



Vulnerability assessment of source code analysis tools for memory corruption vulnerabilities a comparative study

Johan Tejning

This thesis at the Faculty of Computing at Blekinge Institute of Technology is a part of the degree of Bachelor Science in computer science and is essentially equal to 20 weeks of full time studies.n.

The authors are the only authors of this thesis. They have only used the source listed in the references. The authors state that they haven't used the thesis to obtain a degree of bachelor at any other institution

Contact Information:

Author(s):

Johan Tejning

E-mail: jote@student.bth.se

University advisor:

Senior Ph.D.Oleksii Baranovskyl

Department of Department of DIDA

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

Background. One of the main reasons for memory corruption vulnerabilities lies in the lack of built in safety measures for the C/C++ programming language which is often time used to develop performance critical software. Static source code analysis tools perform a review of static(not running) source code usually by identifying sources of user input and data flow analysis in order to highlight potential security issues.

Objectives. In this thesis we will also try to figure out which types of vulnerabilities related to memory corruption that could be discovered by these kinds of tools as well as which types that appears to be difficult to discover by using this approach. We will also investigate some suggestions for improvements.

Methods. A comparative results of source code analysis tools written in C/C++ will take place for this thesis. The information needed in order to select the appropriate tools and test data will be derived from

1. Detection of the core reasons for each vulnerability.
2. Enumeration and separation of vulnerability cases.

Results/conclusion. All of the tools were able to find less then half of the buffer overflow vulnerabilities that existed in the data set and none of the tools were able to find any vulnerabilities related to integer overflow or use after free. The reason for this could very well be due to the tools limitations in their ability to find vulnerabilities caused by absence of correct processing.

Keywords: Vulnerabilities, Memory corruption, buffer overflow, integer overflow, use after free, double free

Acknowledgments

I would like to thank the Senior Ph.D. Oleskii Baranovskyl for being the supervisor of this thesis and supporting with feedback on the process

Contents

Abstract	i
Acknowledgments	iii
1 Introduction	1
1.1 Background	1
1.2 Aim and objectives	2
1.3 Research questions	2
1.4 Overview	2
2 Related Work	3
2.1 Comparison of source code analysis tools	3
2.2 Memory corruption detection	4
2.3 Summary	5
3 Procedures	7
3.1 Common causes of Memory corruption	7
3.1.1 Buffer overflow	7
3.1.2 Integer overflow	8
3.1.3 Format string	9
3.1.4 Double free	9
3.1.5 Use after free	10
3.2 Enumeration and separation of vulnerability cases	11
3.2.1 Presence of dangerous or vulnerable function	11
3.2.2 Absence of correct processing	11
3.3 Selecting the test data	12
3.4 Selecting the source code tools	12
3.4.1 First selection	12
3.4.2 Second selection	14
3.5 Searching for vulnerabilities	14
4 Method and results	15
4.1 Overview	15
4.2 Discovered vulnerabilities	18
4.2.1 Buffer overflow	18
4.2.2 Format string	19
4.2.3 Double free	19
4.3 Work environment	20

5	Analysis and discussion	21
5.1	Explanation of results	21
5.2	Sugestion for improvements	22
5.3	Threats to validity	22
5.4	Answers for research questions	23
6	Conclusions and Future Work	25
A	Tested files	29
A.1	Buffer overflow	29
A.2	Format string	33
A.3	Integer overflow	33
A.4	Double Free	33
A.5	Use after free	34

1.1 Background

Several memory corruption vulnerabilities were ranked in Cwe top 25 2020[2]. Between 2015 and 2020 the Chromium project found that around 70 percent of serious security bugs were memory corruption related issues[8]. In 2019 Microsoft security response center stated that about 70 percent of all security updates for Microsoft products had memory safety vulnerabilities[3]. The core reason for these issues lies in the lack of built in memory safety for system programming languages such as C/C++ which is oftentimes used for performance critical programs such as the operating system,firmware and device drivers. These languages are generally described as a low level language which means that the programmers are responsible for memory consumption which makes it very prone to vulnerabilities that are related to memory corruption. This opens up the possibility for attackers to modify the behaviour of the program and alter the flow of control in order to exploit the memory.

This thesis will compare the detection capabilities of static source code analysis tools in regards to memory corruption. static source code analysis tools are oftentimes used to analyze source code or compiled versions of code in order to find security flaws. They are useful for finding certain vulnerabilities such as buffer overflows[4], SQL injection[5] etc as well as providing developers with the source files, line numbers and subsections that are affected. The results however need to be analyzed by a human since it frequently does generate a high number of false positives and it can not always differentiate between a minor implementation bug from an actual vulnerability.

Several source code analysis tools such as Flawfinder[6] and Cppcheck[7] are able to detect vulnerabilities related to memory corruption although it is common knowledge in the software industry that many vulnerabilities exists that the tools struggles to identify such as for instance those related to configuration issues and dependencies on libraries etc. This study will hopefully provide useful information for system developers regarding the tools performance within that area.

1.2 Aim and objectives

The aim of this study is to find potential improvements for vulnerability detection in regards to memory corruption for non commercial static analysis tools by performing a comparative study. This will be done using the following steps..

1. Identify the memory corruption vulnerabilities with a high detection ratio for the tools and identify the root cause.
2. Identify the memory corruption vulnerabilities with a low detection ratio for the tools and identify the root cause .
3. Provide suggestions for improvement.

1.3 Research questions

1. Which memory corruption vulnerabilities could be found through static analysis tools and why is this the case?
2. Which memory corruption vulnerabilities could not be found through static analysis tools and why is this the case?

1.4 Overview

This thesis will be structured in the following way. In chapter 2 there will be a presentation of the related works that was used as a source of inspiration for this thesis. We will then see how the necessary information regarding source code vulnerabilities was derived in order to select the appropriate tools and test data in chapter 3 as well as how the selection of test data and source code analysis tools took place followed by a brief discussion of how the testing took place. In chapter 4 there will be a presentation of the method and the results. In chapter 5 there will be a discussion regarding the results in which suggestion for improvements will be provided. The thesis will later end with a conclusion and recommendation for future work in chapter 6.

Detecting software vulnerabilities at an early stage is an essential component for any organisation in order to ensure authenticity, integrity and availability of their software.

2.1 Comparison of source code analysis tools

Prof Arvinder Kaur and Ruchikaa Nayyar evaluated and compared commonly used open source static code analysis tools namely Flawfinder, Rats and Yasca for C/C++ and Findbugs, PMD, Lapse+ and Yasca for Java for the purpose of exploring their pros and cons.[18]. The test data were different kinds of approved and candidate source codes that are listed on the US Department of Homeland security's software assurance Metrics and tool evaluation. The test data contained common security vulnerabilities such as cross site scripting, OS command injection, null pointer dereference's etc. Each java source code tool had its own specific abilities. Lapse+ was not able to detect many security holes although it was proven useful for path traversal and injection type vulnerabilities. PMD was very effective in detecting poor coding practises, Null pointer dereference's and resource injection. Yasca managed to detect the most number of vulnerabilities although it also had the highest false positive rate. For the C/C++ source code tools it was found that Rats and Flawfinder were similar in what types of vulnerabilities they detect although Rats managed to find more vulnerabilities than Flawfinder. Yasca managed to find the same vulnerabilities that Rats found as well as other vulnerabilities, The combined results of all the tools in this study showed that there are certain source code vulnerabilities that are not easily detected by the tools. In this study neither of the tools were able to discover vulnerabilities related to integer overflow and the use of hard coded passwords.

Rahma Mahmood and Qusay H. Mahmoud highlight in their comparison of static analysis tools for Java and C/C++[19] the importance of incorporating static code analysis in software development life cycle given the huge increase in the number of discovered vulnerabilities in software products. They also perform a comparative study for these tools in which they find that there were some vulnerabilities that were not detected by either of the tools such as cross site scripting, use of hard coded password etc. The tools that were used in this study was Flawfinder, Cppcheck and Rats for C/C++ as well as Spotbugs and PMD for Java. These comparisons were done on juliet test suite version 1.3 and APACHE tomcat. For the C/C++ tools Rats did

have the highest detection ratio scoring 84/118. Similar to the the study presented by Prof Arvinder Kaur and Ruchikaa Nayyar[18] Rats and Flawfinder were similar in the types of vulnerabilities in which they detected. For the Java source code tools Spotbugs managed to find more vulnerabilities than PMD with a detection ratio of 92/171.

James A. Kupsch and Barton P. Miller performed a study in 2009 called Manual vs. Automated Vulnerability Assessment[24]in which they attempted to evaluate and quantify the effectiveness of 2 commercial automated source code analysis tools Coverity prevent and Fortify SCA by comparing their performance to and in depth manual analysis method on a large distributed system called Condor. The contributions of that study included presenting the limitations of the tools as well as the necessity of performing manual vulnerability assessment in order to accurately audit the security of the application. In their study they found that the tools were only able to identify a few of the vulnerabilities that were discovered by manual analysis and they were mainly implementation bugs which didn,t require any deep understanding of the code such as for instance the use of known vulnerable functions such as strcpy,system and popen etc.Both tools reported thousands of defects although only a few of them were security related.

Even though this study is 12 years old at the time of this writing the source code analysis tools today still suffers from most of the same limitations that were presented in that study.

2.2 Memory corruption detection

David Gens, Simon Schmitt, Lucas Davi and Ahmad-Reza Sadeghi presented in their paper K-Miner: Uncovering Memory Corruption in Linux[9] a new framework in order to perform global static analysis for detection of memory corruption vulnerabilities in operating system kernels. They were focusing on the issue that local intra procedural analysis that were mainly used by static source code analysis toll were not capable of performing global static analysis on of operating system kernels due to scalability issues. Through the use of interprocedural analysis it was able to detect multiple different classes of memory corruption vulnerabilities such as dangling pointers, use after free, double free and double lock vulnerabilities.K-Miner demonstrated an ability to detect recent versions of Linux in different configurations as well as detecting real world vulnerabilities.

Alexander Solzhenitsyn and Leninskie Gory presented in their study Buffer Overflow Detection via Static Analysis: Expectations vs. Reality a buffer overrun detector called svace that was able to perform interprocedural context and path sensitive analysis in which they managed to detect buffer overflow vulnerabilities on static and stack objects with a 65 percent true positive ratio on Android 5.0.2[23].This study also presented a survey on popular methods for buffer overflow detection and on overflow related CVEs. In the end of this study they were able to draw the conclu-

sion that interprocedural analysis, path sensitivity and loop handling are essential for buffer overflow vulnerability detection.

2.3 Summary

The 2 sections above provides information in regards to the general performance of source code analysis tools as well as different kinds of detection capabilities in regards to memory corruption vulnerabilities. This study will however focus specifically on widely used non commercial static source code analysis tools and their performance within the area of memory corruption. This study will not only evaluate their detection capabilities but also provide information as to why or why not they perform well within this area as well as providing information on detection methods that has performed well in previous studies.

In this chapter there will be a step by step explanation for the process of deriving the necessary information in regards to memory corruption vulnerabilities and how to detect them. This information will later be used in order to select the appropriate test data and source code analysis tools.

3.1 Common causes of Memory corruption

In order to select the appropriate tools and test data for this thesis you would have to discover the core reasons for memory corruption. This thesis will focus on some of the common causes for memory corruption which will be listed below.

3.1.1 Buffer overflow

Cause

Oftentimes programmers tend to make the assumption that a user never inputs more characters than a fixed size value defined by the programmer to be stored in a buffer. Buffer overflow generally leads to crashes although it could also overwrite variables used in the program as well as opening up for the possibility of arbitrary code execution[10].

```
int main(int argc, char **argv)
char buf[5];
strcat(buf,argv[1])
```

In the example provided above the argument from the program is directly appended into a buffer that has has the maximum capacity of 5 characters without checking the size of the argument. If the argument should exceed 5 characters then buf would be overwritten.

Detection

Buffer overflow vulnerabilities could be found through the use of both automated static and dynamic analysis tools. Manual analysis could be useful for smaller programs although may not be feasible for larger programs since all inputs would have to be taken into account which could be very time consuming. It could also be useful to generate a large amount of diverse inputs to the program through fuzz testing and see how the program responds.

3.1.2 Integer overflow

Cause

An integer variable has a fixed range of values that it is able to store. If such a variable should be incremented to a value that exceeds its limit it generally leads to undefined behaviour which then often times leads to crashes. Integer overflow could have security consequences if the same variable is being used in for instance size determination of memory allocation, copying, concatenation etc since the variable could potentially wrap to a very small number or negative number[11].

```
int main(int argc, char **argv)
int number=atoi(argv[1]);
char *buf;
buf=(char*) malloc(number);
```

In the example above a char buffer is allocated with a size specified by the user input. If this size would exceed the limit that the integer variable can store then it could potentially for instance cause an allocation of 0 bytes to the buffer which could have severe consequences if subsequent code operates on that buffer.

Detection

Some of the detection methods used for finding Integer overflow is both automated static and dynamic analysis tools and fuzzing. It could also be found by manual analysis through methods such as penetration testing and evaluation of the calculations used in memory allocations.

3.1.3 Format string

Cause

This becomes a vulnerability when a function accepts a format string as an argument that originates from an external source since it could potentially lead to buffer overflow and execution of arbitrary code[12]. An attacker could exploit format string vulnerabilities in order to read and write content to the stack.

```
int main(int argc, char **argv)
char holder[32];
snprintf(holder,32,argv[1]);
```

In the example program above the function `snprintf` copies the userinput into the char array `holder`. If an attacker provides a format specifier such as `%n` as an argument to the example program above he/she could write some arbitrary content to the stack.

Detection

Format string vulnerabilities could be discovered through automated static analysis tools and manual analysis.

3.1.4 Double free

Cause

If the same memory address is being freed twice[13] the programs memory management structures becomes corrupted which could result in a write what where condition in which the attacker has the ability to write an arbitrary value to an arbitrary location[14]. An attacker can exploit this vulnerability by overwriting certain memory spaces or registers in order to manipulate the program to execute malicious code.

```
int main(int argc, char **argv)
char *buf;
buf=(char*) malloc(10);
if(some_condition)
free(buf);
free(buf);
```

In the code example a dynamic char array is allocated and if `some_condition` is true pointer for the allocated memory area will be released twice.

Detection

Double free vulnerabilities could be discovered through automated static analysis tools and manual analysis.

3.1.5 Use after free

Cause

By referencing memory after it has been freed a program could crash, use unexpected values or potentially corrupt valid data if the same memory area has been allocated elsewhere[15].

```
int main(int argc, char **argv)
char *buf;
buf=(char*) malloc(10);
if(some_condition)
free(buf);
strncpy(buf,argv,sizeof(argv));
```

In the code example above a dynamic char array is allocated and if `some_condition` is true the pointer for the allocated memory area will be released and later reused in the `strncpy` function which attempts to copy the content of the userinput into a released memory location.

Detection

Use after free vulnerabilities could be discovered through automated static analysis tools and manual analysis.

3.2 Enumeration and separation of vulnerability cases

In order to understand which vulnerabilities could easily be detected with automatic code analysis tools and which ones that require manual analysis all known cases that could cause the selected categories needed to be enumerated and divided up into the following 2 sections

- Presence of dangerous or vulnerable functions such as the use of `strcpy()`, `system()` etc.
- Absence of correct processing which means that the vulnerability is caused by general poor programming practices but the presence of dangerous or vulnerable functions are absent.

3.2.1 Presence of dangerous or vulnerable function

```
int main(int argc, char **argv)
char buf[20];
strcpy(buf, argv[1]);
```

In the example provided above the vulnerable function `strcpy()` copies the user argument directly into the char buffer without checking that the size of the input does not exceed the size of the buffer. Most tools should be able to find these cases since all that is required is some provided knowledge on the existence of these functions by for example the usage of a database with known vulnerable functions.

3.2.2 Absence of correct processing

```
int main(int argc, char **argv)
int number=30;
char buf[20];
for(int i=0;i<number;i++)
buf[i]='a';
```

In this example a loop is causing a buffer overflow since 30 characters are put into a char buffer which only can handle 20 characters. Vulnerabilities that are related to absence of correct processing are generally more difficult for source code analysis tools to discover since they require some knowledge about the flow of the program. The author of this thesis does not believe however that the above example would be difficult for most tools to discover due to its simplicity.

3.3 Selecting the test data

The test data was selected from Nist software assurance dataset project [16] which maintains a database of vulnerable test cases as well as real software applications with known bugs and vulnerabilities. 100 test cases were selected for each of the vulnerability categories.

3.4 Selecting the source code tools

3.4.1 First selection

Initially 4 static source code analysis tools were selected for analysis due to the fact that they all claim to be able to find one or more instances of memory corruption vulnerabilities but also the fact that they were non-commercial and easy to use and set up. This was especially important due to budget and time limitation at the time of testing which is both a concern for the author of this thesis but also for various other software development companies who want to save time and money. The selected tools ended up being the following.

- **Flawfinder:** Flawfinder is an open source static source code analysis tool written in Python that is capable of quickly examining and report flaws in C/C++ source code and sort them by risk level. The risk level depends upon the vulnerable function and its parameter. In many cases Flawfinder is capable of differentiating between parameters with a constant string (which oftentimes does not provide any real danger) and fully variable strings (This could potentially be dangerous since the values of the variables may have come from user input) and thereby sorting them with different risk levels. Flawfinder is using a built in database of known vulnerable functions such as `strcpy()`, `strcat()`, `gets()`, `sprintf()` etc for which it uses to match the source code text against. It works on unix-like systems and on windows through the use of cygwin. Flawfinder is capable of generating output in the form of textfile, html and comma separated csv files. Flawfinder comes with a couple of useful features. One of them is the ability to only review the difference between an older and newer version of a program by only report hits that relates to lines that has changed [6].
- **Cppcheck;** This tool focuses primarily on undefined behaviour such as integer overflow, uninitialized variables and null pointer dereferences and dangerous coding constructs in C/C++ code however it is also capable of generating alerts for other issues such as for instance style and performance. Cppcheck performs unsound flow sensitive analysis which means that it takes the order of the statements in the program into consideration while at the same time tries to keep the amount of false positive reports to a minimum. Even though Cppcheck primarily focuses on a low amount of false positive it does have an extra analysis method called bug-hunting which is a more sound method for those who wants to do the tradeoff of more false positives for the purpose of finding more actual vulnerabilities. The developers of Cppcheck claims that by using the bug-hunting analysis method you should be able to find most bugs reported in

cve[26](common vulnerabilities and exposures) which is a well known catalog of vulnerabilities that has been discovered and published from organisations worldwide. Cppcheck is cross platform and can be run on both windows and unix operating system. It is available as both command line interface and graphical user interface[7].

- Rats; This is an open source tool capable of quickly scanning various languages such as C,C++,Perl,Php and python for vulnerabilities such as for instance buffer overflow and race conditions[21]. Similar to flawfinder Rats is comparing the source code to its own vulnerability database. You have the option to choose if you want the tool to find more vulnerabilities at the cost of more false positive findings as well as the opposite. It can generate output in html and xml,
- Visual code grepper: Visual code grepper is a windows based automated code security review tool for C,C++,Java,C,VB,PL,SQL. It is capable of quickly scanning vulnerabilities such as for instance buffer overflow and signed/unsigned comparisons.It also has a config file for each language that allows one to add certain functions and text that may be interesting to search for[22].It can be run both as a command line tool in which the results can be outputted into csv files, textfile, or xml file and as graphical user interface. In visual code-grepper you can choose to scan for comments, potential code security issues, dangerous functions or a full scan which is a combined scanning of these[27].

3.4.2 Second selection

The tools were here selected to be used in this study due to their ability to present associated Cwe output for their findings which would make the discovered vulnerabilities easily searchable, they also had to have been used in previous academic research or that they are generally well known and the most recent version had to be from 2021.

Tools	Cwe output	Have been used in previous research	Most recent version from 2021
Flawfinder	yes	yes	yes
Cppcheck	yes	yes	yes
Rats	no	yes	no
Vcg	no	yes	no

Based on the selection criterias from table above the following tools ended up being selected for this thesis.

- Flawfinder version 2.0.15 [6]
- Cppcheck version 2.4.1 [7]

3.5 Searching for vulnerabilities

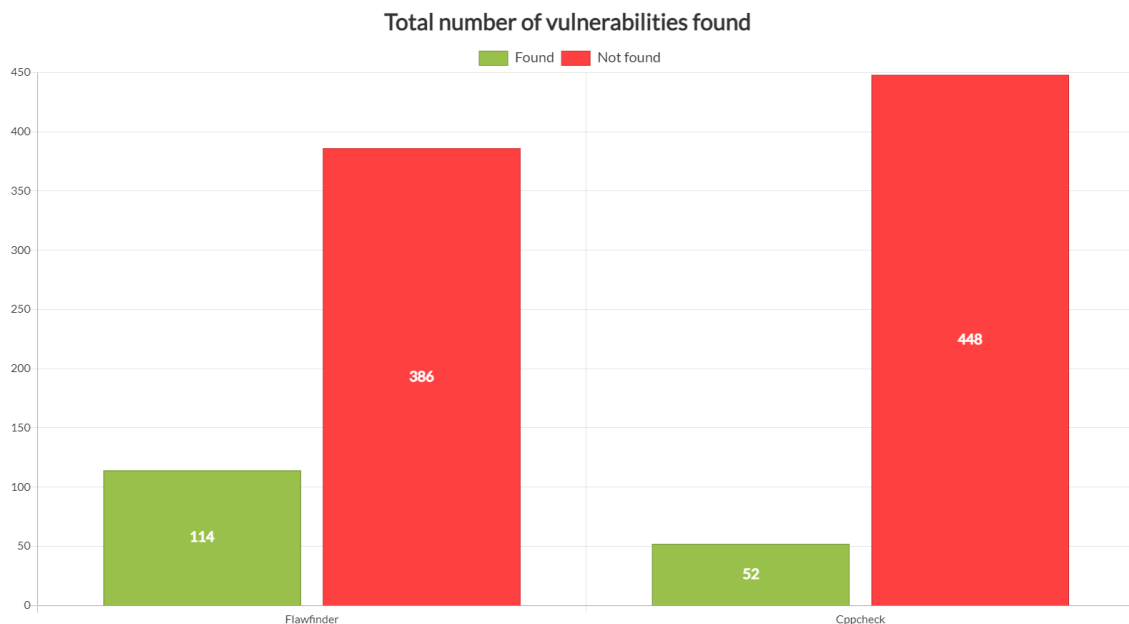
After running both tools on the test data the results needed to be analyzed in order to assess the performance of the tools. The following criterias had to be met in order for the alerts generated by the tools to be classified as a true positive.

- The tool had been able to identify the alert as a vulnerability that belongs to the selected category.
- The tool had been able to correctly identify the line and file of the vulnerability.

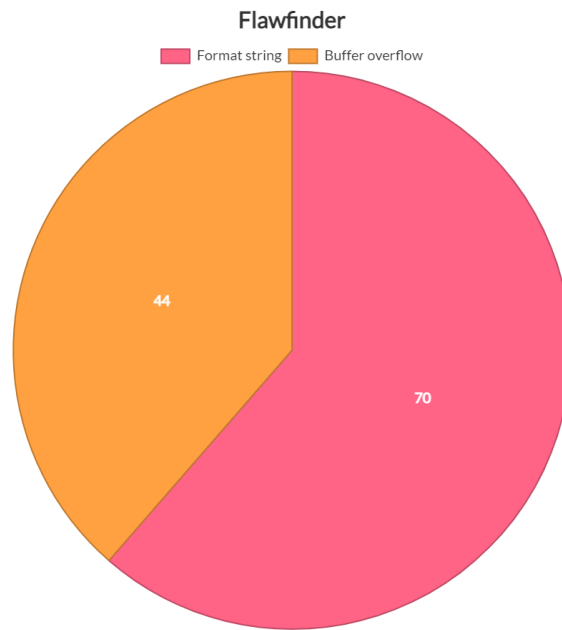
For the duration of this thesis a comparative study will be taking place between 2 different source code analysis tools in regards to their ability to detect memory corruption vulnerabilities. The reason why a comparative study is the best alternative for this thesis is that the only way to gain understanding regarding the overall performance of source code analysis tools is by drawing a conclusion based on the similarities and differences in their results.

This chapter will compare the results for the tools and present the total amount of findings as well as how many of the discovered vulnerabilities were discovered due to the presence of vulnerable function and how many were discovered because of absence of correct processing(Bad practise).

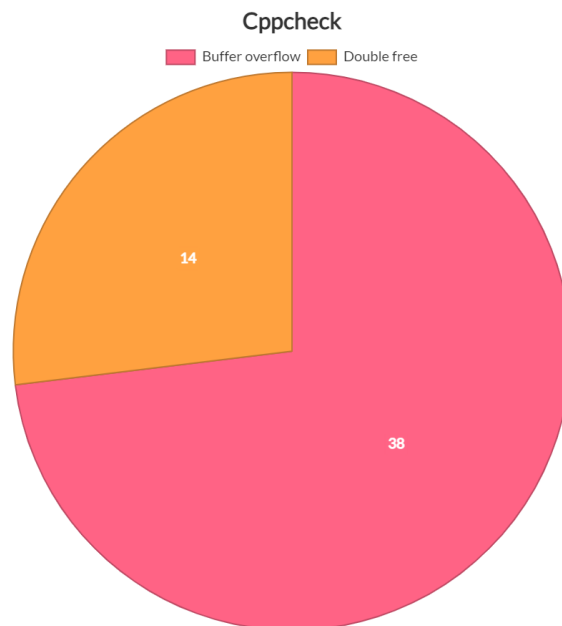
4.1 Overview



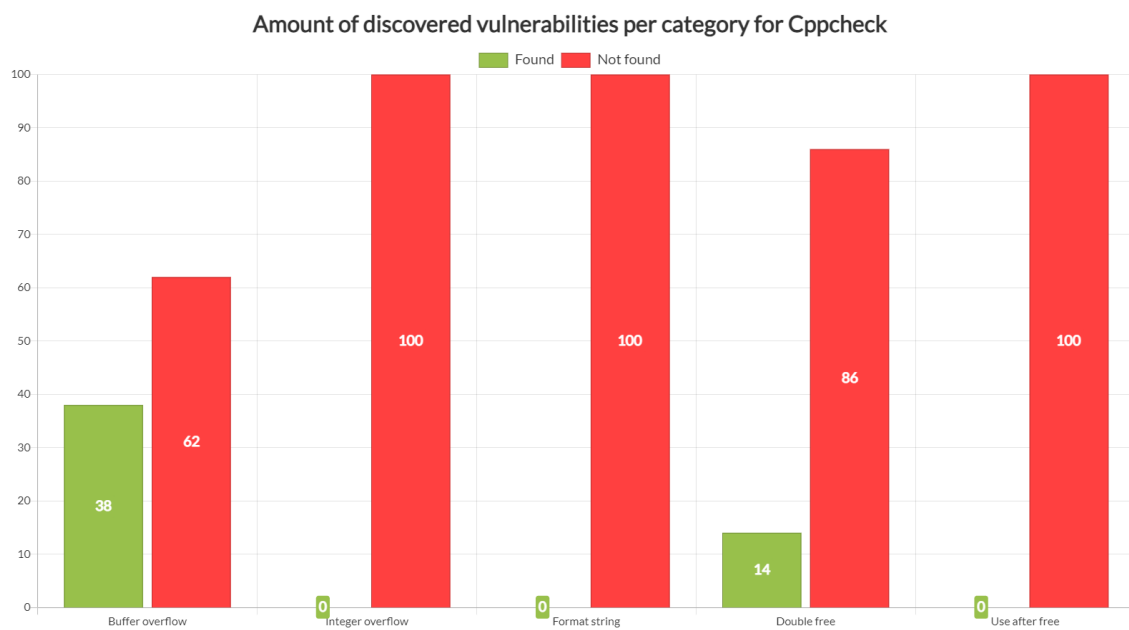
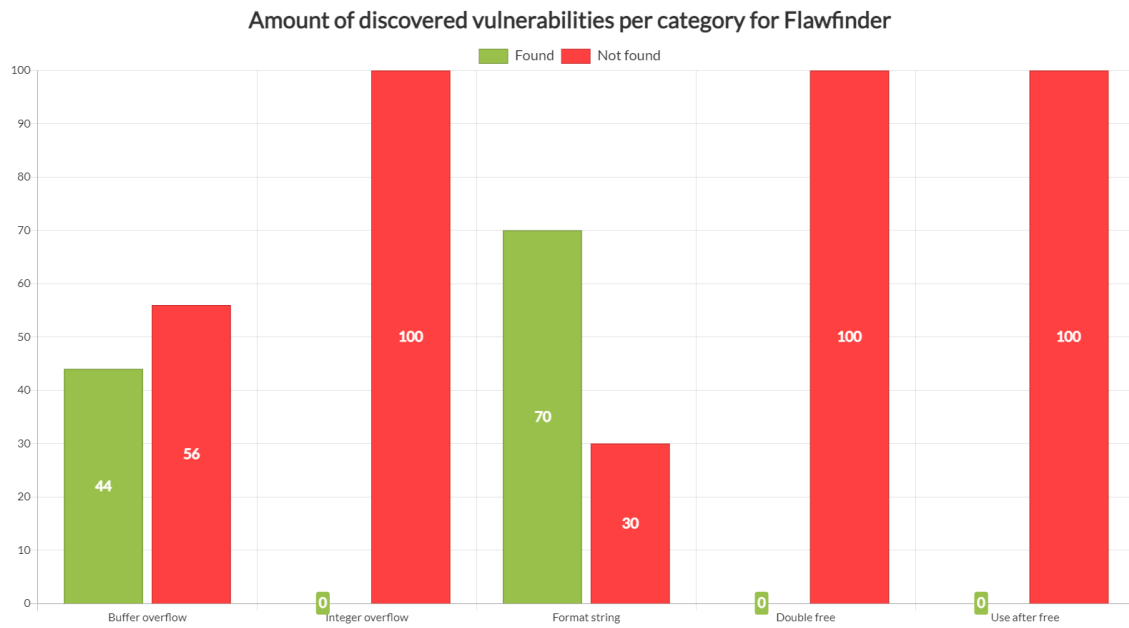
Flawfinder managed to detect approximately 23 percent of the total amount of vulnerabilities while cppcheck managed to detect approximately 10 percent.



Flawfinder had the best performance when it came to the detection of Format string vulnerabilities. The reason for this is because Format string vulnerabilities are often time associated with the presence of known vulnerable function such as for instance printf.



Cppcheck had the best performance when it came to the detection of buffer overflow. Buffer overflow are like format string vulnerabilities also commonly caused by the presence of known vulnerable functions such as strcpy,sprintf,snprintf etc which was the reason for all of its discovery in regards to this vulnerability.

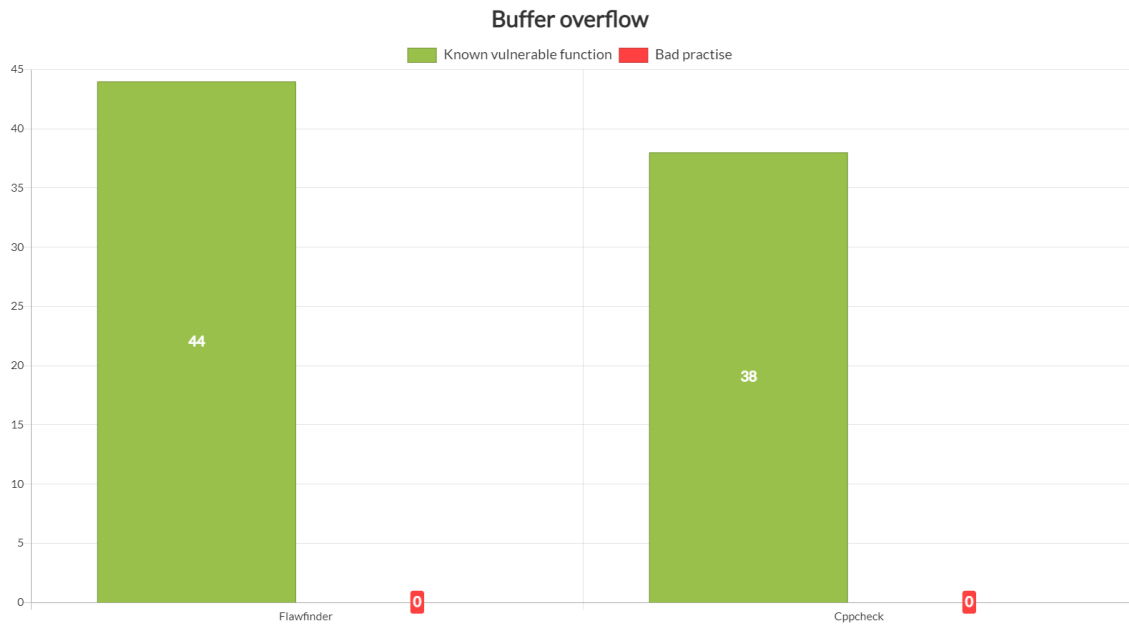


There is no specific category here that both of the tools performed particularly well. Flawfinder was able to discover format string vulnerabilities with a relatively high detection ratio. Neither of the tools were able to provide a high detection ratio of Integer overflow, double free, and use after free. Flawfinder did not find any instances of Integer overflow, Use after free, or double free. Cppcheck did not find any instances of Format string, integer overflow, and double free.

4.2 Discovered vulnerabilities

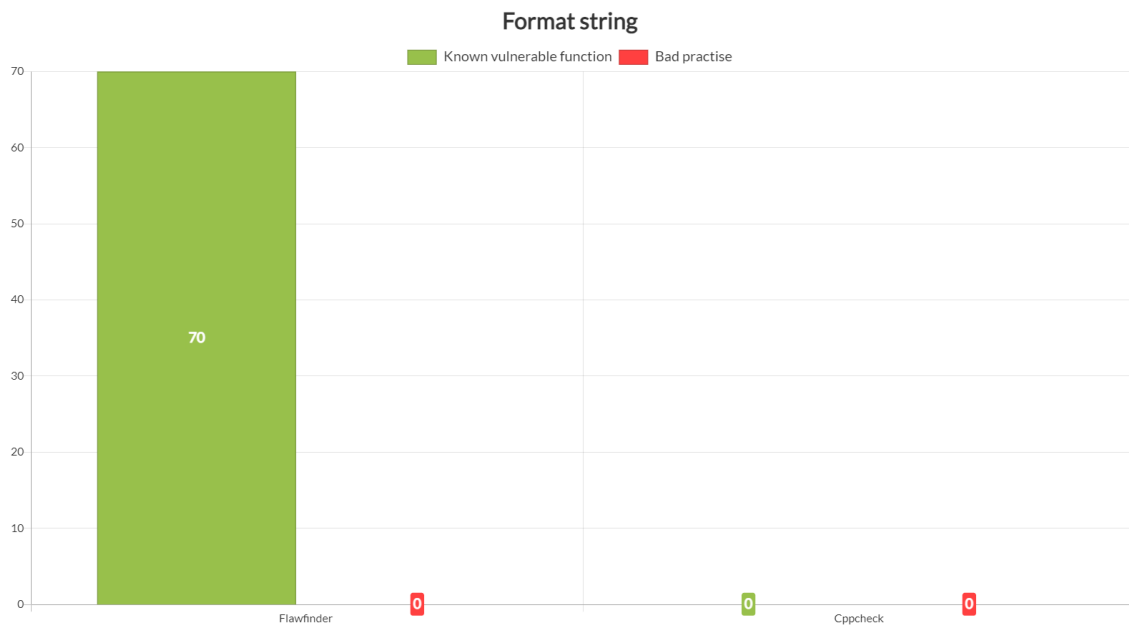
This section will present the vulnerabilities that were found by at least one of the tools as well as how many of the discovered vulnerabilities that were caused by the presence of known vulnerable functions and how many that were discovered due to bad practise.

4.2.1 Buffer overflow



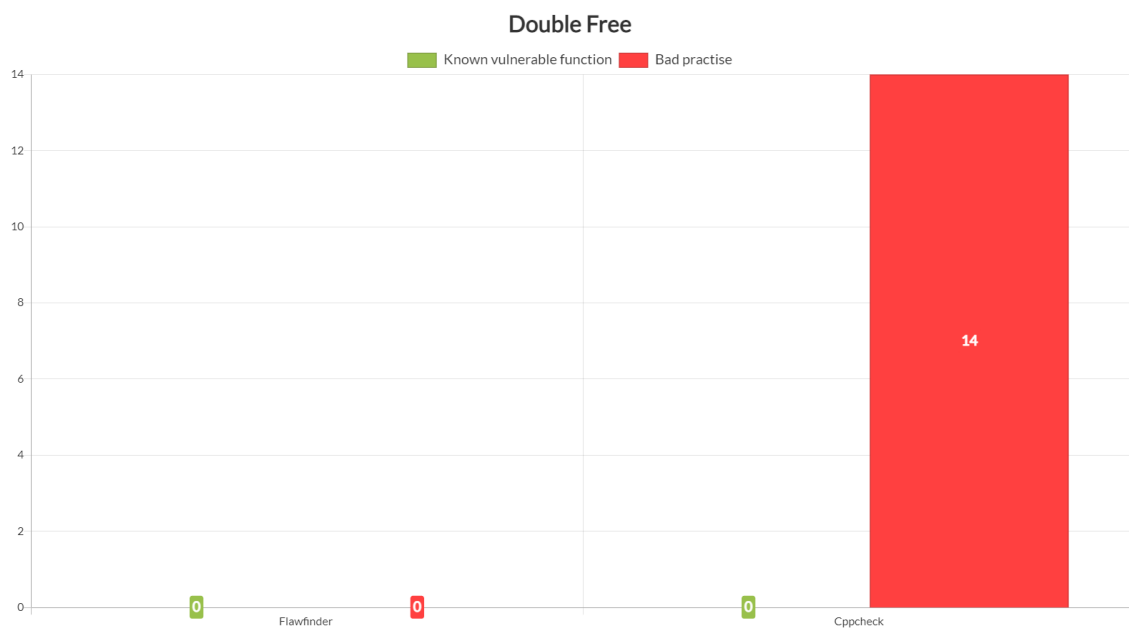
Both tools had an detection ratio of below 50 percent for buffer overflow vulnerabilities. All of the discovered vulnerabilities was because of the usage of known vulnerable functions such as strcpy, strcat, sprintf etc.

4.2.2 Format string



Flawfinder performed reasonable well for the detection of Format string vulnerabilities while cppcheck did not manage to find any. Many of the vulnerabilities in this category that flawfinder managed to find were due to the absence of format string in functions such as `printf`, `vfprintf`, `vsprintf` etc. .

4.2.3 Double free



It is difficult to find a clear pattern for the discovery of double free vulnerabilities more than the fact that none of the discovered vulnerabilities were due to any usage

of vulnerable functions and that it mostly appears to be able to find the vulnerabilities for simple cases similar to the pseudo example provided below.

```
if(condition1)
free(buf);
if(condition2)
free(buf);
```

The consequences for the code example above is that if both conditions are true then buf will be freed twice.

4.3 Work environment

The analysis took place in a virtual environment using Virtual box which is running Kali Linux version 2020.4 64-bit. The host machine is using Windows 10 version 1909 64-bit.

5.1 Explanation of results

All of the discovered vulnerabilities found in regards to buffer overflow was because of the usage of known vulnerable functions such as `strcpy`, `strcat`, `sprintf`. The reason why Flawfinder was able to discover format string vulnerabilities with a relatively high detection ratio was due to the fact that these types of vulnerabilities are commonly associated with the use of vulnerable functions and are therefore relatively easy to find with a pattern based approach. The absence of known vulnerable functions related to Integer overflow, double free and use after along with the fact that neither of the tools are able to perform sufficient dataflow analysis could be the reason why they performed poorly when it came to discovering these vulnerabilities.

Flawfinder does not perform any dataflow analysis at all since it is merely using a built-in database of known vulnerable functions in `c/c++`. Cppcheck is able to perform flow sensitive analysis but it focuses on unsound analysis which is useful if your goal is to have as few false positives as possible although less useful when it comes to finding as many actual vulnerabilities as possible since it makes it unable to detect all potential problems. Cppcheck does however have an extra analysis method called bug hunting which focuses more on finding as many vulnerabilities as possible with the downside of potentially more false positive findings. This analysis was useful in order to improve the detection ratio for buffer overflow vulnerabilities since it was able to generate alerts for the presence of vulnerable functions such as `strcpy` and `sprintf` however it did not report anything for the rest of the vulnerability categories and it seemed to be mostly focused on reporting the potential use of uninitialized variables.

5.2 Sugestion for improvements

It is unlikely that there will be any major improvements in Flawfinders ability of detecting these vulnerabilities that it failed to detect since they are difficult to detect through the use of a pattern based approach however cppcheck does have an interesting concept in their bug hunting analysis so if they could implement these vulnerabilities in that analysis it would most likely improve the results.

The suggestions presented below highlights the methods that has been effective in detection of various vulnerabilities related to memory corruption based on previous studies and could work as a guideline for future developement of source code analysis tools.

- An alternative method to find some of the vulnerabilities that had a low detection ratio for this study is being presented by David Gens, Simon Schmitt, Lucas Davi and Ahmad-Reza Sadeghi[9] in which they present a framework capable of performing inter procedural analysis in order to find multiple different classes of memory corruption vulnerabilities such as dangling pointers, use after free, double free and double lock vulnerabilities in operating system kernels.
- Inter procedural context and path sensitive analysis has been proven to be a rather effective analysis method in order to detect buffer overflow vulnerabilities according to Alexander Solzhenitsyn and Leninskie Gory who implemented this in their buffer overrun detector[23]. They managed to receive a 65 percent true positive ratio on static and stack object.

The author of this thesis do specifically recommend the implentation of interprocedural analysis in source code analysis tools. It would reduce a great deal of false negative if the tools were able to track the flow of the program from a function call to its corresponding function and see which values that are passed in between. //Note more suggestions for improvements according to previous work will be presented for the final thesis due to time limitation.

5.3 Threats to validity

The selection criteras that were used in order to decide upon which tools to use for the duration of this thesis narrowed down the results to 2 tools which may have had an impact on the objectivity of the results. The results may also have been different if a larger amount of test cases per vulnerability were selected for testing although due to time constraint this was not feasible for this work since all of the test cases had to be analyzed manually.

5.4 Answers for research questions

Which memory corruption vulnerabilities could be found through static analysis tools and why is this the case?

Both tools were able to detect less than half of the vulnerabilities related to buffer overflow. All of the discovered vulnerabilities that were found were due to the usage of known vulnerable functions such as strcpy, strcat, sprintf etc which is relatively easy to discover since the tools only need to know the function names which is also the reason why Flawfinder was able to find Format string vulnerabilities with a relatively high detection ratio. Cppcheck was able to detect a few of the double free vulnerabilities since it is capable of performing flow sensitive analysis and is therefore to some extent able to follow the flow of the program and see if the resource in question has been previously released.

Which memory corruption vulnerabilities could not be found through static analysis tools and why is this the case?

Integer overflow and use after free was not discovered by any of the tools in this test data. Flawfinder has however been proven to be capable of discovering Integer overflow vulnerabilities in previous studies such as the one made by Rahma Mahmood and Qusay H[19] so that contradicts the notion that Flawfinder is incapable of finding these types of vulnerabilities. The same study did however not manage to find any instances of Integer overflow for Cppcheck. Even though Flawfinder is based upon previous studies evidently capable of detecting Integer overflow the results from this study does imply that Flawfinder may not be an optimal tool for the discovery of these kinds of vulnerabilities especially since it is using a pattern based approach and there are not that many known vulnerable functions associated with Integer Overflow as for those related to for instance buffer overflow and Format string that the author of this thesis is aware of. The detection of Use after free vulnerabilities would require some type of knowledge about the program flow in order for the tool to be able to discover whether or not a resource has been released so it is difficult to see that Flawfinder could ever discover these types of vulnerabilities.. No answers could be found as to why Cppcheck did not find any instance of integer overflow or double free based upon the results from this study .

Chapter 6

Conclusions and Future Work

The results provided from this study does not imply that the tools are particularly effective in regards to their memory corruption detection capabilities. Flawfinder only managed to detect approximately 23 percent of the total amount of vulnerabilities while Cppcheck managed to detect approximately 10 percent. As discussed in the results section this could very well be due to the tools limitations in their ability to find vulnerabilities caused by the absence of correct processing. According to Owasp[20] source code analysis tools should however be able to find buffer overflow vulnerabilities with high confidence.

There is generally a trade off between multiple factors in the development of source code analysis tools. Oftentimes the results from the tools depends largely on the priorities from its developers since a sound analysis method is able to detect more vulnerabilities but also generates more false positives which then could potentially lead to less usage since it would require manual labour to analyse a huge amount of alerts in order to determine whether or not they are relevant. An unsound analysis method should only generate alerts in case it is absolutely certain that it is an actual concern. This could have potential benefits for software developers who value efficiency more than security although given the core limitations of static analysis this leads to a large amount of false negatives. A tool that tries to fill in the gaps between these methods could potentially add more complexity and may on some instances not be scalable for large software projects .

For future work the author of this thesis recommends a comparison between commercial and non commercial source code analysis tools that are capable of detecting memory corruption vulnerabilities in order to find out if there is any major differences in their analysis methods. This could provide useful information to software companies as to whether or not they should invest money in such a tool or not.

Bibliography

1. Manajit Pal, Prashant Kumar Dey, Memory Corruption- Basic Attacks and Counter Measures, School of Electronics Engineering, KIIT University, India, 2016.
2. https://cwe.mitre.org/top25/archive/2020/2020_top25.html.
3. <https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code>.
4. https://owasp.org/www-community/vulnerabilities/Buffer_Overflow.
5. https://owasp.org/www-community/attacks/SQL_Injection.
6. <https://dwheeler.com/flipfinder/>.
7. <http://cppcheck.sourceforge.net/>.
8. <https://www.chromium.org/Home/chromium-security/memory-safety>.
9. David Gens, Simon Schmitt, Lucas Davi, Ahmad-Reza Sadeghi K-Miner: Uncovering Memory Corruption in Linux, University of Duisburg-Essen, Germany, 2018.
10. <https://cwe.mitre.org/data/definitions/120.html>.
11. <https://cwe.mitre.org/data/definitions/190.html>.
12. <https://cwe.mitre.org/data/definitions/134.html>.
13. <https://cwe.mitre.org/data/definitions/415.html>.
14. <https://cwe.mitre.org/data/definitions/123.html>.
15. <https://cwe.mitre.org/data/definitions/416.html>.
16. <https://samate.nist.gov/SARD/>.
17. <https://samate.nist.gov/SARD/view.php?tsID=108>.
18. Rahma Mahmood, Qusay H. Mahmoud Evaluation of Static Analysis Tools

for Finding Vulnerabilities in Java and C/C++ Source Code, Department of Electrical, Computer Software Engineering University of Ontario Institute of Technology, Oshawa, ON, Canada, 2018.

19. Prof. Arvinder Kaur a , Ruchika Nayyar b A Comparative Study of Static Code Analysis tools for Vulnerability Detection in C/C++ and JAVA Source Code, GGSIPU, Sector 16 Dwarka, New Delhi, 11018, India, 2020.

20. https://owasp.org/www-community/Source_Code_Analysis_Tools.

21. <https://security.web.cern.ch/recommendations/en/codetools/rats.html>.

22. <https://security.web.cern.ch/recommendations/en/codetools/vcg.html>.

23. Alexander Solzhenitsyn st, Leninskie Gory, Buffer Overflow Detection via Static Analysis: Expectations vs. Reality, Ivannikov Institute for System Programming of the Russian Academy of Sciences 25, Moscow, 109004, Russia, Lomonosov Moscow State University, GSP-1, Moscow, 119991, Russia, 2018.

24. James A. Kupsch, Barton P. Miller, Manual vs. Automated Vulnerability Assessment: A Case Study, Computer Sciences Department, University of Wisconsin, Madison, WI, USA. 2009.

25. <http://cppcheck.sourceforge.net/manual.pdf>.

26. <https://cve.mitre.org/about/index.html>.

27. <https://github.com/nccgroup/VCG>

This chapter will provide a list of files that were tested for this thesis with their corresponding test case id. Each test case could be searched for individually on the samaste nist website.

A.1 Buffer overflow

- Test case ID 153829
- Test case ID 153821
- Test case ID 153807
- Test case ID 153805
- Test case ID 153794
- Test case ID 153781
- Test case ID 153772
- Test case ID 153770
- Test case ID 153744
- Test case ID 153741
- Test case ID 153733
- Test case ID 153722
- Test case ID 153716
- Test case ID 153715
- Test case ID 153707
- Test case ID 153696
- Test case ID 153695
- Test case ID 153670

- Test case ID 153669
- Test case ID 153651
- Test case ID 153646
- Test case ID 153645
- Test case ID 153642
- Test case ID 153631
- Test case ID 153630
- Test case ID 153607
- Test case ID 153606
- Test case ID 153600
- Test case ID 153582
- Test case ID 153577
- Test case ID 153550
- Test case ID 153536
- Test case ID 153535
- Test case ID 153532
- Test case ID 153526
- Test case ID 153523
- Test case ID 153520
- Test case ID 153500
- Test case ID 153494
- Test case ID 149201
- Test case ID 149193
- Test case ID 149169
- Test case ID 149167
- Test case ID 149165
- Test case ID 149125
- Test case ID 149123

- Test case ID 149087
- Test case ID 149083
- Test case ID 149081
- Test case ID 149079
- Test case ID 149077
- Test case ID 149069
- Test case ID 149067
- Test case ID 149065
- Test case ID 149059
- Test case ID 149057
- Test case ID 149055
- Test case ID 148953
- Test case ID 148919
- Test case ID 148914
- Test case ID 148896
- Test case ID 2082
- Test case ID 2081
- Test case ID 2065
- Test case ID 2064
- Test case ID 2063
- Test case ID 2062
- Test case ID 1971
- Test case ID 1961
- Test case ID 1958
- Test case ID 1955
- Test case ID 1952
- Test case ID 1903
- Test case ID 1641

- Test case ID 1639
- Test case ID 1637
- Test case ID 1636
- Test case ID 1634
- Test case ID 1632
- Test case ID 1630
- Test case ID 1628
- Test case ID 1626
- Test case ID 1624
- Test case ID 1622
- Test case ID 1620
- Test case ID 1616
- Test case ID 1609
- Test case ID 1607
- Test case ID 1605
- Test case ID 1603
- Test case ID 1600
- Test case ID 1577
- Test case ID 1575
- Test case ID 1572
- Test case ID 1486
- Test case ID 115
- Test case ID 108
- Test case ID 14

A.2 Format string

Test case ID 235337-235328

Test case ID 81951-81942

Test case ID 81844-81835

Test case ID 81416-81407

Test case ID 81279-81270

Test case ID 81031-81022

Test case ID 79117-79100

Test case ID 79455-79446

Test case ID 79551-79542

Test case ID 79919-79910

A.3 Integer overflow

Test case ID 237282-237273

Test case ID 82135-82126

Test case ID 82851-82842

Test case ID 83331-83322

Test case ID 83849-83840

Test case ID 84191-84182

Test case ID 84619-84610

Test case ID 235642-235633

Test case ID 84291-84282

Test case ID 83955-83946

A.4 Double Free

Test case ID 240262-240253

Test case ID 240162-240153

Test case ID 102196-102187

Test case ID 102096-102087

Test case ID 101996-101987

Test case ID 101796-101787

Test case ID 101696-101687

Test case ID 101596-101587

Test case ID 240214-240205

Test case ID 101980-101972

A.5 Use after free

Test case ID 2400400-240391

Test case ID 240309-240301

Test case ID 240300-240291

Test case ID 102639-102630

Test case ID 102539-102530

Test case ID 102439-102430

Test case ID 102344-102335

Test case ID 102510-102501

Test case ID 240380-240371

Test case ID 240272-240263

