

Is Complexity Really the Enemy of Software Security?

Yonghee Shin

Department of Computer Science
North Carolina State University
Raleigh, NC 27695
yonghee.shin@ncsu.edu

Laurie Williams

Department of Computer Science
North Carolina State University
Raleigh, NC 27695
williams@csc.ncsu.edu

ABSTRACT

Software complexity is often hypothesized to be the enemy of software security. We performed statistical analysis on nine code complexity metrics from the JavaScript Engine in the Mozilla application framework to investigate if this hypothesis is true. Our initial results show that the nine complexity measures have weak correlation ($\rho=0.30$ at best) with security problems for Mozilla JavaScript Engine. The study should be replicated on more products with design and code-level metrics. It may be necessary to create new complexity metrics to embody the type of complexity that leads to security problems.

Categories and Subject Descriptors

D.2.8 [Software Engineering]: Complexity measures, Product metrics

General Terms

Measurement, Reliability, Security.

Keywords

Software metrics, security metrics, complexity, reliability, fault prediction, vulnerability prediction

1. INTRODUCTION

Software security is the ability to defend attacker's exploitation of software problems by building software to be secure throughout the whole development life cycle [4]. Software complexity is often hypothesized to be the enemy of software security [8]. The wisdom of security experts is that complexity leads to security problems [3, 8]. Geer [3] stated that "complexity provides both opportunity and hiding places for attackers" and "security failures come from it [complexity] as surely as dawn comes from the east" when he addressed the importance of cybersecurity in a hearing at the Subcommittee of Homeland Security on Emerging Threats, Cybersecurity, and Science and Technology on 23rd, April, 2007. McGraw [8] also points out complexity as one of three major causes of software security problems (the "Trinity of Trouble"); the other causes

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

QoP'08, October 27, 2008, Alexandria, Virginia, USA.

Copyright 2008 ACM 978-1-60558-321-1/08/10...\$5.00.

together are connectivity and extensibility.

The wisdom of these experts, though, has not been substantiated by empirical evidence using quantifiable metrics in terms of software security. However, we cannot control what we cannot measure [2]. Geer [3] also emphasized that a system of security metrics is in the first priority among the tasks for cybersecurity. Software complexity may be related to security problems or may not. If an empirical relationship can be discovered between software security metrics and security problems at any level (e.g. code, design, or architecture level), these metrics could aid organizations in their efforts to fortify their products early in the development lifecycle.

The study of the relationship between software complexity and vulnerabilities is similar to the study of the relationship between software complexity and faults. A fault is an accidental condition that causes a functional unit to fail to perform its required function [5]. A software vulnerability is an instance of a [fault] in the specification, development, or configuration of software such that its execution can violate an [implicit or explicit] security policy [7]. Previous studies have shown that complexity is related with software faults [6, 9]. However, other studies argue that the currently-known complexity metrics seem not to be good indicators of faults. The previous studies [6, 9] can be replicated in the context of vulnerabilities. However, the results might not be necessarily the same as fault prediction, because even though vulnerabilities are a subset of faults [10], the differences in the characteristics of vulnerable code and faulty code have not been investigated quantitatively yet.

The objective of this research is to explore if more complex code is less secure.

We conducted a case study on the JavaScript Engine (JSE) in the Mozilla application framework¹ to identify the relationship between complexity and vulnerabilities.

2. METHODOLOGY

We examine the following two hypotheses in this study.

H1: More complex code has more discovered vulnerabilities.

H2: Vulnerable code has different complexity from faulty code.

If H1 is true, complexity metrics might be helpful to determine how to allocate testing resources. If H2 is true, complexity metrics that can explain faults are not necessarily also can explain vulnerabilities. Therefore, using metrics that are appropriate for vulnerability testing might be more beneficial

¹ <http://www.mozilla.org>

than using the same metrics used to find faults. To test these hypotheses, we collected complexity metrics and vulnerability information from the Mozilla JSE, an open source project, and applied statistical analysis. We chose the Mozilla JSE for our case study because the source code, faults, and vulnerability information are publicly available and the sample size of faults and vulnerabilities reported were large enough for our initial statistical analysis. We performed our analysis at function level.

2.1 Complexity Metrics Collection

We collected nine code complexity metrics using a commercial metrics tool, Understand C++²: (1) McCabe’s cyclomatic complexity, the number of decision statements plus one; (2) modified cyclomatic complexity and (3) strict cyclomatic complexity which are the same as McCabe’s cyclomatic complexity except for the differences in the way that switch statements and conditions in decision statements are counted; (4) essential cyclomatic complexity, a measure of the structuredness of code by counting cyclomatic complexity after iteratively replacing all structured programming primitives to a single statement; (5) nesting complexity, a measure of the deepest level of nested control constructs; (6) paths, the number of possible paths; (7) SLOC, the number of lines of source code excluding comments; (8) SLOC_exe, the number of SLOC excluding declarations; and (9) Stmt_exe, the number of executable statements.

2.2 Faults and Vulnerability Collection

The faults in the Mozilla JSE can be found from the Bugzilla³, a bug tracking system. A bug is a fault in code. The Bugzilla records include a description of identified bugs, related components, developers, and current status of bug fixes and verification. Bugzilla also provides a link to modified code for each bug fix. Figure 1 shows a screen shot of a bug report in the Bugzilla. The bug report tells that the bug 319872 was found from JSE component and the bug fix was verified for the release 1.8.0.1 and the release 1.8.0.2. The changed code can be found by following the *Diff* link.



Figure 1. An example of a bug report in the Bugzilla

² <http://www.scitools.com/>

³ <http://bugzilla.mozilla.org>

The mitigated vulnerabilities in the Mozilla JSE are posted to the Mozilla Foundation Security Advisories (MFSA)⁴ and also listed in Common Vulnerabilities and Exposures (CVE)⁵. Each MFSA has links to one or more bug reports in the Bugzilla. The total number of bug reports in Bugzilla was 51370 (as of 29th, February, 2008) and the number of bug reports linked from the MFSA was 458. The percentage of vulnerabilities among the total bug reports is around 0.89%. There were 106 bug reports on the JSE linked from the MFSA and 15 of them were not accessible from the bug reports due to the security policy of the Mozilla project. These could not be included in our analysis. Figure 2 shows an example of an MFSA. The vulnerability in bug 319872 was on integer overflows found from the Firefox 1.5.0.1 that uses JSE as one of its components. An integer overflow occurs when a numeric value is stored to a storage that is smaller than the value. As a consequence, incorrect value will be stored which could lead to an incorrect financial transaction or unexpected behavior of a system such as denial of service.

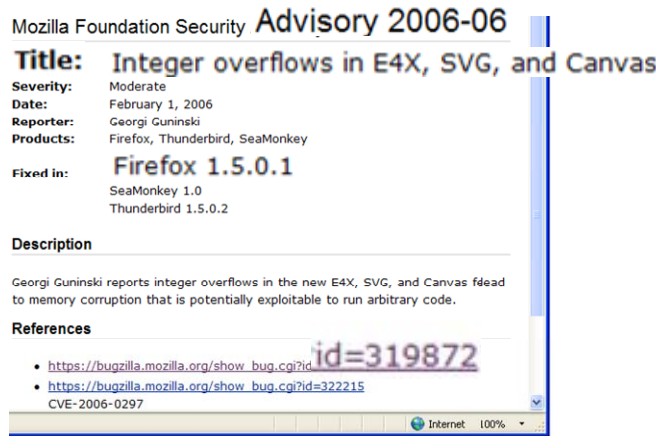


Figure 2. An example of an MFSA

The MFSA report vulnerabilities with product names such as Firefox instead of component names included in the products such as JSE and all the vulnerabilities fixes for JSE were incorporated into Firefox. Therefore, we use release numbers of Firefox in this paper instead of using the release numbers for JSE. At the time of data collection (November, 2007), 28 releases of Firefox had fixes for vulnerabilities starting from Release 1.0 to Release 2.0.0.8. Among these release we chose four releases (R1.0.2, R1.5, R2.0, and R2.0.0.7) that represent each major releases and the last release that had vulnerabilities.

To identify faulty and vulnerable functions, we counted the number of functions that were changed due to faults and vulnerabilities. We know in which release a vulnerability was fixed, however, we do not know when the vulnerability was introduced. Therefore, when there is a vulnerability fix in a release of a function (e.g. v1.5.0.8), we assumed that the same vulnerability existed in all the previous releases of the function (e.g. v1.5.0.7, v1.5.0.6, etc.). After all, 0.5% to 16.6% of functions were changed due to identified faults, and 0.4% to

⁴ <http://www.mozilla.org/projects/security/known-vulnerabilities.html>

⁵ <http://cve.mitre.org/>

9.4% of functions due to vulnerabilities. Table 1 shows the summary of the facts for those releases including the number of functions, lines of source code, the number of functions with faults, and the number of functions with vulnerabilities.

Table1. JavaScript Engine Information

Releases	No of Fun.	LOC	Fun. with faults	Fun. with vulnerabilities
R1.0.2	1352	39,282	224 (16.6%)	125 (9.4%)
R1.5	1680	50,230	289 (17.2%)	148 (8.8%)
R2.0	1848	56,881	172 (9.3%)	61 (3.3%)
R2.0.0.7	1864	57,420	9 (0.5%)	8 (0.4%)

2.3 Analysis Methods and Evaluation

Criteria

H1 can be tested by computing the correlations between complexity measures and the number of vulnerabilities in each function. Pearson correlation coefficient (r) and Spearman rank correlation coefficient (ρ) are the statistics often used to measure the strength of correlations between two variables. Pearson correlation coefficient assumes normal distribution of data, while Spearman rank correlation coefficient is a non-parametric test that does not assume any distribution. Spearman rank correlation is performed on the ranks of the values without considering the magnitudes of the values, and therefore, Spearman rank correlation coefficient is not sensitive to outliers. Because our data was not normally distributed, we used Spearman rank correlation coefficient in this study.

When the value of a Spearman rank correlation is negative, the two data sets are negatively associated (when the values of one data set go up, the values of the other data set go down); and the positive value of a Spearman rank correlation indicates that two data sets are positively associated. The magnitudes of the values tell the strengths of correlations. Cohen [1] suggests that less than 0.3 value of correlation coefficient means weak correlation, 0.3 to 0.5 means medium correlation, and greater than 0.5 means strong correlation. However, the interpretation depends on the context of the usage of correlation. Nagappan et al. [9] considered correlation over 0.4 strong in their study on software failure prediction.

H2 can be tested by comparing the means or ranks of the complexity measures between vulnerable code and faulty code. T-test and Wilcoxon rank sum test [2] are representative statistical methods used to compare the means or ranks of two groups. Similar to Spearman rank correlation, Wilcoxon rank sum test is a non-parametric test that is not sensitive to outliers and does not assume any distribution of sample data. Therefore, we use Wilcoxon rank sum test instead of using t-test.

3. RESULTS & CONCLUSION

To test H1, we computed the Spearman rank correlation coefficients between the measures of complexity and the number of vulnerabilities. Table 2 presents the results. The correlations that are statistically significant at 95% confidence level are presented in bold face. The reason that the correlation at Release 2.0.0.7 is not significant might be because the number of vulnerabilities is too small. Table 2 shows that there

are weak correlations between vulnerabilities and complexity measures. The reason might be because there is not actually strong correlation between complexity measures and vulnerabilities. Another possible reason of low correlation might be because we assumed that all the functions that have been changed due to a vulnerability are vulnerable. However, even though a set of functions changed together, that does not mean all the functions are directly related with the vulnerability. Changes for functional enhancement regardless of vulnerabilities might have been occurred together with vulnerability fixes.

Table 2. Correlation between complexity and vulnerabilities

Complexity metrics	R1.0.2	R1.5	R2.0	R2.0.0.7
McCabe’s cyclomatic	0.300	0.236	0.175	0.044
Modified cyclomatic	0.300	0.236	0.174	0.045
Strict cyclomatic	0.295	0.231	0.170	0.045
Essential cyclomatic	0.307	0.233	0.160	0.022
Nesting	0.280	0.231	0.178	0.055
Paths	0.300	0.238	0.172	0.044
SLOC	0.292	0.240	0.171	0.030
SLOC_exe	0.288	0.240	0.170	0.031
Stmt_exe	0.292	0.240	0.175	0.037

Table 2 also shows that the older versions have stronger correlation than the later versions. The reason might be because we assumed that when there is a vulnerability fix in a release of a function, the same vulnerability exists in all the previous releases of the function. More precise analysis would be necessary to identify when the vulnerability has been introduced. We are currently investigating the ways to count vulnerabilities more precisely. A technique that identifies code changes at function level using the CVS repository such as APFEL [11] might be useful.

To test H2, we performed Wilcoxon rank sum test on the vulnerable functions and the faulty-but-non-vulnerable functions. Table 3 shows the p-values from two-sided Wilcoxon rank sum test. The p-values that are less than 0.05 are presented in bold face.

Table 3. P-values from Wilcoxon rank sum test

Complexity metrics	R1.0.2	R1.5	R2.0	R2.0.0.7
McCabe’s cyclomatic	< .0001	< .0001	0.0921	0.2724
Modified cyclomatic	< .0001	< .0001	0.1224	0.2724
Strict cyclomatic	< .0001	< .0001	0.1334	0.2724
Essential cyclomatic	< .0001	< .0001	0.5798	0.4855
Nesting	< .0001	< .0001	0.0169	0.2636
Paths	< .0001	< .0001	0.1733	0.2724
SLOC	0.0005	< .0001	0.3234	0.1904
SLOC_exe	0.0005	< .0001	0.3019	0.1904
Stmt_exe	0.0006	< .0001	0.2465	0.1904

The measures of complexity for the vulnerable functions and the faulty-but-non-vulnerable functions in Release 1.0.2 and Release 1.5 were significantly different in the nine complexity

metrics. However, in Release 2.0, the measures of complexity for the vulnerable functions and the faulty-but-non-vulnerable functions were significantly different only in the nesting complexity. This result indicates that the nesting complexity can be a differentiating factor of vulnerable functions from faulty functions in JSE. Therefore, giving more attention to highly nested functions than to other functions in security inspection could be an efficient strategy. There was only 1 faulty-but-non-vulnerable function in Release 2.0.0.7. Therefore, drawing any conclusion from Release 2.0.0.7 is difficult.

Even when the results of Wilcoxon rank sum test were not significant, the means and the medians of all the nine metrics for vulnerable functions were equal to or higher than both the faulty functions and the faulty-but-non-vulnerable functions. Table 4 shows the means and medians for the cyclomatic complexity

Table 4. Means and Medians of the cyclomatic complexity

Releases	Criteria	Vulnerable functions	Faulty functions	Faulty-but-non-vulnerable functions
R1.0.2	Mean	32.48	24.52	14.46
	Median	11	8	4
R1.5	Mean	33.42	24.50	15.14
	Median	13	9	5
R2.0	Mean	63.02	34.30	18.51
	Median	12	9	8
R.2.0.0.7	Mean	21.43	18.88	1.00
	Median	9	7	1

Table 5 shows the means and medians for the nesting complexity.

Table 5. Means and Medians of the nesting complexity

Releases	Criteria	Vulnerable functions	Faulty functions	Faulty-but-non-vulnerable functions
R1.0.2	Mean	2.90	2.55	2.11
	Median	2	2	1
R1.5	Mean	3.21	2.73	2.23
	Median	3	2	2
R2.0	Mean	3.33	2.84	2.58
	Median	3	2	2
R.2.0.0.7	Mean	3.14	2.75	0.00
	Median	4	3	0

The above results show that vulnerable functions are more complex than faulty functions. This fact indicates that the fault prediction models that use complexity metrics also might be useful for vulnerability prediction in general. However, some metrics such as nesting complexity metrics might be more effective to locate vulnerable code than to locate faulty code.

To summarize, the results of our study show weak evidence that software complexity is the enemy of software security for the nine complexity metrics we collected. However, vulnerable code seems to be more complex than faulty code. Before we make any conclusion, we will examine more precise way to count vulnerabilities and analyze Mozilla JSE again and will analyze other projects. We will also include more design and code-level metrics available and will devise new metrics.

4. ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation Grant No. 0716176. Any opinions expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

5. REFERENCES

- [1] Cohen, J., *Statistical Power Analysis for the Behavioral Sciences*, 2nd ed. Hillsdale, NJ: Lawrence Erlbaum Associates, 1988.
- [2] Fenton, N. E. and Pfleeger, S. L., *Software Metrics: A Rigorous and Practical Approach*, 1997.
- [3] Geer, D. E., "A Witness Testimony in the Hearing, Wednesday 25 April 07, entitled Addressing the Nation's Cybersecurity Challenges: Reducing Vulnerabilities Requires Strategic Investment and Immediate Action," *submitted to the Subcommittee on Emerging Threats, Cybersecurity, and Science and Technology*, 2007.
- [4] Hoglund, G. and McGraw, G., *Exploiting Software: How to Break Code*. Boston: Addison-Wesley, 2004.
- [5] IEEE, "IEEE Std 982.1-1988 IEEE Standard Dictionary of Measures to Produce Reliable Software," The Institute of Electrical and Electronics Engineers, June 9, 1988.
- [6] Khoshgoftaar, T. M., Allen, E. B., Kalaichelvan, K. S., and Goel, N., "Early Quality Prediction: A Case Study in Telecommunications," *IEEE Software*, vol. 13, no. 1 pp. 65 - 71, Jan., 1996.
- [7] Krsul, I. V., *Software Vulnerability Analysis*, PhD Thesis, West Lafayette, Purdue University, 1998.
- [8] McGraw, G., *Software Security: Building Security In*. Boston, NY: Addison-Wesley, 2006.
- [9] Nagappan, N., Ball, T., and Zeller, A., "Mining Metrics to Predict Component Failures," in *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*, Shanghai, China, May 20-28, 2006, pp. 452-461.
- [10] Viega, J. and McGraw, G., *Building Secure Software*. Boston, NY: Addison-Wesley, 2002.
- [11] Zimmermann, T., "Fine-grained processing of CVS archives with APFEL," in *Proceedings of Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange*, Portland, Oregon, 2006, pp. 16 - 20.