A comparative study of buffer overflow anomaly in Java using findbugs, PMD and checkstyle

R. Singh¹, Rajnish Dashora²* and Shiven Saiwal²
¹NIC, New Delhi, India.
²School of Computing Sciences and Engineering, VIT University, Vellore, India.

ABSTRACT
Static analysis means examining a program code and overcoming all possible errors that tend to occur at run-time. Static analysis tools are efficient in finding bugs and correction of defects that arise due to improper functioning of the code, before the actual execution of the code [10]. In recent times, technology has facilitated us with new tools that do deeper and more efficient code analysis and have a higher defect detection rate along with low false warning ratio. This paper aims to deal with buffer overrun anomalies occurring in many areas of source code in Java.

Introduction
A few decades back, there was no formal method of reviewing source code for software development. Later, an inspection procedure was introduced, termed as “manual reviewing”, accepted as the formalized reviewing methodology. It’s importance is that it provides information about software productivity and quality. For the inspection phase, Fagan has given an expression to calculate error detection efficiency i.e. [1]

Error detection efficiency = (error found by an inspection / total errors before inspection)*100.

The best way to debug a program is to use static analysis tools which maximize the program efficiency by quick analysis and correction of defects. The static analysis approach is meant to check the source code, compliance of the rules, usage of arguments, dynamic inconsistency, dead code, coding standards and also to be able to find bug patterns [2].

The major question that arises is whether testing should precede reviewing or vice versa. Thus, this paper is organized in various sections - Section II: Static Analysis Approach, Section III: Manual Reviewing Process, Section IV: Experimental Tools, Section V: Comparison of Static Analysis Tools, Section VI: Related works important to understand the literature behind static analysis of code using bug finding tools. Section VII: Improvements in existing tools, Section VIII: Conclusion over static code analysis tool for software development.

Static Analysis Approach
Major concern is shown over the Java programming language as it is considered an object oriented language but still is not able to deal with buffer overrun anomaly and tends to produce defects like “stack overflow” and “overridden methods”.

Static analysis tools that have been used in the research to debug the source code are: 1) Findbugs, which analyses Java byte code and produces different types of potential errors. Potential errors are classified into ranks i.e. a) scariest, b) scary, c) troubling, d) of concern. It reports errors like dead code elimination, deadlock detection, null pointer referencing and dereferencing 2) PMD (Programming Mistake Detector), which is a static rule set based Java code analyzer that deals with problems like a) empty try/catch block, b) Dead code, c) empty conditional and looping statements, d) complicated expression, e) wasteful string buffer usage, f) classes with high cyclomatic complexity measurements and g) duplicate code. 3)Checkstyle, which compiles Java code in such a manner so as to result in good programming practices which improve the code quality, readability, reusability and reduce the cost of development. It does not analyze the correctness of code but deals with style constraints that are needed for certain programs. It permits to check a) Naming conventions, b) limit function parameters, c) header files, d) spaces between characters etc [4]. Nowadays, these tools come together in a single package known as SONAR that is efficient in performing the mentioned tasks collectively.

Through this analysis, we can reduce the loss of data by avoiding wasteful buffer usage, infinite loops, and inappropriate conditional and looping statements. In this way we may overcome the buffer overrun anomaly [5].

Several lines of Java code were tested by the above mentioned tools and their behavior towards buffer overrun was noted. A comparative study is also shown between static and dynamic analysis techniques. Lastly, some suggestions are also made for the improvement of the existing tools which would aid in further maximizing program efficiency [4].
Buffer overrun: It is an anomaly in which a program, while writing data to buffer, overruns the buffer boundary and writes to the adjacent boundary and violates the memory safety. For example, Char A[8]: \ A having value null string; Unsigned short B; \ b having value 1212 [15].

<table>
<thead>
<tr>
<th>variable name</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>[null string]</td>
<td>1212</td>
</tr>
<tr>
<td>hex value</td>
<td>00 00 00 00</td>
<td>00 00 00 0C</td>
</tr>
</tbody>
</table>

Figure 1: Variable Buffer

Strcpy (A,"excessive"); // as excessive is of 9 character it overruns the buffer and takes space of buffer B and change its value to 25856

<table>
<thead>
<tr>
<th>variable name</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>e x c e s</td>
<td>25856</td>
</tr>
<tr>
<td>Hex</td>
<td>65 78 63 65</td>
<td>73 73 69 76</td>
</tr>
</tbody>
</table>

Figure 2: Buffer Overflow

Legacy code: It is a source code that is deprecated leading to vulnerabilities or threats [15].

B. Cyclomatic complexity: Cyclomatic complexity is computed using the control flow graph of the program where the nodes of the graph correspond to indivisible groups of commands of a program, and a directed edge connects two nodes. Cyclomatic complexity may also be applied to individual functions, modules, methods or classes within a program [15].

Assume M is the complexity of control flow graph (Directed Graph).

\[ M = E - N + 2P \]

Where,

\[ E = \text{the number of edges of the graph} \]
\[ N = \text{the number of nodes of the graph} \]
\[ P = \text{the number of connected components (exit nodes)} \]

Manual Reviewing

Manual reviewing is a static analysis process that is very time consuming and before going to examine the code, human auditors need to have sufficient knowledge about the errors they are going to analyze. Reviewing is not only the analysis of code but also includes documentation, requirements and designs produced by developers as parts of the reviewing process. This is because there can be hidden errors at each step of software development process [3].

Manual reviewing is divided into 4 steps: (1)Self Review, in which the developer debugs his/her own code and gets it rectified, (2)Walkthrough, which refers to the way to present work to audience, (3)Peer-review, which refers to reviewing done by colleagues and (4) Final auditing, which is the reviewing done by higher committee members.

Experimental tools

Three tools were used to analyze a static buffer overflow in Java code and results were analyzed.

Findbugs

It is a tool that analyzes byte code and produces bug patterns. It is comparatively faster than other tools as it works on byte code. It often finds real defects and also has low false defect rates. There are a maximum of 414 rules declared in Findbugs under the rule categories i.e. Correctness, Dodgy Code, Bad Practice, Multithread Correctness, Performance Malicious, Code Vulnerability, Security Experimental and Internalization.

Weakness: It is not aware of the sources and needs a compiled code. Sometimes it also issues warnings that do not correspond to real bugs in which cases the percentage of these warnings needs to be made small [11][12].

Example 1: Null Pointer Dereferencing

A pointer which is null on an exception path is dereferenced here. This will lead to NullPointerException when code is executed.

Note: Findbugs sometimes does not prune infeasible exception paths and hence may cause false warnings.

Example 2: Method to concatenate strings using + in a loop.

The method seems to be building a String using concatenation in a loop. Here, the String is converted to a StringBuffer and converted back to a String. This can lead to quadratic cost in terms of the number of iterations as the growing string is recopied every time.

Better performance can be obtained using a StringBuffer explicitly.

```java
// Bad Practice
String a = " ";
for (int i = 0; i < field.length; ++i) {
    a = a + field[i];
}

// Better Practice
StringBuffer a = new StringBuffer();
for (int i = 0; i < field.length; ++i) {
    a.append(field[i]);
}
String b = a.toString();
Display (b);
```

Example 3: Hardcode Constant Database Password

```java
// Hardcode Constant Database Password
```
This bug is reported because anyone with access to source code or compiled code can easily learn the password. To overcome this defect, we must provide variable password instead of a static one.

**Programming Mistake Detector (PMD)**

It is a tool that looks for potential problems, possible bugs and sub optimal code and highly complicated expressions in a Java source code. It occasionally finds real defects and bad practice bugs. It has maximum of 234 rules in its rule categories i.e. JSP, XSL, JAVA, EcmaScript and XML.

However, it is somewhat slow in detecting duplicate code and occasionally finds real bugs [6][13].

**Example 1:** Data overflow reported

```
package expections;
public class Dataexceptions {  
    static {
        int import[];  
    
    ...
    }
    public static void main(String[] args) {  
        System.out.println("Dataflow anomaly");
    }
}
```

**Figure 6:** DD anomaly

**Note:** Variable buffer is overflowed due to for loop resulting in dataflow anomaly reported by PMD.

**Example 2:** Method declared final

```
package entry;
public class Data_entry {  
    public static void main(String[] args) {  
        System.out.println("DD anomaly");
    }
}
```

**Figure 7:** Declaration issue

It also processes other bugs like locally declared variable final, system println, for loops necessary to be enclosed braces, avoid printstack tree and use singleton.

**Checkstyle**

It scans the source code and looks for coding standards for example, the sun code convention and the JavaDoc. It has maximum of 132 rules in a rule categories i.e. Annotations, Block Checks, Class design, Coding, Duplicate Code, Headers, Imports, JavaDoc Comments, Metrics, Miscellaneous, Modifiers, Naming Conventions, Regexp, Size Violations, White Space.

**Weakness:** It is not able to find real bugs but only analyzes the coding standards produced by sun systems [7][14].

**Example 1:** Java Coding Standards

```
package expections;
public class Javaexpections {  
    static {
        int import[];  
    
    ...
    }
    public static void main(String[] args) {  
        System.out.println("Java Coding Standards");
    }
}
```

**Figure 8:** Conventions followed in java

Code above states the sun produced coding conventions.

**Comparison between Static Analysis tools**

Tabulated results of Static analyses of JAVA code using software testing tools for JAVA applications.

**Java analysis tool report**

<table>
<thead>
<tr>
<th>Tools</th>
<th>Analyzed Features</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Findbugs</td>
<td>Static analyses of byte code, false warnings, misunderstood API methods and variety mistakes.</td>
<td>37.5%</td>
</tr>
<tr>
<td>PMD</td>
<td>Analyses of source code, detects cyclomatic complexity and displays potential problems.</td>
<td>53.8%</td>
</tr>
<tr>
<td>Checkstyle</td>
<td>Static code analyses for software development, maintains the coding standard produced by SUN</td>
<td>91.3%</td>
</tr>
</tbody>
</table>

**Table 1:** Java Analysis tool report

**Note:** A buffer overflow code is analyzed by above tools and their performance is calculated by:-

Efficiency=Total defects (positives warnings)/Total lines of code * 100

**Comparison between Findbugs and PMD**

A more than 1500 LOC on UID management is analyzed and results are tabulated below:

<table>
<thead>
<tr>
<th>Types of Defects</th>
<th>Findbugs</th>
<th>PMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null Pointer Dereference</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Security Issues</td>
<td>27</td>
<td>98</td>
</tr>
<tr>
<td>Naming Conventions</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>Corrective Defects</td>
<td>3</td>
<td>58</td>
</tr>
<tr>
<td>Compared objects with ==</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Dead store to local variables</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

**Table 2:** Comparison between Findbugs and PMD

After regreous analysis following results are tabulated

<table>
<thead>
<tr>
<th>Defect</th>
<th>Example</th>
<th>Cause</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call to equals with null argument</td>
<td>Matching of variable with empty string</td>
<td>Empty match always returns false</td>
<td>Use == or != to match empty string</td>
</tr>
<tr>
<td>Possible null pointer dereference</td>
<td>Display statement i.e. out.Print();</td>
<td>A false warning</td>
<td>Unavoidable, it does not affect the performance</td>
</tr>
<tr>
<td>Non constant string passed to execute SQL Statement</td>
<td>Mysql database connection</td>
<td>It is more vulnerable to SQL injection attacks</td>
<td>Providing variable value to retrieve result</td>
</tr>
<tr>
<td>Dead store to local variable</td>
<td>Variable declared but not used</td>
<td>Not required declaration, a garbage dump</td>
<td>Remove from the source code the non required items</td>
</tr>
<tr>
<td>Method may fail to close the resource</td>
<td>Connection/Resource not closed</td>
<td>Poor performance and cause communicatio n problem with database</td>
<td>Connection/resourc e after use needs to be closed to clean up the stream or resources</td>
</tr>
</tbody>
</table>

**Table 3:** Defect analyzed by Findbugs and provided solutions
Defect Example Cause Solution

Local variables not declared final Variables whose values are fixed may be declared as final Poor performance and bad style of writing code Declare all local variables to be final

Data flow anomaly analysis Declaring variable to be null Performance not affected but not a good style of writing code Reduction usage of global declaration

Avoid using short names to variables Giving meaningful names Less understandability Names of variables should match its usage/working

Method may fail to close the resource Connection/Resources not closed Poor performance and causes communication problem with database Connection/resourses after use needs to be closed to clean up the stream or resources

Use equals() in comparing string object instead of == or != Comparing null value in a code Bad practice Use comparison method (equals) to compare string objects

<table>
<thead>
<tr>
<th>Defect Example</th>
<th>Cause</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null Pointer Dereferencing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Security Issues</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Naming Conventions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Corrective Defects</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compared objects with ==</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dead store to local variables</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Types of Defects</th>
<th>Findbugs</th>
<th>PMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null Pointer Dereference</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Security Issues</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Naming Conventions</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Corrective Defects</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>Compared objects with ==</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Dead store to local variables</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bug Categories</th>
<th>Scariest</th>
<th>Scary</th>
<th>Troubling</th>
<th>Of Concern</th>
<th>Na Bug Region</th>
</tr>
</thead>
</table>

Table 4: Defect analyzed by PMD and provided solutions

Re-analysed results after following above solutions

<table>
<thead>
<tr>
<th>Types of Defects</th>
<th>Findbugs</th>
<th>PMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>Null Dereferencing</td>
<td>Yes</td>
</tr>
<tr>
<td>Concurrency</td>
<td>Deadlocks</td>
<td>Yes *</td>
</tr>
<tr>
<td>Conditional Loop</td>
<td>Unreachable code</td>
<td>Yes *</td>
</tr>
<tr>
<td>String</td>
<td>Checking equality and inequality</td>
<td>Yes</td>
</tr>
<tr>
<td>Object Overriding</td>
<td>Equal hashcodes</td>
<td>Yes *</td>
</tr>
<tr>
<td>I/O Streams</td>
<td>Stream not closed</td>
<td>Yes</td>
</tr>
<tr>
<td>Unused duplicate statements</td>
<td>Unused local variables</td>
<td>Yes</td>
</tr>
<tr>
<td>Design</td>
<td>Static inner class</td>
<td>Yes</td>
</tr>
<tr>
<td>Unnecessary Statements</td>
<td>Unnecessary return statements</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 5: Results found out after Bug correction

Diagramatic Study of Bug Catagories corresponding to the above Stated Results

Figure 9a: Bug categories before applying bug correcting solutions

Figure 9b: Bug categories after applying bug correcting solutions

Table 6: Types of bugs each tool find

* For specific examples

Related works

Artho is a static and dynamic bug finding tool in multithread programs. It compares tools on small core programs extracted from different Java applications. Artho then proposes extension to various other tools like Jlint, Findbugs, PMD, Checkstyle to greatly improve their bug finding ability to check the multithreaded programs. The focus of this paper is on finding wider variety of bugs across several ways. Thus the absence of warning from the tool does not mean the absence of error. There are several other tools that produce more warnings/bugs than the tools used but ultimately we need to believe that still there is a wider area of open research in understanding the right tradeoffs to make in bug finding tools [16].

Proposed Improvements

Tools used to find bugs often reports false thus making detection of actual positives difficult. High false positive rate frustrates users and discourages them from using helpful tools. Examining the code changes and the state of the code before and after the change may allow matching previous code changes to warnings produced by a bug finding tool. There are several ways to match the code warnings that are functions invoked in the code or data flow may be used to link the flagged code to the code. Warnings that flag code similar to code snippets that have been changed in the past may be more likely to be true positives [8].

Findbugs

It is a highly efficient static analyses tool. But it is only confined to Java-based applications.

PMD

This tool is widely used in correcting and analysing static code. It is not only confined to buffer overflow detection but can also be efficient in analyzing dataflow anomaly, improves the correct style of writing code and also performs code cleanups.

Checkstyle

Its main aim is to maintain high coding standards due to which non-real defects are produced. The false positive rate needs to be reduced in this tool.

Conclusion And future works

The paper attempts to bring out a comparison on how the manual reviewing is replaced by static analysis tools. Experimental results displayed above show that usage of bug finding tools are highly efficient in finding real defects that help in software development industry.
All the tools used above are equally important as each one of them has its own definite independent task to perform i.e. Findbugs, that analyzes a byte code and produce useful bug patterns, PMD is best suited for improving buffer overrun anomaly, suggesting better symbols and perform code cleanups. Checkstyle is best suited for maintain coding standards produced by SUN Systems. All the three tools nowadays come together in a package called SONAR.

In order to understand an error report, users must develop a way to take the information in the report and relate it to the potential problems with the code. Moreover, to decide whether an error report is a false positive, the user has to realize something about the sources of imprecision in the analysis. Therefore, they can create their own ad-hoc, inconsistent procedures that neglect some sources of imprecision. This situation can be addressed by encoding a triaging procedure through a checklist that enumerates the steps required to triage a specific report [9].

Hence, future works include devolving inbuilt libraries for these tools that incorporate the features of automated reviewing which results in high performance, efficiency and code optimization.

References