Software Security-Static Buffer Overflow Analysis in Object Oriented Programming Environment- A Comparative Study

MANAS GAUR
Computer Science and Engineering
Ambedkar Institute of Technology (AIACTR)
DELHI, INDIA

RAMESH SINGH
Senior Scientist, System Software Division
National Informatics Centre
DELHI, INDIA

ABSTRACT

Measurement of efficacy and efficiency of software (code) is one of the most useful and left over exercise in software development life cycle. Testing software regarding its capability to withstand attack is a major concern in the ICT field. There are many threats like threat to information, byte code error, malfunctioning, injections etc. Many tools have been created to combat the problem but the work path is not defined. We survey the research work in this area with the key interest in Buffer Overflow anomaly, a threat to be considered very seriously. We lay our research findings on some live projects in object oriented environment, analyze the test result of static and dynamic tools and try to improve the result of our work through code (statement/branch) coverage analysis. We henceforth attempt an algorithm to provide a checklist of some hot spot area in the software code. We also design a taxonomy of the error generated during our testing and analysis and strengthen the research with conclusion that the buffer overflow occur due to negligence in the code within the realm of taxonomy.

Keywords : Buffer overflow taxonomy, tool performance , tool comparison ,hash code analysis, types of coverage and analysis

1. INTRODUCTION

The internet is constantly under attack as witnessed by recent Blaster and Slammer worms that infected more than 200,000 computers in few hours. Buffer Overflow attack on source code has a terrific relation to network security firewall. Though use of NAT router and firewall prevent hostile attack but when user download the code for use as snippet, malicious code gets attached, which look alike to normal code but are bugs and perform a DOS attack? The best appearance of these bugs are in legacy codes or use in deprecated files. These files contain link to deprecated libraries, use of these may harm the reliability of the software[1]. To review the buffer overflow, approaches have been developed to reduce the buffer anomaly. These approaches can be at compile time called static approach or at run time called dynamic approach. Static approaches are based on source code design and utility. Static testing of the source code eliminates buffer overflow and are used in open source software testing, but requires experience in data flow graph analysis. Many static tools have been developed and evaluated in past, but still have high false alarm rate. Tester generally sees a tool as a panacea to complete bug problem but such wholesome tools don't exist. Testing whether static or dynamic calls for regressive analysis of code through different views. We define both qualitative and quantitative analysis of the static and dynamic tools and try to cure the anomaly by using combination of all tools. We evaluate the output of testing result of one tool and try to reduce it using other complimentary tools within the realm of object oriented paradigm. We try to resolve the problem by using basic software engineering methodology of algorithm and cyclomatic complexity analysis. We lay our research on some live projects developed by our team in object oriented environment and open source application server. We henceforth define an algorithm to manually check the stability of the code and provide taxonomy of error that generates the buffer overflow susceptibility. Our work is based on four tools, two static, one dynamic and one for data flow analysis, our work is mainly consolidated on process to reduce false alarm rate, compare their efficiency and try to achieve above 90 percent coverage in the code so that there is minimal probability for bugs to reside.

1.1 Literature Survey

There were tremendous work done in past in this field and were greatly helpful in developing my research. Experience Using Static Analysis to Find Bugs by david Hovemeyer, William Pugh, John Penix.2008 This research develops the find bug tool and shows survey results of the tool but does not specify 3 subparts to divide findbugs and quantitative comparison between findbugs and Pmd and dynamic tools. Analysis Tool Evaluation-PMD by Allen Hsu, Somakala Jagnathan Carnegie Mellon University:- This paper present detailing about the PMD tool, its interface and rule set primarily qualitative analysis with its pros and cons. Making FindBugs More Powerful by Asheq Hamid 2011: This paper talk about the different categories of bug patterns and how buffer overflow analysis can be detected by analysis one of these patterns..In late 2000 Crispin Cowan published there paper buffer overflow :Attack and defences for the Vulnerability of the decade. They implicitly discuss several of our attack forms but leave out the integer overflow and data structures overflow. Software Unit Test Coverage and Adequacy by Zhu Hal :- The paper speaks about the coverage analysis as a panacea of many coding errors and bending of the program. Apart from mutation testing and unit cases it put forward a research on coverage analysis and branch testing which serves as a module in modern tools. Defining and Providing Coverage of Assertion Based Dynamic Verification JG Tong 2010:- This paper elaborates the relation between the coverage of the assertion-based specification and the specific coverage metrics representing the assertions. Simple Dynamic Assertion For Interactive Program Validation C Hulten 1984: Its speaks about the advantages of a simple, user-friendly system based on dynamic assertions for expressing constraints,

transactions, and transition constraints. The study is merely subjective and less of practical information about assertion at work. Fade to Grey: Tuning Static Program Analysis by Ralf Hucck and Michael Tapp 2011: The information discovers the need of tuning more than reframing the code. It shows the different dimensions in static code analysis and distributes the bug according to properties and severity. DynaMine :Finding Common Error patterns by Mining Software Revision Histories by Benjamin LivShits and Thoman Zimmermann 2006:- This paper is develops a tools DynaMine and is unique for my research. No closely related but still highlight the concepts of Measurement and Maintainence of Software bugs by concept of data Mining and bringing in pattern in error occurrence. Testing Static Analysis

2. EXPERIMENTAL SETUP

2.1. FindBugs

FindBugs is a smart tool used in detecting static intrusion points in the code. Findbugs is generally applied in static testing of java source programs and provides the data defining the priority of error, confidence factor, type of error and its effect on other part of the module. FindBugs also includes some more sophisticated analysis techniques devised to help effectively identify certain issues, such as Categorization of bugs by findbugs is:- Malicious code vulnerability:- code that can be altered by other code. Dodgy :- code that can lead to error. Bad Practice:- code that violates the recommended coding practice. Correctness:- code that might give different results than the developer intended. Internationalization:- code that can inhibit the use of international characters. Performance dereferencing of null pointers that require such techniques and occur with enough frequency.

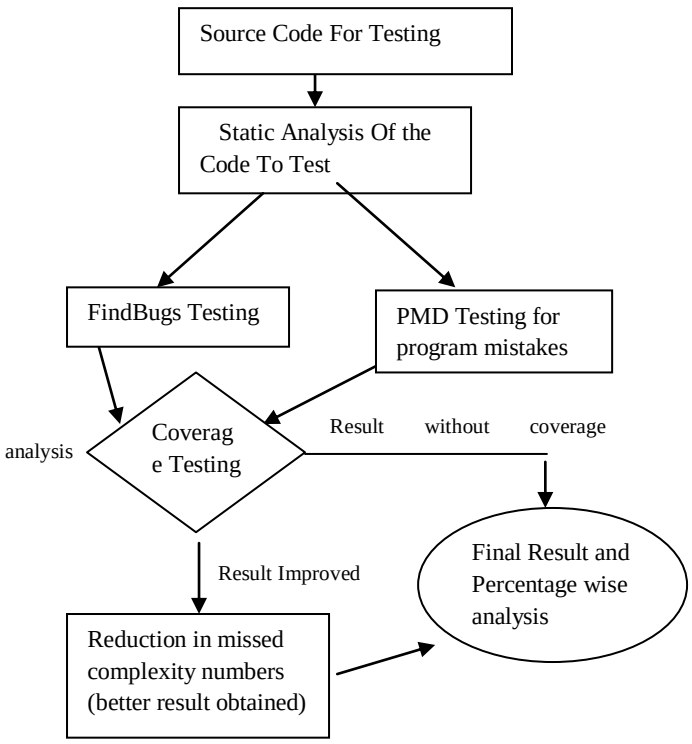

FIGURE 1. Flow Chart resembling the basic concept , motive and gradual advancement in study

code can be transformed to provide better performance. Security:- security problems in the code. Multithreaded correctness-multithreaded environment threats. Experimental- no closing statement of streams, database objects or others require closing statements.

2.2. PMD

Programming Mistake Detector is a static code analyzer for Java. Developers use PMD to make program comply with coding standards and deliver quality code.[2] Team leaders and Quality Assurance folks use it to change the nature of code reviews. PMD has the potential to transform a mechanical and syntax check oriented code review into a to dynamic peer-to-peer discussion.PMD works by scanning Java code and checks for violations in three major areas. Compliance with coding standards :-Naming conventions - class, method, parameter and variable names, Class and method length, Existence and formatting of comments and JavaDocs. Coding antipattern:-Empty try/catch/finally/switch blocks, unused local variables, parameters and private methods, Empty if/while statements. Overcomplicated expressions - unnecessary if statements, for loops that could be while loops, Classes with high Cyclomatic Complexity measurement. Cut and Paste Detector(CPD):- a tool that scans files and looks for suspect code replication. CPD can be parameterized by the minimum size of the code block. [9] **Important**:- PMD comes with 149 rules and 19 ruleset but it also provides the tester to develop his own set of rules to test the code and to bring homogeneity in code. Priority assignment in *PMD*:-PMD assigns violation priority from 1 to 5.VERY HIGH PRIORITY(indicated with red), HIGH PRIORITY(indicated with orange), MEDIUM PRIORITY(indicated with yellow), IGNORANT PRIORITY(indicated with green), NEGLIGIBLE(indicated with blue).

2.2.1 PMD Works

PMD relies on the concept of Abstract Syntax Tree**,** a finite, labelled tree where nodes represent the operators and the edges represent the operands of the operators. PMD creates the AST of the source file checked and executes each rule against that tree. The violations are collected and presented in a report. PMD executes the following steps when invoked from Eclipse. The PMD engine uses the Rule Sets as defined in the PMD preferences page to check the file(s) for violations. In the case of a directory or project (multiple source files) the plug-in executes the following steps for each file in the set.PMD uses JavaCC to obtain a Java language parser.PMD passes an InputStream of the source file to the parser.The parser returns a reference of an Abstract Syntax Tree back to the PMD plugin.PMD hands the AST off to the symbol table layer which builds scopes, finds declarations, and find usages. If any rules need data flow analysis, PMD hands the AST over to the DFA layer for building control flow graphs and data flow nodes. Each Rule in the RuleSet gets to traverse the AST and check for violations. The Report is generated based on a list of Rule Violations. These are displayed in the PMD Violations view or get logged in an XML, TXT, CSV or HTML report.[9]

2.3 JaCoCo

JaCoCo uses a set of different counters to calculate coverage metrics. This approach allows efficient on-the-fly instrumentation and analysis of applications even when no source code is available. In most cases the collected information can be mapped back to source code and visualized down to line level granularity. The smallest unit JaCoCo counts are single Java byte code instructions. **Instruction coverage** provides information about the amount of code that has been executed or missed. This metric is completely independent from source formatting and always available, even in absence of debug information in the class files. JaCoCo also calculates **branch coverage** for all if and switch statements. This metric counts the total number of such branches in a method and determines the number of executed or missed branches. Branch coverage is always available, even in absence of debug information in the class files. Note that exception handling is not considered as branches in the context of this counter definition. No coverage: No branches in the line has been executed (red diamond), Partial coverage: Only a part of the branches in the line have been executed (yellow diamond), Full coverage: All branches in the line have been executed (green diamond)

2.3.1 Complexity Description

JaCoCo also calculates **cyclomatic complexity** for each non-abstract method and summarizes complexity for classes, packages and groups. A cyclomatic complexity is the minimum number of paths that can, in (linear) combination, generate all possible paths through a method. [5] Thus the complexity value can serve as an indication for the number of unit test cases to fully cover a certain piece of software. Complexity figures can always be calculated, even in absence of debug information in the class files.

.JaCoCo calculates cyclomatic complexity of a method with the following equivalent equation based on the number of branches (B) and the number of decision points (D):V (G) = B - D + 1. Based on the coverage status of each branch JaCoCo also calculates covered and missed complexity for each method. [11] Missed complexity again is an indication for the number of test cases missing to fully cover a module. Note that as JaCoCo does not consider exception handling as branches try/catch blocks will also not increase complexity.

2.3.2 JaCoCo works

In the abstract sense, complexity beyond a certain point defeats the human mind's ability to perform accurate symbolic manipulations, and errors result. The same psychological factors that limit people's ability to do mental manipulations of more than the infamous "7 +/- 2" objects simultaneously apply to software. Structured programming techniques can push this barrier further away, but not eliminate it entirely. In the concrete sense, numerous studies and general industry experience have shown that the cyclomatic complexity measure correlates with errors in software modules. Other factors being equal, the more complex a module is, the more likely it is to contain errors. [8] Also, beyond a certain threshold of complexity, the likelihood that a module contains errors increases sharply. Many organizations limit the cyclomatic complexity of their software modules in an attempt to increase overall reliability. Methods Coverage:-Each non-abstract method contains at least one instruction. A method is considered as executed when at least one instruction has been executed. As JaCoCo works on byte code level also constructors and static initializers are counted as methods. Some of these methods may not have a direct correspondence in Java source code, like implicit and thus generated default constructors or initializers for constants. Classes Coverage:-A class is considered as executed when at least one of its methods has been executed. Note that JaCoCo considers constructors as well as static initializers as methods. As Java interface types may contain static initializers such interfaces are also considered as executable classes.

3. ANALYSIS

3.1 Bug Pattern

- Infinite recursive Loop: this is one of serious coding error not detected by compiler but can make your influential to attacker. When a function is called the caller and its address are put to stack and if the called function again makes a call to caller/itself its again puts it address on the stack , if there is no exit the stack overflows and the code is dead.

  Example:-

  Public static void main (String args []){Makeover () ;} Void makeover ()

  {Makeoverdone () ;} Void Makeoverdone (){ if(1) Makeover (); // not EXIT and the condition always point to true and the call goes on loop and stack overflows.}

- Hashcode and Equals

  Java.lang. super class files and default equals method which can be called as it is and does not erupt an error. But since it is a default one you cannot force it to behave according to you, for that you have to override it. This is the step where major errors originate because with every "EQUALS" there is associated hashcode, which is used for hashing (memory management in the operating system) the bytecode of a compiled java code.

  Example: Public static void main (String args []) {String str="hello"; String str1="HELLO" If(str.equals(str1)) Print(" they are equal");}

  The equals called is the default one and sometime the generated output is malignant. If the hashcode is not defined the interpreter will generate is default hashcode and the data and variables will be stored in some anonymous location in the memory. The correct implementation is [15]

  //same code above + Public int hashcode(){Assert false: "string is erroneous"; Return 434 ;}

- Null pointer dereferencing The null pointer analysis is a forward intra-procedural dataflow analysis performed on a control-flow graph representation of a Java method. The dataflow values are Java stack frames containing "slots" representing method parameters local variables, and stack operands. Each slot contains a single symbolic value indicating whether the value contained in the slot is definitely null, definitely not null, or possibly null. Figure

3:Eg: - p=null is null, p="String" is not null and p=f ()/a[i] is NCP (Null on Complex Path). [12].

CODE 1: public static void main (String args []) throws FileNotFoundException {File file = **new** File ("C:\\Documents and Settings\\MAK\\workspace\\BufferOverflowTest\\src\\com\\BOT1\\text1.txt");//FileReader, BufferReader Object defined(br)
Intervals = 0; **Double** [] nvals = **new double** [10]; **double** [] vals = **new double** [10];**try** {**while** (br.readLine ()!= **null**){String str = br.readLine();***vals[newvals]=Double.valueOf(str.trim()).doubleValue(); //bug detected***//rest of the code

- Return values ignored: - this means when the called function returns a value to the caller to pop up from stack the caller should check the value as it might create security breach.
- Inconsistent Synchronisation: - In today's programming arena everyone wants multiprogramming and multithreading, in java it is obtained by wait, sleep, join and the synchronized block. The code involving the above keywords should use conditional block like
- Asynchrony: - indicates if the buffer overflow is potentially obfuscated by an asynchronous program construct (no, threads, forked process, signal handler). [7] The functions that may be used to realize these constructs are often operating system specific (e.g. on Linux, thread functions; fork, wait, and exit; and signal). A code analysis tool may need detailed, embedded knowledge of these constructs and the O/S-specific functions in order to properly detect overflows that occur only under these special circumstances.
- Probable Out of bound Array Indexing [10]:- When array. Length library function is used in the initialization (part b) or in condition checking (part a) of a loop, the probability arises to use the value of the length of array as the array-index inside the loop. Thus we have safely used the term: probable' to give warning for out of bound array indexing.
  int[] array2 = new int[5]; int b7;
  for(int i = 0 ; i<=array2.length; i++){}part(a)
  int[] array9 = new int[5]; int b9;
  for(int i = array9.length ; i>=0; i--){} part(b)

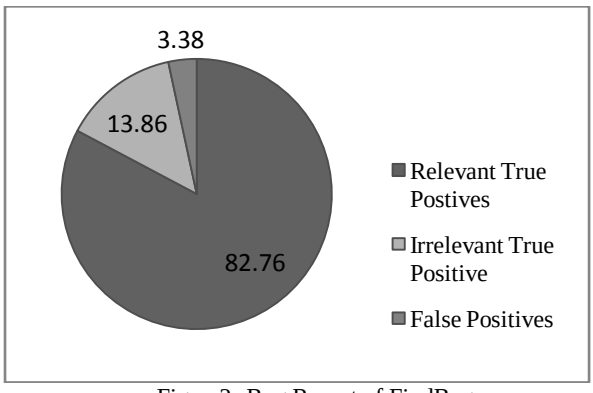

Figure2 Bug Report of FindBugs

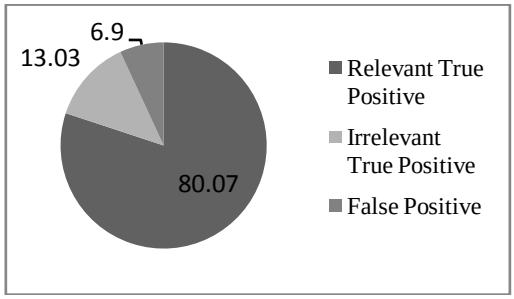

Figure 3 Bug Report of PMD

3.2 Static Analysis

3.2.1 Dimension of Static Analysis

Static program analysis is a term that was coined by the compiler community for a set of techniques to investigate program properties without actually executing the program. We define code optimization approaches which propel complexity analysis that is Flow-sensitive analysis:- takes into account the control flow of a program while a flow-insensitive analysis does not. E.g., taking loops and branching behaviour into account are characteristics of a flow-sensitive analysis while typical text searches are insensitive. Path-sensitive analysis:- considers only valid program paths. This means, more program semantics is considered like variable values conditionals that enable the analysis to distinguish between feasible and infeasible paths. Context-sensitive analysis:- takes the calling context of a function such as the states of input parameters and global variables into account. It is a special case of inter-procedural analysis, because it not only considers whole-program information, but the actual deferent program states in which a function is called.
Static analysis technique based on approximation is:- May-analysis considers over-approximations of program behaviour. May analysis, for example, might return as a result for a loop that indicates a septic variable is written after the loop, even if the analyzer itself cannot decide if this loop ever terminates. Must-analysis considers under-approximations of program behaviour. Must analysis will not return, for the loop example above, that the same variable is written, as it only considers those effects that are guaranteed to happen.

3.2 2 Bug Report of Code 1

Information about the bug
Bug: Dereference of the result of readLine () without nullcheck.in.com.BOT1.OverFlowFile.main(String[])the result of invoking readline() is dereferenced without checking to see if the result is null. If there are no more lines of text to read, readLine()will return null and dereferencing that will generate a null pointer exception. Confidence: Normal, Rank: Of Concern (15).PatternNP_DEREFERENCE_OF_READLINE_VALUE. Type: NP, Category: STYLE (Dodgy code)

3.2.3 Categorization of Bug Report in FindBugs and PMD

Categorization of bug report is done so as to consolidate the study of tool behaviour on some strong grounds. We categorize the bug report as false positive, relevant true positive and irrelevant true positive. False positive is defined as an output of tool regarding error, whose removal is not considerable. Relevant true positive are those error, whose

removal is considerable. Irrelevant true positive restricts itself to structure of the code [10,12]. FindBugs reports 82.76 percent relevant true positive(live projects put to test) 3.38 percent false positive and 13.86 percent irrelevant true positive. PMD reports 11,597 warnings(80.07 percent) relevant true positive, 13.03 percent irrelevant true positives and 6.9 percent false positive. See Figure 2,3,5

4) JaCoCo - Code Coverage Analysis

The results obtained by the use of the above tools(static analysis) didn't fight against the complexity issue. Bugs are not merely by programming error, it can be also due to presence of unreachable area in code. We used JaCoCo analysis and divide the result as Positive results and Negative Results. Positive Results, in an ironic manner tells about left over part of the code during testing by highlighting the statement, so that these statement can

Be improving so as to improve overall complexity of the code. Negative Results tells about the branches left out by this tool or we can say it defines the false behaviour of the tool. Still by combined approach of the tools we manage to achieve 91.7 percent complexity , much higher than earlier 71.3 percent. Note:-high percent unit define more coverage, low complexity of code and vice versa. Result shown in Figure 7 and 8.

5) Complexity Analysis

Explanation of Results:- Coverage is calculated via expression Cyclomatic Complexity:-

(No. Of line(Covered & missed)/ Total lines Covered)*100

Covered Complexity:- this entity tells the lines reached/reachable at compile/runtime. If its value equals to Total Lines Covered, means each branch is Reachable and code is free from threat. Missed Complexity:- defines the branches(LOC) not reachable during testing. If its value equals total LOC the code is vague and nearer to security theft.[8] Total Complexity:- derived by drawing graph nodes(index variables, conditional statement) and edge(flow of data) . it is calculated by:= E(Edges)-N(Nodes) +2 and traversing a graph by dijkastra or warshall algorithm to derive transitive closure , the 0 values in matrix formed shows missed branches. Fig.4

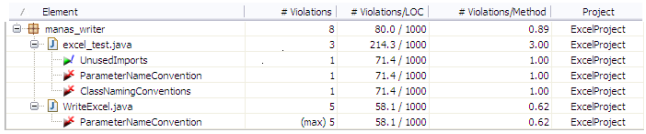

Figure 4 Violation/LOC data by PMD tool

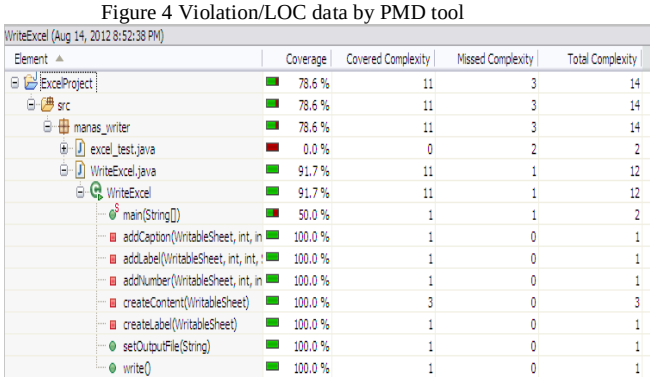

Figure 5 Statement/branch/method coverage of our project

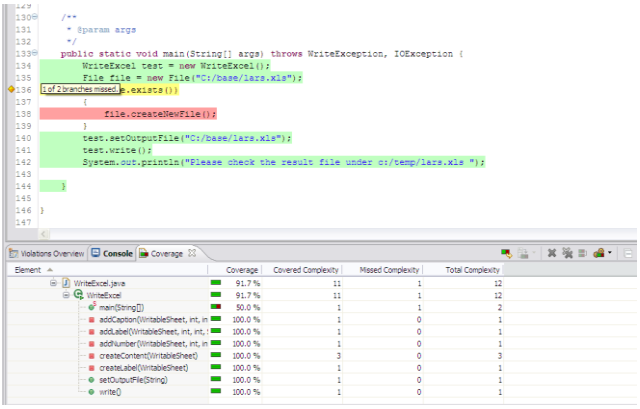

Figure 6 Code coverage improved after FindBugs and PMD Test

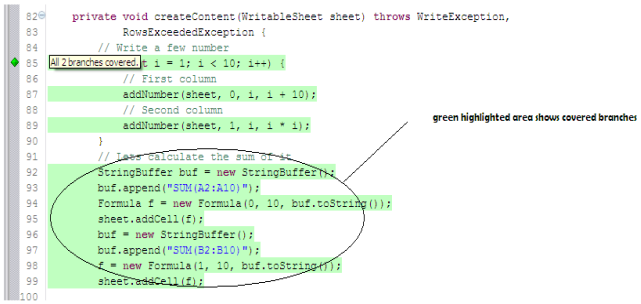

Figure 7 Result of JaCoCo covered branches in code

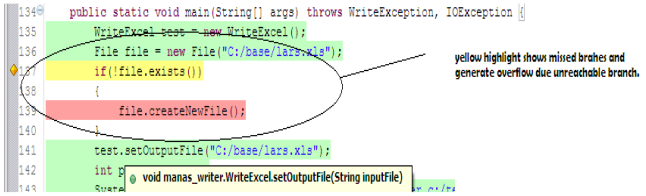

Figure 8 Result of JaCoCo missed branches

3.2.4 Exe File /.class file Analysis

When we compile code it in necessarily converted in .class file which is generally in low level, more esoteric for the operating system. A developer can never rectify errors like heap overflow, stack overflow(though can provide exception handling in code but its internal to code) neither integer overflow. A Overflow situation generally occurs when we use array implementation or size limit in array list like data structures, we have generated some experimental results on stack overflow, heap overflow or integeroverflow.String processing code are ,more vulnerable so we need to impose strong constraint on the size of the stack(for C/C++ user use Strncat instead of strcat, strncmp for strcmp etc.).

TABLE 1. EXAMPLE 1

Stack Address	Value
0000	0049
O004	0088

TABLE 2 EXAMPLE 2

Stack Address	Value
0000	0066
0004	1234

TABLE 3 EXAMPLE 3

Stack Address	Value
0000	9999
0004	9999

The values in **Example 1 represent** the location in the program where execution will resume when the current function completes its tasks and a value passed to a called function. In other words, a function was called during program execution. In order for the program to know where to resume once the called function returns control back to the calling function, the address of the next line of code to be executed is stored in the stack. In this case, the value of that address is 0049. The value of 10 in offset 0004 represents a value passed to the called function. In this example, there's no problem. The programmer assumed that the value passed to the called function would not exceed 4 bytes. The called function executes, and control is returned to the appropriate line of code in the calling function. [4] In **Example 2**, an attacker is taking advantage of buffer overflow vulnerability in the application. The attacker found that the programmer didn't add code to verify the size or data type of the data passed to the called function. By entering the value 12340088 (which exceeds the expected 4-byte limit), the attacker has succeeded in overwriting the return address stored in offset 0000 with the address of a malicious executable. When the called function completes its tasks, control will be handed over to the malicious program at address 0088 instead of the next line in the calling function at address 0049. Not all buffer overflow attacks are designed to cause the execution of malicious code. In **Example 3**, the attacker simply entered a series of 9's. In this case, the program will probably crash when it attempts to return control to the calling function.[9]
The data provided to the called function might come from a variety of sources. The key point to take from this example is that the input was not properly validated. (Note: For you purists out there, I know that certain values in the stack might not be stored most significant digit first. This is just easier for demonstration purposes.) Heap Overflow**:-**When a program retrieves a large amount of data for processing, a portion of memory known as the heap is allocated to handle the loaded data. In low-level languages like C and C++, the programmer is responsible for ensuring the proper amount of memory is set aside. If the loaded data is larger than the allocated heap memory, the system could crash.[14] Integer Overflow:- When adding two integers, the result occasionally exceeds the memory allocated for the result. When added together, the following two eight bit integers (10 + 5) fit nicely into an eight bit result space:

0000 1010 (10)	1100 0000 (208)
+0000 0101 (5)	1101 0000 (192)
0000 1111 (15)	0001 1001 0000 (400)

The sum of 400 won't fit in an 8 bit memory space. The integer overflow is not necessarily a good vehicle for outside attacks. But if your application doesn't return an exception error, your data integrity might be a little off. In this case, you might end up with a value of 144 (1001 0000) instead of 400 in your database or in your next processing step.
Redundant Comparison A finally block in Java is a region of code associated with a try statement which is guaranteed to be executed no matter how control leaves the try block. The Java source to bytecode compiler will emit code for a finally block either by duplicating it in the generated bytecode, or by emitting a jsr subroutine. [12]How to represent jsr subroutines in the control flow graph. This makes jsr and rets instructions used to call and return from jsr subroutines behave like goto instructions as far as the dataflow analysis is concerned. While this could theoretically result in an exponential increase in the size of the resulting control flow graph. The second issue is how to handle warnings for code inside finally blocks. For most kinds of warnings, including null pointer dereferences, the warning is valid. Redundant comparison warnings are only valid if the comparison is redundant for every expansion, and is always redundant for the same reason. We use the method source line number table to keep track of duplicated code, and only emit redundant comparison warnings if all redundant comparisons for a particular line are in agreement.

4. RESULTS

4.1. Algorithm For Buffer Overflow Bound Checking

FINDING ARRAY.
1.1 Mark each array declaration
1.2 For Each array marked above, check all subsequent reference.
INDEX VARIABLES:- legal ranges of an array of size n is 0<i<N
2.1 For each access that uses a variable as an index write legal range of it.
2.2 For each index marked in 2.1 underline all occurrences of that variable.
2.3 sort out any assignments, input or operation that may modify this index variable.
2.4 Mark with a any letter the finding in 2.3
LOOPS THAT MODIFY INDEX VARIABLE.
3.1 Find loops that modify variables used to index arrays.
3.2 For any index that occurs as part of the loop conditional, underline the loop limit.
For example: - for (i=0; i<max+1; i++) if I is the index variable underline i<max+1.
3.3 Write the legal range of the array index next to the loop limit as you did in 2.1. Mark a V if the loop limit could exceed the legal range of the array index.
3.4 Watch out for the loop that goes until i<=max as the largest valid index is max-1.
3.5 If the upper or lower loop limit is a variable, it must be declared, it must be checked just as indices are checked in step 2.

4.2. Comparative Analysis

Efficiency rate is Buffer overflow detection rate.

Calculated as (Total no. of warnings (positive) / Total Lines of Code) X 100

Table 4. EFFICIENY BASED RESULTS

TOOLS	ANALYSIS STRATEGY	EFFICIENCY RATE
FINDBUGS	Static analysis, flow sensitive	42.4%

	analysis, java byte codes	
PMD	Unused variables, symmetricity in code, error in exception handling, garbage collection	51.8%
JaCoCo(Code Coverage)	Date flow, Complexity analysis	17.98%

Table 5. COMPARISON BASED ON ERROR DETECTED

TYPES OF ERROR	FINDBUGS	PMD
Concurrency warning	72	4
Null Dereferencing	27	0
Null Assignment	6	68
Index Out Of Bounds	19	11

Table 6. RESULTS IMPROVED ABOUT TESTING

Note: TT: test time, W: percent Warning, R; Percent repaired

Prog.name	TT(min)	LOC	%W	%R
1. Excel	6	520	70.59	69.21
2. Online Exam	5	320	41.06	41..06
3. ITrust	10	3500	63.51	63.34
4. JBOSS**	29	90000	56.81	54.08

Table7. TYPES OF BUGS DETECTED BY PMD AND FINDBUGS

BUG CATEGORY	EXAMPLE	FINDBUGS	PMD
General	Null Dereference	YES	YES
Concurrency	Possible Deadlock	YES	YES
Array	Length may be zero	YES	NO
Conditional Loop	Unreachable code	YES	YES
String Processing	Check equality (==, =)	YES	YES
Object Overriding	HASCODE check	YES	YES
IO Stream	Streams closed or not	YES	NO
DESIGN	Static-inner classes	YES	NO
Unnecessary	Ignored return	NO	YES

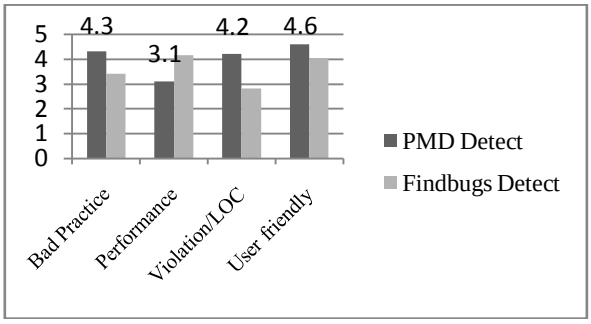

Figure 9. Graphical comparison on 5 point scale

4.2.1 Errors detected by tools-Proof by Code

```
Import java.io.*;
Public class Testing{Private byte[] b; Private int size;
Testing(){size=25; b=new byte[size];}
Public void test(){int z; // Variable unused detected by
//PMD
try{FileInputStream fish-new FileInputStream("XYZ");
x.read (b, 0, size); // Method value ignored detected by
FindBugs
 c.close();}catch(Exception e) // IO stream unclosed on
exception //caught detected by Findbugs
{System.out.println("help, I m caught");} for(int y=1;
y<=size;y++){If(Integer.toString(50)==Byte.toString(b[
i]))//Using == for //comparing string detected by
findbugs
System.out.println(b[i] + "");}}//end of test
method}//end of class
```

5. CONCLUSION AND FUTURE WORK

FindBugs and PMD analysis of the code really lowers the threat of the software (table 5). With the tremendous rise in object oriented programming the threats increase, at syntax level, bytecode and unused part of the code which consumes only memory. We put forward an algorithm that rectify the erroneous lines and perform bounded buffer checking. The coverage analysis of an untested code is merely 21.7 % but after static analysis we can achieve about 91 % coverage. Large unreachable lines in program bring in security breach like buffer overflow and diversion of the program from its intended use. We conclude , findbugs and PMD analysis at static level and assertions at dynamic level retards the fragile lines in program and reduces its time complexity.Findbugs well detected the bugs and it provides clear rationale of the bugs but it detects only syntactic errors but not sematic bugs, highly dependent on the coding standards, these tools is they don't understand what your software is trying to do and there sense of context is extremely limited which can lead to false positives being generated, for which the developer has to spend time to review it.PMD helps in finding programming bugs along with addressing of some complexity issues but it irrelevant true positive rate is very high, works on basis of some rules set, needs experience. Figure 9 personnel(AST tree study) and addresses bad practice problems well but lack performance features when compared to find bugs(in terms false positive rate).

5.1 Future Directions

5.1.1 Future Scope in Findbugs

The fact that Findbugs support only java, limits its uses to java based application. However findbugs support detection of various categories of bugs like:- Performance bugs in embedded applications ,Concurrency bugs in Complex multithreading. Priority Based analysis of Bugs(+200 bugs detection in findbugs).

5.1.2 Future Scope in PMD

In our research we focused on static buffer overflow ,coherent code generation and violation detection using PMD but it has far more scope in future in areas like:-Data Flow Analysis, Better Symbol Analysis and Code Cleanup-detecting and correcting sloppy codes.

5.1.3 Future Scope of JaCoCo

We used this tool for analyzing cyclomatic complexity of code, indentifying statement coverage and branch coverage, however this tool serves a panacea for various white box testing and finds it's utility in detecting missing requirement in software engineering and UML diagrams. It can also serves as tool for Feasibility analysis of software projects.

6. REFERNCES

[1] Cowan C., Wagle P., *et al,* " *Buffer Overflow :Attack and Defenses for the Vulnerability of the Decade* "ACM Transactions on Computer System, Vol 19, No.2,pages 217-251, 2001

[2] Hamid A., "*Making FindBugs More powerful*" , University of Texas at Arlington, USA in IEEE 2nd International Conference on software engineering and service science, 2011

[3] Huuck Ralf, Tap Micheal,"*Fade To Grey: Tuning Static Program Analysis*", Not published

[4] Hsu Allen, "*Analysis Tool Evaluation: PMD*", Not Published

[5] Jacobson Ivar "Object Oriented Software Engineering, information about Code Coverage Analysis", ACM computing Press, 2010

[6] Leek T. *et al "* Testing Static *Analysis Tools using Exploitable Buffer Overflows From open source code*", Proceedings of 12th ACM SIGSOFT international symposium on foundation of Software Engineering,2004

[7] Lippman,"*A Taxonomy of Buffer Overflows For Evaluating Static and Dynamic Software Testing Tools*" Proceedings of Workshop on Software Security Assurance Tools, Techniques, and Metrics, NIST Special Publication 500-265, Eds. P.E.Black, M.Kass and E.Fong, National Institute Of Standards and Technology, pp.44-51, 2005.

[8] McCabe,*Cyclomatic Complexity-Structural Testing*

[9] Ozlak T,"*Web Application Security –Buffer Overflows Are you really at risk*", Not published

[10] Pugh W., Hovemeyer D, "*Experiences Using Static Analysis to FindBugs*",IEEE Journal volume25 issue 5, pages 22-29, 2008

[11] Sommerville I., *"Software Engineering"*, 6th ed, Addison-Wesley, 2001.

[12] Spacco J.,"*Evaluating and Tuning a Static Analysis to Find Null Pointer Bugs*", Proceeding of the 6th ACM SIGPLAN –SIGSOFT workshop on Program analysis for software tools and engineering page 13-19 and in Newsletter ACM SIGSOFT software engineering notes volume 31, issue 1, 2006,

[13]Wilander J. ,KamKar M. ,"*A Comparison of Publicly Available tools for Dynamic Buffer Overflow Prevention*", In Network and Distributed System Security Symposium (NDSS) (February 2003), pp 149-162, 2002

[14] www.cwe.mitre.org/top25 (online) [21st June 2012]

[15]www.cs.toronto.edu/~sme/CSC/302/notes/19-static-analysis.pdf (online) [23rd June 2012]