Volume 5, Issue 7, July 2015 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Static Analysis Tools for Security: A Comparative Evaluation

Hanmeet Kaur Brar
Student, UIET, Panjab University, Chandigarh, India

Puneet Jai Kaur
Asstt. Professor, UIET, Panjab University, Chandigarh, India

Abstract— Static analysis tools check the software for potential vulnerabilities and hence software security is improved with their usage. They prevent a wide range of difficulties those one can face at a later stage, in case, the flaws are not detected at an early stage in Software Development Life Cycle. For static analysis, a number of static analysis tools are available nowadays, which include both commercial as well as open source tools. Their comparative analysis is required for the users so that they can make their choice according to their requirements. This paper differentiates 3 open source tools used for static analysis for security: Cppcheck, RATS, Flawfinder. The comparative evaluation is done on different parameters analyzed on executing demo codes with intentionally introduced vulnerabilities.

Keywords— Static analysis tools; Software security; Software Development Life Cycle; Static Analysis; Vulnerabilities; Evaluation

I. INTRODUCTION

Due to increasing software application in all fields of our life, software security has become critical. A low quality code can make the program functional but will not be secure. Software security problem arises from software vulnerabilities, the root of which is either flawed coding by the programmer [1] or limitations within the programming language itself. In either case, finding and removing these vulnerabilities is crucial in software development as failure to do so can lead to cataclysmic results [2]. So, investment in doing this is justified otherwise lot more damage might arise at later stages which sometimes can be irreversible at that time [3]. Approximately 90% of stated security related hacking attacks occur due to flawed coding [4].

Finding the loopholes in the development of software and rectifying them, is a crucial job and must be done. A number of ways exist for software security improvement which includes the raising of programmers' safety awareness, usage of firm model for development of design, providing safe environment for software to run etc. [5].

Vulnerabilities can be found either by examining the code manually or with the help of automated tools. Manually finding flaws is an extremely time consuming and tiresome job. In some cases, it might not be possible to do it [6]. Static analysis tools help us here. They help the auditor by examining and directing to the likely flaws, thus saving time and energy. Thus, they are a quick and effective way of finding flaws within a code. As a result, a lot of static analysis tools have been designed and brought into use. The importance of static analysis tools can be judged by the fact that a lot of software companies (like Microsoft [7]) have made static analysis an important part of their Software Development Lifecycle (SDLC). One thing that should be kept in mind is tools just find errors for the auditor. Manual operation of tool as well as manual checking of errors is still required, but still they are a valuable source if applied with skill [8].

Studies have been conducted to check the effectiveness of static analysis tools in finding flaws in a code as well as verifying whether security has improved by their use or not but very few focus on examining and gauging various static analysis tools. The tools need to be gauged in order to judge their efficiency and utility.

Most of the tools are able to detect several types of shortcomings but their abilities are not uniform across the range of errors detected by them. Sometimes, a tool is capable of detecting a category of flaws but misses some variants of that particular category [9]. Not all tool user manuals define which particular flaws the tool cannot detect nor do they specify other weaknesses of their tool.

In this paper we have tried to investigate the capabilities of static analysis tools based on grounds of time of execution for each application tested as well as the categories of vulnerabilities these tools can figure out.

The rest of paper is arranged as follows. In section 2, a brief overview of testing is given (with focus on static analysis). In section 3, the tools selected for this paper and the reasons behind their selection have been discussed. In section 4, brief description of the chosen tools has been presented. In section 5, the test results have been deliberated. Section 6 contains conclusion. Future work is in section 7.

II. TESTING

Testing is a process of value addition. It is done with the intention of finding errors in the program, if any. The aim is to find that the program does what it is supposed to do. It is a quality improvement process. It is a *verification & validation* activity [10].

© 2015, IJARCSSE All Rights Reserved

A. Static Testing vs. Dynamic Testing

In static testing, software is not actually executed. In this testing, primarily, the code is inspected to find any errors. It can be done manually as well as with help of automated tools. This is the *verification* part of testing.

In dynamic testing, the software is actually run and it is examined by putting in different input values and checking if the output from the system is as expected or not. It can also be manual or automated. This is the *validation* part of testing.

B. Static Code Analysis

Static code analysis is a software verification activity in which source code is scrutinized for quality and security [11]. It allows software developers and testers to detect and make out various types of flaws—e.g. divide by zero, out of bounds read/write, overflow of bufffer boundaries etc.—without actually executing the code. These flaws can then be removed by the programmer. Thus, static code analysis helps to generate a more efficient code.

Static code analysis exposes "hard" bugs before runtime which may be impossible to detect during rumtime e.g. memory leaks which do not affect the programs functioning but increases memory footprint. If the code is very long finding such errors manually is not feasible [12].

As a result, automatic static analysis have been included in security engineering procedures to detect security vulnerabilities early in development lifecycle [13]. The aim is to decrease the time and effort needed during code reviews. The pursuit to automate the code reviews began with simple program checkers [14], which were then followed by more sophisticated and efficient commercial tools for different languages. These tools detect the most usual flaws in a particular language and help the programmer in making their code stable, secure, dependable and efficient.

Advantages of static analysis:

- The program to be analyzed does not have to be complete.
- We can use the static code analysis very early in the life cycle. Thus, we receive early feedback on software quality.
- The flaws are discovered at an early stage reducing the rework cost. This further boosts the development productivity.
- Test cases are not required and "hard" bugs like memory leaks can be detected [15].
- Tool has access to the actual instructions the software will be executing i.e. it has full access to all of the software's possible behaviors. Thus, it does not need to guess or understand behavior [16].
- Disadvantages of static analysis:
- False positives are generally produced which have to be scrutinized. High false positives are reprted for tools like Flawfinder, RATS, ITS4 [17].
- Also the tester needs good coding knowledge to understand the results of static code analysis.
- It requires access to source code or at least binary code to perform a build.
- Proficiency running software builds are characteristically needed.
- Problems related to operational deployment environments will not be found [16].

III. SELECTION OF TOOLS

Tools for static analysis in the beginning just dealt with lexical analysis but with the passage of time, more and more complex tools with more and better functioning have been introduced. The tools today are easier to use as well as efficient.

The brief description of tools used for this research is presented in the next section. In this section, the reason behind their selection is discussed.

Selection Criteria

The tools selected for research purpose are: Cppcheck, Flawfinder, and RATS. The reason behind their selection is to present diverse approaches to solve the same question. Our work does not involve financial support or budget, thus the tools used are open source/free. These tool are the latest ones amongst the open source static analysis tools for security.

RATS and Flawfinder focus on finding security vulnerabilities while Cppcheck offers broader analysis possibilities than both RATS and Flawfinder. All the three have good support communities and the latest versions available are given in Table I.

TABLE I SELECTED TOOLS

Tool	Version	Time of Release
RATS	2.4	Dec'13
Cppcheck	1.69	May'15
Flawfinder	1.31	Aug'14

IV. TOOLS

The tools used for research are: Cppcheck, Flawfinder and RATS. We will discuss them in a little more detail here.

© 2015, IJARCSSE All Rights Reserved

A. RATS

RATS stands for rough auditing tool for security. It was originally developed by Secure Software [18]. The name itself makes it clear that the tool does a rough analysis of the source code. Thus, it may not find every error and may find things that are not errors. Manual auditing is still necessary but the tool aides the auditor well.

Languages supported: C, C++, Perl, PHP, Python (and soon Ruby).

B. Flawfinder

Flawfinder searches for known security vulnerabilities using a built-in database containing known problems. David A. Wheeler is the author. It states security problems sorted by risk level. Flawfinder works by using a built-in database of C/C++ functions with well-known problems. It is CWE compatible, officially [19].

Languages supported: C,C++

C. Cppcheck

Cppcheck is a static analysis tool for the C and C++ programming languages. Non-standard code can also be checked. The creator and lead developer is Daniel Marjamäki. It does not detect syntax errors in the code. Cppcheck primarily detects the types of bugs that the compilers normally do not detect. The goal is to detect only real errors in the code (i.e. it aims towards zero false positives) [20].

Languages supported: C, C++

V. RESULTS AND ANALYSIS

This section presents the comparative evaluation of three static analysis tools for security that we are considering in this paper. The comparison is done on the basis of time taken by each tool to execute an application and figure out the security related vulnerabilities. Another comparison of these tools is done on the basis of category of vulnerabilities a tool is able to find.

To evaluate we had to be on the same platform and use the same programming language for all the tools. So, the operating system chosen was Ubuntu 14.0.LTS and the programming language chosen was C++. The latest versions of tools were used which are: RATS 2.4, Cppcheck 1.69 and Flawfinder 1.31.

Five applications developed in C++ language were taken for the purpose. These applications were given as input to each tool one by one. The results were recorded and are presented in this section.

A. Execution Time

Table II shows the different readings of time taken by the tools to execute the applications. Three different types of time are recorded in the table. We have considered only real time for analysis and calculations.

Figure I graphically represent the time taken by each tool to execute tests on different applications. This is drawn taking into account only real time in seconds.

It was found that RATS is faster than Cppcheck by 95.05% in executing application 1, 96.06% in executing application 2, 96.35% in executing application 3, 90.68% in executing application 4 and 87.50% in executing application 5. Thus, RATS is faster than Cppcheck by 87% to 96% according to our observations.

Similarly, RATS is faster than Flawfinder by 87.50% in executing application 1, 90.74% in executing application 2, 89.36% in executing application 3, 89.47% in executing application 4 and 87.50% in executing application 5. Thus, RATS is also faster than Flawfinder by 87% to 91% according to our observations.

TABLE II EXECUTION TIME (IN SECONDS)

TOOL	CPPCHECK			FLAWFINDER			RATS		
	REAL	*USER*	*SYS*	*REAL*	*USER*	*SYS*	*REAL*	*USER*	*SYS*
APPLICATION 1	0.101	0.101	0.000	0.040	0.036	0.004	0.005	0.005	0.000
APPLICATION 2	0.127	0.123	0.004	0.054	0.050	0.004	0.005	0.005	0.000
APPLICATION 3	0.137	0.122	0.004	0.047	0.043	0.004	0.005	0.005	0.000
APPLICATION 4	0.043	0.043	0.000	0.038	0.033	0.004	0.004	0.004	0.000
APPLICATION 5	0.040	0.031	0.008	0.040	0.035	0.004	0.005	0.004	0.000

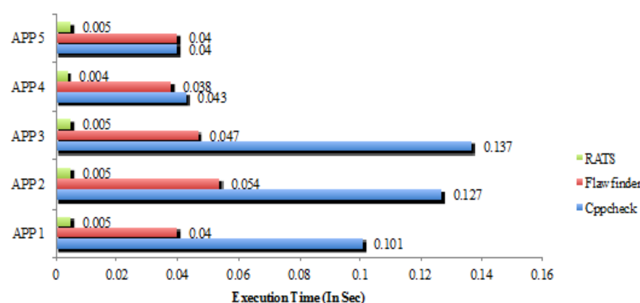

Fig. 1 Comparison of Real Execution Time (In Seconds)

B. Categories of Vulnerabilities

Table III contains the categories of vulnerabilities we have considered for our study and their description. We introduced these categories one by one in various applications and noticed the results to check if the tool under observation can detect that category or not.

Some of these categories have CWE entry associated with them. So, those entries are written along. Few are generalized categories which have wider horizon. So, those are not written with a specific CWE entry.

After the introduction of vulnerabilities in the applications and execution of those applications, the results found were recorded and are presented in Table IV. If the tool could capture that category, a tick (√) is marked corresponding to it otherwise a cross (×) is marked. Few points need explanation beyond a tick or cross which are explained after Table IV.

TABLE III CATEGORIES OF VULNERABILITIES INTRODUCED

Category	***Description***
Dead Code (CWE-561)	Code that is never executed in a program. Writing dead code in a program is considered a poor coding technique.
Divide by Zero (CWE-369)	This is an error that occurs in case of an unexpected input value or unexpected result of a calculation. It is also considered a weak coding technique. It may result in crash.
Uncontrolled Format String (CWE-134)	When the format strings are externally controlled in 'printf' in a program, this can cause buffer overflow or problems regarding representation of data. This error is introduced generally in case of code that needs to maintain log messages.
Classic Buffer Overflow (CWE-120)	Attempting to adjust data more than the buffer capacity allowed by the programmer or attempting to put data in such a way that it exceeds the boundaries of buffer. This coding practice represents ignorance of basic security protection.
Out-of-Bounds Read (CWE-125)	This occurs when the code tries to read data beyond the end of allowed buffer or before the buffer boundary begins. This also happens while incrementing or decrementing the pointer before the buffer boundaries.
Out-of-Bounds Write (CWE-787)	This occurs when the code tries to write data beyond the end of allowed buffer or before the buffer boundary begins. This also happens while incrementing or decrementing the pointer before the buffer boundaries.
Improper Input Validation (CWE-20)	This causes 'no validation' or 'incorrect validation' of input that affects the flow of program. This will cause the system to receive unintended input that can cause alteration in control flow or data flow.
Performance	These are suggestions that if implemented can improve the performance (mainly in the form of speed)
Portability	Information regarding the working of the code on different platforms.

TABLE IV DETECTION OF VULNERABILITIES

Category	***Cppcheck***	***Flawfinder***	***RATS***
Dead Code (CWE-561)	√	×	×
Divide by Zero (CWE-369)	√	×	×
Uncontrolled Format String (CWE-134)	×	√	√
Classic Buffer Overflow (CWE-120)	√	√*	√*
Out-of-Bounds Read (CWE-125)	√	×	×
Out-of-Bounds Write (CWE-787)	√	×	×
Improper Input Validation (CWE-20)	√**	√	√
Performance	√	—***	—***
Portability	√	—***	—***

* Flawfinder and RATS do not allow static character arrays to be declared regardless of their usage in the following code. This automatically requires dynamic array to be declared which in turn permanently avoids buffer

overflow. Flawfinder indicates the buffer overflow in form of 'risk levels'. If the usage of code causes buffer overflow, 'risk level' will be higher and if usage of code does not cause buffer overflow but has the potential to cause (if used in such manner), then the 'risk level' will be lower.

** Cppcheck does the input validation but skips few cases that Flawfinder and RATS can catch.

*** Flawfinder and RATS do not explicitly categorize the errors on the basis of portability and performance.

VI. CONCLUSIONS

The main motive of this research was to facilitate the software testers or the developers to make their choice of tool for static analysis. Depending upon the time of execution of each tool, we found that RATS was the faster than Flawfinder and Cppcheck. Cppcheck was slowest in execution of almost every application. Another comparison on the basis of category of vulnerabilities a tool can detect was done and it was found that Cppcheck was able to detect maximum categories of vulnerabilities introduced by us. Thus, every tool has its strong areas and depending upon the user requirements, the choice of tool can be made.

VII. FUTURE WORK

We have considered time of execution and categories of vulnerabilities as parameters for comparative evaluation. Other parameters such as precision, accuracy of the tools can also be calculated and compared by taking false positives and false negatives into account.

We have used applications developed in C++ for analysis. Other object-oriented languages may also be used which the tools support.

REFERENCES

[1] McGraw, Gary, and John Viega. "Building Secure Software." In RTO/NATO Real-Time Intrusion Detection Symp. 2002

[2] R. Jetley, B. Chelf. "Diagnosing Medical Device Software Defects Using Static Analysis." Published in MD&DI (2009).

[3] H. K. Brar, P. J. Kaur, "Differentiating Integration Testing and Unit Testing", 2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom), IEEE, pp. 796-798.

[4] Wang, J. A., Wang, H., Guo, M., & Xia, M., "Security metrics for software system," Proceedings of the 47th Annual Southeast Regional Conference, pp. 47, New York: ACM, 2009.

[5] P. Li, B. Cui. "A comparative study on software vulnerability static analysis techniques and tools." In Information Theory and Information Security (ICITIS), 2010 IEEE International Conference on, pp. 521-524. IEEE, 2010.

[6] M. Mantere, I. Uusitalo, and Juha Röning. "Comparison of static code analysis tools." In 2009 Third International Conference on Emerging Security Information, Systems and Technologies, pp. 15-22. IEEE, 2009.

[7] M. Howard and S. Lipner, "The Security Development Lifecycle: SDL: A process for developing demonstrably more secure software," Microsoft Press, 2006, ISBN-13: 978-0735622142.

[8] B. Chess and J. West, "Secure programming with static analysis," Addison-Wesley, 2007, ISBN-13: 978-0321424778.

[9] "On analyzing static analysis tools", National security Agency Center for Assured Software, July 26, 2011, pp. 1-13.

[10] Naresh Chauhan, "Software testing principles and Practices," Oxford, 2010; pp. 65-79

[11] A. German, "Software static code analysis lessons learned," Crosstalk, vol. 16, no. 11, 2003.

[12] Vincenzo Ciriello, Gabriella Carrozza and Stefano Rosati, "Practical experience and evaluation of continuous code static analysis with C++ test," In Proceedings of the 2013 International Workshop on Joining AcadeMiA and Industry Contributions to testin Automation, pp. 19-22., ACM New York, 2013.

[13] S. Lipner, "The trustworthy computing security development lifecycle." Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC), 2004.

[14] S.C. Johnson, "Lint, a C program checker" Computer Science Tech. report 65, Bell Laboratories, 1978.

[15] Patrik Hellström, "Tools for static code analysis: A survey," Department of Computer and Information Science, Linköping University, 2009.

[16] Dan Cornell, "Static analysis techniques for testing application security," OWASP San Antonio, 2008.

[17] Misha zitser, Richard Lippmann and Tim Leek, 'Testing static analysis tools using exploitable buffer overflows from open source code," ACM New York, 2004.

[18] RATS Info (Online). Available: https://code.google.com/p/rough-auditing-tool-for-security/

[19] Flawfinder Website (Online). Available: http://www.dwheeler.com/flawfinder/

[20] Cppcheck 1.69 Manual (Online). Available: http://cppcheck.sourceforge.net/manual.pdf

© 2015, IJARCSSE All Rights Reserved