Measuring the Effect of Code Complexity on Static Analysis Results

James Walden, Adam Messer, and Alex Kuhl

Department of Computer Science
Northern Kentucky University
Highland Heights, KY 41099

Abstract. To understand the effect of code complexity on static analysis, thirty-five format string vulnerabilities were selected from the National Vulnerability Database. We analyzed two sets of code for each vulnerability. The first set of code contained the vulnerability, while the second was a later version of the code in which the vulnerability had been fixed. We examined the effect of both code complexity and the year of discovery on the quality of static analysis results, including successful detection and false positive rates. The tool detected 63% of the format string vulnerabilities, with detection rates decreasing with increasing code complexity. When the tool failed to detect a bug, it was for one of two reasons: the absence of security rules specifying the vulnerable function, or the presence of a bug in the static analysis tool. Complex code is more likely to contain complicated code constructs and obscure format string functions, resulting in lower detection rates. However, detection rates did not change substantially from 2000 to 2006, showing that reported format string vulnerabilities are not becoming more difficult to find over time.

Key words: static analysis, code complexity

1 Introduction

As an increasing number of vital operations are carried out by software applications, there is growing pressure on the software industry to provide more secure programs. Because of the complexity of software and the rapidly changing nature of vulnerabilities, it is often impractical to identify vulnerabilities through developer code reviews alone. In addition to the time required, many developers lack the expertise to consistently detect the numerous types of vulnerabilities. To help address this problem, developers have been increasingly using static analysis tools to identify security vulnerabilities.

Static analysis is the process of evaluating code without executing it. A wide variety of static analysis tools for assessing code quality exist, but fewer tools focused on detecting security bugs in source code exist. Security-focused tools have matured in recent years. They can detect a variety of types of security bugs and handle large-scale programs, reducing the amount of time that must be spent in manual code inspection.

As a result of this maturity and a growing concern over the security of software, static analysis tools are increasingly used in the development of software. Visual Studio 2005 includes the Prefast static analysis tool[2]. Static analysis tool vendors like Coverity and Fortify perform free analyses of open source software through their web sites[4, 5].

Static analysis tools detect security errors in code. However, they make two types of mistakes in identifying errors: false negatives and false positives. False negatives occur when a security bug exists in the program but the tool does not report its presence. Such results are an obvious problem as the security bug is not found and developers are left with a false sense of security.

When a tool reports the presence of a security bug when none exists in the code, the result is called a false positive. False positives place an additional burden on software developers by requiring that each result be examined to determine whether it is an actual security bug. When the false positive rate is too high, developers can find it difficult to find real security bugs that are hidden in a mass of false positives.

Static analysis tools must make tradeoffs between precision and scalability. High precision tools reduce the number of false positives and false negatives by examining the context of each potential security bug using control flow analysis and data flow analysis techniques. However, the number of potential paths through a program increases exponentially with the number of branches, so very high precision tools take too long to run on large programs. Data flow analysis also becomes time consuming due to pointer aliasing or copying data through a long set of function calls.

To provide scalability, static analysis tools make simplifying assumptions and limit the depth of the analysis. However, these simplifying assumptions can introduce inaccuracies into the analysis, causing the tool to produce false negative or false positive results. One extreme example is lexical analysis which does not consider context in any way, so tools using only this technique report any potentially dangerous function call as a problem, no matter how securely it is used.

We conducted an evaluation that measures detection and false positive rates using a set of thirty-five format string vulnerabilities found in open source software. These vulnerabilities were reported in the National Vulnerability Database from 2000 through 2006. Each vulnerability was analyzed to determine the impact of code complexity on the error rates of static analysis tools. Given the tradeoffs between precision and scalability described above, we expected that error rates would increase with code complexity.

Another goal of this study was to determine whether the probability of finding security flaws with static analysis tools has changed over time. The number of vulnerabilities reported increased rapidly from 2000 through 2006, which could indicate that the number of security bugs remaining in code is decreasing as reported vulnerabilities are fixed. It is reasonable to suppose that bugs discovered earlier in this period are easier to find than bugs discovered later in this period. If this is the case, then static analysis tools should have higher detection rates in

the early years of this period than in the later years of this period. This would result in static analysis detection rates decreasing with the year of discovery of the vulnerabilities.

Several comparative evaluations of static analysis tools have been published[19, 10]. These studies used two techniques: microbenchmarks, which are small programs containing a single security flaw[10], or samples extracted from known security flaws found in open source software[19]. While Kratkiewicz[10] categorized each test case according to local code complexity categories, such as aliasing depth or type of control flow, the samples are too small to demonstrate the complexity found in complete programs. Zitser[19] extracted small samples containing the vulnerabilities from open source software because the tools evaluated in the study could not successfully analyze the complete source code. This study differs from the ones mentioned above by analyzing complete open source applications, allowing us to study static analysis in the conditions under which it is normally used.

2 Format string vulnerabilities

Format string vulnerabilities are a dangerous class of security bug that can result in the execution of arbitrary code supplied by the attacker. They can occur in any language that supports *printf()*-style formatting functions. We chose to study format string vulnerabilities because it is possible to quickly and conclusively determine whether an application contains a format string vulnerabilty through manual inspection of code.

While the first discovery of a format string vulnerability was reported in September 1999[17], the potential danger of format string vulnerabilities was not widely recognized until the discovery of the WU-FTP format string vulnerability in June 2000[1]. Thirty-nine such flaws were reported in 2000. Format string vulnerabilities arise when untrusted input is used as all or part of the format string in formatted output functions like *printf()*, *sprintf()*, *syslog()*, or variants thereof.

Format strings consist of a combination of ordinary characters and format specifiers that begin with a percent sign. They are used as arguments to variadic functions that take a number of arguments corresponding to the number of format specifiers in the format string. Because the compiler cannot verify that the correct number of arguments is used for variadic functions, it is the responsibility of the programmer to ensure that the function cannot receive too few or too many arguments.

If too many arguments are specified for the number of format specifiers, the extra arguments are ignored. If too few arguments are specified, the result is undefined, but in most cases, the function reads additional data from the stack beyond its arguments. An attacker who controls the format string can alter the number of format specifiers to ensure that there are too many format specifiers for the number of arguments.

While attackers can use format string bugs to view data from the stack or to crash an application by causing it to follow an invalid pointer from the stack, the primary danger of these vulnerabilities is that they can allow an attacker to write directly to memory. The %n format specifier does not cause data to be written to the output stream; instead, it takes an integer pointer argument and sets the corresponding value to the number of characters formatted so far. Attackers can exploit this specifier to write arbitrary values to arbitrary locations in memory. Using multiple %n specifiers allows arbitrarily large amounts of data to be written. This capability can be leveraged by an attacker to cause a process to execute attacker controlled code.

The number of reported format string vulnerabilities has increased each year since the first report in 1999, with the exception of 2003, as can be seen in Figure 1. Note that the number of format string vulnerabilities in the graph is multiplied by a factor of 10 to make changes in that number visible when compared with the trend in the overall number of vulnerabilities. In 2003, both the total number of vulnerabilities and the number of format string vulnerabilities reported decreased. However, that decrease is an artifact of a large backlog of unprocessed vulnerabilities that year[3].

The upward trend in the number of format string vulnerabilities resumed after 2003, and the upward trend in the total number of vulnerabilities resumed after 2004. The rate of increase in the total number of vulnerabilities since 2004 has been much faster than the rate of increase of format string vulnerabilities. This difference is the result of an increasing number of vulnerabilities in web applications, such as cross-site scripting and SQL injection.

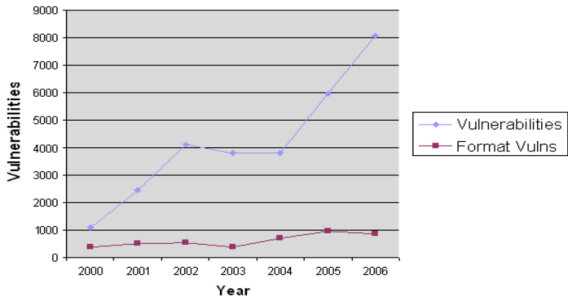

Fig. 1. Format String Vulnerabilities by Year

3 Test cases

Thirty-five format string vulnerabilities in open source Linux software written in C or C++ were selected randomly from the National Vulnerability Database[13]

to evaluate the effectiveness of static analysis. Five vulnerabilities were selected for each year from 2000 through 2006. In order for a test case to be evaluated, source code for both the vulnerable and fixed versions of the software must be available. Furthermore, the software must be able to be compiled with the GNU C compiler on Red Hat Enterprise Linux 4, and it must be possible to run Fortify's Source Code Analyzer on the software. Test cases that did not meet these criteria were replaced with another test case selected randomly. Any piece of software that was selected more than once, even for a different format string vulnerability, was also replaced with another randomly chosen test case.

For each vulnerability, both the version of the software with the reported vulnerability and a later version of the software where the vulnerability was reported to be fixed were evaluated. It was not always possible to find the vulnerable version of the code. Some open source projects remove vulnerable versions of the software from their site once the vulnerability has been fixed to prevent users from downloading dangerous software. Older versions of software are also often removed from sites, so that many of the initially selected samples from 2000 and 2001 had to be rejected because the source code was not available. Source code for some vulnerabilities was retrieved from the Internet Archive[8] when the current project site did not have the source code.

Software was compiled using either gcc 3.4.6 or gcc 3.2.3 on Red Hat Enterprise Linux 4. Some software had to be patched in order to compile with these versions of gcc. Most of the patches were small, fixing minor differences in include files between different versions of Linux or fixing minor differences in gcc, such as older versions of gcc permitting the presence of an empty case at the end of a switch statement[6]. No patch altered lines of code where the format string vulnerabilities existed or where they were fixed. If a software package compiled successfully, there were generally no problems performing a static analysis of the software. Only one test case was eliminated due to the inability to perform a static analysis on it.

Several software packages had multiple entries in the NVD for format string vulnerabilities, most notably Ethereal and Hylafax, with five CVE entries each. Only the first sample selected for such a package was used.

4 Test procedures

The static analysis tool used was Fortify Software's Source Code Analyzer 4.5.0. Older open source static analysis tools, such as flawfinder, ITS4, RATS, were not selected because they rely on simple lexical analysis techniques that produce many false positives[19]. Modern tools, including Fortify, analyze the semantics as well as the syntax of the source code. While numerous modern tools exist, the freely available tools were not suitable for this study, as they were not able to analyze larger pieces of software, did not support common variants of C, or required extensive configuration specific to each piece of software[7].

The analysis of vulnerabilities included both a flawed and a patched version of software. If the flaw was identified in the vulnerable version of software, the

result was recorded as a successful detection. If the flaw was not found, a false negative was recorded. In the analysis of the patched software, detection of the vulnerability constituted a false positive. If no vulnerability was found, the analysis was marked as correct. We also measured the discrimination of the tool, a count of how often the analyzer did not report a vulnerability in the fixed test case when it also did not report the vulnerability in the vulnerable test case[12].

We began each analysis by finding the vulnerability in the National Vulnerability Database[13] and identifying the version of the software that contained the vulnerability and the closest accessible version of the software that had fixed the vulnerability. Some NVD entries provided the complete information necessary to isolate the lines of code containing the vulnerability. However, many other entries only provided the name of the problematic function.

In these instances the source code was examined manually to determine if the vulnerability could be clearly identified. Vulnerable and patched versions of the program file or function were compared to isolate the location of the security bug. In a few cases, there was insufficient information to definitively identify the flaw, so we excluded the vulnerability and selected another to replace it.

Once a vulnerability was isolated, we examined the fixed version of the code to isolate the changes made to fix the vulnerability. No large code changes were observed between the vulnerable and patched versions in most cases, making it trivial to determine if the flawed code snippet had been fixed. In a few of the cases functions were shifted changing line numbers significantly, but the file names and function names remained constant making it easy to confirm the fix in the patched version.

Both versions of the software were compiled using the gcc compiler. Fortify Source Code Analyzer (SCA) relies on the compiler to identify the source code for analysis. Once the source code was compiled, SCA was run with the default rule set. While SCA can be customized to maximize its effectiveness on each particular program, we wanted to evaluate its capabilities in the same way for every program.

To score a successful detection, SCA had to either detect a format string vulnerability in the particular format string function described in the NVD entry or it had to detect a vulnerability in the chain of function calls to led to that format string being called. A false positive result was recorded if SCA continued to detect the same format string vulnerability in the fixed version of the source code. Most format string vulnerabilities are fixed by replacing the dangerous call with a call using a fixed format string. However, some vulnerabilities were fixed by validating the user-controlled format string before the call, and Fortify was unable to detect such fixes.

5 Results

Two complexity measures were computed for each application. The first metric was Source Lines of Code (SLOC), which is the number of lines of code excluding comments or blank lines. We measured SLOC using SLOCCount[18]. The

second metric was McCabe's cyclomatic complexity, the classical graph theory cyclomatic number[11]. The tool used to measure cyclomatic complexity was PMCCABE[14]. We divided the applications into five classes by size and complexity values as described in Figures 2 and 3.

Class	**Lines of Code**	**Samples**	**Detections**
Very Small	< 5000	9	7
Small	5000-25000	9	7
Medium	25,000-50,000	7	4
Large	50,000-100,000	6	2
Very Large	$> 100,000$	4	0

Fig. 2. Size Class Table

Class	**Complexity**	**Samples**	**Detections**
Very Small	< 1000	10	7
Small	1000-5000	10	8
Medium	5000-10,000	5	3
Large	10,000-25,000	6	2
Very Large	$> 25,000$	4	0

Fig. 3. Complexity Class Table

Of the 35 vulnerabilities examined, 22 (63%) were detected by the static analysis tool. Figure 3 shows that detection rates of format string vulnerabilities decreased with increasing code complexity. We found that the detection rates for different classes came from different statistical distributions with a p-value of 0.02. While there was no substantial difference in the quality of static analysis results between the very small and small categories, the quality of results declined noticeably as complexity increased beyond the small category, declining to zero for the very large category, which included software larger than 100,000 lines of code.

We found two causes of failed detections of format string vulnerabilities. The first cause was the use of a format string function that was not in the rule base of Source Code Analyzer. Four of the thirteen (31%) failed detections resulted from this problem. There are hundreds of format string functions, provided by a variety of libraries coming from different sources. One of the format string functions that was not detected was the `ap_vsnprintf()` function from the Apache Portable Runtime. It is impossible for a tool to track all potential format string functions. However, these mistakes could be fixed by the developer adding rules to detect the format string functions that are used in the developer's application.

The second cause was a bug in how Source Code Analyzer counts arguments that are passed into a function using the C language's varargs mechanism. In these cases, the application contains a variadic function that wraps the call to the dangerous format string function. Nine of the thirteen (69%) failed detections resulted from this cause. These mistakes could also be fixed one by one through adding appropriate rules to detect the dangerous functions, but it would also be possible to fix the entire class of flaws by modifying the static analysis tool to correctly analyze data passed through the varargs mechanism. This bug has been reported to Fortify.

Neither of these causes has a necessary relationship to code size or complexity. However, larger projects are more likely to use their own specialized functions for input and output instead of directly using functions from the standard library. This means that larger projects are more likely to call format string functions through wrappers. Large software projects also tend to include a larger number of developers, which typically provides a wide range of knowledge. This knowledge can lead to use of a broader subset of a language's features than would be used in smaller projects, resulting in heavier usage of the varargs feature.

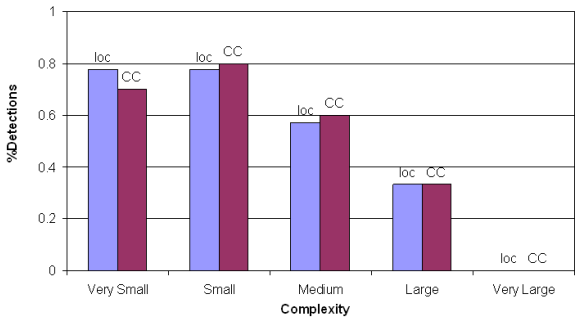

Fig. 4. Detections by Complexity Class

Divided into five classes, there was no significant difference ($p < .01$) between the results for lines of code and those for cyclomatic complexity. Other studies, such as [9], have found a strong correlation between these two complexity metrics.

Discrimination[12], a measure of how often an analyzer passes the fixed test case when it also passes the matching vulnerable test case, is shown in figure 5. This metric provides an important check on the results of a static analysis tool, as a tool can achieve a high detection rate through overeager reporting of bugs, which also produces many false positive results. Discrimination ensures that the tool accurately detected both the vulnerability and its fix. The discrimination graph closely resembles the complexity graph above, as Source Code Analyzer returned only two false positives during the analysis of the 35 fixed samples.

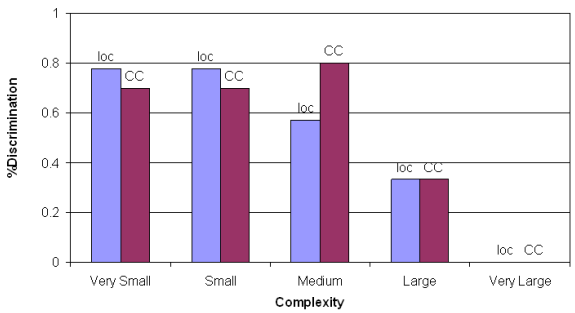

Fig. 5. Discrimination by Complexity Class

We found that the discrimination values for different complexity bins came from different distributions with a p-value of 0.02.

Developers remove security bugs from software as they are discovered. It is likely that vulnerabilities that are easier to find by static analysis tools are discovered earlier than vulnerabilities that such tools cannot detect, for a couple of reasons. First, the effort of developers to find vulnerabilities is hindered by some of the same factors that cause problems for static analysis tools, such as code complexity and the use of obscure language features. Second, static analysis tools have become much more widely used over the years examined in this study. As a result of the remaining vulnerabilities being more difficult to discover, one might expect static analysis detection rates to decrease with the year that a vulnerability is discovered. However, we found no correlation between static analysis detection rates and the year in which the vulnerability was detected as can be seen in Figure 6.

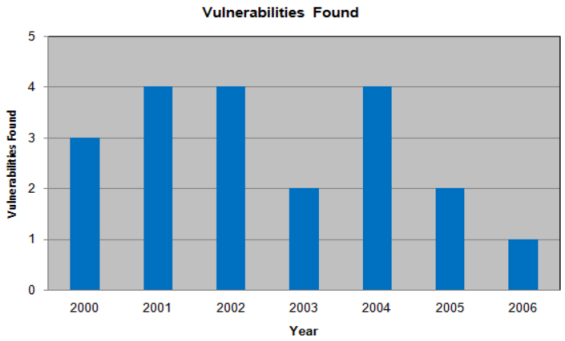

Fig. 6. Vulnerabilities Detected by Year

6 Future Work

We have conducted an experiment on the effect of code complexity on static analysis results. However, there is much yet to be done. In order to determine how far these results can be generalized, we need to measure the effect of code complexity using different types of vulnerabilities and software written in other languages. Analyzing software written in a language such as Java would also enable us to use open source static analysis tools that are not available for C.

To better study the effects of time on the quality of static analysis results, we could study a small number of projects over a number of revisions, measuring the detection rates of vulnerabilities for each revision. We would also like to use more sophisticated statistical analyses for such work in order to disentangle the effects of code complexity from time.

Another area of future interest is the effect of adding customized rules. Is there a set of rules that could be learned from the false negative results in a subset of applications that would substantially improve the detection rates for a particular vulnerability class in other applications? This set of rules will be specific to the vulnerabilities studied and the static analysis tools used.

7 Conclusion

Thirty-five format string vulnerabilities selected from 2000 through 2006 were analyzed to determine the limitations of the usefulness of static analysis tools. We examined the effect of both code complexity and the year of discovery on the quality of static analysis results. Our results show that successful detection rates of format bugs decreased as code complexity increased. Code complexity was measured using both lines of code and cyclomatic complexity.

Static analysis tools are an attractive alternative to formally proving the correctness of a program since it is easier to specify how to detect specific security bugs than it is to generate specifications for the entire program. However, it is still a difficult problem to specify how to detect security bugs. While the concept of format string vulnerabilities is simple, there are hundreds of format string functions, and developers on large projects frequently write their own wrappers for these functions in large programs. A static analysis tool needs to know about all of these functions to detect all format string vulnerabilities.

The other problem is that static analysis tools are large, complex programs that have bugs. While the bug mentioned in this paper will be fixed, there are almost certainly other bugs in Source Code Analyzer. Other static analysis tools will have their own idiosyncratic bugs. Complex programs are more likely to contain unusual code constructs that will trigger new bugs in static analysis tools.

However, there does not appear to be a trend showing that static analysis tools are becoming less useful as time progresses. Detection rates did not change

substantially from 2000 to 2006, showing that format string vulnerabilities are not becoming more difficult for the tool to find over time. Programmers appear to be making the same types of mistakes when using format string functions, as the number of bugs continues to increase and the detection rate of static analysis tools is not decreasing.

Acknowledgements

The authors thank Brian Chess, Maureen Doyle, and Pascal Meunier for their comments on the paper. We also appreciate Steve Christey's help in understanding trends in vulnerability statistics. Finally, we would like to thank Fortify Software for allowing use of their Source Code Analysis tool for this project.

References

1. CERT, `http://www.kb.cert.org/vuls/id/29823`, June 2000.
2. Chess B., West, J.: Secure Programming with Static Analysis, Addison-Wesley, New York (2007)
3. Christey, S.M., personal communication.
4. Coverity, `http://scan.coverity.com/`
5. Java Open Review Project, `http://opensource.fortifysoftware.com/`
6. Gcc 3.4 changes, `http://gcc.gnu.org/gcc-3.4/changes.html`
7. Heffley, J., Meunier, P.: Can Source Code Auditing Software Identify Common Vulnerabilities and Be Used to Evaluate Software Security? In: Proceedings of the 37th Hawaii International Conference on System Sciences. IEEE Press, New York (2004)
8. Internet Archive, `http://www.archive.org/`
9. Herraiz, I., Gonzalez-Barahona, J.M., Robles, G.: Toward a theoretical model for software growth. In:Proceedings of the Fourth International Workshop on Mining Software Repositories. IEEE Press, New York (2007)
10. Kratkiewicz, K., Lippmann, R.: Using a Diagnostic Corpus of C Programs to Evaluate Buffer Overflow Detection by Static Analysis Tools. In:2005 Workshop on the Evaluation of Software Defect Tools. (2005)
11. McCabe, T. J.: A Complexity Measure. In:IEEE Transactions on Software Engineering, vol. 2, no. 4, pp. 308-320. IEEE Press, New York (1976)
12. Newsham, T., Chess, B.: ABM: A Prototype for Benchmarking Source Code Analyzers. In:Workshop on Software Security Assurance Tools, Techniques, and Metrics. U.S. National Institute of Standards and Technology (NIST) Special Publication (SP) 500-265. (2006)
13. NVD, `http://nvd.nist.gov/`
14. PMMCABE, `http://www.parisc-linux.org/~bame/pmccabe/overview.html`
15. Seacord, R.: Secure Programming in C and C++, pp. 203-245. Addison-Wesley, New York (2006)
16. Exploiting Format String Vulnerabilities, `http://julianor.tripod.com/teso-fs1-1.pdf`, March 24, 2001.
17. Exploit for proftpd 1.2.0pre6, `http://seclists.org/bugtraq/1999/Sep/0328.html`, September 1999.
18. SLOCCount, `http://www.dwheeler.com/sloccount/`
19. Zitser, M., Lippmann, R., Leek, T.: Testing Static Analysis Tools using Exploitable Buffer Overflows from Open Source Code. In:SIGSOFT Software Engineering Notes, 29(6):97-106. ACM Press, New York (2004)