Software Engineering Institute | Carnegie Mellon University

Predictive Models for Identifying Software Components Prone to Failure During Security Attacks

Laurie Williams
Michael Gegick
Mladan Vouk

October 2008

ABSTRACT: Sometimes software security engineers are given a product that they not familiar with and are asked to do a security analysis of it in a relatively short time. A knowledge of where vulnerabilities are most likely to reside can help prioritize their efforts. In general, software metrics can be used to predict fault- and failure-prone components for prioritizing inspection, testing, and redesign efforts. We believe that the security community can leverage this knowledge to design tools and metrics that can identify vulnerability- and attack-prone components early in the software life cycle. We analyzed a large commercial telecommunications software-based system and found that the presence of security faults correlates strongly with the presence of a more general category of reliability faults. This, of course, is not surprising if one accepts the notion that security faults are in many instances a subset of a reliability fault set. We discuss a model that can be useful for identifying attack-prone components and for prioritizing security efforts early in the software life cycle.

Please note that, although this article is within the Best Practices section of BSI, the work described in it is exploratory and not yet mature enough to be a recommended practice.

ACKNOWLEDGEMENTS: This work is supported by the National Science Foundation under CAREER Grant No. 0346903. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. Thanks to Jason Osborne for his review of the statistical modeling.

INTRODUCTION

Software reliability engineering is concerned with the design, delivery, and maintenance of software at a reliability level requested by a customer [14]. In the past, principle concerns revolved around functional and performance reliability, availability, and dependability. However, in recent years, the spotlight has also included software security.

Software Engineering Institute
Carnegie Mellon University
4500 Fifth Avenue
Pittsburgh, PA 15213-2612

Phone: 412-268-5800
Toll-free: 1-888-201-4479

www.sei.cmu.edu

In the general reliability realm, internal and external software metrics can be used to successfully predict fault- and failure-prone components relatively early in the software life cycle (SLC), e.g., [16, 20]. A fault-prone component is likely to contain faults [4]. A failure-prone component is likely to exhibit execution-time anomalous behavior or failure due to a fault [19]. However, if faulty code is never executed, or otherwise exploited, the component will not fail. In the context of security, and this paper, a vulnerability is a fault (of either commission or omission) that can result in a security event. If such a fault is exploited during execution, the software experiences a security failure. If that happens often, we say that the software is attack-susceptible or attack-prone.

A predictive capability that identifies fault- and failure-prone components early in the software life cycle can present a significant advantage to a software organization because the costs of finding and fixing problems increases as one progresses through the SLC [3]. Prediction models can illuminate these problematic components for software engineers to prioritize testing, inspections, and redesign and thus mitigate the most significant problems first. However, inspection of all fault-prone components may cause the development team to expend valuable and limited verification resources in low risk areas of the code that, while identified as fault-prone, in reality may be of sufficiently high quality in the context of a normal operational profile (or usage) [14]. Identification of failure-prone components may reduce that overhead by focusing on software behaviors in the context of the expected operational profile.

We define a vulnerability-prone component as a component that is likely to contain one or more vulnerabilities. A vulnerability-prone component is analogous to a fault-prone component in that vulnerabilities may remain latent (similar to faults) until encountered by an attacker (or tester/customer) during software code execution. Vulnerabilities can come in a wide range of severity and likelihood of exploitation. It is quite possible that a vulnerability may be never found, may be difficult to exploit, or may not become exploitable in the field until after the user either changes the system configuration or environment or makes another change in a future revision of the software. (The latter is known as the "next release effect" [11, 12].)

The parallel between the classical software reliability engineering definition of faults and failures [15] and our definition of vulnerability and attack is illustrated in Figure 1. Static analytical methods and metrics can be used to identify physical faults in the code—either actual mistakes (errors of commission) or missing code (errors of omission). If the security requirements are well defined, such analysis will also uncover a subset of faults that represent security vulnerabilities. During software use, faults that are encountered by the flow of execution

(and result in a visible anomaly) become software failures. If vulnerabilities are exploited during software execution, we report a security failure either during testing or during an actual attack. Attack-prone components exhibit comparatively high rates of security failures in security testing or in the field. This may be because their vulnerabilities are an easy target or are of special interest to attackers.

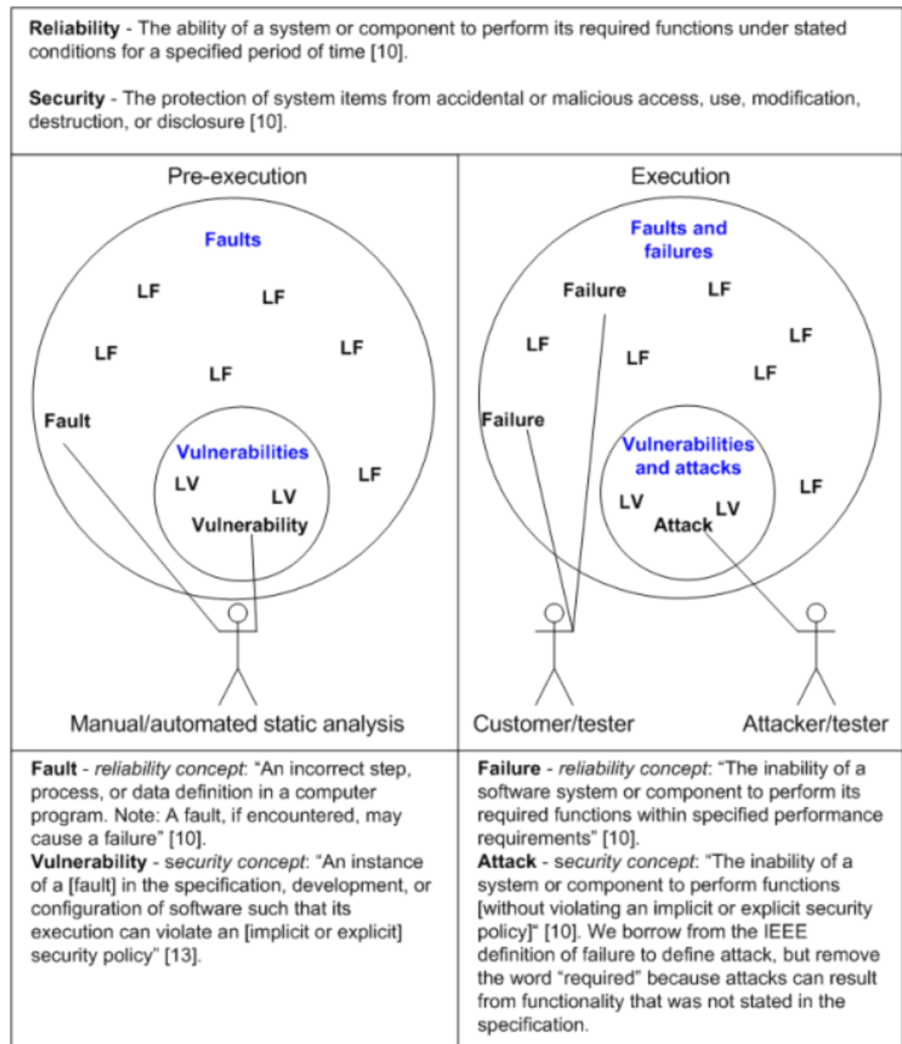

Figure 1. Defining security as a subset of reliability

LF (latent fault) and LV (latent vulnerability)

A relationship exists between vulnerability-prone and attack-prone components similar to the one between fault- and failure-prone classifications. A vulnerabil-

ity-prone component can become attack prone if an attacker (or customer/security tester) is likely to exploit one or more vulnerabilities in that component. However, there is a danger that vulnerability-finding techniques can lead verification efforts to waste resources in low attack risk areas—areas that may be adequately fortified, may be uninteresting to an attacker, or contain difficult-to-exploit vulnerabilities. Attack-prone prediction models may help focus vulnerability-finding efforts. For example, one could assume that if a component has a low index of vulnerability-proneness, it also has a low index of attack-proneness. One could also assume that most of the security problems will be in a small percentage of the software components according to the Pareto law [6], as illustrated in Figure 2. While this general finding is supported by statistical evidence [1], it may not be always true. It is quite possible that field security failures, even very serious ones, derive from components that are not predicted to be either fault- or failure-prone nor vulnerability- or attack-prone.

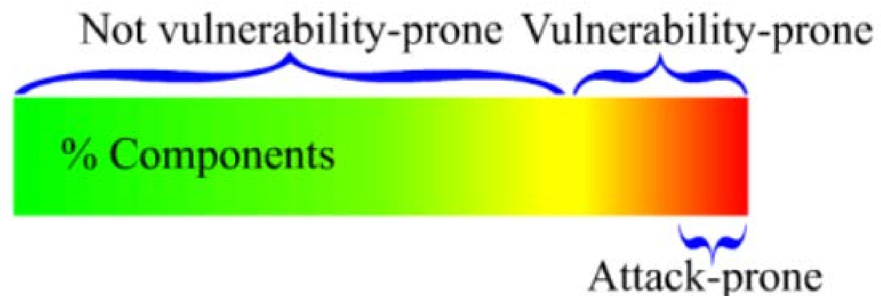

Figure 2. Most security problems are found in a small percentage of the software components

In this paper, we compare the predictive values of external metrics (failure report counts) to the predictive values of internal metrics (static analysis tool warnings, code churn, and SLOC) that we found in our earlier work [8]. We used data collected for a large commercial software telecommunications product. Our results should not be interpreted as applicable to all software because they are based upon the data from only one, though very large, software system.

DATA

We analyzed a large industrial telecommunications software system containing over 1.2 million lines of C and C++ source code that had been deployed in the field for two years. Customers required that the system be highly reliable and secure. The system contained 38 well-defined components, each of which consisted of multiple source files. Complete information necessary for our analysis

was only available for 25 (66%) of the components of the system, and thus the study focuses on those components.

Our analyses are based on two types of software failure reports: those reported by testers and those reported by customers. We analyzed 1255 failure reports that included pre- and post-release failures. A pre-release failure is a failure discovered by internal testers during robustness testing at the feature and system level. A post-release failure is a failure that occurred in the field and was reported by a customer. However, testers did not always indicate in their failure reports whether a testing failure was a security problem or a not. That required a manual follow-up to distinguish security from non-security failures. All failure reports explicitly identified the component where the fault or vulnerability was fixed.

Failure reports that were explicitly labeled as security related problems accounted for approximately 0.5% of the total system failures. Failure reports that were explicitly tagged as security problems but could do no harm were classified as non security-problems. For example, we did not classify a failure report as a security problem if a security mechanism had a default configuration that denied access to all users. For the remainder of the failure reports the first author and an additional research assistant, both doctoral students in software security, independently reviewed each pre- and post-release failure report and classified a failure report as a security problem using the four decision categories listed below. We based our criteria on system functionality and the content of the failure reports that detailed the impact. If there was no conclusive evidence that the failure was due to a security vulnerability, then we classified the report as a non-security problem. The software vendor has since fixed all of the faults in the failure reports.

We found that many reports contained the following keywords often seen in security literature: crash, denial-of-service, access level, sizing issues, resource consumption, data loss, flooding, integrity, overflow, null problems, overload, protection, and leakage. These keywords increased our suspicion that a failure could be a security problem but did not necessarily indicate a security problem. We compiled a list of these keywords and used it to match against all failure reports that were not explicitly labeled as a security problem.

We decided to use the following four categories of security failures:

- **Remote attacks**. The failure reports explicitly indicated whether the failure was due to a remote user or machine. Pre- and post-release failure reports that contained security-related keywords and could be remotely initiated had the highest probability of an exploitable vulnerability.

- **Insider attacks**. If the failure report did not indicate that a failure was due to an external user or machine, we looked for attacks that did not require remote access to the system. For example, one report indicated that an insider attack was possible if a disgruntled employee was to abuse a privilege in the system.
- **Audit capability**. Weak or absent logging of important information was considered a security vulnerability. An example is inadequate logging of a financial transaction that may result in an attacker obtaining a service for free. The absence of logs has been demonstrated as a security problem when audits are required to identify an attack [17].
- **Security documentation**. We also considered whether fundamental principles of software security were followed. For instance, in two failure reports testers indicated that a problem would occur if the users were not "well-behaved." This breaks the principle of Reluctance to Trust [2]. Additionally, we looked to see whether documented descriptions of vulnerabilities could apply (e.g., those listed in the Common Weakness Enumeration at http://cwe.mitre.org), or if any documented attack patterns [9] could match to the software.

A vendor security engineer audited our report and eliminated false positives (the general reliability faults that we claimed to be security vulnerabilities). In the end, 46 (3.7%) of all the failure reports were reported as having security vulnerabilities. It needs to be noted that there is some subjectivity involved in the failure analysis, but the validation from the security engineer and the cross-examination from the two software security students reduces error in our classification. While the other faults could facilitate an attack, they were not directly exploitable. Our analysis included only 3.7% of the defects reported for the product; therefore statistical assessment required extra attention. Failure reports were based upon the testers' abilities to find problems that open the door for an attacker. Of course, we can only know detected faults; we do not know which faults still remain [5].

MODELS

The models we developed are an extension of the concepts and models discussed in [10]. In developing our models, we manually classified the 25 components into failure-prone and attack-prone based on the reported severity levels of the failures in the components. The vendor we worked with used a four-tier hierarchy for classifying failure reports. A priority 1 failure (P1) is the most severe failure, while P4 is the least severe. A P1 or P2 failure was considered a "show stopper" by the vendor, so we used that threshold for our classification of failure-

prone components. Components with no failure reports were classified as not failure-prone.

We manually classified attack-prone components as those components with at least one pre-release test failure that was considered to be a security problem. We use attack-prone instead of vulnerability-prone because a tester discovered the vulnerability during program execution. The threshold of just one attack can create extraordinary damages, such as brand damage and loss of market share, and we thus chose one as a threshold. We could not verify from the failure reports whether the security-based post-release failures reported by customers were attacks that led to an attacker reaching their goal. However, according to the failure reports and vendor security engineers, the vulnerability could have been exploited, and thus we considered the failure an "attack" in the context of our model construction. It is worth noting that no real attacks were reported by the organization for the software system. Our analysis indicated that there were ten attack-prone components. Four of the attack-prone components were associated with customer-reported security failures.

The system we studied was scanned by the static analysis tool FlexeLint. We term the use of static analysis tools automated static analysis (ASA). Although FlexeLint is a reliability-focused ASA tool, we sought to determine if the reliability alerts could be early warnings of security vulnerabilities on a per component basis. As two additional internal metrics, we include code churn—the sum of added and changed source lines of code (SLOC)—and SLOC as internal metrics to our models. ASA can be performed early in the software life cycle and has been shown to be a good predictor of the fault- and failure-proneness of components [15, 21].

To construct our models, we used classification and regression trees (CART). We performed two CART analyses on our data, one using internal metrics to predict attack-prone components and the other with external metrics. Below are the internal and external metrics:

1. ASA produced internal metrics of (a) count and density of null pointer, memory leak, and buffer overflow alerts (both audited and unaudited), (b) the sum of the count and density of the previous three security-based alerts, and (c) the count and density of all alerts.
2. The count of SLOC and churn internal metrics.
3. External metrics of count and densities of pre-release, post-release, and total of pre- and post-release failures (security and non-security).

CART was run against all components in the system that we had already manually classified. The objective was to identify predictors (metrics) that best separate attack-prone components from not attack-prone components.

Classification and Regression Trees

Classification and regression trees (CART) is a statistical technique that recursively partitions data according to X and Y values. The result of the partitioning is a tree of groups where the X values of each group best predicts a Y value. The leaves of the tree are determined by the largest likelihood-ratio chi-square statistic. The threshold or split between leaves is chosen by maximizing the difference in the responses between the two leaves [18].

RESULTS

The following results apply in their specificity to the environment in which they were obtained. However, they also support construction of more models based on CART, and thus provide a generalization of the concepts and the approach.

The first model shows that 100% of the attack-prone components (with an 8% false positive rate) are those that have at least 17 total pre-release failures, as illustrated in Figure 3. The failures include security and non-security failure counts to show that a manual classification between security and non-security failures is not required for the model. While in general further splitting (differentiation) with the other metrics can be done for larger component sample than we had, in our case it was not possible with p-values below 0.05 (95% significance level). The results indicate that the pre-release failure count can be a good metric for classifying components that also have a high probability of being attacked.

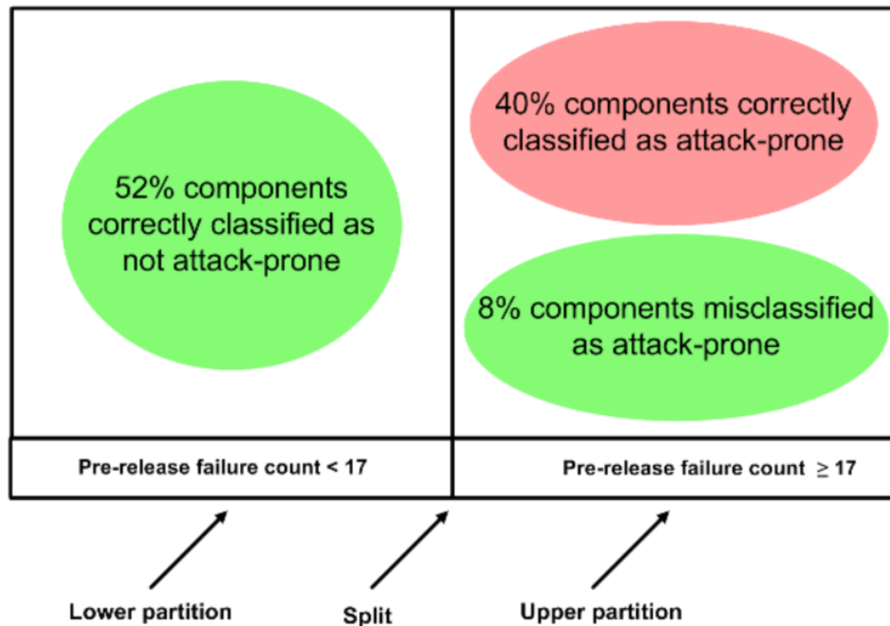

Figure 3. The result of CART after using the pre-release failure count to partition attack-prone components

A not unexpected observation is that the more failure reports that exist per component, the more likely that some of the failures can cause security problems. It is also worth noting that security problems did not appear to exist in components that did not have many problems. From these observations, we can define a discrimination threshold [20]; in this case 17 could be used as a serious warning about potential for security problems. In our setting, the attack-prone components identified by pre-release testing were also associated with security-related failures reported in the field. Thus, the attack-prone components identified by the model are also useful for predicting potential attacks in the field.

In the second model, we compare the analysis to using ASA metrics and churn, as shown in Figure 4. The results are similar, suggesting that the internal metrics are as good indicators as external metrics. Additionally, the internal metrics can be obtained before testing begins, and thus the model can be used to prioritize late-cycle testing efforts to the attack-prone components. The performance of the models is given in Table 1. Type I indicates the number of false positives or the number of not attack-prone components misclassified as attack-prone components. Type II errors indicate the number of attack-prone components misclassified as not attack-prone. The R2 value for the model indicates how much variance the model can account for in the data. The cross-validate R2 is a test to determine the validity of the R2 value. If the R2 value and cross-validated R2 value are similar, then R2 is an accurate measure of the model. Finally, the ROC

curve indicates how well the metrics estimate the probability that a component is attack-prone.

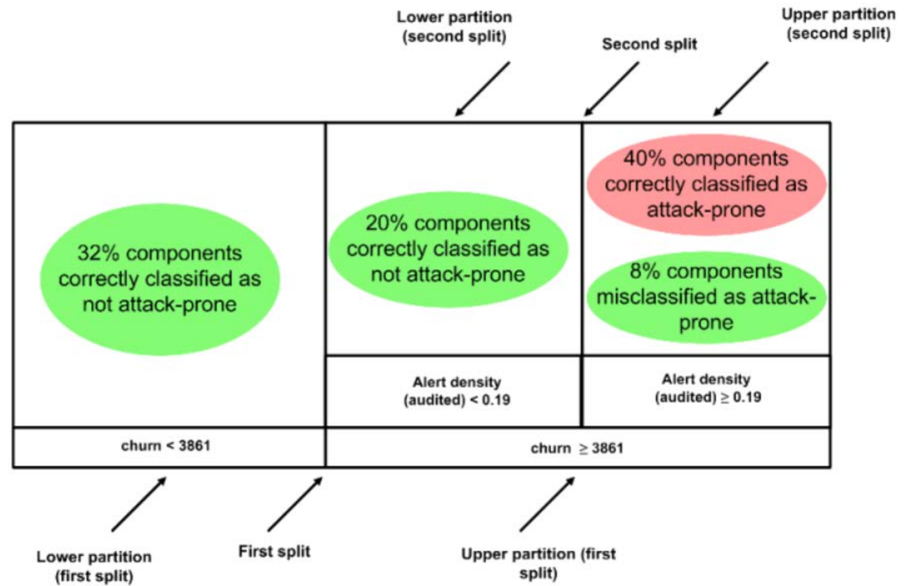

Figure 4. CART with alert density and churn

Table 1. Performance of the CART models

Metric	Type I Error Rate	Type II Error Rate	R^2	Cross-Validated R^2	ROC
Density of alerts and count of code churn	8%	0%	67.9%	61.1%	93%
Security and non-security pre-release failure count	8%	0%	68%	51.6%	93%

We can use the negative binomial distribution to predict the count of pre- and post-release attacks per component based on the count of pre-release non-security failures. For example, in our case the parameters were standard error=0.0108, p<.0089, value/DF=0.92. Having an estimated count of security problem count based on non-security failure count can indicate how many security issues testers should look for in a component. Although not all of the faults in the system have surfaced to cause failures, the system had been in the field for two years, indicating which faults are most likely to surface. We attempted to assign a probability to a component that was attack-prone using logistic regression and Poisson regression, but in the context of our data the models did not produce probabilities that separated attack-prone from not attack-prone components.

We now show where the thresholds from the models lie with respect to failure-prone components. Figure 5 shows the failure count groupings (non-security and security) that best divide the components into four approximately equal segments of components. The green sections indicate components that were classified as not failure-prone or not attack-prone. The column titled "Attack-prone" shows the number and percentage of failure-prone components for that segment that are also attack-prone. We also present the quartiles as a measure of density to negate the effect of count of SLOC. As shown in Figure 5, density evenly distributes attack-prone components among failure-prone components.

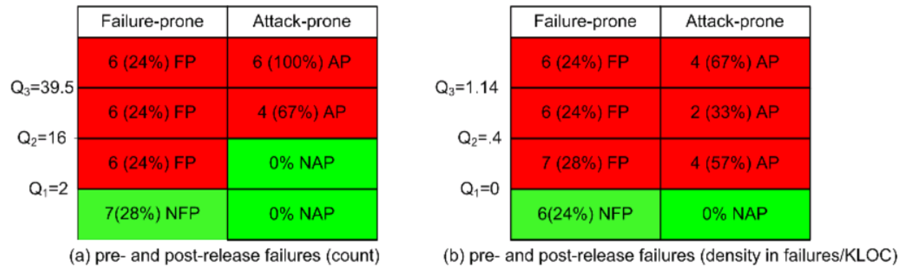

Figure 5. Failure-prone and attack-prone components juxtaposed according to failure counts and densities

Failure-prone (FP), not failure-prone (NFP), attack-prone (AP), not attack-prone (NAP). Q is the quartile value.

We measured correlations between non-security failures and security failures as shown in Table 2 as an extension to our earlier work [7]. A strong correlation (0.8 ? r ?1.0) indicates that an increase in non-security failures is accompanied with an increase in security failures. We found that non-security failure count is strongly correlated to security failure count (0.82). The strong correlation represents that the more failure reports present, the more likely that some could cause security failures. Additionally, the correlations suggest that a developer who is likely to inject a non-security fault is also likely to inject a security fault.

We calculated correlations with failure density, too. However, density can dilute the significance of a vulnerability because a single vulnerability can have a large business impact for a vendor or customer. We could not find a correlation between non-security failure density and post-release security-based failures. Different software systems behave differently. In some systems, the count of SLOC

is directly related to failure count. However, in our system, SLOC was only moderately associated with security failure count.

Table 2. Correlations between non-security and security failures[1]

Metric	Non-security failures	Security failures	Spearman rank correlation (p-value)
Failure count	pre- and post-release	pre- and post-release	0.82 (p < .0001)
	pre-release	pre-release	0.80 (p <.0001)
	pre- and post-release	post-release	0.49 (p =.01)
Failure density	pre-release	pre- and post-release	0.57 (p =.003)
	pre-release	pre-release	0.58 (p =.002)
	pre-release, post-release, pre- and post-release	post-release	no correlation
SLOC	--	Post-release	0.42 (p=0.03)
Churn	--	pre-, post-release, or both	no correlation

[1] Correlations produced by SAS® 9.1.3

SUMMARY AND CONCLUSIONS

We found that security failures are in the components with the highest count of non-security failures and that our internal metrics have nearly identical predictive power as our external metrics. If software organizations can predict their failure-prone components, then they might consider assigning security efforts to the most failure-prone components. Many metrics are available for predicting attack-prone components and can be based on much of the work achieved in reliability engineering.

REFERENCES

1. O. H. Alhazmi, Y. K. Malaiya, and I. Ray, "Measuring, analyzing and predicting vulnerabilities in software systems," *Computers & Security,* vol. 26, no. 3, pp. 219-228, May 2006.
2. S. Barnum and M. Gegick, "Design Principles," https://buildsecurityin.us-cert.gov/portal/article/knowledge/Principles, 2005.
3. B. Boehm, *Software Engineering Economics*, New Jersey, Prentice-Hall, 1981.
4. G. Denaro, "Estimating software fault-proneness for tuning testing activities," *International Conference on Software Engineering*, St. Malo, France, pp. 269-280, 2000.
5. E. Dijkstra, *Structured Programming*, Brussels, Belgium, 1970.
6. A. Endres and R. D. Rombach, *A Handbook of Software and Systems Engineering*, Harlow, England, Pearson Education, Limited, 2003.
7. M. Gegick, "Failure-prone Components are also Attack-prone Components," *OOPSLA - student research competition to appear*, Nashville, Tennessee, October 2008.
8. M. Gegick, L. Williams, J. Osborne, and M. Vouk, "Prioritizing Software Security Fortification through Code-Level Security Metrics," *Workshop on Quality of Protection to appear*, Alexandria, VA, 2008.
9. G. Hoglund and G. McGraw, *Exploiting Software*, Boston, Addison-Wesley, 2004.
10. ISO/IEC 24765, "Software and Systems Engineering Vocabulary," 2006.
11. W. Jones and M. Vouk, *Software Reliability Field Data Analysis*, McGraw Hill, 1996.
12. G. Q. Kenney, "Estimating Defects in Commercial Software During Operational Use," *IEEE Transactions on Reliability,* vol. 42, no. 1, pp. 107-115, 1993.

13. I. Krsul, "Software Vulnerability Analysis," PhD Thesis in Computer Science at Purdue University, West Lafayette 1998.
14. J. D. Musa, *Software reliability engineering: More reliable software faster and cheaper Second Ed.*, Bloomington, Indiana, AuthorHouse, 2004.
15. N. Nagappan and T. Ball, "Static Analysis Tools as Early Indicators of Pre-release Defect Density," *International Conference on Software Engineering*, St. Louis, MO, pp. 580-586, 2005.
16. T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Where the bugs are," *International Symposium on Software Testing and Analysis*, Boston, Massachusetts, pp. 86-96, 2004.
17. V. Prevelakis and D. Spinellis, "The Athens Affair," IEEE Spectrum, vol. 44, no. 7, pp. 26-33, July, 2007.
18. SAS Institute Inc., "The Partition Platform," SAS Institute, Inc., Cary, NC, 2003.
19. A. Schroter, T. Zimmermann, and A. Zeller, "Predicting Component Failures at Design Time," *International Symposium on Empirical Software Engineering*, Rio de Janeiro, Brazil, pp. 18-27, September 21-22 2006.
20. M. Vouk and K. C. Tai, "Some Issues in Multi-Phase Software Reliability Modeling," *Center for Advanced Studies Conference (CASCON)*, Toronto, pp. 512-523, October 1993.
21. J. Zheng, L. Williams, W. Snipes, N. Nagappan, J. Hudepohl, and M. Vouk, "On the Value of Static Analysis Tools for Fault Detection," *IEEE Transactions on Software Engineering,* vol. 32, no. 4, pp. 240-253, April 2006.

Copyright © Carnegie Mellon University 2005-2012.

This material is based upon work funded and supported by Department of Homeland Security under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center sponsored by the United States Department of Defense.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of Department of Homeland Security or the United States Department of Defense.

References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute.

NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN “AS-IS” BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

This material has been approved for public release and unlimited distribution except as restricted below.

Internal use:* Permission to reproduce this material and to prepare derivative works from this material for internal use is granted, provided the copyright and “No Warranty” statements are included with all reproductions and derivative works.

External use:* This material may be reproduced in its entirety, without modification, and freely distributed in written or electronic form without requesting formal permission. Permission is required for any other external and/or commercial use. Requests for permission should be directed to the Software Engineering Institute at permission@sei.cmu.edu.

* These restrictions do not apply to U.S. government entities.

DM-0001120