Comparing Static Security Analysis Tools for Network Connection Based Applications: A Replication Study

Xuying Swift
Montana State University
Bozeman, Montana
Email: xuyingswift@gmail.com

Clem Izurieta
Gianforte School of Computing
Montana State University
Bozeman, Montana
Email: clemente.izurieta@montana.edu

Abstract

Software vulnerabilities are an impediment to the reliable operation of a large number of computer applications, both proprietary and open source. Fortunately, there are many static analysis available to identify possible security issues. This paper presents a replication of a study conducted by 2nd Lt Ryan K. McLean where the author focused exclusively on comparing tools used to analyze network connection-based applications. The author reports the results of evaluating three subsets of open source code using security focused static analysis tools. The study found that these static analysis tools give developers a way to extensively test code for known flaws including buffer overruns, format string bugs, weak random number generation, and system access via shell. Tools such as RATS (Rough Auditing Tool for Security) and Flawfinder find these insecure design patterns. Flawfinder reports 2.25 times the vulnerabilities discovered by RATS, and Flawfinder's ability to provide valuable information to the user is better than that of RATS. In order to confirm and expand upon these findings, we replicate the study on three open source C software packages, PuTTY, Nmap and Wireshark. Our results are not fully in agreement with the original study by McLean when using a similar experimental setup. Static analysis tools used to discover security vulnerabilities suffer from false negatives and false positives. A false positive is a reported vulnerability in a program that is not really a security issue. A false negative is a vulnerability in code which is not detected by the tool. However, when we evaluate the metrics using the original setup we found the metrics offer no consideration on false negatives and false positives.

I. Introduction

Developers have a lot of challenges as they work on software systems that are constantly evolving[1]. They need to be creative and they also need to avoid security problems and bugs. Complex source code and the common errors made when programming make software error-prone. Modern programming languages give developers the ability to produce complex results when creating applications. Developers can make use of memory management, multi-process control, and other operating system functionality[2]. However, not using this type of functionality can lead to security vulnerabilities by end users and malicious attackers. One way to improve the quality of source code and to reduce bugs are code reviews. Typically, this requires human effort. This human aspect takes a lot of work and is error-prone. The work can also be hard and complex: source code analysis, quality assessment, and debugging. Static security analysis tools can find these issues early in the development cycle before they affect end-users. Static analysis tools give software developers a number of opportunities to quickly assess source code for common vulnerabilities. So, it is important that developers understand which tools can identify security flaws. This paper examines two static analysis tools, Flawfinder and RATS (Rough Auditing Tool For Security), and their ability to accurately detect security flaws in network connection-based applications in C code.

This paper shows the approach that we used to do the comparison among static security tools. First, we gave an introduction of static security analysis as a whole, including common security vulnerabilities and tools currently used to treat these vulnerabilities. Then, we discuss why open source software likely contains security flaws, and we list the software tested in this experiment. The experimental setup explains factors considered in choosing the targeted vulnerabilities, software language and applications, and the actual analysis tools. Next, we elaborate on the process used to test the selected tools against code. The fourth section lists the experiment results, compares analyzer output and performance, and provides analysis tool suggestions.

II. Background

A. Static Security Analysis

1) Common Vulnerabilities: Buffer overrun vulnerabilities are one of the most common security flaws[3]. A buffer overrun flaw typically happens when a programmer fails to do bounds checking when writing data into a fixed length buffer[3].

Buffers are memory that temporarily store data while shitting from one location to another. A buffer overflow happens when its data exceeds it storage. As a result, the program attempts to overwrite the data to the adjacent memory locations. Buffer overruns can affect all types of software and range from not allocating enough space for buffer to altered inputs. If the malicious code is executed, the program will behave unpredictably.

Buffer overflow allow attackers to overwrite application memory, which leads to execution path being changed. Private information might be exposed and files might be damaged because of that[3]. Attacker can intentionally alter user input and gain access to memory layout of a system, then they can inject their own executable code[3].

The C programming language has been in use since 1972, and it still is one of the important building blocks of many software[4]. C is behind of most of we do, powering the Linux Kernel, and a large number of popular projects, such as FFmpeg, curl, and ImageMagick. Nowadays, 10% of the new software code is written in Java and C[5].

C is also a language with the highest number of reported vulnerabilities. Since 2019, all the vulnerabilities reported in C is more than 50% of all reported open-source vulnerabilities[4]. C is a lightweight language and C provides little syntactic checking of bounds, and C functions tend to have a minimum number of error checking[4]. Because of this, when developers fail to implement proper logic, they often assume C has handled these issues. These assumptions often lead developers to fail at preventing array out of boundaries[4].

C is highly susceptible to buffer overflow attacks, since there are no overwriting protection build in. The following code is an example of a buffer overflow vulnerability[6]:

```c
#include <stdio.h>
#include <string.h>

int main(void)
{
 char buff[12];
    int password = 0;
    string name = "Sam Smith";

    printf("\n Please enter the password : \n");
    gets(buff);
    return 0;
}
```

In the above code, the $get()$ function does not check the array bounds and can even write string of length greater than the size of the buffer. Attackers can supply an input of length greater than what the buffer can hold that overwrites the memory integer 'password' and gains the root privileges.

Another common vulnerability in the programming language C is a format string bug. A format string is an ASCIIZ string contains text and format parameters. A format function evaluates the format string, it also accesses the extra parameters in the variable argument. Format String bugs happen when an attacker can provide the format string to an format function in part or in whole. Hackers use this exploitation tactic to enable stack memory overflow and inject arbitrary memory into the stack[7]. The main cause behind this is not properly validating the user-supplied input[7]. When hackers control the arguments of the $Printf$ function, it enables them to analyze or overwrite arbitrary data listed in the variable argument. Consider a program where we try to print out the number of characters written in a string.

```c
1 #include <stdio.h>
2 
3 int main() {
4     int num;
5     printf("ABCDE %n, &num));
6     return 0;
7 }
```

At line 5 in the above example, $\%n$ is a special format specifier. It causes printf to write 5 into variable num. An attacker can replace the $\%n$ with $\%s$ to view memory at any location, and cause printf to write an integer into any location. This way, attacker can overwrite flags which have access privileges. In general, format string bugs are the consequence of bad programming practice, allowing externally supplied data in the format string argument that results in exploitable format string vulnerabilities[7].

Another common vulnerability often caused by developers is using a pseudorandom number generator(PRNG) unsafely. PRNG function $rand()$ in C uses deterministic algorithms to create random numbers[8].

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
   printf("%D\n", rand());
   return 0;
}
```

The rand function declared in stdlib.h, returns a random integer in the range 0 to $RANDMAX$ inclusive every time you call it. $rand()$ does not guarantee the quality of random numbers it returns. Since $rand()$ is a pseudorandom number generator, the sequence of values it returns is predictable if you know its starting state[9]. PRNG's algorithm is not cryptographically strong. When PRNG functions are used for authentication, such a seed for a cryptographic key, then an attacker may easily guess the ID or crytographic key and gain access to restricted functionalities[9].

Shell functions such as $system()$, $proen()$ and $execv()$ used unsafely can result in exploitable vulnerabilities[2]. When passing an unchecked or improperly sanitized command string into the C standard $system()$, in the worst case it may result in execution of arbitrary system commands[4].

2)*Static Application Security Testing Tools*

Static Application Security Testing Tools analyze, detect source code and report weakness findings may lead to security issues. They are often engaged to provide immediate feedback during software development stages. *Static Application Security Testing Tools* are designed to scale and run repeatedly with continuous integration[10]. They can find security vulnerabilities with great confidence and report outputs with line numbers and file names[10].

From the National Institute of Standards and Technology (NIST)'s source code security analyzers list, we chose two tools, RATS and FlawFinder, based on certain conditions[2]. Flawfinder is free for C source code and it tries to find or check uses of risky functions such as buffer overflow, format string, race conditions, shell functions and poor random numbers[2]. RATS (Rough Auditing Tool for Security) is a tool for scanning C source code and reporting vulnerabilities such as buffer overflows, race conditions and TOCTOU (Time of Check, Time of Us)[11]. RATS and Flawfinder are both open source tools and detect similar security vulnerabilities.

B. Open-Source Software

Open-source software has the ability to allow developers to modify code, exchange code, and distribute their own releases of the program based on the open-source licensing agreement[13]. Open-source software is also publicly accessible for any one. Currently, a great number of applications in the enterprise market use open source software. Open-source software makes it easier for developers to implement the modified source code into the business's proprietary software, but it also presents risks and reveal vulnerabilities. Open-source software often involves poorly written code that easily leads to malicious attacks. The impact of open source on security influence many fields[13].

III. METHODOLOGY

A. Experimental Setup

The following section details our experimental setups as McLean. First, we selected a set of vulnerabilities exists in the targeted language. Next we discussed the logic behind choosing open source applications and the operating system on which we run the test. Thirdly, we gave a criteria for selecting static analysis tools. Finally we walked through the method for executing this experiment.

1) *Target Specific Vulnerabilities*:

For our experiment, we only focus on 4 categories of high potential vulnerabilities. The ability to detect these four categories of vulnerabilities is the basis of our experiment. *Buffer overflow* vulnerabilities in a program can overwrite other important data into its system and lead to abnormal behavior[3]. *Format string overflow* attacks can exploit the vulnerability of string format functions[7]. *Poor random number generator* may fail to protect private information[8], and shell function can be poorly implemented by developers that allowing attackers to inject code to manipulate the system.

2)*Choosing a Vulnerable Language*:

Since C programming language make it extremely easy for programmers to introduce security vulnerabilities into their code, therefore we chose C over other languages to run our experiment.

3) *Selecting Test Software Package*:

The rapid growth of remote access has become security problems for information protections. There are many security risks with this increased connectivity, such as unauthorised access to systems, data monitoring, and more. Many networking connection applications are targets of these attacks. We chose three open-source networking connection projects written in the C programming language.

Network Mapper(Nmap), is a free, open-source tool for administrators to scan and discovery networks. Nmap identify and discovery available hosts and their services[14].

PuTTY is a free and open-source application that support different network protocols[15].

Wireshark is a free, open-source application that analyze network communications and protocol developments[16].

We use these applications to demonstrate four common vulnerabilities in actual C projects. Table I lists these tools and their descriptions.

Application	Description
Nmap 7.91[14]	Network security scanner
PuTTY 0.74 [15]	SSH and telnet client
Wireshark 3.4.5 [16]	Network protocol analyzer

TABLE I
SELECTED OPEN SOURCE TEST APPLICATIONS

4) *Operating System*:

The selected test software packages and the selected static application security testing tools are all open-source projects and can be easily installed on Linux. Linux is the most used open-source operating system and widely available for download.

5)*Filtering Static Security Analysis Tools*:

Static application security testing tools help us find common software errors such as memory overruns, cross site scripting attacks, injections, and various other boundary cases[10]. There are many static code analyzers that work in different ways. We decided to use the taxonomy tree from McLean in TABLE II[2].

Flawfinder finds potential security vulnerabilities by using a database which has a list of insecure C functions[4]. Typical error types found by Flawfinder are buffer overflow vulnerabilities, string formatting attacks, and potential race conditions[4]. Flawfinder provides type (buffer, format, etc.) and risk-based (ranked zero, or lowest risk, through five, or highest-risk) results categorization, and a suggestion on how to fix the vulnerability[4]. It runs from the command line and the format of its output can be customized . Finally, Flawfinder summarizes its findings according to total hits, hits at each risk level, cumulative hits at each risk level, and the cumulative number of hits per thousands of lines of source code at each level[15].

RATS has the ability to find vulnerabilities in C, C++, Perl, PHP, and Python source code[12]. Buffer overflows, race conditions, and other major vulnerabilities are typical issues found by RATS. Vulnerability databases and output level can be configured. RATS have three different risk levels (Low, Medium, High), and does not offer vulnerability types[14].

Criteria	Filter	Reason
Rules	Security	Security is a must for performing analysis most relevant to this experiment. Some tools might use style- or interoperability-related rules, but searching for primarily security-based too.
Technology	Syntax	All targeted vulnerabilities a matter of syntax, i.e. insecure function calls
Language(s)	C/C++	C is the language of choice; C++ is a superset of C
Releases	N/A	Not concerned with regular releases.
Input	Source code	we desire to download source and analyze it without compiling.
Configurability	N/A	Only desire pre-packaged capabilities.
Extensibility	N/A	Not concerned with ability to extend tool past this experiment.
Availability	Free, open source	Commercial tools typically require a costly license undesirable for this experiment
User experience	Command line UI	Command line implementations likely require the least dependencies on additional UI libraries.
Output	List, text	We desire an ability to quickly filter and sort textual or itemized results using grep. HTML and XML output only complicate this task.

TABLE II
CRITERIA FOR FILTERING STATIC ANALYSIS TOOLS

B. Testing Process

We downloaded and installed each analysis tool and each network connection application using the Linux operating system. First we ran each tool on each application's source code. The output from running RATS was saved as html files and xml files. We wrote a Python parser to clean all of the xml files and then converted them to cvs files. The output from running Flawfinder was saved as html files and cvs files. We ran different execution commands with RATS and Flawfinder to get the vulnerabilities data:

Command	Description
rats −−quiet −−xml(html) −w 3 <path>	• −−xml, −−html generate output in the specified format • −w 1(high), 2(medium), 3(low) set the warning level
flawfinder −−html, −−csv <path>	−−csv, −−html generate output in the specified format

TABLE III
COMMANDS TO GET OUTPUT FILES

We organized the output data into Excel spreadsheets to analyze data in great detail. Flawfinder puts its findings into vulnerability types, such as buffer, format, shell, random, etc. RATS does not organize function calls by type, however. For example, Flawfinder notes $char[]$, $memcpy()$, and $strcyp()$ as all being buffer overflow type. RATS does not have to the ability to categories functions into vulnerability types. We manually mapped RATS' findings to the desired types based on RATS messages and Flawfinder outputs. For example, in running RATS on PuTTy-0.61 result file, we got a vulnerability called *"fixed size global buffer"*, we mapped it to type "buffer" based this message *"Extra care should be taken to ensure that character arrays that are allocated on the stack are used safely. They are prime targets for buffer overflow attacks."* Luckily, RATS finds most of the same functions as Flawfinder. After sorting the data on type, we measured the number of detected vulnerabilities for each test application package.

IV. RESULTS AND DISCUSSION

In this section, we compared RATS and Flawfinder based on their ability to detect and report findings. First, we analyzed their outputs according their ability to identify and quantify vulnerabilities[2]. Second, we analyzed each tool's findings base on its vulnerability type and summarized it overall performance. Lastly, we ran a t-test on 30 Wireshak releases, analyze its results and made our recommendation.

A. Output Helpfulness

For this comparison, we chose to sample the output of analyzing PuTTY-0.61 $cmdgen.c$ source file as McLean[2].

From TABLE IV, we can see that Flawfinder provides more output features than RATS. It lets you save your output into comma-separated value(CSV) format directly, and

Feature	RATS	Flawfinder
Function	YES	YES
Line Number	YES	YES
Cause	YES	YES
Risk Level	YES	YES
Type	No	YES
CWE	No	YES
Suggestion	No	YES
Detailed Summary	No	YES
CSV	No	YES
XML	YES	YES
HTML	YES	YES

TABLE IV
RATS AND FLAWFINDER OUTPUT FEATURE

categorizes findings into types. Both RATS and Flawfinder provide html output and we will use it to demonstrate the detailed difference of their outputs.

In Fig 1, RATS detected a buffer overflow vulnerability, categorized it as ”High” severity. These vulnerabilities happened on line 54, 946, and 976 in the form of $strcpy()$, in cmdgen.c file. Even though RATS gives a severity level and explains the cause of the vulnerability, however, it does not generate vulnerability type nor offers advice on how to solve the vulnerability.

Using the same PuTTY source file as Figure 2, Flawfinder's output shows the risk level is 4 out 5 and the buffer overflow type of vulnerability. Flawfinder also provides which CWE type this finding belongs to and gives a recommendation for how to improve this security issue.

In Fig 3, the RATS analysis summary is simple and short. It only contains the total lines analyzed and the time it used to scan the file. It does not show the total number of vulnerabilities at each severity level nor the number of each vulnerability type.

In Fig 4, Flawfinder produces more detailed ouput for the files analyzed. Flawfinder reports the total number of hits for each trial, the total number of hits at each risk level, number of hits at that risk level plus the sum of hits at all higher risk levels, and hits at each level per 1,000 lines of source code.

B. Tools

RATS consistently discovered *buffer overflow*, *format string*, *poor random number generator*, and *shell function* as the main types of vulnerabilities in PuTTY, Nmap and Wireshark source codes. We put the analysis results from running RATS on PuTTy, Nmap and Wireshark together to analyze the over behavior of RATS.

In TABLE V, we can see that *buffer overflow* as a type takes 83.9% of the overall 7459 vulnerabilities found in these 3 networking connection applications.

Type	Sum	Percent
buffer	6257	83.9%
format	311	4.2%
random	36	0.5%
shell	94	1.3%
race	576	7.7%
misc	185	2.5%

TABLE V
RATS FINDINGS ON PUTTY, NNAP AND WIRESHARK

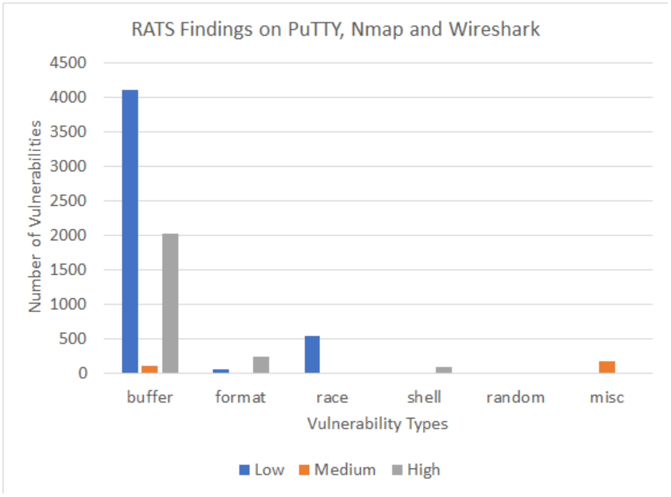


Fig. 5. Risk level by functions in buffer in RATS findings

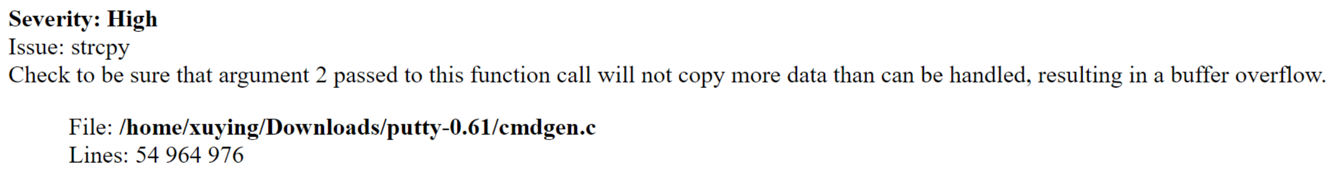
In Fig 5, we realized that most of the vulnerabilities found by RATS are risk level low and high in buffer overflow. Of the 2388 vulnerabilities found in risk level high, 2031 of them were in type *buffer overflow*, 247 were in type *format string*, 91 of them were in type *shell*, 8 of them in type *race*, and 0 in type *random*. Therefore, the majority of the high risk findings were in *buffer overflow* and *format string*.

Flawfinder also consistently discovered *buffer overflow*, *format string*, *poor random number generator*, and *shell function* as the main types of vulnerabilities in PuTTY, Nmap and Wireshark source codes. We put the analysis results from running RATS on PuTTy, Nmap and Wireshark together to analyze the over behavior of Flawfinder.

Type	Sum	Percent
buffer	6699	89.25%
format	310	4.13%
random	43	0.57%
shell	41	.55%
race	83	1.11%
misc	318	4.40%

TABLE VI
FLAWFINDER FINDINGS ON PUTTY, NNAP AND WIRESHARK

In TABLE VI, we can see that *buffer overflow* as a type takes 89.25% of the over all 7506 vulnerabilities found in these 3 networking connection applications.

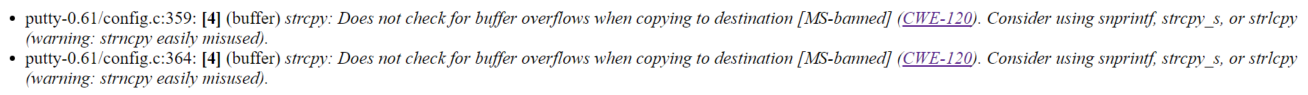


Severity: High
Issue: strcpy
Check to be sure that argument 2 passed to this function call will not copy more data than can be handled, resulting in a buffer overflow.

File: **/home/xuying/Downloads/putty-0.61/cmdgen.c**
Lines: 54 964 976

Fig. 1. RATS output of buffer overflow in html format

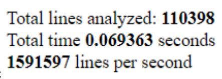
Figure



- putty-0.61/config.c:359: **[4]** (buffer) *strcpy: Does not check for buffer overflows when copying to destination [MS-banned] (CWE-120). Consider using snprintf, strcpy_s, or strlcpy (warning: strncpy easily misused).*
- putty-0.61/config.c:364: **[4]** (buffer) *strcpy: Does not check for buffer overflows when copying to destination [MS-banned] (CWE-120). Consider using snprintf, strcpy_s, or strlcpy (warning: strncpy easily misused).*

Fig. 2. Flawfinder output of buffer overflow in html format

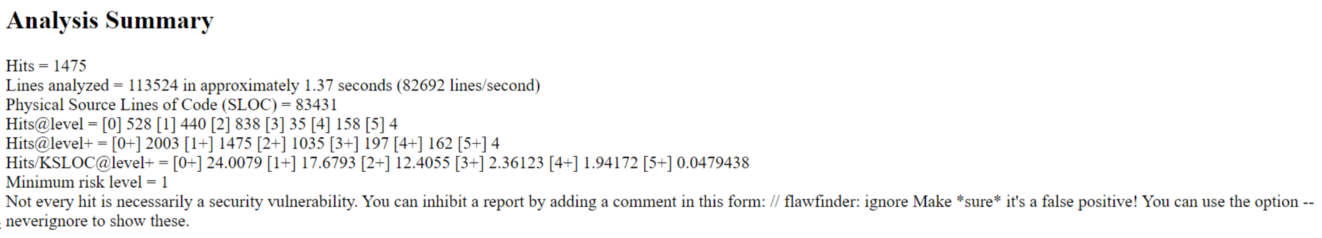
Figure



Total lines analyzed: **110398**
Total time **0.069363** seconds
1591597 lines per second

Fig. 3. RATS analysis summary

Figure



Analysis Summary

Hits = 1475
Lines analyzed = 113524 in approximately 1.37 seconds (82692 lines/second)
Physical Source Lines of Code (SLOC) = 83431
Hits@level = [0] 528 [1] 440 [2] 838 [3] 35 [4] 158 [5] 4
Hits@level+ = [0+] 2003 [1+] 1475 [2+] 1035 [3+] 197 [4+] 162 [5+] 4
Hits/KSLOC@level+ = [0+] 24.0079 [1+] 17.6793 [2+] 12.4055 [3+] 2.36123 [4+] 1.94172 [5+] 0.0479438
Minimum risk level = 1
Not every hit is necessarily a security vulnerability. You can inhibit a report by adding a comment in this form: // flawfinder: ignore Make *sure* it's a false positive! You can use the option --neverignore to show these.

Fig. 4. Flawfinder analysis summary

In Fig 6, we realized that most of vulnerabilities found by Flawfinder were risk level 1 and risk level 2 in buffer overflow. 16 vulnerabilities were found in risk level 5, with 10 of these vulnerabilities in type *race*. 568 vulnerabilities were found in risk level 4, 310 of them were in type *format string*, 148 were in type *buffer overflow*, 72 were in type *race* and 35 of them were in type *shell*. Even though the majority of these findings were in buffer, the majority of the high risk findings were in *format string* and *race*.

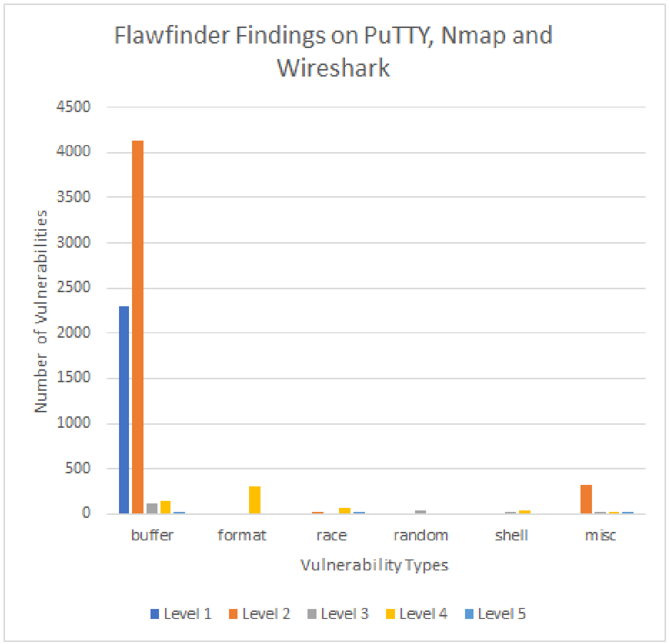


Fig. 6. Flawfinder analysis summary

1) Buffer Overrun:
The percentage of buffer overflow vulnerabilities detected in these 3 network connection applications by RATS and

Flawfinder were very close. Both tools detect buffer overflow violations by recognizing unsafe string library function calls. Even though secure methods exist which allow developers to safely manipulate buffers, unsafe function calls such as $memcpy()$, $strcat()$, and $sprintf()$ exist throughout all three software packages.

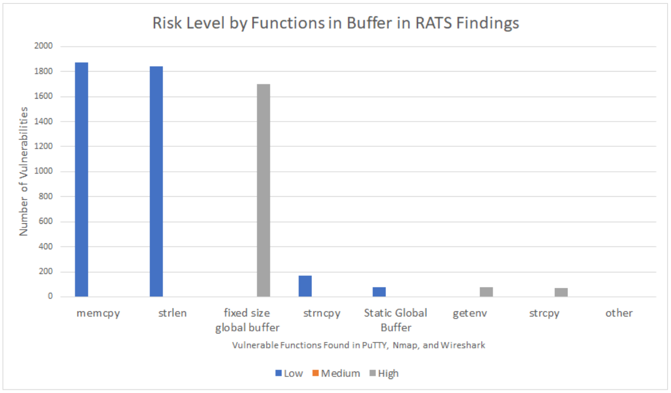


Fig. 7. Risk Level by Functions in Buffer in RATS Findings

Fig 7 shows that of the buffer overflow vulnerabilities in these three applications, 29.9% were caused by $memcpy()$ and 29.46% were caused by $strlen()$. These two functions are both in the low risk levels. $strncpy$ is identified by RATS as low risk level. $fixedsizeglobalbuffer$ was 27.19% and $strcpy$ was 2.73%. Both were identified as high risk.

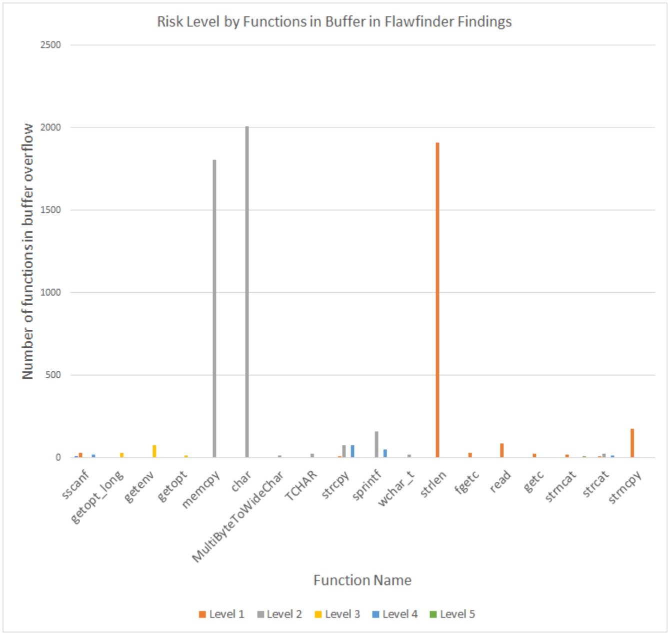


Fig. 8. Risk Level by Functions in Buffer in Flawfinder Findings

Fig 8 shows that the majority of buffer overflow finding were cause by $memcpy$ 24.95%, $char$ 26.75% , $strlen$ and $strcpy$. They are all level 1 and level 2 findings. Unless these buffers are directly accessible to user input, these items have little risk to security. $strcpy$, $sscanf$, $strcat$, $sprintf$ and $strncpy$ had the highest risk level in Flawfinder findings. However these high risk functions only made up 1.56% of the total *buffer overflow* vulnerabilities found.

We can see that RATS and Flawfinder rank $strncpy$ and $strcpy$ differently. In RATS, $strncpy$ is ranked low and $strcpy$ is ranked high. In Flawfinder $strncpy$ is ranked in level 5 and $strcpy$ is ranked in level 4. $strncpy$ and $strcpy$ both belong to CWE-120. CWE is a community-developed list of software and hardware weakness types[9]. CWE-120 is defined as buffer copy without checking size of input[9]. $strncpy$ and $strcpy$ are easily misused by developers. Even though overall Flawfinder found more *buffer overflow* vulnerabilities, RATS found a higher percentage of high-risk buffer overflow vulnerabilities. RATS found 29.5% high risk buffer overflow vulnerabilities while Flawfinder only found 1.56%. This contradicts the original findings of McLean. In the original findings, the author stated that Flawfinder found more buffer overflows in RATS overall in three test applications.

2)Format String:

In TABLE V and VI, RATS found 311 *format string* vulnerabilities which is 4.2% of all vulnerabilities found. Flawfinder found 310 *format string* vulnerabilites which is 4.13% of all vulnerabilities found by Flawfinder in the three test applications. The percentage of *format string* vulnerabilities was similar. Based on Fig. 9 and 10, the majority of the *format string* findings derive from potentially unsafe usage of the $printf$ family of functions such as $fprintf$ and $vfprintf$. *format string* can enable attackers to alter memory from stack. The main cause behind this is validating the user-supplied input. Both RATS and Flawfinder rank *format string* vulnerabilities as high risk findings.

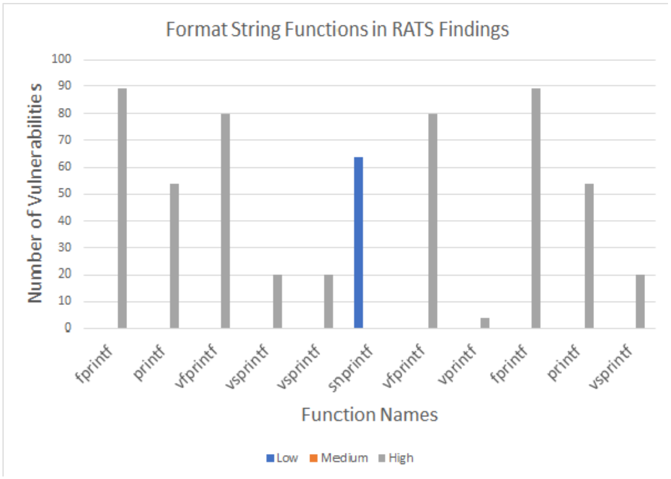


Fig. 9. Risk Level by Functions in Format String in RATS Findings

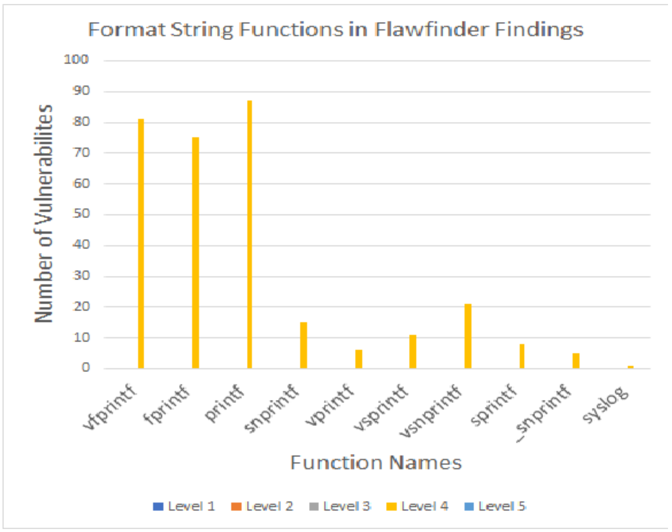


Fig. 10. Risk Level by Functions in Format String in Flawfinder Findings

3)Random Number:

In TABLE V and VI, RATS found 36 *random* vulnerabilities, which was 0.5% of all vulnerabilities found. Flawfinder found 43 *random* vulnerabilities which was 0.57% of all vulnerabilities found by Flawfinder in three test applications. Both tools rank *random* as a medium to low level risk as shown in Fig. 11 and 12.

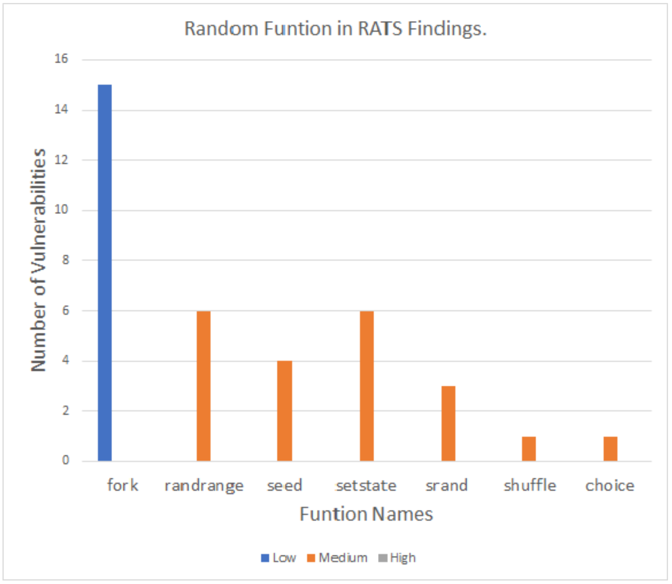


Fig. 11. Risk Level by Functions in Random in RATS Findings

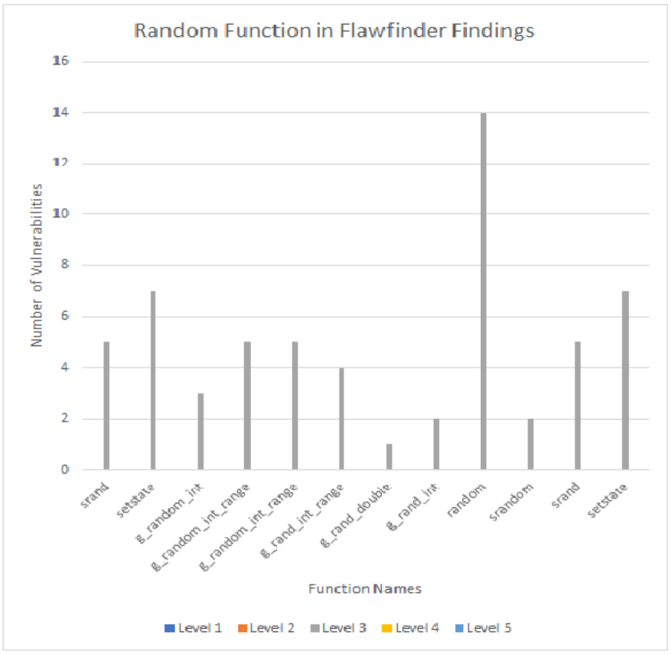


Fig. 12. Risk Level by Functions in Random in RATS Findings

4)Shell:

In TABLE V and VI, RATS found 94 *format string* vulnerabilities, which is 1.3% of all vulnerabilities found. Flawfinder found 41 *format string* vulnerabilities, which is 0.55% of all vulnerabilities found by Flawfinder in three test applications. In Fig. 13, RATS ranks *shell* vulnerabilities as medium to high risk, but in Fig. 14 Flawfinder ranks all *shell* vulnerabilities as level 3 across three different test applications.

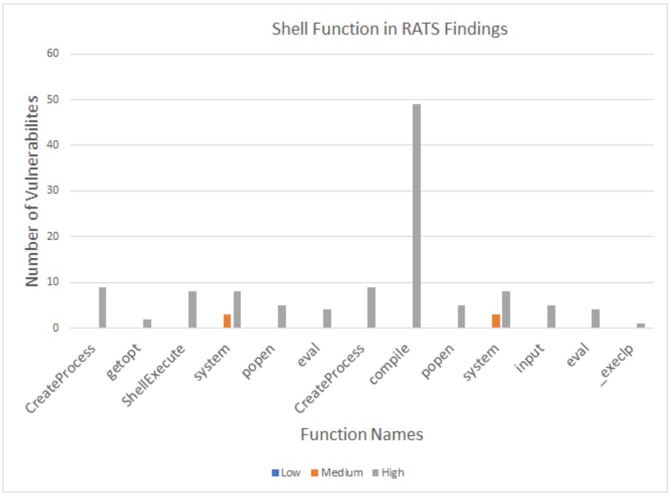


Fig. 13. Risk Level by Functions Shell in RATS Findings

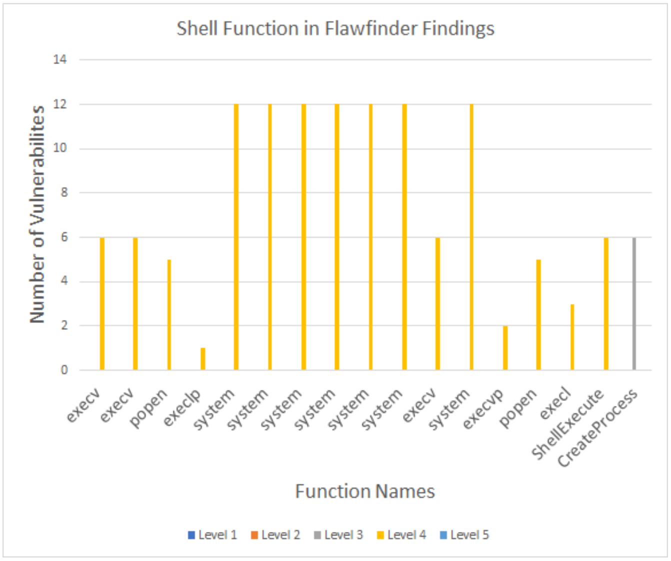


Fig. 14. Risk Level by Functions in Shell in Flawfinder Findings

5)Race:

In TABLE V and VI, RATS found 576 *race* vulnerabilities, which is 7.7% of all vulnerabilities found. Flawfinder found 83 *format string* vulnerabilities which is 1.11% of all vulnerabilities found by Flawfinder in three test applications. A race condition happens when a system has to perform a specific sequence of tasks at the same time[11]. Time of Check/Time of Use or TOC/TOU attacks are also used to refer this vulnerability[11]. In the original study, the author did not mention this vulnerability. Based on our findings, we should include *race* as a consideration.

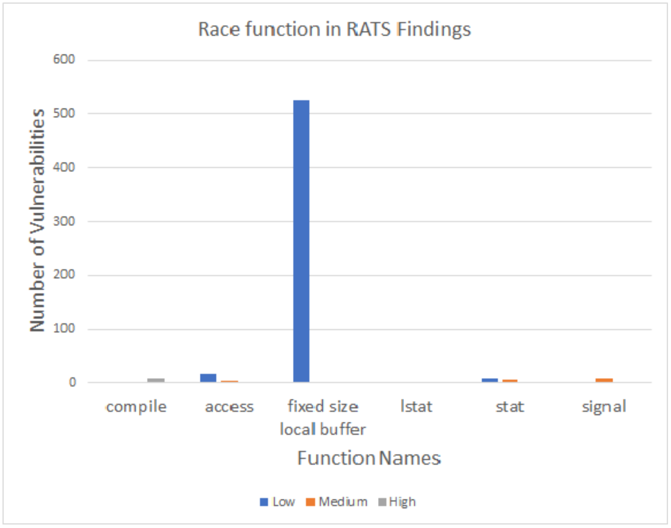


Fig. 15. Risk Level by Functions Race in RATS Findings

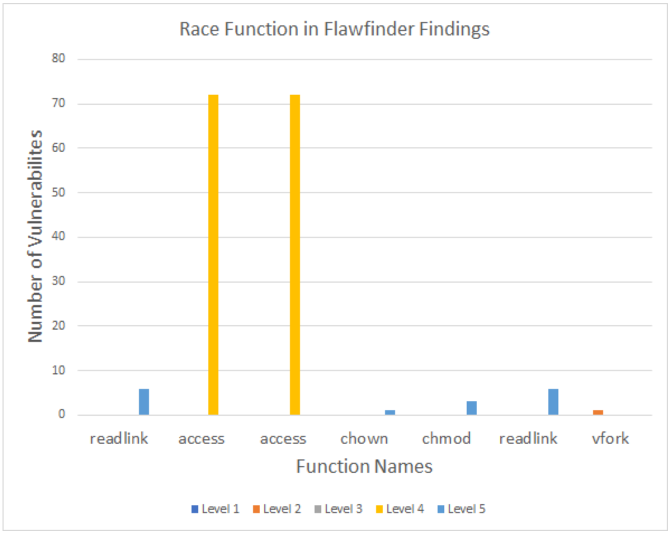


Fig. 16. Risk Level by Functions in Race in Flawfinder Findings

In Fig. 15 and 16, RATS labels *race* as a low risk vulnerability across three test applications, but Flawfinder labels *race* as a level 4 risk vulnerability.

C. *T test to compare RATS and Flawfinder*

After Comparing RATS and Flawfinder based on 5 different type of vulnerabilities, we realized that they sometimes gave the same vulnerability with different risk levels, even though they found close to the same amount of vulnerabilities on each vulnerability type. To determine whether there was a statistically significant difference in the means of Wireshark vulnerabilities between RATS and Flawfinder, we ran an independent sample t-test. The reason we chose Wireshark over Putty and Nmap to do the independent sample t-test is that when trying to run Flawfinder on an older release of Nmap, PuTTY and Wireshark, we got error messages saying $utf-8$ *codec can't detect byte X in position Y: invalid continuation byte*. With the time limitation of this study, we simply did not have to time to solve these error for older releases of these three test applications. RATS has no issues running any versions of these three test applications. Therefore, we downloaded 30 versions of Wireshark ranging from 2009 to 2021. For our t-test, we used the total number of vulnerabilities found by each tool on each version of Wireshark as seen in TABLE VII.

Wireshark Version	RATS	Flawfinder
fw-1.2.10[17]	2457	2942
fw-1.2.18[17]	2364	2941
fw-1.3.2[17]	2432	3058
fw-1.3.5[17]	2463	3033
fw-1.4.6[17]	2576	3193
fw-1.14.15[17]	2734	3189
fw-1.5.[17]1	2753	3201
fw-1.6.2[17]	2821	3238
fw-1.7.1[17]	3053	3479
fw-1.8.8[17]	3034	3466
fw-1.8.15[17]	3034	3465
fw-1.9.2[17]	3004	3391
fw-.10.14[17]	2995	3371
fw-1.11.3[17]	3005	3395
fw-12.13[17]	3017	3383
fw-1.99.9[17]	3196	3644
fw-2.0.16[17]	3222	3674
fw-2.1.1[17]	3075	3424
fw-2.2.6[17]	3194	3435
fw-2.2.17[17]	3086	3340
fw-2.5.1[17]	3092	3431
fw-2.6.20[17]	3222	3462
fw-2.9.0[17]	3111	3273
fw-3.0.14[17]	3133	3288
fw-3.1.1[17]	3222	3354
fw-3.2.7[17]	3170	3307
fw-3.2.13[17]	3133	3307
fw-3.31[17]	3238	3361
fw-3.4.0[17]	3268	3387
fw-3.4.5[17]	3268	3387

TABLE VII
NUMBER OF VULNERABILITIES OF WIRESHARK FOUND BY RATS AND FLAWFINDER

Tools	mean	sd	n
Flawfinder	3327.300	173.9576	30
RATS	2979.067	273.4536	30

TABLE VIII
T-TEST DESCRIPTIVES

mean
348.2333

TABLE IX
MEAN-DIFFERENCE

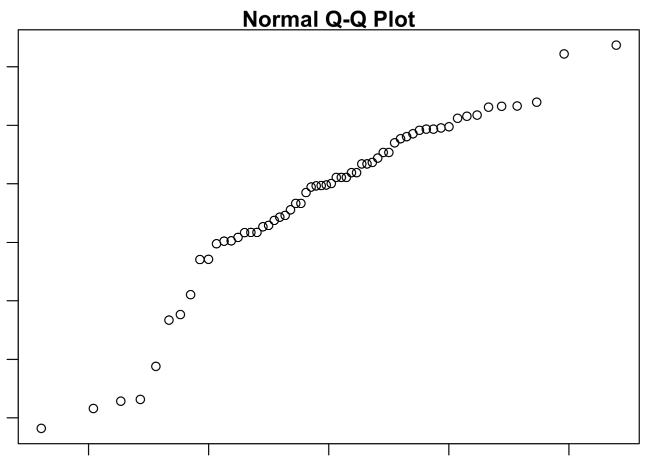


Fig. 17. Wireshark sample distribution

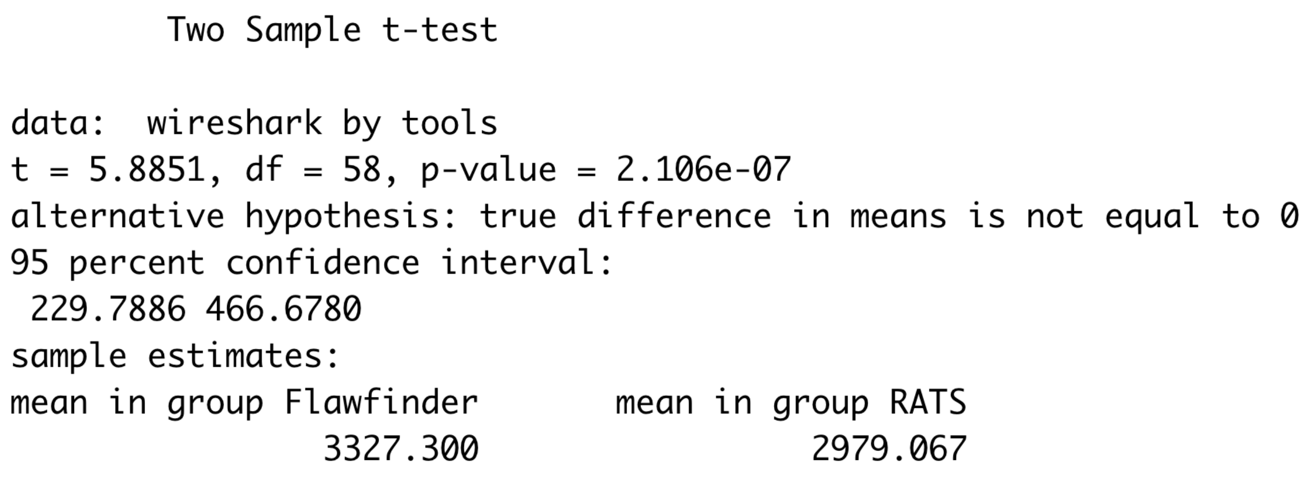
In Fig. 17 shows that our data met the assumptions of no significant outliers, and our data is approximately normally distributed for each tool and our data is independent.

In TABLE VIII, the t-test results showed that the mean Wireshark vulnerabilities in Flawfinder was 3327.300 with a standard deviation of 173.9576. There were 30 samples in Flawfinder. In RATS, the mean Wireshark vulnerabilities was 2979.067 with a standard deviation of 273.4536. There were 30 samples in RATS. Therefore, there is a mean difference of 348.2333(TABLE IX) between Flawfinder and RATS, with Wireshark vulnerabilities 348.2333 higher than RATS. Therefore, in our sample of 30 Wireshark releases it appears that Flawfinder finds a higher number of Wireshark vulnerabilities. In Fig. 18, with a 95% confidence interval, the mean difference was 229.7886 to 466.6780.

In Fig. 18, the obtained t-vlaue(t) were 5.8851 and the degrees of freedom were 58. The statistical significance p-value of our independent samples test is 0.0000002106. Our p-value is smaller than 0.05, therefore, there is a statistically significant difference in mean Wireshark vulnerabilities between Flawfinder and RATS.

D. Tool Usage Recommendation

Based on the RATS and Flawfinder's output helpfulness, since Flawfinder provide more output features and output formats, we recommend Flawfinder. If users need are to detect majority of *buffer overflow* and *race* vulnerabilities, we recommend RATS, based on the experiment ran on three different test applications, RATS detects more of these two types of vulnerabilities. For Wireshark users, we recommend Flawfinder based on the independent sample t-test on 30 releases of Wireshark. For users who want to be able to run any versions of the software without issues, we recommend RATS.



```
        Two Sample t-test

data:  wireshark by tools
t = 5.8851, df = 58, p-value = 2.106e-07
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 229.7886 466.6780
sample estimates:
mean in group Flawfinder       mean in group RATS
                3327.300                 2979.067
```

Fig. 18. t-test result

V. Threats to Validity

A. Internal Threats to Validity

Static security analysis uses patterns, rules, existing security issues in a language as guidance to detect vulnerabilities in source code. If a pattern or a rule has not been added to the database(Both RATS and Flawfinder use database), the toll will never find that vulnerability.

B. External Threats to Validity

Different tools have different behaviors and characteristics. Different characteristics might be preferred depending on where in the company or development cycle the tool is deployed. Here, researchers predefined a taxonomy of static code analysis tools. Within this taxonomy, they chose Flawfinder and RATS. Both Flawfinder and RATS are open source tools. It is obvious that these two tools won't be able to detect all of the different types of vulnerabilities or be good for all kinds of users.

C. Construct Threats to Validity

Operation during the experiment may not get all of the correct data. During the experiment, data is sorted based on their function, but RATS does not organize function calls by type. Therefore, we had to manually map RATS discoveries to the desired categories. This operation may produce errors or lose important information.

Static analysis tools often suffer from false negatives and false positives. A false positive is a reported vulnerability which is a security issue. A false negative is a vulnerability in code which is not detected by the tool. False negatives are more dangerous than false positives because they may lead developers to a false sense of security.

D. Conclusion Threats to Validity

In the paper, we recommended Flawfinder based on output helpfulness, vulnerability type analysis, and the Wireshark t-test summary. A static tool's output requires human evaluation and human judgement on which vulnerabilities represent an acceptable level of risk[18].

VI. Conclusions

Source code is the one common feature that every software shares. Code quality and security should rank very high on developers' priorities. Static security tools helps developer spotting common vulnerabilities such as *buffer overflow*, *format string*, *shell functions*, *poor random number generator*, *race conditions*, and more. Static security analysis tools can evaluate programs much more faster and much more frequently than developers[18]. Addition to that, they encapsulate security expertise and their results, so developers without a lot security knowledge can use these tools with ease. Static analysis tools for security should be applied regularly as part of the software development process.

VII. Future Work

A. Expand Test Set

In this study, we only examined three publicly available vulnerable software. Many different kinds of vulnerable software exist in different fields, such as web applications, online banking software, government software, and more. Using static security analyses on these software and other software applications would help developers prevent potential vulnerabilities. Also, collecting security analysis after running static security tools on these software would help us better understand the pros and cons of each static security tool and make better decision when choosing a static security tool.

B. Use Additional Analysis Tools

Testing more static security tools would present a much more comprehensive and accurate assessment. In this study, we only examined open-source tools, and it would be beneficial to include proprietary software in a future study.

References

[1] Mens T., ”Introduction and Roadmap: History and Challenges of Software Evolution” *Software Evolution(2008).* Springer, Berlin, Heidelberg, https://doi.org/10.1007/978-3-540-76440-31.

[2] R. K. McLean, ”Comparing Static Security Analysis Tools Using Open Source Software” 2012 IEEE Sixth International Conference on Software Security and Reliability Companion, Gaithersburg, MD, USA, 2012, pp. 68-74, doi: 10.1109/SERE-C.2012.16.

[3] C. Cowan, F. Wagle, Calton Pu, S. Beattie and J. Walpole, ”Buffer overflows: attacks and defenses for the vulnerability of the decade,” Proceedings DARPA Information Survivability Conference and Exposition. DISCEX’00, 2000, pp. 119-129 vol.2, doi: 10.1109/DISCEX.2000.821514.

[4] Stephen Turner, ”Security vulnerabilities of the top ten programming languages: C, Java, C++, Objective-C, C, PHP, Visual Basic, Python, Perl, and Ruby,” Turner Associates, Inc, Known-Quantity.com

[5] ”Most Secure Programming Language.,” WhirtSource, 2020, https://www.whitesourcesoftware.com/most-secure-programming-languages

[6] Himanshu Arora, ”Buffer Overflow Attack Explained with a C Program Example.,” 2013, https://www.thegeekstuff.com/2013/06/buffer-overflow/

[7] AJ Kumar, ”Format String Bug Exploration”, Infosec Resources, 2001, https://resources.infosecinstitute.com/topic/format-string-bug-exploration/

[8] C/Randomization, https://www.cs.yale.edu

[9] ”Common weakness enumeration”, 2021, https://cwe.mitre.org/data/definitions/338.html

[10] Brucker, A. Sodan, U., (2014). Deploying static application security testing on a large scale. In: Katzenbeisser, S., Lotz, V. Weippl, E. (Hrsg.), Sicherheit 2014 – Sicherheit, Schutz und Zuverlässigkeit. Bonn: Gesellschaft für Informatik e.V.. (S. 91-101).

[11] RATS, CERN Computer Security, https://security.web.cern.ch/recommendations/en/codetools/rats.shtml

[12] Eric Von Hippel, ”Innovation by User Commnunities: Learning from Open-Source Software”, adaptknowledge.com

[13] Annalen der Physik, 322(10):891–921, 1905. 2Gurbani VK, Garvert A, Herbsleb JD 5-WOSSE: proceedings of the fifth workshop on open source software engineering. In A case study of open source tools and practices in a commercial setting. volume 30 edition. New York, NY, USA: ACM; 2005:1–6.Return to ref 1 in article

[14] Nmap.org, https://nmap.org/

[15] PuTTY.org, https://www.putty.org/

[16] Wireshark.org, https://www.wireshark.org/

[17] Wireshark Downloads, https://www.wireshark.org/download/src/all-versions/

[18] Gray McGraw, ”Static Analysis for Security,” IEEE Xplore Full-Text PDF: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=1366126