A Comparative Study on Software Vulnerability Static Analysis Techniques and Tools

Peng Li
Institute of Electricity and Information Engineering,
Beijing Institute of Civil Engineering and Architecture,
Beijing, China
E-mail: bicealp@sina.com

Baojiang Cui
Institute of Computer,
Beijing University of Posts and Telecommunications,
Beijing, China
E-mail: cuibj@bupt.edu.cn

***Abstract*—Using static analysis tools can detect software vulnerabilities, which is important for improving the security of software. Static analysis technology has developed rapidly, but the comparison and evaluation of static analysis techniques and tools are not much. This paper focuses on software vulnerability static analysis techniques and tools. First we discuss the commonly-used static analysis techniques and tools, and compare these tools in a technical perspective, and then we analyze the characteristics of these tools through the experiment, finally, combining dynamic analysis, we propose an efficient software vulnerability detection method.**

Keyword-Static Analysis; Software Security; Vulnerability; Static Analysis Tools

I. INTRODUCTION

With the advent of information society and the increasing application of software, its security becomes an aspect of user concerning. In general, the software security problem is caused by software vulnerabilities, these vulnerabilities could be caused by flaws developer designs or deficiencies the programming language itself has, or may be back door developers set aside. According to statistics, the vast majority of hacking attacks are caused by software vulnerabilities. Network worms and Trojans can use the software flaw to compromise computers, resulting in information disclosure, the system running slowly, or even collapse.

So, how to find loopholes in software development and application, and make up for these vulnerabilities is an important work. There are many ways to improve software security, such as raising the safety awareness of programmers, using strict development model to design software, make software run in a safe environment and so on. But in recent years, static analysis technology as an efficient procedure for analyzing software vulnerabilities is attended by the people [1]. Static analysis method does not require running the program, but through direct code analysis to detect loopholes.

Static analysis is simple, fast and can be effective to find bugs in the code. Therefore, many software analysis tools are designed and achieved by static analysis technology. The first static analysis tool is FlexeLint in 1980s years; it used pattern matching method to identify gaps. In recent years, a number of complex and powerful static analysis tools began to appear.

Some researchers have evaluated security by looking at some software vulnerabilities. For example, J. Zhang and L. Williams analyzed the effectiveness of static analysis tools and concluded that static analysis tools are effective at finding loopholes in software [2]; A. Ozment and S. E. Schechter examined the code based on OpenBSD, measured the rate of vulnerability reports, and determined whether the security was improved by using the tools [3]. But few studies have analyzed and evaluated different static analysis tools, so it is quite negative to promote and apply static analysis tools.

Generally, no tool can find all the defects in the software. Each tool has different advantages in finding vulnerabilities. Ciera Jaspan and I-Chin Chen designed a coverage model of the defects and selected several tools that will compliment each other into the model [4], which can improve performance to some extent. Static analysis technology has many advantages, but some software vulnerabilities can not be found by static analysis. Therefore, in order to improve detection efficiency, we combined the static analysis and dynamic testing together.

For our study, we selected some well-known, publicly available static analysis tools. Our study focused on identifying static analysis functionality provided by the tools and surveying the underlying supporting technology. First we discuss commonly-used static analysis techniques and tools, and then from a technical point of view we compare these tools, after that through the experiment we analyze the characteristics of these tools, and finally, combining dynamic analysis, we propose an efficient method for software vulnerability detection.

II. STATIC ANALYSIS

In recent years, static analysis technology has rapidly developed from early lexical analysis to formal verification method and its detection capability is also improved a lot. With the continuous in-depth research of static analysis algorithms and static analysis model, software static analysis tools become more and more powerful. Static analysis techniques currently-used are lexical analysis, type inference, theorem proving, data flow analysis, model checking and symbolic execution. The following descriptions are their main features.

A. Lexical Analysis

Lexical analysis is based on grammar structure analysis, similar to the C compiler. This analysis method is to divide program into several small fragments, and then compares these fragments with loopholes libraries to determine

978-1-4244-6943-7/10/$26.00 ©2010 IEEE

whether there are loopholes. Without considering the program's syntax, semantics and the interaction between subroutines, false positive rate is high.

B. Type Inference

Type inference refers to inferring the type of variables and functions by the compiler, and judging whether its access of variables and functions is in accordance with type rules. Programming language type system includes a mechanism for defining data types and a set of type rules.

C. Data Flow Analysis

Data flow analysis refers to collecting semantic information from the program code, and with algebraic method to determine the definition and usage of the variables at compiling time. By using the control flow graph data flow analysis determines whether a value in the program is assigned to the possible vulnerability of the variables.

D. Rule Checking

Rule checking is to check security of the program using pre-established safety rules. There are some safety rules in program designing, such as in the 'root' privileges if the program calls 'exec' function it will bring on security implications.

E. Constraint Analysis

Constraint analysis is divided into constraint generation and constraint solving in program analysis process. Constraint generation is to establish variable types or analyze restraint system between different states using the rules of constraint generation; constraint solving is to solve the constraint system.

F. Patch Comparison

Patch comparison includes source code patch comparison and binary code patch comparison, and is mainly used to find "known" loopholes. After the software security vulnerabilities are found, the manufacturers usually release corresponding patches, so you can compare the code with patches to determine the location and causes of vulnerability.

G. Symbolic Execution

Symbolic execution is to represent the program's input by using symbol values rather than actual data, and produce algebraic expressions about the input symbols in the implementation process. By constraint solving method symbolic execution can detect possibility of errors.

H. Abstract Interpretation

Abstract interpretation is a formal description of program analysis. Because it only tracks program attributes users concern, the interpretation of semantic analysis is similar to its actual semantic meaning.

I. Theorem Proving

Theorem proving is based on semantic analysis of the program, and can solve problems of infinite state systems. Theorem proving first converts the program into logic formulas, then proves the program is a valid theorem by using axioms and rules.

J. Model Checking

Model checking process first constructs formal model for the program such as state machine or directed graph, then traverses and compares the model to verify whether the system meets pre-defined characteristics.

III. Static Analysis Tools

Here eight types of widely-used software vulnerabilities of static analysis tools from open source are chosen for analysis and comparison. First the main features of these tools are briefly described, then compared from a technical point.

A. ITS4

ITS4 [5] is a tool based on lexical analysis technique. It maintains a vulnerability database to read out the contents of the database at runtime and compare with the program codes. The database can be added, modified and deleted.

B. SPLINT

SPLINT (Secure Programming Lint) [6] is the expansion of LCLINT tool (for detecting buffer overflows and other security threats). It employs several lightweight static analyses. SPLINT need to use notes to perform cross-program analysis. SPLINT set up models for control flow and loop structure by using heuristic technology.

C. UNO

UNO [7] uses model checking to find loopholes in the code. UNO is named for the first character of three software defects: the use of Uninitialized variables, dereferencing Nil-pointers, and Out-of-bound array indexing.

D. Checkstyle

Checkstyle is the most useful tool to help programmers write standard Java coding. Programmers can integrate Checkstyle in development environment and use it to automatically check whether the Java codes are standard. Checkstyle is configurable and can almost support all the coding standards.

E. ESC / Java

ESC / Java (Extended Static Checker for Java) [8] is a static detection tool based on theorem proving, and can find run-time error in Java code. Programmers can build ESC / Java into the program verification environment, or install ESC / Java plug-in in the Eclipse.

F. FindBugs

FindBugs [9] is an open source static detection tools, which check the class or JAR files??. By comparing binary codes with the defect model set, FindBugs can detect latent problems. FindBugs is not to find loopholes through analyzing the form and structure of class files, but by using the visitor pattern. At present FindBugs contains about 50 error pattern detectors.

G. PMD

PMD is an open source, rule-based static detection tool. PMD scans Java source codes and finds some potential problems, such as wrong code, duplicate code, fussy code or code to be further optimized. PMD includes a default rule set. In addition, it allows users to develop new rules and use it.

We analyze and compare these tools from a technical point of view, as shown in Table 1.

TABLE1. COMPARISON FROM A TECHNICAL POINT OF VIEW

Tools	Technology	Program	Advantages	Disadvantages	Ability to detect
ITS4	Lexical analysis	C/C++	High efficiency, and without pre-treating the source code.	The analysis is imprecise, and false positive rate is high.	Inferior
SPLINT	Rule checking	C/C++	It can find a large scale program's faults using rules.	Due to restricted rules, it can only detect the given faults.	Powerful
UNO	Model checking	C	It can detect complex program, and test strictly.	State space explosion may happen.	Powerful
Checkstyle	Lexical analysis	Java	High efficiency, and without pre-treating the source code.	The analysis is imprecise, and false positive rate is high.	Inferior
ESC/Java	Theorem proving	Java	The strict reasoning is used to control the conduct of testing, and false positive rate is low.	It is different to process automatically, and need career man to analyze; The scalability is not high.	Powerful
FindBugs	Lexical analysis and Dataflow analysis	Java	High efficiency.	The analysis is not precise enough, and applying only to specific problem-solving.	Powerful
PMD	Lexical analysis	Java	High efficiency, and without pre-treating the source code.	The analysis is imprecise, and false positive rate is high.	Inferior

IV. EXPERIMENTS AND ANALYSIS

Here we mainly compare the functions of ITS4, SPLINT and UNO. We focus on several common loopholes: buffer overflow, arithmetic overflow; type mismatch, format string overflow, competitive conditions, null pointer references, uninitialized variable references, and use of storage after the release. We compile the test codes, scan them, and compare the three tools by analyzing the scanning output. There are two important criteria to evaluate the performance of these test tools: false positive and false negative. False positive means to report the vulnerability, but it does not exist in the program. False negative means that the program has loopholes, but the test tool does not report.

From the scanning results, we find that SPLINT has a relatively low false positive rate and more accurate test results, but there are some false negatives. ITS4 can find loopholes in the program, but there are some false positives. False positive rate and false negative rate of UNO's are intermediate. This result occurs primarily because of the different technology they use. SPLINT fails to detect the vulnerabilities associated with competitive conditions, because there are no defined notes related to competitive conditions. ITS4 can detect the vulnerabilities that are pre-defined, such as buffer overflow, arithmetic overflow, format string overflow, but can not detect unpre-defined loopholes that aren't pre-defined. UNO can find loopholes of limited types. From this result, it proves that no analytical tool is perfect, and can detect any loopholes in the program with a false positive rate of 0.

The experiment finds static analysis techniques have the following disadvantages: (1) static test often depends on the established database or set rules; but for the unknown loopholes, they cannot be checked with the established database or rules, which will produce a large number of false negatives; (2) static test does not really run the program, and cannot position the exact holes which will produce a large number of false positives. For example, Coverity has reported that the false positive rate of its commercial vulnerability detection tool-Coverity Prevent is about 20%. Some simple tools might have a higher false positives rate; (3) Because of the complex of program design language, some features such as pointer, dynamic memory, are difficult to be analyzed by the static test technique; (4) The current static test is mainly for detection of the source code, but nowadays most of commercial software is delivered to users by the executable files and its source code is not obtainable, therefore it cannot be effectively analyzed by using static analysis tools.

V. SOFTWARE VULNERABILITY DETECTION METHODS

In order to improve the detection efficiency of software vulnerabilities, one method is to employ a variety of static test technologies and tools. Another method is to combine static analysis with dynamic detection. Dynamic testing refers to running software in real circumstance, inputting test data, to determine whether there are loopholes in the program by observing its operating state and result. Because dynamic detection tests the program in real operation, instead of treating it abstractedly, so the result can be very accurate. The software flaw detection method in this paper is a combination of static analysis and dynamic detection; the detection process is shown in Fig.1.

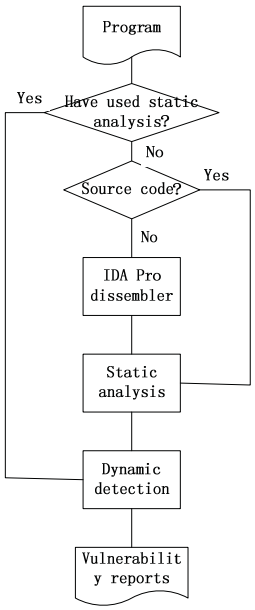

Figure1. Software vulnerability detection methods combining static analysis and dynamic detection

The main characteristic of this method is to test the program by static analysis procedures first, then through the dynamic detection to analyze suspicious results found in static analysis, verify the authenticity of vulnerability, thus reduce the false positives. Meanwhile, dynamic detection can help find the loopholes that are not found in the static analysis, thus reduce the false negatives. For the executable program code, IDA Pro dissembler tool can be employed to transfer the binary file into compiled file or intermediate code for easier analysis. From the Fig.1 we can see that a key part of the model is static analysis and dynamic testing modules.

A. Static Analysis Module

We use static analysis tools to analyze program and generate suspicious loopholes reports. Using the control flow analysis and data flow analysis, we detect the structure of the code, such as subroutines, control flow information, data flow information and the call, etc., and establish control flow graph (CFG) and the function call graph (FCG). In the FCG graph each node record needs to include the starting position in the function, local variables, function parameters and function return address.

B. Dynamic Detection Module

The functionality of dynamic detection module is to further test the code and reduce the false negative. On the other hand, it can verify the suspicious software vulnerabilities which are found by static analysis, and reduce false positive. Through dynamic testing, the accuracy of the testing process can improve. Dynamic test methods are commonly used, such as Fuzzing test, grammar test, defect injection and stain detection. The detection method is to analyze the input data, constructs special data, and run the target program in the debugger. Firstly sets a breakpoint and dynamically tracks the target program, and then detects the defective function calls and their parameter values, observes the implementation process and memory usage to determine whether there are loopholes in the code.

VI. CONCLUSIONS

It proves that most software security problems are caused by vulnerabilities of the software itself. Software security maintenance can be considered from both internal and external factors. Internal factors mean the loopholes in the software and problems of software development methods. External factors mainly refer to the insecure software operating environment. To solve software security issues the first step is to build robust software by improving its development and adopting project-based approach. Then use software vulnerability analysis tools to detect and take measures. The static analysis tools are not universal. In view of it, one method is to use a variety of static analysis tools and compensate for deficiency of each tool. Another method is to combine static testing with dynamic testing (such as Fuzzing---a commonly-used dynamic analysis tool).

ACKNOWLEDGMENT

The project is fund by Beijing Education Innovation Team, and the project number: PHR201007128.

REFERENCES

[1] Xia Yiming, “Security Vulnerability Detection Study Based on Static Analysis,” Computer Science, 2006, 33(10), pp. 279-283, Symposium, 18-22 May 2008, pp. 143-157.

[2] J. Zheng, L. Williams, N Nagappan, W. Snipes, J. P. Hudepohl, and M. Vouk, “On the Value of Static Analysis for Fault Detection in Software,” IEEE Trans. On Software Engineering, v. 32, n. 4, Apr. 2006.

[3] A. Ozment and S. E. Schechter, “Milk or Wine: Does Software Security Improve with Age?,” 2006 Usenix Security Symposium, Vancouver, B.C., Canada, 31 July-4 Aug. 2006.

[4] J. Ciera, C. I-Chin, and S. Anoop. “Understanding the Value of Program Analysis Tools,” Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion, Montreal, Canada, 2007, pp. 963-970.

[5] Llion Yi-liang Hsiao and Chenin-wei Jen, “A new Hardware Design and FPGA Implementation for Internet Routing Towards IP over WDM and Terabit Routers,” IEEE International Symposium on Circuits and Systems, Geneva, Switzerland, 2000, pp. 28-31.

[6] D. Evans and D. Larochelle, “Improving Security Using Extensible Lightweight Static Analysis,” IEEE Software, 2002, 19(1), pp. 42-51.

[7] G. Holzmann, “Static Source Code Checking for User-defined Properties,” Pasadena, CA, USA, June 2002.

[8] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, “Extended Static Checking for Java,” In Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation, Berlin, Germany, June 2002, pp. 234-245.

[9] D. Hovemeyer and W. Pugh, “Finding Bugs is Easy,” In SIGPLAN Notices, December 2004, vol. 39, pp. 192-206.