Finding and Fixing Problems Early: A Perspective-Based Approach to Requirements and Design Inspections

Dr. Jeffrey C. Carver
Mississippi State University

Dr. Forrest Shull and Dr. Ioana Rus
Fraunhofer Center for Experimental Software Engineering

Viewing security vulnerabilities as a specific type of software defect allows proven software engineering techniques for finding and fixing them to be used early in the development of the product. Finding and fixing these problems early (i.e. at the requirements or design phase) will reduce the overall risk and cost of the product. This article describes the application of a previously successful early life cycle software inspection approach (perspective-based reading [PBR]*) to the problem of software security. Excerpts from this tailored approach are provided along with guidance on it use.*

Developing secure software requires engineering and management attention, as well as extra effort and resources. Rather than being treated as an *add-on*, security must be built in throughout the software life cycle, using a mixture of good software engineering practices (to ensure quality software in general) and good security practices (to ensure that exploitable vulnerabilities are not present). Integrating security practices with software engineering practices throughout the development process helps an organization anticipate security failures and develop software that can sustain attacks.

Currently, the predominant method for finding vulnerabilities is to *penetrate and patch*. This approach focuses most of the effort on the latter stages of the software life cycle, e.g., testing for a set of known vulnerabilities or waiting for a vulnerability to be exploited in delivered code. Only after such a vulnerability is found does the development team modify the software to address that vulnerability [1]. The effectiveness of this approach is highly correlated with the quality of the testing activities. An experienced tester can be quite effective, but an inexperienced tester can miss many important issues.

As evidence of the problems associated with testing for security vulnerabilities, although sensitivity to and focus on security problems has increased in recent years, the number of software vulnerabilities found in deployed software has actually increased – not decreased. For example, according to the United States Computer Emergency Readiness Team (US-CERT), the number of new vulnerabilities discovered in software has been growing at rates close to 140 percent, recently in excess of 5,000 per year. Other vulnerability collections show similar trends – e.g., the National Vulnerability Database has nearly 13,000 documented vulnerabilities, and Bugtraq, a moderated mailing list where vulnerabilities are reported and discussed, holds discussions on about 400 items per month. These figures make it clear that there is a national need to better address software security.

An improvement over this *penetrate and patch* mentality is to *build security in* from the beginning of the product lifecycle [2, 3]. Governmental organizations like the Department of Homeland Security (DHS) have already realized the benefits of this approach and are promoting research to make it happen. Standard software engineering approaches for building in software quality provide some insight into building in security. Traditionally, software engineering has defined a *defect* as an error, fault, or failure in the software system or its related artifacts. Security vulnerabilities are a special type of these more general software engineering defects and therefore benefit from similar detection and removal approaches.

Although no concrete guidelines exist, there is a growing recognition that early life-cycle issues do play an important role in the development of secure software. Problems in the early life-cycle phases have caused some costly and highly visible problems leading to untrustable systems. A well-publicized example of this type of problem was the theft of personal data from a database of background files on most American citizens maintained by the ChoicePoint corporation. The theft of data occurred when criminals set up fake companies (e.g. debt collectors, insurance agencies) and gained access to ChoicePoint's databases [4]. This security vulnerability can be viewed as a requirements and design problem. Had the creators of the software included requirements for verifying the legitimacy of a company prior to allowing access to private information, this theft could have likely been prevented.

Security experts have also clearly recommended the need for early life cycle work to address security vulnerabilities [5]. All of this evidence combines to create a powerful message: Although often ignored, decisions made (or missed) in the early life cycle have a large impact on the level of security achievable in the final system.

This message is quite familiar to software engineers. Software engineering practitioners and researchers have observed (and measured) that the earlier defects are found in the software life cycle the easier and cheaper they are to repair [6]. Many software engineering best practices are concerned with how to apply early life-cycle verification and validation (V&V) practices to *build quality in* throughout the life cycle rather than *test it in* during late life-cycle testing phases. Viewing security vulnerabilities as a special type of defect allows for building security into systems in the same way as building quality into systems. The ultimate goal is to integrate the best practices from the security engineering and software engineering communities into a set of techniques for identifying and removing security vulnerabilities early in the software life cycle. This article illustrates the adaptation of one such technique, PBR, to address the security vulnerability problem during a requirements inspection process.

An Early Life-Cycle Approach to Security

A large number of software engineering studies have established inspections as an effective method for reducing defects in software systems [6]. An inspection is a static review process in which a software artifact (e.g. requirements document, design document, or code) is reviewed by one or more inspectors to verify that it meets a set of quality properties. Companies like Microsoft have recognized the potential reduction of vulnerabilities that inspections can cause in the early life-cycle phases and have created their own checklists focused on early life-cycle issues.

One of the most successful examples of inspection use came from the Software Engineering Laboratory (SEL) at NASA's Goddard Space Flight Center. Figure 1 (see page 26) illustrates the impressive reduction in defect rates (per thousand developed lines of code [DLOC]) achieved on software projects over a nineteen year period at the SEL. This chart shows sustained continuous improvement over a long period of time in an

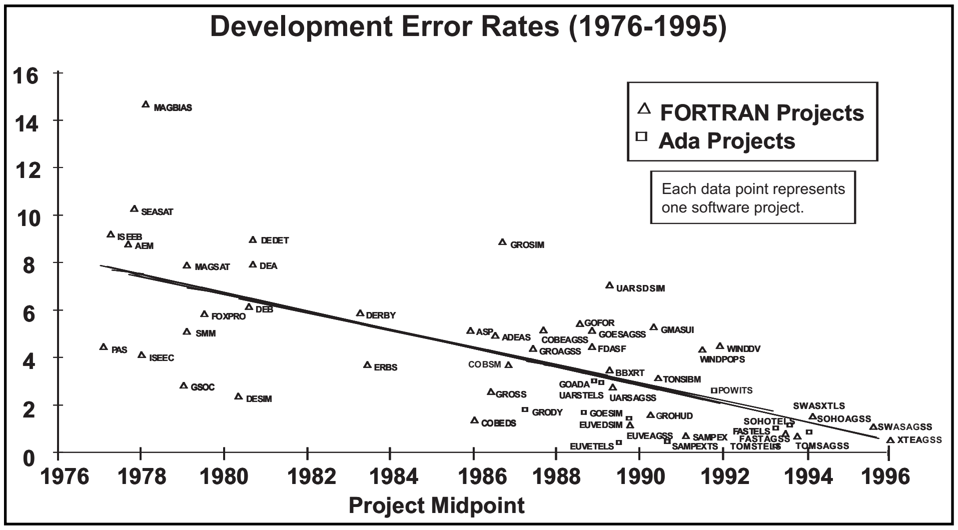

Figure 1: *Demonstrated Defect Reduction in Errors per 1,000 DLOC at NASA's SEL*

organization that built real-time software systems to support safety critical missions costing millions of dollars. Although much effort was spent on software quality improvement at Goddard during this time period, Frank McGarry and Mike Stark, two former directors of the SEL, have both credited the introduction and maintenance of inspections as being the most important factor in the error rate reduction.

One of the difficulties in performing early life-cycle reviews is the amount of human judgment required. Static analysis tools do exist, but these tools focus mainly on code. Early life-cycle V&V by its very nature is human intensive: Identifying and predicting the impacts of requirement or design decisions is not easily amenable to an automated checking approach. To be done effectively, these tasks require the experience and knowledge necessary to make good judgment calls. For security problems, this means that security expertise is essential, but often, lacking in software engineers. Therefore, a mechanism is needed to supplement the software engineers' knowledge with security specific knowledge. PBR is a technique that can be used for encoding and transferring security-relevant expertise.

A Tailorable, Perspective-Based Approach

Perspective-based inspection is a variant of a formal technical review, which provides a useful framework for integrating security concerns. This type of inspection is based on explicitly defining the important stakeholders for a particular artifact and the types of issues that are of importance to the team. Rather than asking each reviewer to search for all types of problems, the perspective-based approach requires inspectors to examine the document using a role-based *scenario* based on how one specific stakeholder would use the document to build the system. For example, an inspection of system requirements includes an inspector using a tester perspective. This inspector reviews the requirements by following a scenario in which he/she considers how to generate test cases based on the requirements. Any time the inspector experiences difficulty in using the document to complete his scenario he records this difficulty as an issue that must be fixed. This list of issues then becomes the list of items that is returned to the author of the document for repairs.

The set of appropriate stakeholders and issue types must be tailored for different environments. In this sense, the perspective-based approach provides a set of guidelines for creating an effective, tailored inspection technique, not a one-size-fits-all process.

By emphasizing the stakeholders, the perspective-based approach provides a helpful way to make the need for collaboration between software and security engineers explicit. Each of these roles must be represented by at least one inspector on the review team. This approach is also useful when it is not possible to get an inspector with sufficient security expertise to participate in the inspection process. Creating a scenario based on the way each stakeholder uses the document captures that stakeholder's expertise about early life-cycle indicators of potential security vulnerabilities that can be used by more novice inspectors.

To conduct a perspective-based inspection, practitioners first need to decide what documents should be inspected. Inspection of software-specific documents (like the requirements specification or design document) needs to be augmented with the inspection of security-specific artifacts (such as threat models). The goal of the inspection must be to ensure that the security models are internally consistent and correct, as well as that their implications for the system are adequately reflected in the software work products, to support construction of a correct end-product.

For the documents that have been selected, a list of stakeholders must be compiled, considering the following:

- *Stakeholders in downstream phases* who need the document to perform their own job. Related to security, the following are some examples: testers, who need to ensure that security requirements are clearly stated in terms of their expected behavior and the functionalities to which they should be applied; designers and/or developers, who need to ensure that security policies are specified in enough detail to allow for correct implementation; and users, who need to ensure that the final behavior of the system, including all security policies, will meet their needs.
- *Stakeholders from previous phases* who want to ensure that their decisions relevant to system design are adequately reflected in the document. Related to security, an important stakeholder is the developer of threat models and other early life-cycle security artifacts, who needs to ensure that the behavior identified by these models is in the system artifacts.
- *Stakeholder's specific types of expertise* need to be correctly reflected in the software work documents. An important security-related perspective is a *black hat* user who focuses specifically on issues that could lead to exploitable security vulnerabilities and increases the risk of a successful attack on the software.

For each perspective identified, a scenario that reflects the normal day-to-day work activity of the respective stakeholder must be created. Different stakeholders' scenarios require the inspectors to focus on different aspects of the document, while the entire set of perspectives covers the whole artifact. By focusing each reviewer on a separate task, the overlap among the responsibility of various inspectors is reduced. At the same time, the importance and necessity of the role played by each inspection member increases because no two reviewers focus on the same set of defects. This approach combats the assumption that simply having more eyes on a document increases the chances of finding problems. In reality, we find the opposite to be true: Having more people look at the same document without a specific focus allows each inspector to assume that his/her expertise and time are not crucial, and that someone else will find any problems they miss. This is a potentially dangerous assumption.

Our previous experiences in organizations such as NASA have shown perspective-based inspections to be especially useful for helping bring novice reviewers up to speed, by giving them a clear direction of *how* to get started on their analysis of the document, and by giving them a clear subset of concerns to focus on rather than having to *wear all the hats*. Perspective-based inspections were evaluated at Goddard with a controlled experiment comparing different approaches for reviewing requirements specifications. Using the perspective-based approach allowed individual reviewers to find up to 30 percent more defects, on average, than when they used the standard NASA review approach. When team results were statistically simulated from the individual data, teams using perspectives also found up to 30 percent more defects than teams using the standard NASA approach. These differences were statistically significant [7].

This study has been replicated multiple times with similar results. A series of other studies provide support for the benefits of using a perspective-based approach for inspections of different types of artifacts and different organizations:

- A study conducted at Lucent Technologies indicated that inspection teams using perspectives to find defects in formal requirements models were significantly more effective than teams using only checklists or unstructured techniques [8].
- A study at a U.S. government organization in 1998 showed that teams found about 30 percent more usability problems in Web interfaces when using a perspective-based inspection approach [9].
- A study of German software professionals from various companies producing object-oriented designs showed that teams using the perspective-based approach found 41 percent more defects than teams using only checklists, and the cost per defect was significantly lower [10].
- A study of code inspections at Bosch Telecom in Germany indicated that teams using perspectives were more effective at finding defects than teams using the baseline inspection approach with the *improvement* being statistically significant in two of the three experimental runs. The cost per detected defect was also significantly lower for the perspective-based approach in all runs [11, 12].
- A study of object-oriented design inspections at Ericsson in Sweden showed that teams using perspectives found more defects than those using their usual approach, although the perspective-based inspections took more time [12].

Tailoring Perspective-Based Reading for Security

Based on the previous proven success of using the perspective-based approach to find generic software quality defects, we have tailored this approach to focus on software security. In this paper, we describe a set of perspectives for a requirements inspection. To perform this tailoring, we augmented two of the *standard* PBR perspectives (the designer and the tester) with additional security specific questions. In addition, we created a new perspective based on the needs of a *black hat* tester. The remainder of this section briefly describes each technique.

The *designer* perspective has the goal of ensuring that there is enough, consistent information present in the requirements to successfully create a system design. The existing scenario is augmented with questions that focus on whether important security-related information has been correctly specified rather than being left up to the designer, who may not be familiar with all details of the security policy. Examples of the new questions that the reviewer using the designer perspective should consider when following this perspective include the following:

- Have the requirements specified enough information about the security policies for the designer to understand whether a layered security policy is required instead of a single point of vulnerability?
- If several administrator roles are defined, have they been defined as separate accounts with limited access to security resources, or a single account with comprehensive *super user* permissions?

In a similar fashion, the scenario for the *tester* perspective remains unchanged, but is augmented with security specific questions. The inspector using the tester perspective has the goal of ensuring that the trustworthiness of the system will be knowable during the testing phase. Examples of the new questions that the reviewer using the tester perspective should consider include the following:

- Have the requirements specified appropriate exception-handling functionality?
- Have the requirements specified adequate safeguards that would take effect once a malicious user has gained unauthorized access to the system?
- Does the system have a well-defined status, either a secure failure state or the start of a plausible recovery procedure, after a failure condition?

Finally, the *black hat* perspective is a new one (i.e. not tailored from the previous set of perspectives) which focuses the reviewer on finding weaknesses in the requirements that could be exploited via an attack. The scenario that the reviewer follows is to create a set of malicious attack scenarios that seek to exploit system vulnerabilities. While creating this scenario, the reviewer focuses on three types of information relevant at the requirements stage: Cryptography, Authentication/Authorization, and Data Validation. These types of information, along with the related questions were adapted for requirements from Araujo and Curphey's article on Security Code Reviews [13].

Cryptography relates to the encoding mechanisms specified for data items within the system. During the review, the inspector is looking for underspecified or incorrectly specified features that could be exploited. Example questions include the following:

- Can the encoding mechanisms specified for transmission and storage of data be broken?
- Do the cryptographic mechanisms specified follow well-known, well-documented, and publicly scrutinized algorithms, and if not, can they be easily broken?

Authentication/Authorization focuses the reviewer on determining how unauthorized users could gain access to the system. Example questions include the following:

- Can the protocols for validating user identity be broken?
- If account lockout is specified, are there requirements in place to prevent denial-of-service attacks?
- Can user privileges be artificially elevated due to omissions or poorly specified requirements?

Data Validation is an important source of security vulnerabilities and focuses the reviewer on determining whether invalid data could be entered into the system. An example question: Do the requirements leave any opportunities for invalid data to be entered by the lack of validation of external data?

These excerpts from the requirements review techniques illustrate that formulating inspection methods must fully leverage the intelligence and flexibility of the human beings involved in the procedure. As much as possible, inspectors should avoid using algorithmic heuristics that would be candidates for tool support instead (for example, keep coupling low by ensuring that no class calls more than six others). Rather, inspectors should be given tasks that require both semantic understanding of the system and judgment calls. The flexibility of such an approach allows inspectors to focus both on bad things that should be avoided (excessive coupling) and good things that are omitted (exception handling for appropriate functionalities). Similar ideas can be incorporated into the reviews of the threat models, architecture and design documents by adding similar questions specific to each artifact.

Conclusions

The decisions made in early life-cycle development phases have a large impact on whether secure systems are achievable or not. Just as for other types of quality issues, early life-cycle V&V activities can detect and repair security issues early on, for example, by checking that security requirements are well thought-out, feasible, and consistent with user needs; that system architectures exhibit good design principles, making them easier to maintain and fix without introducing vulnerabilities; or that components are designed so that if security measures are breached in one component, an attacker gains access only to a limited part of the system. Also, as for other types of quality issues, finding security problems early saves time and effort for the development team by avoiding the need to fix the many downstream documents that instantiate early decisions.

A perspective-based inspection, as illustrated in this article, is one approach which has been very effective at early life cycle defect detection and can be used to tailor V&V techniques to focus on security. While other approaches are certainly possible, the explicit reliance of the perspective-based approach on the needs of the stakeholders gives practitioners a way to bring the right expertise to bear and capture best practices so they can be employed by a larger set of inspectors.◆

References

1. McGraw, G. "Building Secure Software: Better Than Protecting Bad Software." IEEE Software 19.6 (2002): 57-58.
2. Beaver, K. and Sima, C. "Software Development: Building Security In." Security.itworld. 5 Sept. 2006.
3. Grance, T., Hash, J., and Stevens, M. "Security Considerations in the Information System Development Life Cycle." NIST Special Publication 800-64, 2004.
4. Sullivan, B. "Data Theft Affects 145,000 Nationwide." MSNBC. 18 Feb. 2005.
5. McGraw, G. Software Security: Building Security In. Addison-Wesley Professional, 2006
6. Shull, F., et al. "What We Have Learned About Fighting Defects." Proceedings of IEEE Symposium on Software Metrics. 2002.
7. Basili, V., et al. "The Empirical Investigation of Perspective Based Reading." Empirical Software Engineering – An International Journal 1.2 (1996): 133-164.
8. Porter, A. and Votta, L. "Comparing Detection Methods for Software Requirements Inspections: A Replication Using Professional Subjects." Empirical Software Engineering – An International Journal 3.4 (1998): 355-379.
9. Zhang, Z., Basili, V., and Shneiderman, B. "Perspective-Based Usability Inspection: An Empirical Validation of Efficacy." Empirical Software Engineering – An International Journal 4.1 (1999): 43-70.
10. Laitenberger, O., Atkinson, C., Schlich, M., and El Emam, K. "An Experimental Comparison of Reading Techniques for Defect Detection in UML Design Documents." Journal of Systems and Software 53.2 (2000):183-204.
11. Laitenberger, O., El Emam, K., and Harbich, T.G. "An Internally Replicated Quasi-Experimental Comparison of Checklist and Perspective Based Reading of Code Documents." IEEE Transactions on Software Engineering 27.5 (2001): 387-421.
12. Conradi, R., et al. "Object-Oriented Reading Techniques for Inspection of UML Models – An Industrial Experiment." Proceedings of European Conference on Object-Oriented Programming (ECOOP '03), Darmstadt, Germany, 2003.
13 Araujo, R. and Curphey, M. "Software Security Code Review: Code Inspection Finds Problems." Software Magazine: July 2005.

About the Authors

Dr. Jeffrey Carver is an Assistant Professor in the Computer Science and Engineering Department at Mississippi State University. His research interests include software process improvement, software quality, software inspections, and software engineering for high performance computers. Carver's research has been funded by the U.S. Army Corps of Engineers, the U.S. Air Force, and the National Science Foundation. He received his doctorate from the University of Maryland in 2003.

Department of Computer Science and Engineering
300 Butler Hall
Box 9637
Mississippi State University, MS 39762
Phone: (662) 325-0004
Fax: (662) 325-8997
E-mail:carver@cse.msstate.edu

Dr. Forrest Shull is a senior scientist at the Fraunhofer Center for Experimental Software Engineering, Maryland (FC-MD). At FC-MD he is project manager and member of the technical staff for projects with clients that have included Fujitsu, Motorola, NASA, and the U.S. Department of Defense. He is responsible for research projects in the areas of software defect reduction and software practice evaluation. He received his doctorate degree from the University of Maryland, College Park.

Fraunhofer Center for Experimental Software Engineering Maryland
University of Maryland
4321 Hartwick RD STE 500
College Park, MD 20742-3290
Phone: (301) 403-8970
Fax: (301) 403-8976
E-mail: fshull@fc-md.umd.edu

Dr. Ioana Rus is a scientist at the FC-MD where she serves as a technical area lead for safety and security. Rus has represented FC-MD as a member of the Software Assurance Processes and Practices Working Group and as a reviewer for the DHS Software Assurance Common Body of Knowledge. She received her doctorate from Arizona State University.

Fraunhofer Center for Experimental Software Engineering Maryland
University of Maryland
4321 Hartwick RD STE 500
College Park, MD 20742-3290
Phone: (301) 403-8971
Fax: (301) 403-8976
E-mail: irus@computer.org