Supply Chain Risk Management

Understanding Vulnerabilities in Code You Buy, Build, or Integrate

Paul R. Croll, CSC

© 2011 IEEE. Reprinted, with permission, from the Proceedings of the 5th Annual IEEE Systems Conference, April 2011

Abstract. This paper describes the scope of the problem regarding software vulnerabilities and the current state of the practice in static code analysis for software assurance. Recommendations are made regarding the use of static analysis methods and tools during the software life. Static code analysis touch points during lifecycle reviews and challenges to automated static code analysis are also discussed.

Section 1. Introduction

Managing software risk in the supply chain is in large part about discovering and understanding the vulnerabilities that might exist in code that you might buy as standalone applications or integrate into other systems or products. It is also about vulnerabilities you might build into code that you develop in-house. Static code analysis can be an effective means for determining the vulnerabilities in your code.

a. Scope of the Problem

Capers Jones [1] described the results of a survey of the U.S. software industry as of 2008. Based on those data, Tables 1 and 2 address the number and severity of software vulnerabilities in several classes of application projects. For military projects, as one approaches systems the size of typical large combat systems (expressed as function points), the estimated number of security vulnerabilities rises to above 3000 and the probability of serious vulnerabilities rises to 45%. The statistics are much worse for civilian and commercial systems. These systems have tended to make much more extensive use of COTS. As we move more and more into COTS and open source software for our national defense and critical infrastructure systems, one might expect that the extent of vulnerabilities in these critical systems might nearly double.

In a study by Reifer and Bryant [2], 100 packages were selected at random from 50 public open source and COTS libraries. These spanned a full range of applications and sites like SourceForge. The packages were analyzed by college students using a variety of tools.

Size in FP	Commercial Projects	Civilian Government Projects	Military Projects	*Average*
1	1	1	1	*1*
10	5	5	4	*5*
100	28	29	14	*24*
1,000	150	155	55	*120*
10,000	794	832	209	*600*
100,000	4,217	4,467	794	*3,031*
1,000,000	22,387	23,988	3,020	*15,412*
Average	*3,940*	*4,211*	*585*	*2,742*

Table 1. Estimated Number of Vulnerabilities in Software Applications

Size in FP	Commercial Projects	Civilian Government Projects	Military Projects	*Average*
1	15.00%	25.00%	5.00%	*11.29%*
10	30.00%	35.00%	15.00%	*26.00%*
100	40.00%	45.00%	20.00%	*33.57%*
1,000	60.00%	62.00%	30.00%	*54.57%*
10,000	77.00%	80.00%	35.00%	*74.00%*
100,000	85.00%	87.00%	40.00%	*80.14%*
1,000,000	96.00%	92.00%	45.00%	*86.29%*
Average	*57.57%*	*60.86%*	*27.14%*	*52.27%*

Table 2. Probability of Serious Security Vulnerabilities in Software Applications

The objectives were to:

- Determine if the packages were up-to-date with respect to vendor identified vulnerabilities and patches
- Assess if packages were free of known viruses, worms, Trojans and spyware
- Assess if the packages had weaknesses in the code and backdoors, using reverse engineering techniques
- Assess if the packages had potential dead code, malware, unwanted behaviors, or undesired functionality

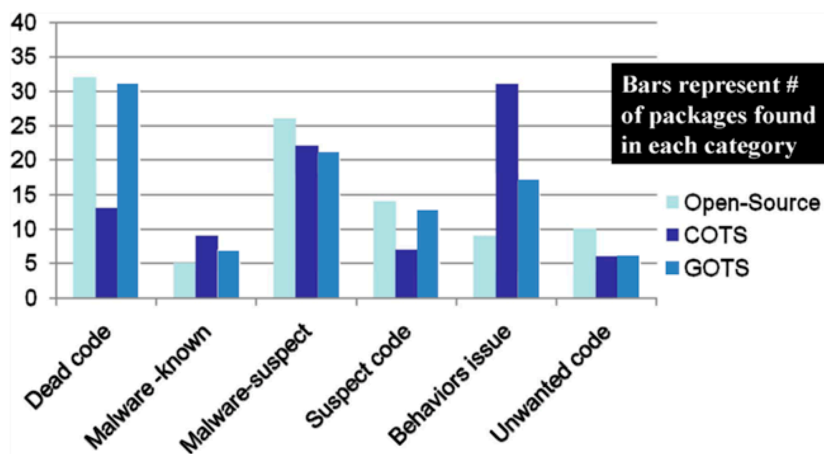

Figure 1. COTS Study Findings. Source: D. Reifer and E. Bryant, Software Assurance in COTS and open source Packages, DHS Software Assurance Forum, October 2008

Figure 1 describes the results of this small study. Over 30% of open source and Government Off the Shelf (GOTS) packages analyzed had dead code, an anathema to the software safety community, and a concern of the software security community as well. Over 20% of the open source, COTS, and GOTS packages had suspected malware, and over 30% of the COTS packages analyzed had behavioral problems.

Reifer and Bryant conclude that the potential for malicious code in applications software is large as more and more packages are used in developing a system. They have been developing a tool for analyzing software executables, often the only thing available from COTS suppliers. They have a method and tool that is available now. These focus on analyzing software executables, often the only thing available from COTS suppliers.

b. What is Static Source Code Analysis?

Static analysis is the process of evaluating a system or component based on its form, structure, content, or documentation [3]. From a software assurance perspective, static analysis addresses weaknesses in program code that might lead to vulnerabilities. Such analysis may be manual, as in code inspections, or automated through the use of one or more tools. A static analysis tool is a program written to analyze other programs for flaws [4]. Such analyzers typically check source code. There is also a smaller set of analyzers that check byte code and binary code as well. While testing requires code that is relatively complete, static analysis can be performed on modules or unfinished code. Manual analysis, or code inspection, can be very time-consuming, and inspection teams must know what security vulnerabilities look like in order to effectively examine the code. Static analysis tools are faster and do not require the tool operator to have the same level of security expertise as a code inspector [5].

Section 2. Strategies for Effective Source Code Analysis

a. What Code Do You Analyze?

How do you prioritize a code review effort when you have thousands of lines of source code, and perhaps object code to review? From a software assurance perspective, looking at attack surfaces is not a bad place to start [6]. A system's attack surface can be thought of as the set of ways in which an adversary can enter the system and potentially cause damage. The larger the attack surface, the more insecure the system [7]. Higher attack surface software requires deeper review than code in lower attack surface components. Howard [8] proposes several heuristics as an aid to determining code review priority, that is, given a large amount of code to review, what kinds of code do you emphasize for review. They are summarized below:

Legacy code: Howard points out that legacy code may have more vulnerabilities than newly developed code because security issues likely were not as well understood when the legacy code was created.

Code that runs by default: Howard suggests that attackers will often attempt to exploit code that runs by default. He also suggests that code running by default increases an application's attack surface, which is a product of all code accessible to attackers.

Code that runs in elevated context: Code that runs with elevated privileges, e.g. root privileges, for example, should also be reviewed earlier and deeper because compromise of such code can allow attackers to execute commands that are intended only for privileged users such as a site administrator.

Anonymously accessible code: Howard suggests that code that permits anonymous access should be reviewed in greater depth than code that only allows access to valid users and administrators.

Code connected to a globally accessible network interface: Howard strongly states that code that interfaces with a network, especially uncontrolled networks like the Internet, presents substantial risk. Such code increases the potential attack surface for the system.

Code written in a language whose features facilitate building in vulnerabilities: Howard suggest that code written in languages like C and C++, have features, like direct memory access, that allow programmers to inadvertently insert vulnerabilities, like buffer-overflow vulnerabilities. Howard also points out other language vulnerabilities, such as SQL-injection vulnerabilities in Java, or C# code. ISO/IEC TR 24772:2010 [9] specifies software programming language vulnerabilities to be avoided where assured behavior is required. These vulnerabilities are described in a generic manner that is applicable to a broad range of programming languages.

Code with a history of vulnerabilities: Code that has had a number of past security vulnerabilities should be suspect, unless it can be demonstrated that those vulnerabilities have been effectively removed.

Code that handles sensitive data: Code that handles sensitive data should be analyzed to ensure that weaknesses in the code not compromise such data by disclosing it to untrusted users.

Complex code: Complex code has a higher bug probability, is more difficult to understand, and may likely have more security vulnerabilities.

Code that changes frequently: Howard points out that frequently changing code often results in new bugs being introduced. Not all of these bugs will be security vulnerabilities, but compared with a stable set of code that is updated only infrequently, code that is less stable will probably have more vulnerabilities in it.

b. A Three-phase Code Analysis Process

Howard [8] also suggests a notional three-phase code analysis process that optimizes the use of static analysis tools.

1. Phase 1 – Run all available code-analysis tools

Howard suggests that multiple tools should be used to offset tool biases and minimize false positives and false negatives. This makes great sense if your organization can afford it. Strengths and weaknesses vary from tool to tool [10, 11]. Warnings from multiple tools may indicate code that needs closer scrutiny through manual inspection.

Additionally, these tools are most effective when run early in the lifecycle and run often [12]

Howard also suggests that code should be evaluated early, and re-evaluated throughout its development cycle.

2. Phase 2 – Look for common vulnerability patterns

Howard recommends that analysts make sure that code reviews cover the most common vulnerabilities and weaknesses. Sources for such common vulnerabilities and weaknesses include the Common Vulnerabilities and Exposures (CVE) and Common Weaknesses Enumeration (CWE) databases, maintained by the MITRE Corporation and accessible on the web at: <http://cve.mitre.org/cve/> and <http://cwe.mitre.org/>. MITRE, in cooperation with the SANS Institute, also maintains a list of the "Top 25 Most Dangerous Programming Errors [13]" that can lead to serious vulnerabilities. The top three classes of errors as of December 2010 were cross-site scripting, SQL injection, and buffer overflows. Static code analysis tool and manual techniques should at a minimum, address these Top 25.

3. Phase 3 – Use manual analysis for risky code

Howard also suggests that analysts should also use manual analysis (e.g. code inspection) to more thoroughly evaluate any risky code that has been identified based on the attack surface, or based on the heuristics described earlier. Manual analysis allows detailed tracing of code paths and data usage.

Section 3. The Assurance Case

An Assurance Case is a set of structured assurance claims, supported by evidence and reasoning that demonstrates how assurance needs have been satisfied [14].

- It shows compliance with assurance objectives.
- It provides an argument for the safety and security of the product or service.
- It is built, collected, and maintained throughout the lifecycle.
- It is derived from multiple sources.

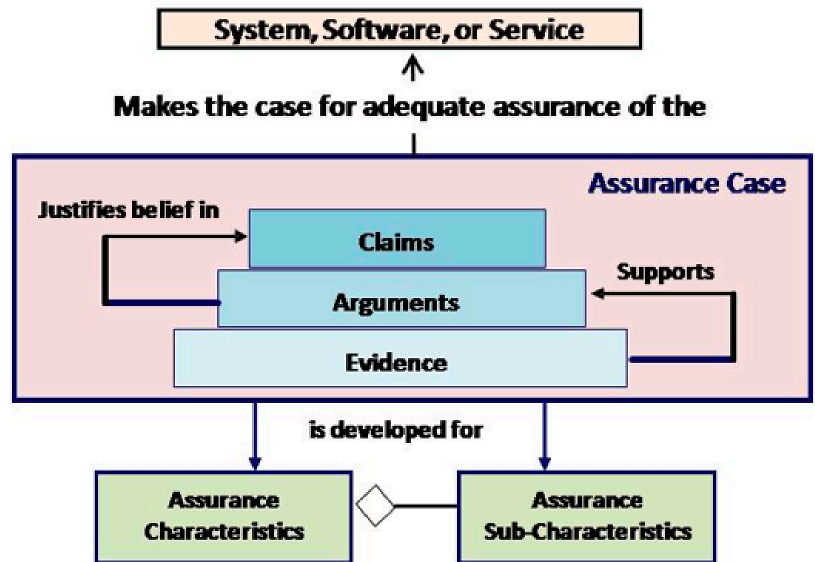

Figure 2. The Assurance Case

As shown in Figure 2, the Assurance Case should be used to document claims about the security of a software product or system. Those claims must be supported by arguments regarding the security characteristics of the software, and those arguments must be firmly supported by evidence.

The results obtained from static code analysis provide evidence regarding vulnerabilities in code, and should be documented as part of the Assurance Case.

The Sub-parts of an assurance case include:

- A high level summary
- Justification that product or service is acceptably safe, secure, or dependable
- Rationale for claiming a specified level of safety and security
- Conformance with relevant standards and regulatory requirements
- The configuration baseline
- Identified hazards and threats and residual risk of each hazard and threat
- Operational and support assumptions

An Assurance Case should be part of every acquisition in which there is concern for IT security. It should be prepared by the supplier and describe the assurance-related claims for the software being delivered, the arguments backing up those claims, and the hard evidence supporting those arguments.

The details of how static code analysis was used in the development process and the results of such static analysis should be included to support assurance arguments.

Section 4. Static Code Analysis in the Software Lifecycle

Project Managers (PMs) have a responsibility to ensure that security requirements are addressed throughout the software lifecycle. This responsibility includes conducting risk assessments; documenting system threats and vulnerabilities, including test and remediation plans on a continuing basis. Static code analysis contributes to documenting system weaknesses and vulnerabilities.

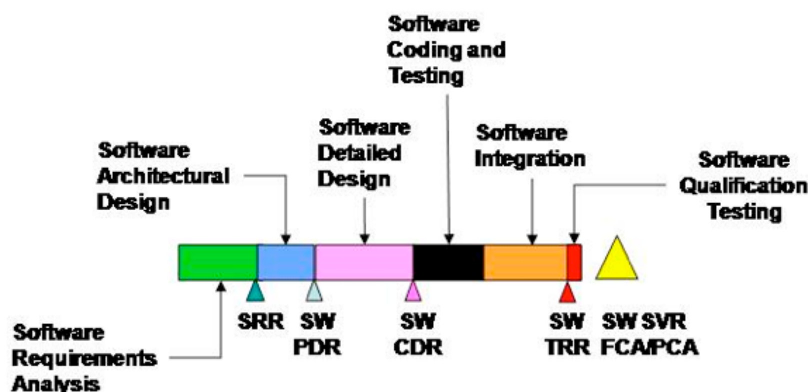

Figure 3. Reviews in the Software Lifecycle

Static code analysis should be applied at several points in the software acquisition and development lifecycle.

The reviews that are associated with software are shown in Figure 3 [15]. The following discussion addresses the objectives and expected outcomes of these reviews, describing the touch points for static code analysis in the software lifecycle review process [16].

a. System Requirements Review (SRR)

1. Objectives

The SRR helps the PM understand the scope of the software assurance landscape (assurance requirements, elements to be protected, the threat environment) in which context static code analysis should be applied.

2. Outcomes

• Establishment of the System Assurance Case

The Assurance Case both sets the context for static code analysis and provides a repository for analysis results. As discussed earlier and emphasized here, the Assurance Case should include:

-Specification of the top-level system assurance claims that address identified threats.

-Identification of the approach for developing the system assurance case.

-Identification of all critical elements to be protected.

-Identification of all relevant system assurance threats and their potential impact on critical system assets.

-Identification of high-level potential weaknesses in the system.

-Determination and derivation of system assurance requirements (as a subset of the system requirements).

• Test and Evaluation Master Plan (TEMP) addressing system assurance.

The TEMP or establishes the test strategy for testing throughout the development lifecycle.

-Examine the TEMP to ensure testing processes are sufficient for system assurance. This may include planning for static code analysis.

• Support and Maintenance Concepts

Support and Maintenance concepts addresses the need to address assurance concerns beyond development, throughout the life of the system. Outcomes include:

-Documentation of the support and maintenance concepts including a description of how assurance will be maintained.

-Description of what static code analysis tools will be used post deployment and how and when they will be applied.

b. Preliminary Design Review (PDR)

1. Objectives

The PDR is a multi-disciplined technical review to ensure that the system under review can proceed into detailed design, and can meet the stated performance requirements within cost (program budget), schedule (program schedule), risk, and specific assurance requirements and constraints.

2. Outcomes

• Information security technology evaluation of all critical COTS/GOTS elements.

As discussed earlier, COTS/GOTS components might present security risks. As part of the analysis of alternatives process, candidate components should be vetted with respect to their security characteristics. The Assurance Case should also be updated based on the components selected, and any new weaknesses and vulnerabilities identified.

The outcomes from the evaluation of COTS/GOTS elements should include:

-Specification of assurance-specific static analysis and assurance-specific criteria to be examined during code reviews.

-Documentation of the results of static code analyses performed on GOTS/COTS components.

-Documentation regarding which tools were used to perform static code analysis.

-Documentation of weaknesses and vulnerabilities that were discovered.

-Documentation of code reviews performed during implementation.

• Configuration management.

The preliminary configuration management plan must support protection of each configuration item, addressing vulnerabilities that might creep in during the change process. This includes requirements, architectures, designs, and code. The outcomes associated with configuration management include:

-Discussion regarding at which stages of the configuration management process static code analysis will be applied.

-Discussion of what configuration change events will trigger code analysis.

-Description of which components will be analyzed.

-Description of how the results of the analyses will be documented.

The Assurance Case should also be updated with relevant evidence as a result of the PDR.

3. Other Considerations

Use of COTS and open source presents a supply chain assurance challenge. As part of an analysis of the supplier and its processes, the following should be determined.

- Will the supplier perform static code analysis as part of its code development and/or code integration processes?
- Which components will be analyzed? Which will not?
- What tools do they plan to use?
- What are the details of their code inspection process for manual security analysis?
- How will they mitigated any discovered vulnerabilities

or weaknesses?

COTS source code is rarely available to the acquirer for independent code review.

PMs should request COTS vendors provide Assurance Cases for their COTS products detailing both the vendor's secure coding practices and the results of internal static code analysis or third party assessment (e.g. Common Criteria certification).

In cases where such information is unavailable, and there is still a desire to use the COTS component, the PM should consider analyzing the executables using binary code analysis.

c. Critical Design Review (CDR)

1. Objectives

The CDR is a multi-disciplined technical review to ensure that the system under review can proceed into system fabrication, demonstration, and test, and can meet the stated performance requirements within cost (program budget), schedule (program schedule), risk, and specific assurance requirements and constraints.

From a software perspective, the CDR focuses on the completeness of the detailed design and how it supports functional, performance, and assurance requirements.

2. Outcomes

With respect to software security and code analysis, the CDR should document:

• Identification and use of the selected static analysis tools for source code evaluation.

• Selection of additional development tools and guidelines to counter weaknesses and vulnerabilities in the system elements and development environment(s), including:

-Definition and selection of assurance-specific static analyses and assurance-specific criteria to be examined during peer reviews performed during implementation.

-Planning for training for assurance-unique static analysis tools and peer reviews.

The Assurance Case should also be updated with relevant evidence as a result of the CDR.

d. Test Readiness Review (TRR)

1. Objectives

The TRR is a multi-disciplined technical review to ensure that the subsystem or system under review is ready to proceed into formal test. The TRR also examines lower-level test results, test plans, test objectives, test methods, and procedures to verify the traceability of planned tests to program requirements.

2. Outcomes

•Verification of static code analysis.

Verification regarding static code analysis determines if assurance-specific static analyses and peer reviews of assurance criteria have been completed. Such verification includes:

-Documentation of evidence that static analysis has been performed (both source and binary) to identify weaknesses and vulnerabilities such as cross-site scripting, SQL injection, and buffer overruns.

-Verification that another party other than the developer (such as a peer) performed static analysis and peer review.

-Documentation regarding the selection of any additional static analysis tools to identify or verify weaknesses and vulnerabilities in the system elements and development environment(s).

-For COTS/GOTS software products with no source code, identification of industry tools and test cases to be used for the testing of any binary or machine-executable files.

The Assurance Case should also be updated with relevant evidence as a result of the TRR.

Even for those with less formal lifecycle review processes, there will generally be a requirements development phase, one or more design phases, and implementation and testing phases. For some organizations there will be operations and maintenance phases as well. The objectives and outcomes of the lifecycle touch points described above for static code analysis should provide guidance and help set expectations, no matter how formal or informal the lifecycle review process.

Section 5. Challenges to Automated Static Code Analysis

There are two challenges to the effective uses of automated static code analysis.

a. Procurement and Maintenance of Tools

The better static code analysis tools are expensive. However, the best results are obtained when multiple tools are used to offset tool biases and minimize false positives and false negatives. Use of multiple tools can quickly become cost prohibitive for a single project.

In addition, maintenance agreements to ensure a tool is up to date with respect to the spectrum of threats, weaknesses, and vulnerabilities add long term costs.

The concept of "buy it once, use it often" provides the most bang for the buck. Pooled resources analysis labs that support multiple projects within organizations may make the most economic sense.

b. Training

Static code analysis is not for sissies, although it may be for CISSPs® (Certified Information System Security Professionals). This tongue-in-cheek statement belies the difficulty in using static code analysis tools to their best advantage.

Chandra, Chess, and Steven [17] point out that when static code analysis tools are employed by a trained team of code analysts, false positives are less of a concern; the analysts become skilled with the tools very quickly; and greater overall audit capacity results.

In addition, in order to determine the validity of static code analysis results, it is important for PMs to understand the level of training that code analysts have had with the tools employed for static code analysis as well as their understanding of code weaknesses and vulnerabilities. Even a good tool in the hands of a poorly trained or inexperienced code analyst can produce misleading results. A tool is just a tool. How it is used and how its results are interpreted are key to useful and valid results.

Section 6. Useful Links

a. NIST Software Assurance Metrics and Tool Evaluation (SAMATE) Static Analysis Tool Survey

The NIST SAMATE project provides tables describing current static code analysis tools for source, byte, and binary code

analysis <http://samate.nist.gov/>.

b. DHS Build Security In Web Site

This site contains a wealth of software and information assurance information, including white papers on static code analysis tools. More information on Build Security In can be found at:

<https://buildsecurityin.us-cert.gov/daisy/bsi/home.html>

c. CWE

This site provides a formal list of software weakness types created to:

- Serve as a common language for describing software security weaknesses in architecture, design, or code.
- Serve as a standard measuring stick for software security tools targeting these weaknesses.
- Provide a common baseline standard for weakness identification, mitigation, and prevention efforts. <http://cwe.mitre.org/>

d. CWE/SANS Top 25 Most Dangerous Software Errors

The 2010 CWE/SANS Top 25 Most Dangerous Software Errors is a list of the most widespread and critical programming errors that can lead to serious software vulnerabilities. They are often easy to find, and easy to exploit. They are dangerous because they will frequently allow attackers to completely take over the software, steal data, or prevent the software from working at all.

<http://cwe.mitre.org/top25/archive/2010/2010_cwe_sans_top25.pdf>

Section 7. Summary

This paper has described the scope of the problem regarding vulnerabilities in the code we buy, build, or integrate. As more and more COTS and open source components are integrated into our systems, the problem becomes ever more exacerbated.

The paper has also discussed strategies for effective static code analysis as a means to understand an manage supply chain risk, and has described the expected outcomes regarding such analysis at appropriate touch points in the software lifecycle. Although the lifecycle reviews described were fairly formal, the activities associated with those reviews apply to any software development, integration, or maintenance effort. In addition, the paper has described the Assurance Case, the repository for, among other things, the empirical results of static analysis.

Lastly, the paper touched on challenges to automated static code analysis, regarding the procurement and maintenance of tools and the training required for tool users in order to facilitate accurate results. Such analysis is most effective when multiple tools are used to offset tool biases, and are employed by analysts with proper training in both tool use and in security-related code inspection.

To be sure, there are other means for assessing and managing supply chain risk with respect to software, but at the bottom line, it is all about the code and the vulnerabilities it might contain.

The Software Assurance Community Resources and Information Clearinghouse contains links to free Pocket Guides on other aspects of supply chain risk management, including:

- Software Assurance in Acquisition and Contract Language
- Software Supply Chain Risk Management and Due Diligence
- Key Practices for Mitigating the Most Egregious Exploitable Software Weaknesses
- Software Security Testing
- Secure Coding

Guides on other aspects of software assurance include:

- Requirements and Analysis for Secure Software
- Architecture and Design Considerations for Secure Software
- Software Assurance in Education, Training & Certification

All of these guides can be found at:

<https://buildsecurityin.us-cert.gov/swa/pocket_guide_series.html>

CALL FOR ARTICLES

If your experience or research has produced information that could be useful to others, CrossTalk can get the word out. We are specifically looking for articles on software-related topics to supplement upcoming theme issues. Below is the submittal schedule for three areas of emphasis we are looking for:

Resilient Cyber Ecosystem
Sept/Oct 2012 Issue
Submission Deadline: Apr 10, 2012

Virtualization
Nov/Dec 2012 Issue
Submission Deadline: June 10, 2012

Please follow the Author Guidelines for CrossTalk, available on the Internet at <www.crosstalkonline.org/submission-guidelines>. We accept article submissions on software-related topics at any time, along with Letters to the Editor and BackTalk. To see a list of themes for upcoming issues or to learn more about the types of articles we're looking for visit <www.crosstalkonline.org/theme-calendar>.

ABOUT THE AUTHOR

Paul Croll is a Fellow in CSC's Defense Group and Chief Scientist of the Defense & Maritime Enterprise Technology Center, where he is responsible for researching, developing and deploying systems and software engineering practices, including practices for cybersecurity.

Paul has over 35 years experience in mission-critical systems and software engineering. His experience spans the full life cycle and includes requirements specification, architecture, design, development, verification, validation, test and evaluation, and sustainment for complex systems and systems-of-systems. He has brought his skills to high profile, cutting edge technology programs in areas as diverse as surface warfare, air traffic control, computerized adaptive testing, and nuclear power generation.

Paul is also the IEEE Computer Society Vice President for Technical and Conference Activities, and has been an active Computer Society volunteer for over 25 years, working primarily to engage researchers, educators, and practitioners in advancing the state of the practice in software and systems engineering. He was most recently Chair of the Technical Council on Software Engineering and is also the current Chair of the IEEE Software and Systems Engineering Standards Committee. Paul is also the past Chair and current Vice Chair of the ISO/IEC JTC1/SC7 U.S. Technical Advisory Group (SC7 TAG).

Paul is also active in industry organizations and is the Chair of the NDIA Software Industry Experts Panel and the Industry Co-Chair for the National Defense Industrial Association (NDIA) Software and Systems Assurance Committees. In addition, Paul is Co-Chair of the DHS/DoD/NIST Software Assurance Forum Processes and Practices Working Group advancing cybersecurity awareness and practice.

E-mail: pcroll@csc.com

© 2011 IEEE. Reprinted, with permission, from the Proceedings of the 5th Annual IEEE Systems Conference, April 2011

CIVILIAN TALENT IS MISSION-CRITICAL.
LET'S GET TO WORK.
NAVAIR
CIVILIAN
CHOICE IS YOURS.
Discover more about Naval Air Systems Command today.
Go to **www.navair.navy.mil**
Equal Opportunity Employer | U.S. Citizenship Required

Work for Naval Air Systems Command (NAVAIR) and you'll support our Sailors and Marines by delivering the technologies they need to complete their mission and return home safely. NAVAIR procures, develops, tests and supports Naval aircraft, weapons, and related systems. It's a brain trust comprised of scientists, engineers and business professionals working on the cutting edge of technology.

You don't have to join the military to protect our nation. Become a vital part of NAVAIR, and you'll have a career with endless opportunities. As a civilian employee you'll enjoy more freedom than you thought possible.

REFERENCES

1. Jones, Capers. Overview of the United States Software Industry Results Circa 2008, DHS Software Assurance Forum working paper, June 20, 2008.
2. Reifer, D, and Bryant, E. "Software Assurance in COTS and Open Source Packages," Proceedings of the DHS Software Assurance Forum, October 14-16, 2008.
3. ISO/IEC JTC1/SC7. ISO/IEC 24765:2009, Systems and software engineering vocabulary.
4. Black, P. Static "Analyzers in Software Engineering," CrossTalk, The Journal of Defense Software Engineering, pp. 16-17, March-April 2009.
5. McGraw, G. "Automated Code Review Tools for Security," Computer, vol. 41, no. 12, pp. 108-111, Dec. 2008.
6. Howard, M. Mitigate Security Risks by Minimizing the Code You Expose to Untrusted Users, <http://msdn.microsoft.com/msdnmag/issues/04/11/AttackSurface, November, 2004>.
7. Manadhata, P., Tan, K, Maxion, R, and Wing, J. An Approach to Measuring a System's Attack Surface, CMU-CS-07-146, Carnegie Mellon University, August 2007.
8. Howard, M. "A Process for Performing Security Code Reviews," IEEE Security & Privacy, pp. 74-79, July-August 2006.
9. ISO/IEC TR 24772:2010, Guidance to avoiding vulnerabilities in programming languages through language selection and use.
10. U.S. Department of the Navy, Software Security Assessment Tools Review, March 2009, <https://buildsecurityin.us-cert.gov/swa/downloads/NAVSEA-Tools-Paper-2009-03-02.pdf>
11. Okun, V., Delaitre, A, Black, P. The Second Static Analysis Tool Exposition (SATE) 2009, NIST Special Publication 500-287, National Institute of Standards and Technology, June 2010.
12. Goertzel, K., Winograd, T., Enhancing the Development Life Cycle to Produce Secure Software, Rome, NY: DACS Data & Analysis Center for Software, 2008.
13. Cristey, S. 2010 CWE/SANS Top 25 Most Dangerous Software Errors. The MITRE Corporation, 2010, <http://cwe.mitre.org/top25/archive/2010/2010_cwe_sans_top25.pdf>
14. ISO/IEC/IEEE 15026-2:2010, Systems and software engineering – Systems and software assurance – Part 2: Assurance case.
15. Program Executive Office (PEO) Integrated Warfare Systems (IWS) Technical Review Manual (TRM) (Draft), Department of the Navy, Naval Sea Systems Command, Program Executive Office, Integrated Warfare Systems, December 2008.
16. National Defense Industrial Association (NDIA) System Assurance Committee. Engineering for System Assurance, Version 1.0, 2008.
17. Chandra, P., Chess, B., and Steven, J. "Putting the Tools to Work: How to Succeed with Source Code Analysis, "IEEE Security & Privacy, pp. 80-83, May-June 2006.