrency Bugs. Third Swedish Workshop on Multi-Core Computing (MCC-10).

Markettos, A. T., Watson, R. N. M., Moore, S. W., Sewell, P., & Neumann, P. G. (2019). Through Computer Architecture, Darkly, Communications of the ACM, Vol. 62 No. 6, Pages 25-27, 10.1145/332528.

Nordström M., and Börstler J. (2010). Heuristics for Designing Object-Oriented Examples for Novices. Submitted to ACM Transactions on Computing Education (TOCE).

Nordström M., and Börstler, J. (2011). Improving OO Example Programs. Submitted to IEEE Transactions on Education, VOL. 54.

Oskouei, Elmira Hassani and Kalıpsız, Oya (2018): Comparing Bug Finding Tools for Java Open Source Software. Journal Contribution.

Rutar, N., Almazan, C. B. and Foster, J. S. (2004). A comparison of bug finding tools for Java. 15th International Symposium on Software Reliability Engineering, Saint-Malo, Bretagne, 2004, pp. 245-256. doi: 10.1109/ISSRE.2004.1.

Sajaniemi, J. and Kuittinen, M. (2008). From procedures to objects: A research agenda for the psychology of object-oriented programming education. Human Technology, 4(1):75—91.

Saydjari, O. Sami. (2019). Engineering Trustworthy Systems: A Principled Approach to Cybersecurity. Communications of the ACM, Vol. 62 No. 6, Pages 63-69, 10.1145/3282487**.**

Seacord, R. C. (2013). Secure Coding in C and C++. Addison-Wesley.

Static Code Analysis Tools. (2019). Retrieved from https://security.web.cern.ch/security/recommendations/en/code_tools.shtml.

Source Code Security Analyzers. (2019). Retrieved in July 2019. Retrieved from https://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html**.**

Stamat, M. L. & Humphries, J. W. (2009). Training ≠ Educating Secure Software Engineering Back in the Classroom. WCCCE '09 May 1-2, 2009, Burnaby, BC, Canada. ACM 978-1-60558-415-7.

Westfall, R. (2001). 'hello, world' considered harmful. Communications of the ACM, 44(10):129–130.

Yang, J., Lodgher, A., & Lee, Y. (2018). Secure Modules for Undergraduate Software Engineering Courses. *2018 IEEE Frontiers in Education Conference (FIE)*, San Jose, CA, USA, doi: 10.1109/FIE.2018.8658433.

Yang, J. & Lodgher, A. (2019). Fundamental Defensive Programming Practices with Secure Coding Modules. *2019 International Conference on Security and Management*, Las Vegas, NV.

Yuan, Xiaohong; Yang, Li; Jones, Bilan; Yu, Huiming; & Chu, Bei-Tseng. (2016) "Secure Software Engineering Education: Knowledge Area, Curriculum and Resources," *Journal of Cybersecurity Education, Research and Practice*: Vol. 2016 : No. 1 , Article 3.

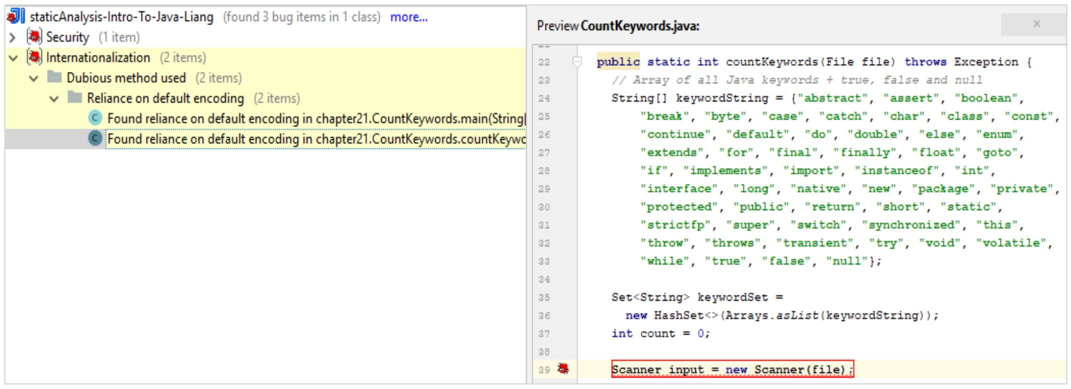
Yu, H., Jones, N., Bullock, G., & Yuan, X. (2011). Teaching secure software engineering: Writing secure code. *2011 7th Central and Eastern European Software Engineering Conference (CEE-SECR),* doi: 10.1109/CEE-SECR.2011.6188473.

APPENDIX A: Overview of Bug Categories Present in Text Source Code

Internationalization (I): Found in *CountKeywords.java* in Chapter 21 of Text 2

Bug Description: Found reliance on default encoding - new java.util.Scanner(InputStream) & java.util.Scanner(File) which will perform a byte to String (or String to byte) conversion and will assume that the default platform encoding is suitable. This will cause the application behavior to vary between platforms (e.g., Windows to Linux).

Recommended Solution**:** To resolve this issue, it is recommended to use an alternative API and specify a charset name or charset object explicitly.



```
staticAnalysis-Intro-To-Java-Liang (found 3 bug items in 1 class) more...
> Security (1 item)
v Internationalization (2 items)
  v Dubious method used (2 items)
    v Reliance on default encoding (2 items)
        Found reliance on default encoding in chapter21.CountKeywords.main(String|
        Found reliance on default encoding in chapter21.CountKeywords.countKeywc
```

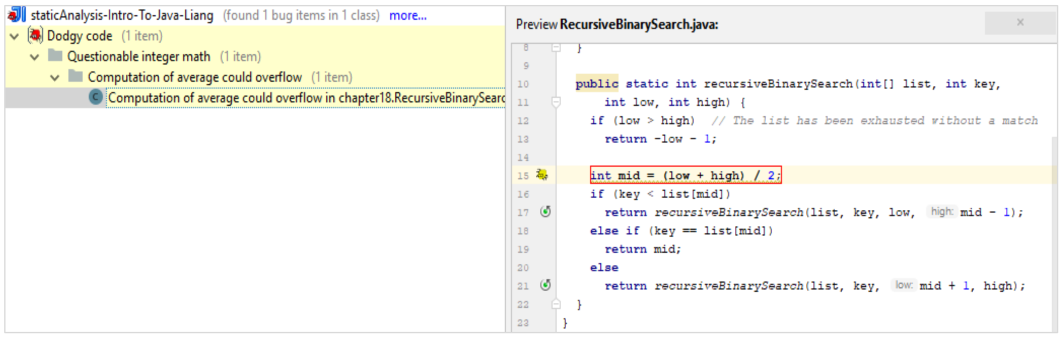
Preview CountKeywords.java:

```java
public static int countKeywords(File file) throws Exception {
  // Array of all Java keywords + true, false and null
  String[] keywordString = {"abstract", "assert", "boolean",
      "break", "byte", "case", "catch", "char", "class", "const",
      "continue", "default", "do", "double", "else", "enum",
      "extends", "for", "final", "finally", "float", "goto",
      "if", "implements", "import", "instanceof", "int",
      "interface", "long", "native", "new", "package", "private",
      "protected", "public", "return", "short", "static",
      "strictfp", "super", "switch", "synchronized", "this",
      "throw", "throws", "transient", "try", "void", "volatile",
      "while", "true", "false", "null"};

  Set<String> keywordSet =
    new HashSet<>(Arrays.asList(keywordString));
  int count = 0;

  Scanner input = new Scanner(file);
```

Dodgy Code (D): Found in *RecursiveBinarySearch.java* in Chapter 18 of Text 2

Bug Description: Computation of average could result in overflow - The code computes the average of two integers using either division or signed right shift, and then uses the result as the index of an array. If the values being averaged are very large, this can overflow (resulting in with a negative average).

Recommended Solution**:** If the result is intended to be non-negative, an unsigned right shift can be used instead. In other words, rather than using (low+high)/2, use (low+high) >>> 1.



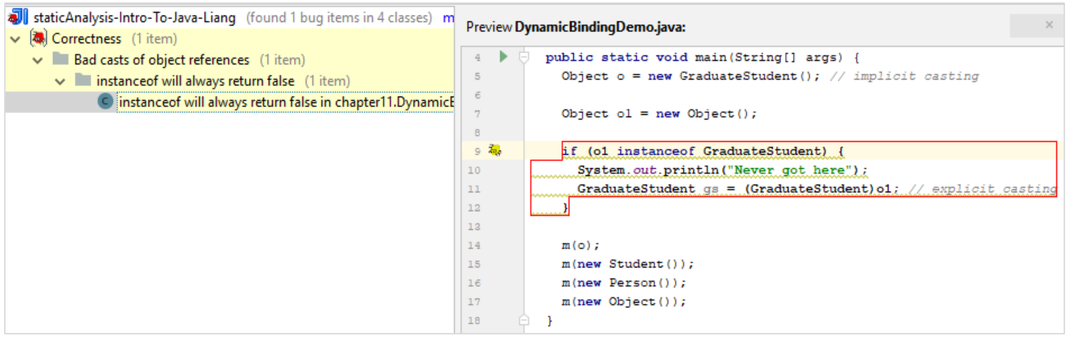
```
staticAnalysis-Intro-To-Java-Liang (found 1 bug items in 1 class) more...
v Dodgy code (1 item)
  v Questionable integer math (1 item)
    v Computation of average could overflow (1 item)
        Computation of average could overflow in chapter18.RecursiveBinarySearc
```

Preview RecursiveBinarySearch.java:

```java
  }

  public static int recursiveBinarySearch(int[] list, int key,
      int low, int high) {
    if (low > high)  // The list has been exhausted without a match
      return -low - 1;

    int mid = (low + high) / 2;
    if (key < list[mid])
      return recursiveBinarySearch(list, key, low, high: mid - 1);
    else if (key == list[mid])
      return mid;
    else
      return recursiveBinarySearch(list, key, low: mid + 1, high);
  }
}
```

Correctness (C): Found in *DynamicBindingDemo.java* in Chapter 11 of Text 2

Bug Description: This instanceof test will always return false. Although this is safe, make sure it isn't an indication of some misunderstanding or some other logic error.



```
staticAnalysis-Intro-To-Java-Liang  (found 1 bug items in 4 classes)
  Correctness  (1 item)
    Bad casts of object references  (1 item)
      instanceof will always return false  (1 item)
        instanceof will always return false in chapter11.DynamicE
```

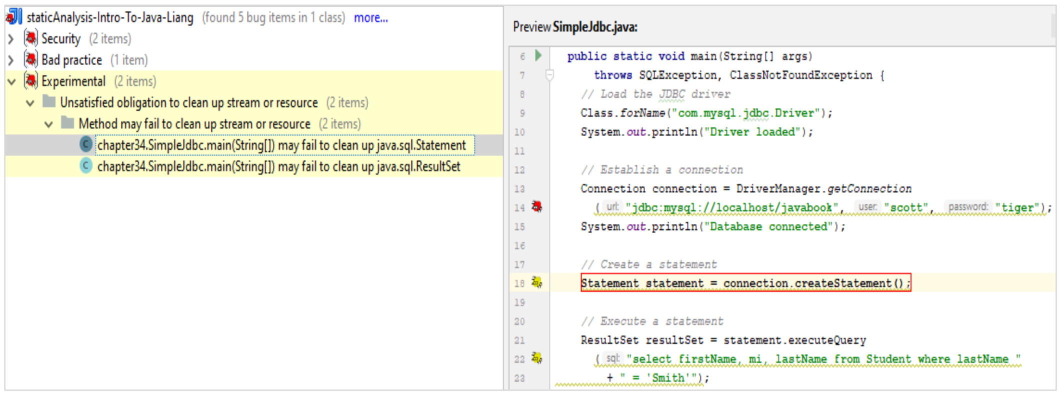
Preview DynamicBindingDemo.java:

```
public static void main(String[] args) {
  Object o = new GraduateStudent(); // implicit casting

  Object o1 = new Object();

  if (o1 instanceof GraduateStudent) {
    System.out.println("Never got here");
    GraduateStudent gs = (GraduateStudent)o1; // explicit casting
  }

  m(o);
  m(new Student());
  m(new Person());
  m(new Object());
}
```

Experimental (E): Found in *SimpleJdbc.java* in Chapter 34 of Text 2

Bug Description: main (String []) may fail to clean up java.sql.ResultSet / main (String []) may fail to clean up java.sql.Statement - This method may fail to clean up (close, dispose of) a stream, database object, or other resource requiring an explicit cleanup operation.

Recommended Solution: In order to make this method compliant, the usage of a try/finally block should be implemented to ensure that the stream or resource is cleaned up before the method returns.



```
staticAnalysis-Intro-To-Java-Liang  (found 5 bug items in 1 class)  more...
  Security  (2 items)
  Bad practice  (1 item)
  Experimental  (2 items)
    Unsatisfied obligation to clean up stream or resource  (2 items)
      Method may fail to clean up stream or resource  (2 items)
        chapter34.SimpleJdbc.main(String[]) may fail to clean up java.sql.Statement
        chapter34.SimpleJdbc.main(String[]) may fail to clean up java.sql.ResultSet
```

Preview SimpleJdbc.java:

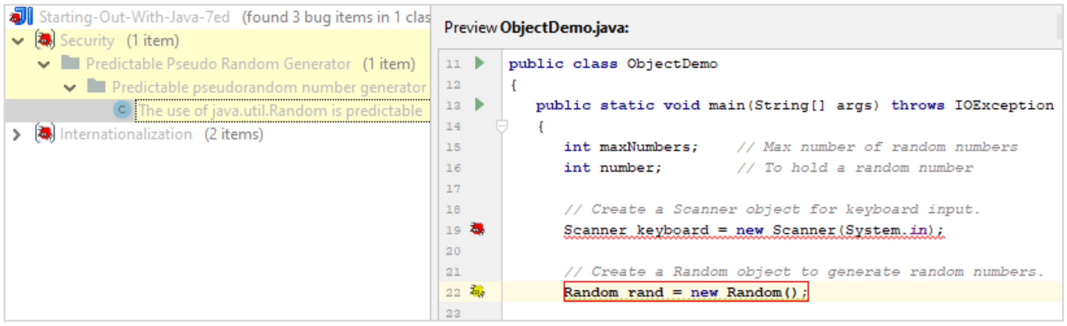
```
public static void main(String[] args)
    throws SQLException, ClassNotFoundException {
  // Load the JDBC driver
  Class.forName("com.mysql.jdbc.Driver");
  System.out.println("Driver loaded");

  // Establish a connection
  Connection connection = DriverManager.getConnection
    ( url: "jdbc:mysql://localhost/javabook", user: "scott", password: "tiger");
  System.out.println("Database connected");

  // Create a statement
  Statement statement = connection.createStatement();

  // Execute a statement
  ResultSet resultSet = statement.executeQuery
    ( sql: "select firstName, mi, lastName from Student where lastName "
      + " = 'Smith'");
```

Predictable Random (PR): Found in *ObjectDemo.java in* Chapter 6 of Text 1

Bug Description: The Random class is susceptible to returning predictable values which can lead to vulnerabilities if used in a critical security component.

For example, if this class was used to generate a CSRF token, then an attacker can easily isolate the value of the token by using a password reset and eventually lead to an account breach through a series of guesses by examining the URL of the change password form.

Recommended Solution: To prevent the predictability of values generated, the use of the *java.security.SecureRandom* class should be substituted for the *java.util.Random* class. The SecureRandom class is cryptographically stronger at generating random numbers that produces non-deterministic output. Therefore, any seed material passed to the SecureRandom object must be unpredictable and all output sequences intrinsically are cryptographically strong.



```
Starting-Out-With-Java-7ed   (found 3 bug items in 1 clas
  Security   (1 item)
    Predictable Pseudo Random Generator   (1 item)
      Predictable pseudorandom number generator
        The use of java.util.Random is predictable
  Internationalization   (2 items)
```

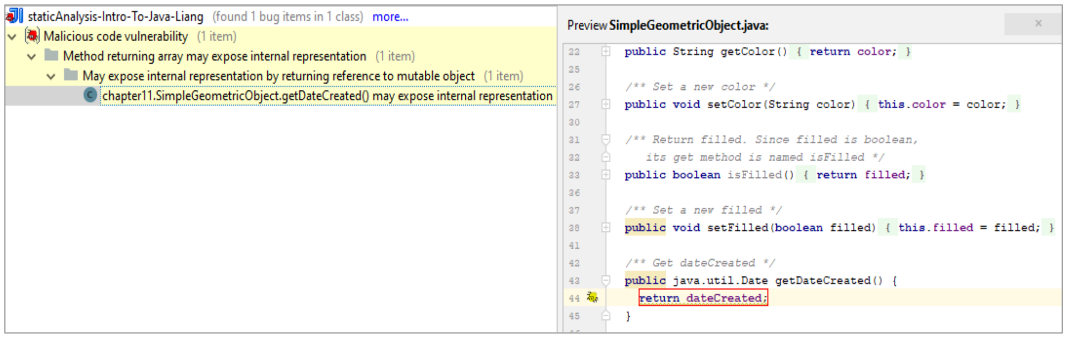
Preview ObjectDemo.java:

```
public class ObjectDemo
{
   public static void main(String[] args) throws IOException
   {
      int maxNumbers;     // Max number of random numbers
      int number;         // To hold a random number

      // Create a Scanner object for keyboard input.
      Scanner keyboard = new Scanner(System.in);

      // Create a Random object to generate random numbers.
      Random rand = new Random();
```

Malicious Code Vulnerability (M): Found in *SimpleGeometricObject.java in* Chapter 11 of Text 2

Bug Description: Returning a reference to a mutable object value stored in one of the object's fields exposes the internal representation of the object. If instances are accessed by untrusted code, and unchecked changes to the mutable object would compromise security or other important properties, you will need to do something different.

Recommended Solution: Instead of returning a reference to the object, you can return a new copy of the object which would prevent revealing internal information about the object.



```
staticAnalysis-Intro-To-Java-Liang   (found 1 bug items in 1 class)   more...
  Malicious code vulnerability   (1 item)
    Method returning array may expose internal representation   (1 item)
      May expose internal representation by returning reference to mutable object   (1 item)
        chapter11.SimpleGeometricObject.getDateCreated() may expose internal representation
```

Preview SimpleGeometricObject.java:

```
public String getColor() { return color; }

/** Set a new color */
public void setColor(String color) { this.color = color; }

/** Return filled. Since filled is boolean,
   its get method is named isFilled */
public boolean isFilled() { return filled; }

/** Set a new filled */
public void setFilled(boolean filled) { this.filled = filled; }

/** Get dateCreated */
public java.util.Date getDateCreated() {
  return dateCreated;
}
```