SYM-AM-17-047

Proceedings
of the
Fourteenth Annual Acquisition Research Symposium

Wednesday Sessions
Volume I

Acquisition Research:
Creating Synergy for Informed Change

April 26–27, 2017

Published March 31, 2017

Prepared for the Naval Postgraduate School, Monterey, CA 93943.

Acquisition Research Program
Graduate School of Business & Public Policy
Naval Postgraduate School

Software Vulnerabilities, Defects, and Design Flaws: A Technical Debt Perspective

Robert L. Nord—is a Principal Researcher at the Carnegie Mellon University's Software Engineering Institute (SEI). He is engaged in activities focusing on managing technical debt, agile and architecting at scale, and works to develop and communicate effective practices for software architecture. He is co-author of *Applied Software Architecture and Documenting Software Architectures: Views and Beyond* and lectures on architecture-centric approaches. Dr. Nord is a recognized leader in the software engineering community, both as a thought leader in software architecture and through his work helping industry and government customers at the SEI. He is a distinguished member of the ACM. [rn@sei.cmu.edu]

Ipek Ozkaya—is a Principal Researcher and Deputy Technical Lead of Software Architecture Practices at Carnegie Mellon University's Software Engineering Institute (SEI). With her team at the SEI, she works to help government and industry organizations improve their software development efficiency and system evolution through better use of agile architecting and technical debt management. Dr. Ozkaya serves as the chair of the departments of the *IEEE Software* magazine and as an adjunct faculty member for the Master of Software Engineering Program at Carnegie Mellon University. She is the co-author of articles on software architecture and technical debt management. [ozkaya@sei.cmu.edu]

Forrest Shull—is Assistant Director for Empirical Research at Carnegie Mellon University's Software Engineering Institute. His role is to lead work with the U.S. DoD, other government agencies, national labs, industry, and academic institutions to advance the use of empirically grounded information in software engineering, cybersecurity, and emerging technologies. He has been a lead researcher on projects for the DoD, NASA's Office of Safety and Mission Assurance, the Defense Advanced Research Projects Agency (DARPA), the National Science Foundation, and commercial companies. He serves on the IEEE Computer Society Board of Governors and Executive Committee. [fjshull@sei.cmu.edu]

Abstract

Technical debt describes a universal software development phenomenon: "Quick and easy" design or implementation choices that linger in the system will cause ripple effects that make future changes more costly. Although DoD software sustainment organizations have routine practices to manage other kinds of software issues, such as defects and vulnerabilities, the same cannot be said for technical debt. In this work, we discuss the relationships among these three kinds of software anomalies and their impact on software assurance and sustainable development and delivery. Defects are directly linked to external quality, and vulnerabilities are linked to more specific security concerns, but technical debt concerns internal quality and has a significant economic impact on the cost of sustaining and evolving software systems. Emerging research results and industry input demonstrate there are clear distinctions that call for different detection and management methods for defects, vulnerabilities, and technical debt. We draw from concrete examples and experience to offer software development practices to improve the management of technical debt and its impact on security.

Introduction

Software engineers face a universal problem when developing and sustaining software: weighing the benefit of an approach that is expedient in the short-term, but which can lead to complexity and cost over the long-term. In software-intensive systems, these tradeoffs can create technical debt (Kruchten, 2012), which is a design or implementation construct that is expedient in the short-term, but which sets up a technical context that can make future changes more costly or even impossible. Accumulating technical debt in the form of design shortcuts can be a strategic approach for software developers to accelerate

development and optimize resource management without impacting overall quality—as long as the debt is eventually paid off (i.e., the time is taken to improve the software quality). The results of recent practitioner-focused empirical industry studies reveal that most systems are also suffering from *unintentional* technical debt, that is, the quick and dirty choices that accumulate with no strategic thought.

An increasingly important cost driver for DoD systems is the effort that is put into developing, acquiring, and sustaining software-intensive systems. Most DoD systems are in operation for extensive periods of time—likely for multiple decades—and continue through sustainment to evolve to incorporate new functionality. Even in mature systems, ongoing sources of software changes may include changing mission profiles, the need to incorporate more effective or efficient technologies into the system, or the need to repair newly discovered software vulnerabilities. For all of these issues, software tends to be the logical and cost-effective way to make the change, meaning that in effect, software is never "done." Consequently, dealing with technical debt is an unavoidable phenomenon for the DoD. For systems on which technical debt has been allowed to accumulate (or said another way, sufficient emphasis has not been placed on maintaining software quality), dealing with this never-ending stream of changes becomes increasingly less cost-effective, as more and more effort is required to comprehend and work within a poor quality system rather than on focusing on implementing new capabilities. This can result in cost and schedule slippage or diminished abilities to field new capabilities for the same amount of effort.

The conventional approach many organizations take to managing such cost and schedule issues is to assess software project and process performance through metrics. An important class of these metrics focuses on software defects, since correcting defects (especially late in the software life cycle) can represent significant expenditures of unplanned effort. Cyber vulnerabilities, once detected, are typically candidates for focused effort to repair or mitigate quickly. Our overarching research question is this: **Will DoD systems see improved outcomes if they manage technical debt explicitly, along with these other classes of software anomalies?**

Impacts of Technical Debt

Indeed, a growing body of research indicates that focusing on defect management provides insight to only one perspective of the schedule, cost, and quality management problem. Empirical studies have shown that if technical debt is not paid back in a timely manner, it correlates with greater likelihood of defects (Falessi, 2015), unintended rework (Li, 2014), and increased time for implementing new system capabilities in software (Kazman, 2015).

For example, code quality issues such as dead code or duplicate code add to the technical debt. They do not affect the functionality seen by the end user but can impede progress and make development more costly over time. Software architecture plays a significant role in the development of large systems; flaws in a software system's design, such as a frequently changing interface between two classes (an unstable interface; Xiao, 2014), can also add significantly to the technical debt.

Technical debt can have observable adverse consequences on software security as well, an issue of high priority for DoD software-intensive systems, meaning that allowing debt to accumulate may be even more costly. Some vulnerabilities may be inadvertently introduced as the result of technical debt: for example, if a vulnerability is fixed in one location but is not fixed in a similar duplicated code fragment, or if overly complex code makes it harder to reason about whether a dangerous corner-case condition is feasible or not. Alternatively, as we will show, technical debt can also be caused by addressing a

vulnerability's symptoms rather than its root cause. Both of these relationships motivate a better understanding of the complex relationship between software vulnerabilities and technical debt. Therefore, we advocate that in order to get a better handle on their software quality, DoD programs should move toward also tracking their technical debt, similar to how they may be tracking defects and vulnerabilities (Figure 1).

In the remainder of this paper, we review the state of practice in managing technical debt and illustrate the need to manage all three types of software anomalies by summarizing results from our previous study, looking at the relationship between software vulnerabilities (Nord, 2016) and technical debt to address the following question: Are software components with accrued technical debt more likely to be vulnerability-prone? We present findings from a study of the Chromium open source project that motivates the need to examine a combination of evidence: quantitative static analysis of anomalies in code, qualitative classification of design consequences in issue trackers, and software development indicators in the commit history. Understanding this relationship can provide DoD programs with (a) ideas for improving their software engineering practices in better facilitating software quality assessment initiatives through data-driven analysis as presented in this work, and (b) an approach to better take advantage of existing analysis tools to help them focus what areas of their software to improve.

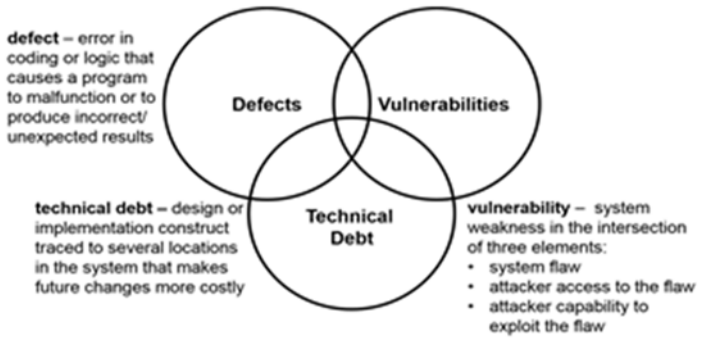

Figure 1. Software Anomalies

State of Practice in Managing Technical Debt

The technical debt metaphor is widely used to encapsulate numerous software quality problems. The metaphor is attractive to practitioners as it communicates to both technical and nontechnical audiences that if quality problems are not addressed, things may get worse. It is also very applicable in government sustainment contexts, as often the organizations that deal with the debt and those that take on the debt are not the same. While there has been significant progress made in creating an empirical and theoretical basis for identifying, quantifying, and managing technical debt (Spinola, 2012), there is still a lot of opportunity for improvement (Avgeriou, 2016).

Major software failures—for example, the recent United Airlines failure and New York Stock Exchange glitch or the National Security Agency's call data collection discrepancy—are being recognized in the popular media as the result of accumulating technical debt (Felten, 2014; Tufekci, 2015). In 2012, researchers conservatively estimated that for every 100 KLOC, an average software application had approximately US$361,000 of technical debt, the cost to eliminate the structural-quality problems that seriously threatened the

application's business viability (Curtis, 2012). The undeniable message is that technical debt is real and significant. Industry and government organizations have started to respond to this message, most significantly demonstrated by increasing initiatives focusing on analyzing code quality. Yet repeatable, data-driven studies that can help quantify this understanding, especially in the context of government systems, still lag behind.

The results of our recent, broad practitioner survey of 1,831 software engineers and managers, including industry and government participants, demonstrate that they share a common understanding of the concept of technical debt (Ernst, 2015). According to participants, the lack of proven tool support to accurately identify, communicate, and track technical debt is a key issue and remains a gap in practice. More than half of the participants of our survey reported using issue trackers to communicate technical debt either explicitly ("technical debt" is mentioned) or implicitly (the concept of "technical debt" is discussed but not explicitly mentioned). This is consistent with anecdotal feedback from our own experiences of working with organizations, as well as case studies represented in literature on technical debt (Zazworka, 2013). In the absence of validated tools to concretely communicate technical debt and its consequences, developers resort to practices they are familiar with. Our work in this paper contributes to closing the gap between system analysis and understanding of observed problems of technical debt, demonstrated as security issues.

Analysis Approach

To understand whether software components with accrued technical debt are more likely to be vulnerability-prone, we need to take into account data from multiple sources: quantitative static analysis of anomalies (faults, vulnerabilities, design flaws) in code, qualitative classification of design consequences in issue trackers, and software development indicators in the commit history. In this paper we present results from our analysis with Chromium open source project (Barth, 2008; Camillo, 2015). This is a complex web-based application that operates on sensitive information and allows untrusted input from both web clients and servers. We use it as a representative test bed of typical technical debt issues and types of vulnerabilities. The Chromium open source project released Version 17.0.963.46 (referred to as Chromium 17 from here on) on February 8, 2012. This release contained 18,730 files. From February 1, 2010, to February 8, 2012, there were 14,119 bug issues reported as fixed (Chromium 2017).

A challenge we observe with some DoD programs is that this type of data is not always available and different development parties are not incentivized correctly to share this information in a timely way with key decision-makers. Therefore, replicating this study with a DoD software-intensive system has its challenges, although we expect the underlying relationships would hold equally well in the DoD context.

Our analysis approach is as follows:

1. Identify software vulnerabilities.
    a. Enumerate issues in the Chromium issue tracker (Chromium 2017) that have the `security` label.
    b. Classify each issue in terms of its Common Weakness Enumeration (CWE) using the issue's description, comments, metadata, and patch.
    c. For each issue, identify the set of files changed by commits that reference the issue.
2. Identify technical debt.
    a. Classify issues for technical debt.

b. Classify the type of design problem and rework based on the issue description, comments, and metadata.
c. Detect design flaws that co-exist in the same files changed to fix the issues labeled `security`.

3. Model the relationships between technical debt issues and vulnerabilities in the common artifacts they represent (code files, issues, commits).
   a. Extract concepts related to vulnerability types.
   b. Test whether technical debt indicators (e.g., number and type of design flaws, number of traditional bugs, number of bugs labeled security, and the lines of code that change to fix a bug) correlate with the number of vulnerabilities reported.
   c. Manually investigate how selected vulnerabilities are influenced by the correlated technical debt indicators.

We will show how design knowledge can help identify other related issues and files so that developers can more efficiently diagnose the root cause of vulnerabilities and provide a long-term fix.

Analysis Results

To identify vulnerabilities, we used the issues labeled security. Using the Chromium project's issue tracker, we identified 79 software vulnerability issues, which were related to 289 files in which we detected design flaws (described in the next section). An issue labeled security may have a well-identified security bug, such as a null pointer exception. Such an issue may not represent technical debt but could simply be an implementation oversight. On the other hand, some issues may manifest themselves with multiple symptoms. This can hint that technical debt contributed to the vulnerability.

Following this exercise we classified whether each issue was technical debt or not using the classification approach we developed (Bellomo, 2016).

To classify source code files, we used the results of a study that analyzed Chromium 17 and reported 289 files associated with design flaws that can be detected in the code. The approach analyzes a project's repositories—its code and its revisions—to calculate a model of the design as a set of design rule spaces (DRSpaces; Xiao, 2014). These DRSpaces are automatically analyzed for design flaws that violate proper design principles. Four types of design flaws can be identified from the DRSpace analysis: modularity violation, unstable interface, clique, and improper inheritance. A modularity violation occurs when files with no structural relation frequently change together. This suggests that those files share some secret or knowledge and that information has not been encapsulated or modularized. An unstable interface occurs when there is an important class or interface that many other files depend on, and this class is buggy and changes frequently, requiring its "followers" to also change. Clique refers to a cross-module cycle that prevents groups of modules from being independent of each other. Improper inheritance occurs when the parent class depends on the child or when another file depends on both a parent and its child class. We consider these flaws as indicators of technical debt.

Table 1. Design Flaws and Issues Classified as Technical Debt

	Classified Not TD	Classified TD
No Design Flaw	8	6
Design Flaw	50	15

Table 1 shows our results where we found 15 issues classified as technical debt that also demonstrate design flaws. When we analyze these results for correlations using Pearson correlation coefficient, we see promising results. Design flaws demonstrate correlations with number of bugs (0.921), bug churn (0.908), number of security bugs (0.988) and security churn (0.826). Our further analysis shows that for three of the four types of design flaws—modularity violation, clique, and improper inheritance—files with vulnerabilities are also more likely to have design flaws. The more types of design flaws a file is involved in, the higher the likelihood of it also having vulnerabilities. We look at the design concepts represented in these issues related to vulnerabilities to better understand overarching correlations. Table 2 summarizes the vulnerabilities of those issues that also reported design problems from the 79 issues we classified, in the form of CWE categories (CWE 2017).

Table 2. Affinity Groups of Vulnerability Types

Affinity	CWE	#Issues
interface	200: Information Exposure	1
resource arbitration	362: Concurrent Execution using Shared Resource with Improper Synchronization	3
	400: Uncontrolled Resource Consumption	3
invalid result	20: Improper Input Validation	2
	451: User Interface (UI) Misrepresentation of Critical Information	2
	476: NULL Pointer Dereference	1
	704: Incorrect Type Conversion or Cast	1
	825: Expired Pointer Dereference	1
boundary conditions	125: Out-of-bounds Read	1
	703: Improper Check or Handling of Exceptional Conditions	4
	787: Out-of-bounds Write	2
privilege	250: Execution with Unnecessary Privileges	2
	269: Improper Privilege Management	1
	285: Improper Authorization	1

Our study revealed that developers are already using concepts related to technical debt when investigating security issues, including the following:

- getting to the *root cause*
- understanding the *underlying design* issues
- recording symptoms where changes are taking *longer than usual* or problems are reoccurring
- predicting consequences for the *longer term*
- building evidence for a more *substantial fix*

Furthermore, when we studied the issues in detail, we observed that finding the true design root cause of the problems, i.e., the underlying technical debt, took a substantial amount of resources of the developers. DoD software deals with these challenges, where many small issues like these add up daily to accumulate to not only jeopardize sustainment

resources, but also operational issues such as vulnerabilities that cause significant risks for the DoD.

Conclusions and Future Work

Software system vulnerability and technical debt are high priority concerns for our DoD software base and industry alike, leading us to address research questions such as:

- Are software components with accrued technical debt more likely to be vulnerability-prone?
- Does understanding the difference and similarities between technical debt, defect, and vulnerabilities lead to their better management?

Our studies on open source, industry, and government software data demonstrate that a conscious focus on understanding design issues that accumulate consequences in the form of vulnerabilities, extensive rework, and maintenance issues create the most risky technical debt items. The state of the practice in industry and the DoD alike is that such issues are not explicitly tracked and understood. Our results demonstrate that it is those areas that in the long run create both the highest operational and sustainment risks.

Understanding and calling out similarities and differences between defects, vulnerabilities, and technical debt has a direct impact on acquisition practices such as software risk management and technical tradeoffs that impact contracting decisions. Measurement and analysis techniques for managing technical debt in the long run improves sustainability of systems and impacts better buying power. When they address security issues, software developers use technical debt concepts to discuss design limitations and their consequences on future work. One time-consuming relationship between vulnerabilities and technical debt is tracing a vulnerability to its root cause when it is the result of technical debt. Introducing technical debt measurement and analysis potentially improves finding such root causes.

Our ongoing and future work focuses on creating intelligent mechanisms to extract and analyze this data for software development professionals and provide guidelines that DoD decision-makers can use to allocate their scarce resources most appropriately.

References

Avgeriou, P., Kruchten, P., Nord, R., Ozkaya, I., & Seaman, C. (2016). Reducing friction in software development. *IEEE Software, 33*(1), 66–73.

Barth, A., Jackson, C., & Reis, C. (2008). The security architecture of the Chromium browser. Stanford Web Security Research.

Bellomo, S., Nord, R. Ozkaya, I., & Popeck, M. (2016). Got technical debt? Surfacing elusive technical debt in issue trackers. In *Proceedings of the 13th International Conference on Mining Software Repositories* (pp. 327–338).

Camilo, F., Meneely, A., & Nagappan, M. (2015). Do bugs foreshadow vulnerabilities? A study of the Chromium Project. In *Proceedings of the 12th Working Conference on Mining Software Repositories* (pp. 269–279).

Chromium issues. (2017). Retrieved from https://code.google.com/p/chromium/issues/list

Common weakness enumeration. (2017). Retrieved from https://cwe.mitre.org/about/sources.html

Curtis, B., Sappidi, J., & Szynkarski, A. (2012). Estimating the principal of an application's technical debt. *IEEE Software, 29*(6).

Ernst, N., Bellomo, S., Ozkaya, I., Nord, R., & Gorton, I. (2015). Measure it? Manage it? Ignore it? Software practitioners and technical debt. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering* (pp. 50–60). New York, NY: ACM.

Falessi, D., & Reichel, A. (2015). Towards an open-source tool for measuring and visualizing the interest of technical debt. In *Proceedings of the Seventh International Workshop on Managing Technical Debt*.

Felten, E. (2014). Technical debt in the NSA's phone call data program? Retrieved from https://freedom-to-tinker.com/2014/02/10/technical-debt-in-the-nsas-phone-call-data-program/

Kazman, R., Cai, Y., Mo, R., Feng, Q., Xiao, L., Haziyev, S., et al. (2015). A case study in locating the architectural roots of technical debt. In *Proceedings of the 37th IEEE International Conference on Software Engineering* (pp. 179–188).

Kruchten, P., Nord, R., & Ozkaya, I. (2012). Technical debt: From metaphor to theory and practice. *IEEE Software, 29*(6), 18–21.

Li, Z., Liang, P., Avgeriou, P., Guelfi, N., & Ampatzoglou, A. (2014). An empirical investigation of modularity metrics for indicating architectural technical debt. In *Proceedings of the 10th International ACM SIGSOFT Conference on Quality of Software Architectures* (pp. 119–128).

Nord, R., Ozkaya, I., Schwartz, E., Schull, F., & Kazman, R. (2016). Can knowledge of technical debt help identify software vulnerabilities? CSET @ USENIX Security Symposium.

Spínola, R. O., Zazworka, N., Vetro, A., Seaman, C., & Shull, F. (2012). Investigating technical debt folklore: Shedding some light on technical debt opinion. MTD@ICSE 2012, 1–7.

Tufekci, Z. (2015). Why the Great Glitch of July 8th should scare you. *The Message*. Retrieved from https://medium.com/message/why-the-great-glitch-of-july-8th-should-scare-you-b791002fff03

Xiao, L., Cai, Y., & Kazman, R. (2014). Design rule spaces: A new form of architecture insight. In *Proceedings of the 36rd International Conference on Software Engineering* (pp. 967–977).

Zazvorka, N., Spínola, R., Vetro, A., Shull, F., & Seaman, C. (2013). A case study on effectively identifying technical debt. In *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering* (pp. 42–47). Porto de Galinhas, Brazil.

Disclaimer & Distribution Statement

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

Acquisition Research Program
Graduate School of Business & Public Policy
Naval Postgraduate School
555 Dyer Road, Ingersoll Hall
Monterey, CA 93943

www.acquisitionresearch.net