Comparative Assessment of Static Analysis Tools for Software Vulnerability

Peter Miele, Mohammed Alquwaisem, Dae-Kyoo Kim*
Department of Computer Science and Engineering, Oakland University, Rochester, MI, 48309, USA.

* Corresponding author. Tel.: +1-248-370-2863; email: kim2@oakland.edu
Manuscript submitted January 10, 2018; accepted March 8, 2018.
doi: 10.17706/jcp.13.10.1136-1144

Abstract: Software security is a continuous and growing field within software development, maintenance, and operation. Vulnerabilities in software provide significant risk to the operation of software. Software tools have been developed over time to assist in identification and rectification of software vulnerabilities through static analysis of source code. Static analysis tools provide a software development team a means to rapidly review their project for the vulnerabilities that exist, but unknown to the team. In this paper, we present comparative assessment of three commonly used static analysis tools for software vulnerability using open source software for the purpose to aid software developers in choosing a suitable tool for their needs.

Key words: Software, static analysis, vulnerability.

1. Introduction

The risk of security vulnerability continues to increase as software development grows. Security vulnerability exposes not only the software to risk, but also users to malicious attackers. Tools that provide static analysis of security vulnerability can help one identify different types of vulnerabilities before the software is released to the customer. However, the developer should first understand the features of tools in terms of their strengths and weaknesses in order to be able to select a suitable tool for his needs.

Improper exposure of accessing or controlling the software allows the attacker to compromise the data and functions of the software. Other vulnerabilities also exist using external functions of modern computing such as attacking the timing of multi-threaded applications or violating access policies. To cope with these vulnerabilities, software developers should be aware of the nature of existing vulnerabilities and have access and knowledge of tools for identifying and correcting potential vulnerabilities.

This paper presents comparative assessment of three static analysis tools – ITS4 [1], Flawfinder [2], and RATS [3] for their ability to detect vulnerabilities in applications written the C programming language. The goal of this work to aid software developers in determining an appropriate security tool for their needs. These tools are chosen because they are capable of automating code review for security and open source. we use three applications – PuTTY [4], Wireshark [5], and Nmap [6] to assess the tools. This work is based on the work by McLean [7].

The remainder of the paper is structured as follows. Section 2 gives an overview of related work. Section 3 describes experiment setup, data collection, and analysis of the collected data. Section 4 concludes the paper with a discuss on the results of the static analysis and topics for future exploration.

2. Related Work

Several techniques exist to carry out static analysis on security. Nagy and Mancoridis [8] suggested an approach for determining security vulnerabilities by scanning the code that involves user input. Their work was motivated by that security attacks often involve sending faulty input data to the application. Their technique also enables developers to determine buffer overflow faults with high accuracy.

The work by Wagner *et al.* [9] propose a tool for detecting buffer overflow vulnerabilities. The tool involves program analysis procedures for identifying potential security holes. It also includes an algorithm for finding a solution for a vulnerability. The security knowledge used in the tool creates a generic class of relevant security-bugs that might occur in applications.

Tevisand and Hamilton [10] use a method of code scanning and checking to analyze source code. They propose a method for static security scanning of source code and identifying vulnerabilities. The method searches for known types of vulnerabilities and suggests corresponding defense techniques for each type. However, there is little validation (e.g., completeness) on identified vulnerabilities.

The work by Okun *et al.* [11] evaluates the effect of using static analysis tools on software security. Security is measured in terms of the number of reported vulnerabilities as opposed to other works where the number of weaknesses is used for measuring security.

3. Static Analysis on Security

In this section, we describe (i) testing applications and tools, (ii) the types of vulnerabilities concerned in this work, and (iii) the results of the testing.

3.1. Testing Applications

In this section, we describe (i) testing applications and tools, (ii) the types of vulnerabilities concerned in this work, and (iii) the results of the testing.

In this work, we use applications written in C language for testing. We chose C programming language due to its high vulnerability nature as C functions have little security mechanism. Because of that, many applications implemented in C inherently exhibit high risk of vulnerability. For testing applications, we use PuTTY 0.68, Wireshark 1.12.1, and Nmap 6.47 which are open source. Table 1 describes these applications. These applications are well known in the network software community and widely used with active user feedbacks (including those pertaining to vulnerability). Also, they have a number of stable open source versions available, which makes them suitable for the experiment in this work.

Table 1. Selected Open Source Test Applications

Application	Description
PuTTY 0.68 [4]	SSH and telnet client
Wireshark 1.12.1 [6]	Network protocol analyzer
Nmap 6.47 [3]	Network security scanner

We use Linux as the base operating system for experiment. Linux allows easy installation of applications with ample functionality and commands that help in creating test cases and viewing the results. Furthermore, Linux is open source and easily accessible.

3.2. Static Analysis Tools

There are many static analysis tools with different functionalities and algorithms to choose from. We use the criteria by Nagy and Mancoridis [8] for selecting tools for testing vulnerability. Table 2 shows the criteria. Per the criteria, we chose Flawfinder [1], RATS [5], and ITS4 [2]. Flawfinder examines C/C++

source code and reports possible security flaws sorted by risk level. The tool is efficient in finding and removing security problems. RATS is similar to Flawfinder in finding security vulnerabilities. ITS4 scans function calls in C/C++ source code to identify security vulnerabilities and generates statistical reports with informative description.

Table 2. Criteria for Selecting Static Analysis Tools [8]

Criteria	Filter	Rationale
Rules	Security	Security is the primary focus of this work.
Technology	Syntax	Most vulnerabilities are syntactical (e.g., insecure function calls).
Language(s)	C/C++	C is the chosen language in this work. C++ is a superset of C.
Releases	N/A	Not concerned.
Input	Source code	We use source code as input without compilation.
Configurability	N/A	Not concerned.
Extensibility	N/A	Not concerned.
Availability	Free, Open Source	We use open source for each access.
User Experience	Command Line, UI	Command line implementation requires the least dependencies on UI libraries.
Output	List, Text	We use grep in Linux for quick filtering and sorting of textual results.

Each tool provides its own form of output. RATS and ITS4 address the same types of vulnerabilities and group them for outputting. Flawfinder outputs each vulnerability as it finds it, which generates much more output than the other tools. However, the output contains significant redundant information for the same type of vulnerability. The below shows an example of redundant information from Flawfinder.

```
color_filters.c:520: [1] (buffer) getc:
  Check buffer boundaries if used in a loop.
color_filters.c:542: [1] (buffer) getc:
  Check buffer boundaries if used in a loop.
```

Unlike Flawfinder, ITS4 and RATS collapse such redundant information so that it appears only once when the first vulnerability is detected. The below shows an example of grouped redundant information from RATS.

```
color_filters.c:520: Medium: getc
color_filters.c:542: Medium: getc
  Check buffer boundaries if calling this
  function in a loop and make sure you are
  not in danger of writing past
  the allocated space.
```

Redundant descriptions in Flawfinder causes the size of output files to be almost 6.4 times larger than the average output size of the other tools, which might be of interests to users to consider in selecting a tool though it is not relevant to the tool's capabilities. Grouping redundant information also helps in identifying where a certain type of vulnerability exists.

Each tool provides as output the description of each vulnerability and in some cases how the vulnerability can be properly handled. Additionally, Flawfinder provides categorical information for identified vulnerabilities. For example, the below shows the buffer category from Flawfinder.

```
color_filters.c:542: [1]
(buffer) getc:
```

While RATS and ITS4 provide no such information directly, it can be derived from the description of vulnerabilities. The following shows an example of vulnerability information from ITS4.

```
color_filters.c:542:(Some risk) getc
```

An example of vulnerability information from RATS is shown below. The description specify that the *getc*() function has a potential vulnerability of buffer overflow, which helps to maintain the boundary of the butter. With this information, the additional categorical information provided by Flawfinder can be used as a basis for defining generic vulnerability types. We use the generic types in the comparison of the three tools in Subsection 3.3.

```
color_filters.c:542: Medium: getc
```

3.3. Testing Vulnerability

Vulnerabilities that affect software system are numerous in cause and result. Both syntax and semantics influence vulnerabilities. In this work, we focus on buffer overflow, format string, random number generation, shell, race condition vulnerabilities. Using the three tools, we conduct testing on PuTTY, Wireshark, and Nmap as follows. We first run the test on each software using each tool. The output of the test is stored in a spreadsheet that contains the list of vulnerabilities found by the three tools. The spreadsheet is then expanded to identify similar vulnerabilities across the tools and map them to a generic type (e.g., Buffer Overflow) for categorization. The results are used to define metrics by comparing the number of vulnerabilities using the built-in features in the spreadsheet.

For analysis, we compare the capability of the three tools in detecting and reporting vulnerabilities. Then, we analyze the raw results generated by each tool to determine its strengths and weaknesses. Fig. 1(a) shows the results. The figure shows that in terms of the absolute number of identified vulnerabilities, Flawfinder found the most vulnerabilities with 40% more than ITS4 and 200% more than RATS. The results demonstrate the effectiveness of the tools in discovering vulnerabilities with an indication of how well they recognize and ignore false positive results.

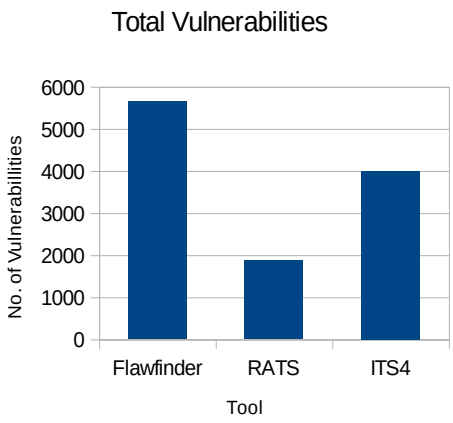
Buffer Overflow Vulnerability. Buffer manipulation is a source of vulnerability that can serve as a medium to high risk of security. A buffer is a portion of memory set aside for storing information during the execution of the software. Buffer usages include copying data from one buffer to another, joining two buffers together, storing user input, and writing program output (e.g., screen buffers). Buffer overflow is caused by incoming data larger than the size of the buffer in memory or an operation accessing memory outside of the allocated range. Buffer overflow may allow an attacker to write and execute a new instruction on the running software and gain full control over the system. The following shows an example of buffer overflow.

```
#include<stdio.h>
int main(int argc, char* argv[])
{
  char buf[16];
  gets(buf);
  printf("%s\n",buf);
  return 0;
}
```

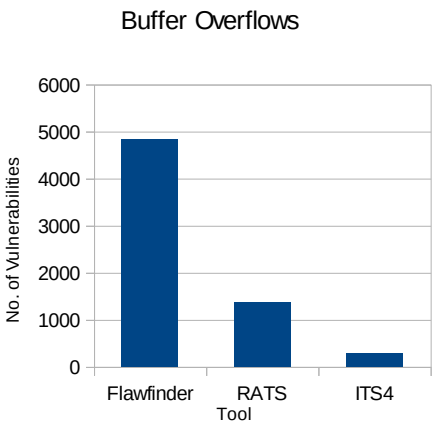
The code uses the *gets()* function to receive a sequence of characters from user input. The entered sequence is then stored in a buffer *buf* and echoed back to the user. However, the *gets()* function does not check the boundaries of the buffer, allowing the user to enter data of an arbitrary length. As a consequence, the function accepts any size of the input and stores it in the buffer. Any data beyond the allowed 16 characters is written past the boundaries of the buffer, which allows an attacker to overwrite arbitrary memory.

Buffer over-read, which is the counter vulnerability of buffer overflow, may allow an attacker to bypass the access control of the system and read data that is not supposed to be accessible. This vulnerability was identified in the Heartbleed vulnerability in OpenSSL, allowing an attacker to read secure memory that contains security data such as user account credentials and website security certificates [1].

All the three tools detect buffer overflow vulnerabilities. Flawfinder detected the most with a factor of 3.5 more than RATS, while ITS4 detected the least. The latter might be attributed to the fact that ITS4 does not recognize static sized buffers as vulnerabilities. Fig. 1(b) shows the effectiveness of the tools on buffer overflow. The graph shows the superiority of Flawfinder over the other tools. Flawfinder detected 4,945 vulnerabilities of buffer overflow, while RATS and ITS4 found only 1,183 and 158, respectively. ITS4 detected the least because it does not recognize fixed sized buffers (e.g., char[]) as vulnerabilities. Of the total number of detected vulnerabilities, buffer overflow vulnerabilities account for 56% which is heavily weighted by ITS4 having only 7% of its detected vulnerabilities as being of buffer overflow.



(a) Number of VULNERABILITIES



(b) Buffer Overflow Vulnerability

Fig. 1. The number of vulnerability and buffer overflow vulnerability.

Format String Vulnerability. Another type of vulnerability is concerned with format string which can be defined as trusting user input without validation. An attacker can take advantage of this vulnerability by submitting a hexadecimal input instead of the string format expected by the application. This might return the top of stack address to the attacker. The attacker can use the information to create a buffer overflow attack.

Fig. 2(a) shows the number of format string vulnerabilities detected by the tools. The graph shows that ITS4 detected the most with 3,519 vulnerabilities identified, while Flawfinder and RATS detected 327 and 391 vulnerabilities, respectively. The large discrepancy between ITS4 and the other tools is attributed to false negatives in the detection of ITS4. For example, ITS4 reports the *fprintf()* call in the following message from Wireshartk as a non-constant format string, and thus vulnerable.

```
color_filters.c:713:(Urgent) fprintf
Non-constant format strings can often be
attacked. Use a constant format string.
```

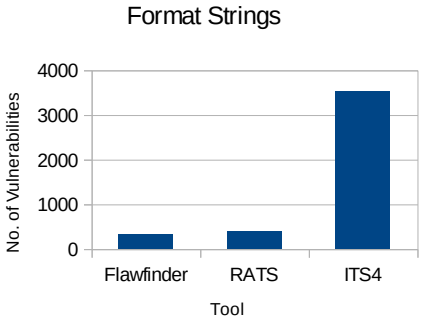
However, the line in question obviously uses a constant format string as shown below, which can be easily determined. Such a false positive makes the report less credential.

```
fprintf(f,"%s@%s@%s@[%d,%d,%d][%d,%d,%d]\n",...
```

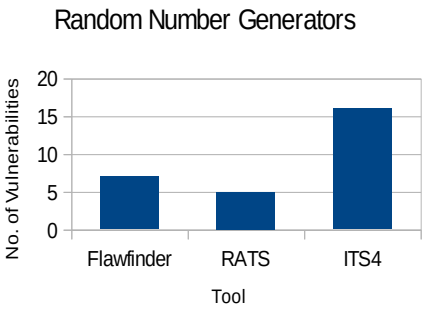
As shown in the above, ITS4 does not properly determine the type of format string and simply marks every call as a vulnerability. On the other hand, Flawfinder and RATS are capable of detecting **printf* calls whose source cannot be determined as constant. As such, each identification should be reviewed per the message generated by the tool.

Random Number Generation Vulnerability. Programming languages provide different ways of generating random numbers. Many of them are not truly random, but rather predictable based on an input seed. Random number generation is widely used in software development such as game development and cryptographic key generation. A common method for initializing a random function in a Unix environment is to use the system time as the input seed. However, the initializing time can be obtained through simple analysis of the random function and if the time is used for generating security keys, that can be a risk.

The **rand* and **random* functions in C use a deterministic method for generating the sequence of numbers. For random number generation, all the three tools identified the *srand()* function as vulnerable. ITS4 also identifies the *rand()* function as vulnerable. Similarly, Flawfinder also recognizes the *random()* function as vulnerable. There might be other methods of generating random numbers that the tools missed detecting. Fig. 2(b) shows the results. ITS4 stands out with 16 identifications, while Flawfinder and RATS identified 7 and 5 vulnerabilities.



(a) Format String Vulnerability



(b) Random Number Generation Vulnerability

Fig. 2. Format string and random number generation vulnerability.

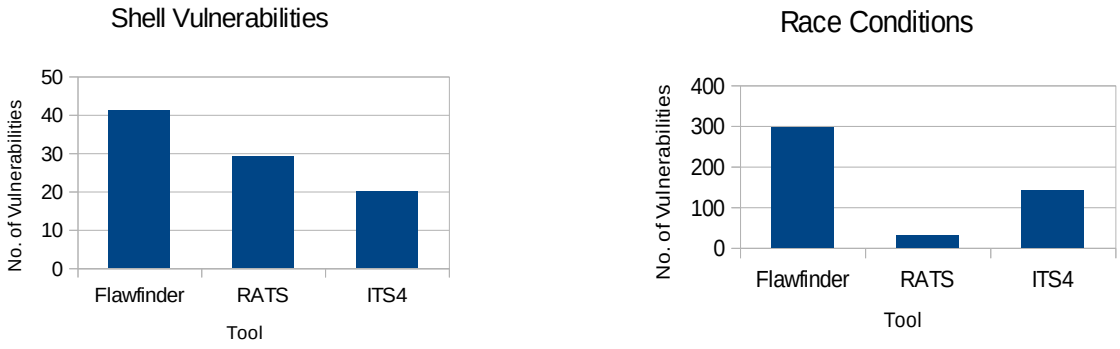
Shell Vulnerability. A shell function enables a single command execution for executing multiple commands/programs, which provides convenience to the user. However, it exposes environment variables or the location of the commands/programs in the system, which makes the system vulnerable. Such information can be used for creating a system shell for gaining authentication as the system administrator or it can be used for executing an arbitrary process for a malicious task.

Fig. 3(a) shows the results of shell vulnerability testing. The graph shows that Flawfinder is superior to the other tools, identifying nearly double the number of shell vulnerabilities as ITS4, while RATS identified roughly the average of the other two tools. RATS and ITS4 had different sets of vulnerabilities detected. RATS detected a variety of library loading functions and some direct shell execution functions as vulnerable.

ITS4 detected the exec* functions as vulnerable.

Race Condition Vulnerability. A race condition is a case in which two competing processes attempt to use the same resource at the same time. The resource can only be available to one process at a time, but there is no external control that enforces the limitation. A common example is Time of Check in which there is a time gap between when a check is performed and when the action based on the result of the check is taken. An example is modifying files. Suppose that a program functioning as a system administrator works with a user-controlled file after making a certain modification to the file as passing the security test. The user of the file can figure out when the program has modified the file and replace the file with a link to a protected system file, which enables the user to access the protected file.

Race condition is difficult to test as it requires precise timing. All the three tools detected race conditions in file usage as vulnerable. Fig. 3(b) shows the results of race condition assessment. Flawfinder stands out identifying 296 race conditions as vulnerable, while ITS4 and RATS identified only 141 and 30, respectively. More specifically, for the *fopen()* function, Flawfinder detected 64, while ITS4 detected only 40. This implies that ITS4 was more aggressive in eliminating potential false positives, and thus its report is more credential. For the *access()* function, Flawfinder identified 168, while ITS4 detected only 14. The vulnerability of the *access()* function can be a security flaw. If anything along the path between an *access()* call and the actual use of the file (e.g., moving the file), the race condition can be exploited. This can be prevented by setting up the correct permissions (e.g., using *setuid()*) and allowing only direct access to the file. Flawfinder provides such a resolution along with a general description of race condition. Flawfinder identifies all the instances of the *access()* function as vulnerable.



(a) Shell Vulnerability (b) Race Condition Vulnerability

Fig. 3. Shell and race condition vulnerability.

In terms of categories, ITS4 identified the most (which is not captured in the graph). ITS4 provides a list of the functions that involve a file in a race condition. The following shows an example which provides the locations where a race condition occurs.

```
packaging/macosx/ScriptExec/main.c:462:
(Very Risky) access
Potential race condition on: path
Points of concern are:
packaging/macosx/ScriptExec/main.c:462: access
tools/lemon/lemon.c:3096: access
wsutil/filesystem.c:669: access
epan/app_mem_usage.c:99: open
Manipulate file descriptors, not symbolic names, when possible.
```

3.4. Discussion

From the results of the experiments, RATS turns out to identify the least vulnerabilities overall. ITS4 outperformed the other tools on detecting **printf* vulnerabilities by a factor of 9 against RATS and 10.7 against Flawfinder. Flawfinder outperformed on identifying buffer overflow vulnerabilities with 85% of its discovered vulnerabilities falling into that category. From a statistical standpoint, RATS is the weakest among the three tools. However, it has the lowest rate of false positives. Given the analysis, Flawfinder, RATS, and ITS4 are recommended in the order of preference.

From the results, it is observed that each of the tools has different strengths and weaknesses. That is, combinatory use of the tools based on needs can give a synergistic performance. For example, among the three tools, Flawfinder identified the most vulnerabilities on buffer overflow including false positives. The results of Flawfinder can be fed into ITS4 and RATS to filter out false positives. In this way, the tools complement each other, providing more credential results. It should be noted that the identified vulnerabilities by the tools are meant to be potential rather than actual. Therefore, it is highly desirable to conduct manual assessment of the results to further filter out false positives. However, such manual analysis can be exhaustive, especially for a large-scale project. Hence, the development of tools that can automate the assessment is desired.

There also exist many programming languages in addition to the C programming language. Although many modern languages abstract away some of vulnerable behaviors of the C programming language, security flaws still exists in software written in the languages. There exist a variety of tools for other languages and comparisons of these tools should be carried out.

4. Conclusion

In this paper, we have presented comparative assessment of three static analysis tools for identifying software vulnerabilities using three open source applications for the purpose of aiding software developers to choose appropriate tools for their needs. The results show that Flawfinder, RATS, and ITS4 are recommended in the order of preference. Also, combinatory use of the tools is recommended to complement their weaknesses for a synergistic performance. We acknowledge that the selection of the applications may not have exposed the full functionality of the tools. We plan to expand this work by including a wide range of open source applications with more tools.

References

[1] It's the software stupid! (security Scanner) (ITS4). Retrieved November 10, 2017, from http://www.math.utah.edu/cgi-bin/man2html.cgi?/usr/local/man/man1/its4.1

[2] Flawfinder. Retrieved November 10, 2017, from http://www.dwheeler.com/flawfinder/

[3] Rough auditing tool for security (RATS). Retrieved November 10, 2017, from https://github.com/andrew-d/rough-auditing-tool-for-security

[4] PuTTY: A free Telnet/SSH client. Retrieved November 10, 2017, from http://www.chiark.greenend.org.uk/ ~sgtatham/putty/

[5] Wireshark: Go deep. Retrieved November 10, 2017, from https://www.wireshark.org/

[6] Nmap - Free security scanner for network exploration & security audits. Retrieved November 10, 2017, from http://nmap.org/

[7] McLean, R. (2012). Comparing static security analysis tools using open source software. *Proceedings of the 6th IEEE International Conference on Software Security and Reliability Companion*. Gaithersburg, MD.

[8] Nagy, C., & Mancoridis, S. (2009). Static security analysis based on input-related software faults.

Proceedings of the 13th European Conference on Software Maintenance and Reengineering. Kaiserslautern, Germany.

[9] Wagner, D., Foster, J., Brewer, E., & Aiken, A. (2000). A first step towards automated detection of buffer overrun vulnerabilities. *Proceedings of the Network and Distributed System Security Symposium* (pp. 3–17). San Diego, CA.

[10] Tevisand, J., & Hamilton, J. (2004). Methods for the prevention, detection and removal of software security vulnerabilities. *Proceedings of the 42nd Annual Southeast Regional Conference* (pp. 197-202). Huntsville, Alabama.

[11] Okun, V., Guthrie, W., Gaucher, R., & Black, P. (2007). Effect of static analysis tools on software security: preliminary investigation. *Proceedings of the ACM workshop on Quality of Protection* (pp. 1-5). Alexandria, VA.

Dae-Kyoo Kim is an associate professor of the Department of Computer Science and Engineering at Oakland University. He received the Ph.D in computer science from Colorado State University in 2004. During his Ph.D work, he worked as a technical specialist at NASA\Ames Research Center in 2003. He is a senior member of the IEEE Computer Society.