Idea: 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 studied. We analyzed two code samples for each vulnerability, one containing the vulnerability and one in which the vulnerability was fixed. We examined the effect of code complexity on the quality of static analysis results, including successful detection and false positive rates. Static analysis 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.

Key words: static analysis, code complexity

1 Introduction

As an increasing number of vital operations are carried out by software, the need to produce secure software grows. Because of the complexity of software and the rapidly changing nature of vulnerabilities, it is impractical to identify vulnerabilities through developer code reviews alone. To help address this problem, developers have been increasingly using static analysis tools to identify security vulnerabilities.

We conducted an experiment that measured detection and false positive rates of static analysis tools using a set of thirty-five format string vulnerabilities from the National Vulnerability Database[6]. Two code samples were analyzed for each vulnerability, one containing the vulnerability and one in which the vulnerability had been fixed. These vulnerabilities were analyzed to determine the impact of code complexity on the error rates of static analysis tools.

Several comparative evaluations of static analysis tools have been published[9, 10, 3]. These studies used two techniques: microbenchmarks, which are small programs containing a single security flaw[3], and samples extracted from known security flaws found in open source software[10]. While Kratkiewicz[3] 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[10] 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 Test Procedures

Thirty-five format string vulnerabilities in open source Linux software written in C or C++ were selected randomly from the National Vulnerability Database. For a test case to be evaluated, source code for both the vulnerable and fixed versions of the software had to be available. The software must also be able to be compiled with the GNU C compiler on Red Hat Enterprise Linux 4. Test cases that did not meet these criteria were replaced with another test case selected randomly.

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. Several software packages had multiple entries in the NVD for format string vulnerabilities. Only the first sample selected for such a package was used, so no software was evaluated twice.

Software was compiled using either gcc 3.4.6 or 3.2.3 on Red Hat Enterprise Linux 4. Some software had to be patched in order to compile with these versions of gcc. The patches fixed differences in include files or C language variants, such as older versions of gcc permitting the presence of an empty case at the end of a switch statement[1]. No patch altered lines of code where the selected vulnerabilities existed or where they were fixed.

We used Fortify Software's Source Code Analyzer 4.5.0. Older open source static analysis tools, such as flawfinder and ITS4, were not selected because they rely on simple lexical analysis techniques that produce many false positives[10]. 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[2].

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. If the flaw continued to be detected in the patched software, a false positive was recorded. If no vulnerability was found in the patched version, the analysis was marked as correct.

3 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[8]. The second

metric was McCabe's cyclomatic complexity[4]. The tool used to measure cyclomatic complexity was PMCCABE[7]. We divided the applications into five classes by size and complexity values as described in Fig. 1.

Class	**Lines of Code**	**Samples**	**Detections**	**Complexity**	**Samples**	**Detections**
Very Small	< 5000	9	7	< 1000	10	7
Small	5000-25000	9	7	1000-5000	10	8
Medium	25,000-50,000	7	4	5000-10,000	5	3
Large	50,000-100,000	6	2	10,000-25,000	6	2
Very Large	$> 100,000$	4	0	$> 25,000$	4	0

Fig. 1. Size and Complexity Class

Of the 35 vulnerabilities examined, 22 (63%) were detected by the static analysis tool. Fig. 2 shows that detection rates of format string vulnerabilities decreased with increasing code complexity ($p = 0.02$). 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. Four of the thirteen (31%) failed detections resulted from the first cause, format string functions that were not in the rule set of SCA. 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. 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 language's 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.

Divided into five classes, there was no significant difference ($p < .01$) between the results for lines of code and those for cyclomatic complexity.

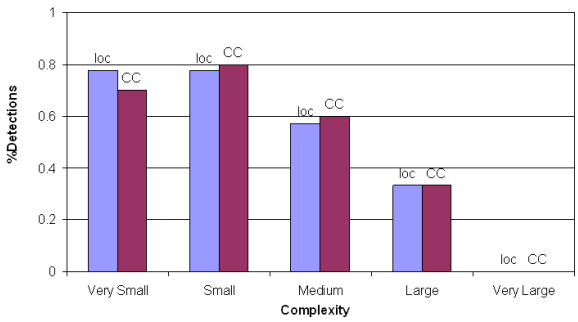

Fig. 2. Detections by Complexity Class

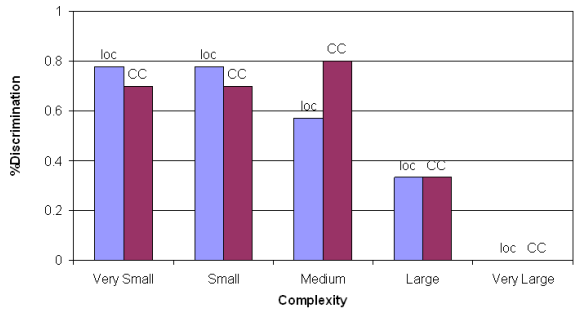

Fig. 3. Discrimination by Complexity Class

Discrimination[5], a measure of how often an analyzer passes the fixed test case when it also passes the matching vulnerable test case, is shown in Fig. 3. 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. Like detection rate, discrimination rate decreases with complexity ($p = 0.02$).

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 a broader range of open source static analysis tools.

4 Conclusion

Thirty-five format string vulnerabilities were analyzed to determine the limitations of the usefulness of static analysis tools. We examined the effect of code complexity, measured using lines of code and cyclomatic complexity, on the quality of static analysis results. Our results show that detection rates of format bugs decreased with increasing code complexity. There were two reasons why the tool failed to detect vulnerabilities: use of format string functions absent from the tool's rule set and a bug in processing parameters submitted to variadic functions.

Acknowledgements

The authors thank Brian Chess, Maureen Doyle, and Pascal Meunier for their comments on the paper. We would also like to thank Fortify Software for permitting use of their Source Code Analysis tool for this project.

References

1. Gcc 3.4 changes, `http://gcc.gnu.org/gcc-3.4/changes.html`
2. 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)
3. 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)
4. McCabe, T. J.: A Complexity Measure. In:IEEE Transactions on Software Engineering, vol. 2, no. 4, pp. 308-320. IEEE Press, New York (1976)
5. 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)
6. NVD, `http://nvd.nist.gov/`
7. PMMCABE, `http://www.parisc-linux.org/~bame/pmccabe/overview.html`
8. SLOCCount, `http://www.dwheeler.com/sloccount/`
9. Wilander, J., Kamkar, M.: A Comparison of Publicly Available Tools For Static Intrusion Prevention. In:Proceedings of the 7th Nordic Workshop on Secure IT Systems, pp. 68-84. Karlstad, Sweden (2002)
10. 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)