Effect of Static Analysis Tools on Software Security: Preliminary Investigation

Vadim Okun William F. Guthrie Romain Gaucher Paul E. Black
National Institute of Standards and Technology
Gaithersburg, MD 20899, USA
{vadim.okun, will.guthrie, romain.gaucher, paul.black}@nist.gov

ABSTRACT

Static analysis tools can handle large-scale software and find thousands of defects. But do they improve software security?

We evaluate the effect of static analysis tool use on software security in open source projects. We measure security by vulnerability reports in the National Vulnerability Database.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics – *product metrics*; D.2.4 [**Software Engineering**]: Software/Program Verification; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms

Measurement, Security

Keywords

Software Security, Static Analysis Tools, Vulnerability

1. INTRODUCTION AND RELEVANCE

Security vulnerabilities are discovered every day in commonly used software. The current publication rate in the National Vulnerability Database (NVD) [11] is about 20 vulnerabilities per day. These vulnerabilities may lead to costly security failures.

Since roughly half of all security defects are introduced at the source code level [10], coding errors are a critical problem. The ability of static analysis to infer information about a program without execution makes it a good complement to testing to discover defects in source code. This paper is concerned with static analysis tools that work on source code (as opposed to other artifacts) and look for defects that may affect security [4]. Such tools are mature: they can handle large-scale software and find thousands of defects. But does their use improve software security?

One can think of several potential problems with the use of such tools in practice. A tool may report many defects, but miss the small number of serious security defects. If a developer takes a mechanical approach to fixing defects reported by tools, he may not think as much about the program logic and miss more serious flaws. Also, the developer may spend time identifying false positives (correct code reported as a defect) and correcting unimportant defects reported, making other mistakes in the process and neglecting harder security challenges. Recognizing such problems, Dawson Engler asked the question: "Do static analysis tools really help?" [6].

QoP'07, October 29, 2007, Alexandria, Virginia, USA.
ACM 978-1-59593-885-5/07/0010.

While Engler's question concerns both security and quality, we are primarily interested in its security aspect. The goal of this study is to *evaluate the effect of tools on software security*. We examine this relationship on open source software projects. We measure security by vulnerability reports in the NVD.

1.1 Related Studies and Experiments

A number of studies have compared different static analysis tools for finding security defects, e.g., [13].

Researchers have evaluated security by looking at number of reported vulnerabilities or failures. Zheng et. al [15] analyzed the effectiveness of static analysis tools by looking at test and customer-reported failures for three large-scale network service software systems. They conclude that static analysis tools are effective at identifying code-level defects.

Ozment and Schechter [12] examined the code base of OpenBSD to determine whether its security is improving. They measured the rate of vulnerability reports and found that, for foundational vulnerabilities (introduced prior to the period covered by the study), it decreases slowly. Our study also looks at vulnerability reports, but our goal is to establish the effect of tools.

1.2 Definitions

The following definitions are adapted from [3]. Any event which is a violation of a particular system's security policy is a *security failure*, or simply a *failure*. A *vulnerability* is a property of system security requirements, design, implementation, or operation that could be accidentally triggered or intentionally exploited and result in a security failure. A vulnerability is the result of one or more *weaknesses* in requirements, design, implementation, or operation. Sometimes we use term defect to refer to code weakness.

2. MEASURING SOFTWARE SECURITY

There are many possible measures of software security.

Counting the number of weaknesses is not satisfactory because some weaknesses can never result in a failure. Exploits of vulnerabilities are more concrete, but are not widely reported and are affected by adversaries' tactics, which may change dramatically.

We chose to use the number of reported vulnerabilities as a measure of security because a vulnerability is a real problem and it depends less on the choice of the adversary.

Number of vulnerabilities that remain in a program is a more useful measure than number of discovered vulnerabilities, but it is unknown, except for trivial programs. To estimate it, [2] proposed vulnerability discovery models, similar to software reliability growth models used by the dependability community.

Thus our goal can be restated as being: *evaluate the effect of tool use on the number of reported vulnerabilities*.

3. STUDY DETAILS

In order to discern the effect of tool use, we compare rates of vulnerability reports:

- before and after the introduction of a static analysis tool
- between projects that use tools and those that do not

3.1 Data Collection

We use vulnerabilities reported in the NVD as an indicator of software security. The NVD has thousands of vulnerabilities, mostly from CVE [5]. The database lists vulnerable software and versions along with other information. For example, in 2006 there were 103 vulnerability reports for Firefox and two for Python. The NVD contents are available for download as XML files; we wrote scripts to parse and analyze the files.

Several assessments of open-source projects by static analysis tools have been reported recently [1][7][8][9]. In particular, Coverity[1], with Stanford University, began using its Prevent tool to analyze dozens of open-source projects in March 2006 [1], resulting in the identification of many weaknesses. For example, the Coverity scan reported over 600 defects in Firefox and over 70 defects in Python. The scan is ongoing. More details of the scan are in Section 4.2. We use project web sites and mailing lists to determine when fixes based on tool reports are made.

3.2 Confounding Factors

There are many confounding factors that may affect our conclusions, some of which we list here.

Our data is based mostly on the use of Coverity tool and therefore may not be representative of all static analysis tools.

Accurate vulnerability discovery dates may be hard to obtain. First, a vulnerability may be reported long after it was discovered. Second, prior to July 2005, vulnerability publication date in the NVD represented the date when the vulnerability was analyzed, which may be later than it was reported to CVE. For example, the number of vulnerability reports in the NVD in May 2005 was about 10 times as high as in the previous month. Some of those vulnerabilities were actually reported in the preceding months.

Additionally, the NVD often does not have accurate information about which versions of the software are vulnerable.

For some projects, fixes may come long after the tool feedback.

An increase in the program size or the user base can increase the number of discovered vulnerabilities without an actual change in security.

[1] Any commercial product mentioned is for information only. It does not imply recommendation or endorsement by NIST nor does it imply that the products mentioned are necessarily the best available for the purpose.

Some vulnerabilities may be discovered and reported through the use of the tool itself. If not clearly identified, these vulnerabilities could bias our evaluation of the tool effects.

Extrapolation of results is difficult because the scanned software is not a random sample of projects. Also, the scope of the study is limited to open-source projects.

Currently, the severity of vulnerabilities is not considered.

The rate of vulnerability reports may depend on the time of the year (seasonal effects).

A major redesign may have a larger effect than the use of tools. Also, project developers may be using other static analysis tools.

4. DATA ANALYSIS

4.1 Before vs. After Tool Use

We analyzed vulnerability reports for two projects: MySQL and Samba. We call *version X* the first version of a project that contains fixes based on static analysis tool reports.

Coverity scanned MySQL version 4.1.8 in early 2005 [9]. Version 4.1.10 (version X), released 15 Feb 2005, contains fixes based on Coverity reports.

Coverity [1] and Klocwork [7] scanned Samba in the first half of 2006. Samba version 3.0.23 (version X), released 10 July 2006, contains fixes based on Coverity and Klocwork reports.

Table 1 compares vulnerabilities discovered in version X or later versions (the "after fix" row) with vulnerabilities discovered before version X (the "before fix" row). "Discovery" means it was reported in the NVD. We present data for four periods: six months immediately before the version X release, six months immediately after the version X release, and twelve months immediately before and after. The 12-month periods include the corresponding 6-months periods. The before period includes the date of release of version X.

Table 1. Number of vulnerability reports before and after fix

Project		**# vuln-s per period length**	
		6-month	**12-month**
MySQL	Before fix	9	12
	After fix	5	8
Samba	Before fix	2	2
	After fix	0	6

The choice of period length is important. A 6-month period minimizes effects of changes in project size and user base. A 12-month period has the advantage of controlling for seasonal effects.

For MySQL, we used as the discovery date the earlier of the discovery dates in the NVD and in the SecurityFocus database [14]. There was one vulnerability that was discovered after the release of version X, but was only present in versions before version X; this vulnerability was omitted from the counts.

For Samba, we used as the discovery date the earlier of the discovery dates in the NVD and on the Samba web site.

The data for a 6-month period suggest some positive impact from tool use, while the data for a 12-month period suggest some negative impact. More data will be necessary to draw definitive conclusions.

4.2 Projects with and without Tool Use

Figure 1 compares aggregates of vulnerability reports for projects scanned by Coverity (left axis) with the aggregates for all other projects in the NVD (right axis).

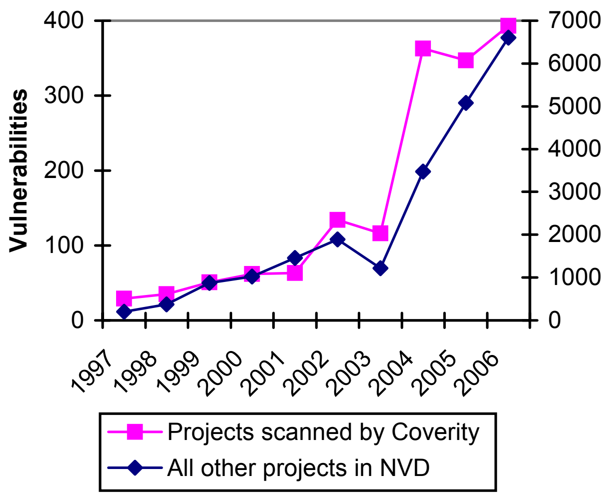

Figure 1. Vulnerabilities in the NVD during 12-month periods

Of the projects scanned by Coverity, we chose 45 which are being scanned beginning in March 2006 and for which a meaningful number of weaknesses have been verified and/or fixed by the developers. Each point on the plot contains the number of vulnerabilities discovered in the 12 months starting June 1, e.g., in 2006 the period is from 1 June 2006 to 30 May 2007. We chose June 1 as the start of the period to better show the effect of tool scans and allow for delay in fixes.

Figure 1 shows that the two curves are similar. This suggests that the set of projects, chosen for the Coverity scan, has similar characteristics to the rest of software projects. The divergence between the two curves in the last years may be due to confounding factors, such as the rapid increase in popularity of Firefox and some other projects. The drop in vulnerability reports for 2003 is attributed to a low output for CVE that year.

Table 2 is a stem-and-leaf plot that presents the changes in the numbers of vulnerability reports for the 45 projects from the Coverity scan in two most recent 12-month periods (1 June 2006 to 30 May 2007 and 1 June 2005 to 30 May 2006). The choice of June 1, as opposed to May 1, as the start of the period allowed us to exclude the effect of an unusually high number of vulnerability reports in the NVD in May 2005 (see Section 3.2). Mapping project names in the Coverity scan to the corresponding names in the NVD involved human judgement and may have introduced errors. For some projects, there were multiple names in the NVD, either because a project was renamed or because of the inconsistent data entry in the NVD.

A stem-and-leaf plot is similar to a histogram, but it has an advantage of displaying individual values. In this plot, the leaf is the last digit of a number; the other digits to the left of the leaf form the stem. A positive number represents increase in the number of vulnerabilities. The stem of 0 is for numbers between 0 and 9, the stem of -0 is for numbers between 0 and -9. Values that are exactly 0 (no change in the number of vulnerabilities) are split as evenly as possible between the "0" and "-0" rows. For example, the "-1" row represents the number -10 (the number of vulnerabilities decreased by 10). There were no values in the 50s, 40s, 30s, -20s, or -30s, so those rows are omitted. The top row of the table represents the biggest increase in the number of vulnerabilities, from 24 to 90, which occurred for PHP. The bottom row represents the biggest decrease, from 119 to 73, which occurred for Linux kernel. The table shows a nearly symmetric distribution, with slightly more increases than decreases, which suggests no positive effect from tool use on the number of vulnerabilities reported for a given project over time.

Table 2. Changes in the numbers of vulnerability reports in two most recent 12-month periods

Stem	Leaf
6	6
…	
2	2
1	0
0	0 0 0 0 0 0 0 0 1 1 1 1 1 2 2 3 3 3 3 6 7
-0	0 0 0 0 0 0 0 0 0 1 1 1 1 2 2 2 4 8 8
-1	0
…	
-4	6

To try to ensure that the changes observed in the number of vulnerability reports were not masked by the general increase in the number of vulnerabilities reported over time, we compared the results for the scanned projects to similar sets of projects drawn at random from the NVD. In these comparisons 40 of the 45 projects in Table 2 were used, omitting 5 projects that had multiple corresponding names in the NVD. The two 12-month periods used in these comparisons are the same as in Table 2.

The specific quantities compared for the scanned projects versus the randomly sampled projects included the average change in the number of reported vulnerabilities for sets of 40 projects and the percentage of projects for which the number of reported vulnerabilities decreased over time. Figure 2 and Figure 3 show histograms of the results for 1000 sets of 40 projects sampled completely at random from the NVD. The analogous results for the 40 scanned projects are indicated by the dashed line shown on each histogram.

As indicated by Figure 2, the distribution of the average change in the reported number of vulnerabilities is significantly less for the randomly selected sets of projects than for the scanned projects. The scanned projects had about one more vulnerability on average after scanning than was reported before scanning, while the distribution of average change is centered near zero for the randomly selected sets of projects. We cannot necessarily conclude that scanning increases the number of vulnerabilities, however, because the scanned projects may differ in some critical way from the projects used to construct the baseline distribution.

Figure 3 shows the percentage of projects which had a decrease in the number of reported vulnerabilities. In this case, however, the

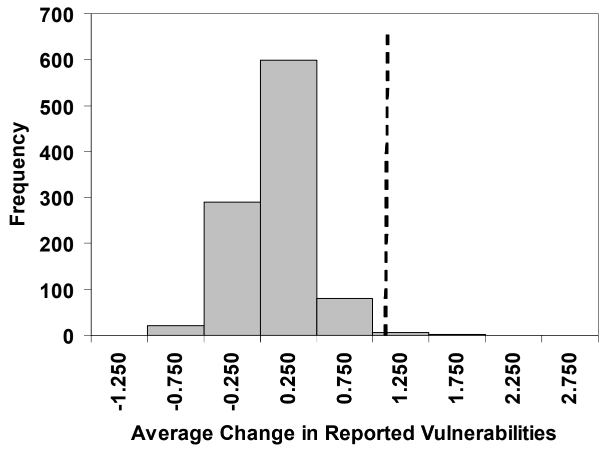

Figure 2. Average change in reported vulnerabilities for 1000 sets of 40 projects drawn completely at random (histogram) versus the 40 scanned projects (dashed line).

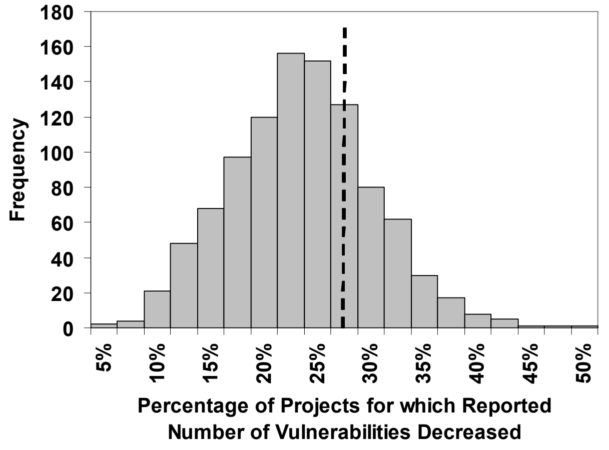

Figure 3. Percentage of projects with a decrease in reported vulnerabilities for 1000 sets of 40 projects drawn completely at random (histogram) versus the 40 scanned projects (dashed line).

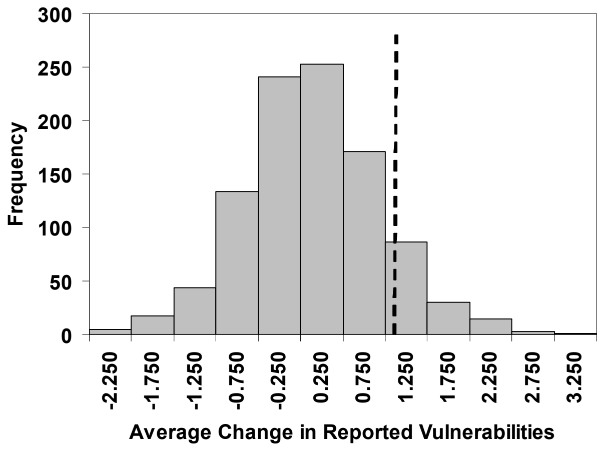

Figure 4. Average change in reported vulnerabilities for 1000 sets of 40 projects drawn using stratified random sampling (histogram) versus the 40 scanned projects (dashed line).

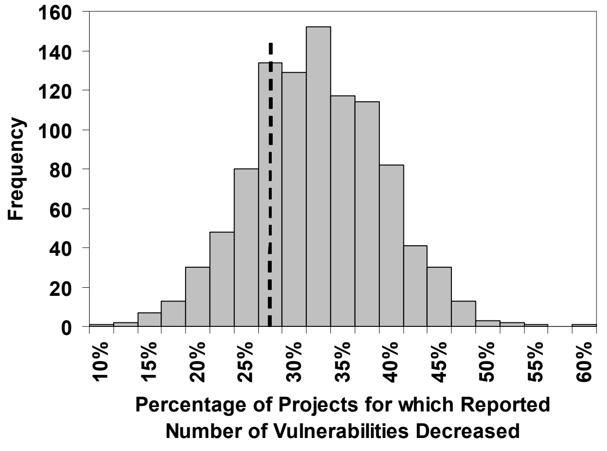

Figure 5. Percentage of projects with a decrease in reported vulnerabilities for 1000 sets of 40 projects drawn using stratified random sampling (histogram) versus the 40 scanned projects (dashed line).

comparison of the randomly drawn sets of projects and the results for the scanned projects indicates that the scanned projects do not appear to be atypical from those for the randomly selected projects.

Because the appropriate population to serve as a baseline for the scanned projects is not completely clear due to the non-random sampling of the projects, we also repeated these comparisons with sets of randomly selected projects from the NVD that were sampled using stratified random sampling. The stratification was based on a categorization of the number of vulnerabilities reported for each project since the beginning of 2005 and was matched to the analogous categorization of vulnerabilities reported for the scanned projects. The results of these comparisons are shown in Figure 4 and Figure 5.

The results shown in Figure 4 indicate that the average change in the number of reported vulnerabilities is slightly more variable for projects similar to the scanned projects. The projects chosen for scanning tended to be better known than the projects selected completely at random and, presumably as a result, typically had more reported vulnerabilities.

The difference in the average number of vulnerabilities is still a bit larger than the corresponding results for the projects selected using stratified random sampling, but the difference is not as significant as with the projects sampled completely at random. The results shown in Figure 5 again illustrate that the percentage of scanned projects for which there was a decrease in the number of vulnerabilities does not look atypical.

Taken as a whole, these comparisons further confirm that the data collected using the techniques outlined here does not indicate that the use of source code scanners has a strong effect on the number of vulnerabilities reported for projects in the NVD. It is always important to keep in mind the possible confounding factors, discussed in Section 3.2 that may affect these results and complicate their interpretation.

5. CONCLUSIONS AND FUTURE WORK

We devised an approach to evaluate the effect of static analysis tools on software security. Our approach could also be applied to other classes of tools and techniques. It harnesses the public vulnerability data to measure security, as well as assessments of open source projects using static analysis tools. This allows us to apply the study to a large number of popular open source projects, which will help generalize the results.

We presented aggregated data for many projects and more detailed data for just two projects. The data did not indicate that the use of static analysis tools has a strong effect on software security. However, it is difficult to draw strong conclusions using this data due to the large number of confounding factors. We plan to collect more data from different sources including bug reports, project web sites, and communication with developers, in order to better control for confounding factors and allow more detailed statistical analyses.

One useful approach to help control for confounding factors may be to compare two projects for the same application type (e.g., two web browsers), where one project uses tools and the other does not.

Our approach complements other approaches to evaluating the effect of tools, such as industrial case studies [15] and controlled experiments, which may also help answer the question of whether the use of static analysis tools really affects software security.

Another future aim of this work is to find the types of vulnerabilities on which tool use has the most or least impact.

6. ACKNOWLEDGMENTS

We thank Cyril Lan for many helpful suggestions on this paper. We would also like to thank Ben Chelf for providing valuable information about the Coverity scan of open source software and Peter Mell for explaining the details of data collection in the NVD.

7. REFERENCES

[1] *Accelerating Open Source Quality,* http://scan.coverity.com/

[2] O. H. Alhazmi, Y. K. Malaiya and I. Ray, *Security Vulnerabilities in Software Systems: A Quantitative Perspective*, IFIP WG 11.3 Working Conference on Data and Applications Security, Aug. 2005.

[3] P.E. Black, M. Kass, and M. Koo, *Source code security analysis tool functional specification version 1.0*, NIST SP 500-268, May 2007.

[4] B. Chess and G. McGraw, *Static Analysis for Security,* Security and Privacy Magazine, IEEE, 2(6), pp 76-79, 2004.

[5] CVE – Common Vulnerabilities and Exposures, The MITRE Corp., http://cve.mitre.org/

[6] D. Engler, *Weird things that surprise academics trying to commercialize a static checking tool,* http://www.stanford.edu/~engler/spin05-coverity.pdf

[7] C. Frye, *Klocwork static analysis tool proves its worth, finds bugs in open source projects,* SearchSoftwareQuality.com, June 2006.

[8] *Java Open Review Project*, Fortify Software, http://opensource.fortifysoftware.com/

[9] S. M. Kerner, *Study: MySQL Hard on Defects*, Internetnews.com, Feb. 2005.

[10] G. McGraw, Software security, Addison-Wesley, 2006.

[11] *National Vulnerability Database,* NIST, http://nvd.nist.gov/

[12] A. Ozment and S. E. Schechter, *Milk or Wine: Does Software Security Improve with Age?* 2006 Usenix Security Symposium, Vancouver, B.C., Canada, 31 July-4 Aug. 2006.

[13] N. Rutar, C. B. Almazan and J. S. Foster, *A Comparison of Bug Finding Tools for Java*, 15th IEEE Int. Symp. on Software Reliability Eng. (ISSRE'04), France, Nov 2004.

[14] SecurityFocus, http://www.securityfocus.com/vulnerabilities

[15] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. Vouk, *On the Value of Static Analysis for Fault Detection in Software*, IEEE Trans. on Software Engineering, v. 32, n. 4, Apr. 2006.