Locking Down the Software Development Environment

Madeline Wright, MCTD
Dr. Carl Mueller, Texas A&M

Abstract. The goal of this paper is that configuration management is a simple and cost effective method to secure the development environment without impeding innovation, creativity, or schedule. Software development is a business, and it is reasonable to assume both developers and customers want systems that are protected because there will always be attempts to gain access to software and the data/information residing in the computer systems.

Introduction

Software security is to prevent the world from harming the system, malicious or unintentional, [1] and encompasses the ability to prevent, detect, and react to malicious indicators. Many security analysts view external threat-agents as the primary source of harm. However, the greatest risk to any system is from those who are developing the system. Developers introduce security exposures either maliciously or unintentionally into the system. Mitigating these risks requires security analysis and an investigation of all vulnerabilities in the Software Development Life Cycle (SDLC) models. Investigating the vulnerabilities for all of the SDLC models is extremely labor intensive, but there are three areas common to most models: Programming personnel, Configuration Management (CM) practices, and Quality Assurance (QA) practices. Each area of the SLDC reviewed emphasizes the difference between secure coding and securing the environment by reviewing threat agents and threat exposure, assessing configuration management's role in securing the environment, and finally, discussing methods to monitor for malicious intent with lessons from Stuxnet and the Heartworm Bug. The authors look at the SDLC from a lens that strives to guide software teams to implement a comprehensive security policy to make penetration of the system's information security perimeter more difficult [2].

Programmers as Threat Agents

Microsoft reports more than 50% of the reported security defects are introduced in the design of a component [3] and this is critical to looking at where the threats exist and opportunities for threat exposure in the in the software development environment. Microsoft's finding suggests both designers and programmers are threat agents in the development environment. According to Microsoft's data, designers and programmers introduce vulnerabilities into an application; it is therefore appropriate to identify all of the software developer roles (analysts, designers, programmers, testers) as potential threat-agents. Viewing software developers as threat-agents should not imply the individuals filling these roles are careless or criminal, but they have the greatest opportunity to introduce source code compromising the systems confidentiality, integrity, or availability. Software developers can expose assets accidently, by introducing a defect or intentionally through the introduction of malicious functionality. Defects have many causes, such as oversight or lack of experience with a programming language, and are a normal part of the development process.

Development processes are the activities, constraints, resources, and techniques to produce an intended output [4]. Software developers perform the daily development activities habitually; they know their jobs, so why question whether there are flaws in the development process? The answer is that team members have their own view of the process, based on what is important to them personally. For example, someone's financial circumstances may have changed or other personal hardships could result in an angry, vulnerable, or distracted team member. Many organizations conduct background checks, credit checks, and drug tests when hiring new employees as part of the company's security policy, but the company may not perform periodic background checks. Developers need to understand that as organizations uses locks on their doors to protect their physical property; there is a need to conduct periodic security screenings to protect intellectual property and financial assets from those with the greatest access. Although these actions are intrusive, they serve to sustain a secure environment, provide stability through structure, and reduce risk.

During development of a software application, there are many opportunities to introduce security exposures. To address these exposures, many researchers recommend enhancing an organization's QA program. One frequent recommendation is expanding the inspection practice by introducing a checklist for the various exposures provided by the programming languages [3] [5]. Items added to a security inspection checklist typically include functions such as Basic's Peek () and Poke () functions, C's string copy functions, exception handling routines, and programs executing at a privileged level. Functions like Peek() and Poke() make it easier for programmers to access memory outside of the program, but a character array or table without bounds checking produces similar results. A limitation of the language specific checklist is each language used to develop the application must have a checklist. For some web applications, this could require three or more inspection checklists, and this may not provide safeguards for all the vulnerabilities. Static analyzers, such as the Software Assurance Metrics And Tool Evaluation (SAMATE) research, sponsored by the National Institute of Standards and Technology (NIST), is an approach to automating some of the objectives associated with an inspection checklist, but static analyzers have a reputation for flagging source statements that are not actually problems [6]. Using a rigorous inspection process as a safeguard identifies many defects, but does not adequately protect from exposures due to malicious functionality. An inspection occurring before the source code is placed under configuration control provides substantial exposure. In this situation, the developer simply adds the malicious functionality after the source code passes the inspection or provides the inspectors a listing without the malicious functionality.

Report Documentation Page	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **DEC 2014**	2. REPORT TYPE	3. DATES COVERED **00-00-2014 to 00-00-2014**

4. TITLE AND SUBTITLE **Locking Down the Software Development Environment**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Army Operational Test Command ,Mission Command Test Directorate (MCTD),Fort Hood,TX,76544**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The goal of this paper is that configuration management is a simple and cost effective method to secure the development environment without impeding innovation, creativity, or schedule. Software development is a business and it is reasonable to assume both developers and customers want systems that are protected because there will always be attempts to gain access to software and the data/information residing in the computer systems.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**6**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

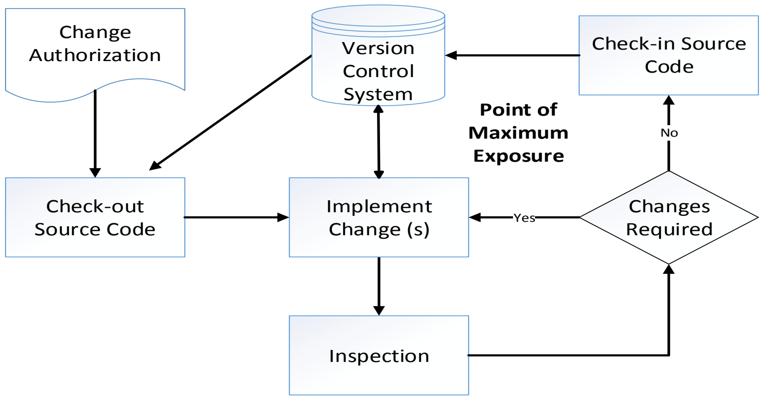


Figure 1. Traditional Unit Level Development Process

Programming languages and development processes offer a number of opportunities to expose assets; but many of the tools, such as debuggers and integrated development environments, can expose an asset to unauthorized access. Many development tools operate at the same protection level as the operating system kernel and function well as a worm to deposit a root kit or other malicious software. Another potential exposure, not related to programming languages, is testing with 'production data'. Using 'production data' may permit access to information the developers do not need to know. Now that we have addressed some areas to apply QA, we will look at the next step, addressing vulnerabilities and back doors.

Software developers address software vulnerabilities by loading the latest virus definitions and implementing all of the security guidelines [7], but if the environment is not secure, the product is still at risk for back doors or malicious intent resulting in security breaches. Acknowledging the software development environment's threat agents are often the software engineers is a major step in reviewing security processes. There is a tendency to focus on the existing vulnerabilities and threats since this provides managers and customers a sense that all security bases are covered. The development environment permits many opportunities to discover a business's processes and, in turn, their vulnerabilities, and how we protect those process leads to our next topic, Configuration Management.

Configuration Management

One tool for securing the environment is auditing through CM so the development team can focus on building functionality, not managing the change [8]. Software CM (SCM) is the traditional technique for controlling the content of deliverable components and is an essential element of a robust security policy [9]. Figure 1 illustrates a traditional unit-level development process indicating possible points of vulnerability. As illustrated in Figure 1, a developer receives a change authorization to begin the modification or implementation of a software unit. Generally, the "authorization" is verbal and the only record of the authorization appears on a developer's progress report or the supervisor's project plan. To assure another developer does not update the same source component, the developer "reserves" the necessary source modules. Next, the developer modifies the source code to have the necessary features. When all of the changes are complete, the developer informs the supervisor who assembles a review panel consisting of three to five senior developers and/or designers. The panel examines the source code to evaluate the logic and documentation in the source code. A review committee recommends the developer make major changes to the source code that requires another review, minor changes that do not require a full review, or no changes are required. It is at this point in the development process where the source code is the most vulnerable to the introduction of malicious functionality, because there are no reviews or checks before the software is "checked-in."

Another limitation of inspections is that the Agile methodology recommends formal inspections and Scrum uses pair programming and testing based on the Backlog list to determine what functionality is priority for developer resources [5]. Using inspections as the primary safeguard from development exposures limits the cost savings promised by Agile development methodologies and does not provide complete protection from a developer wishing to introduce malicious software. Of the six areas of CM, the two areas having the greatest effect on security are configuration control and configuration audits. Version control tools, such as Clearcase and CVS, provide many of the features required by configuration control. Most version-control systems permit anyone with authorized access to check source code "in" and "out" without an authorized change request and some do not even track the last access to a source module. However, in a secure environment, a version control system must integrate with the defect tracking system and record the identification of the developers who accessed a specific source module. Integrating the version control system with the defect tracking system permits only the assigned developer to make a specified change and access the related source code. It is also important for the version control system to track the developers who access the code – traceability. Frequently, developers copy source code from a tested component or investigate the approach used by another developer to address a specific issue in their work, or need access to read source modules they are not maintaining. This access provides an opportunity for insiders to research and learn how to introduce malicious functionality into another source module. By logging source module access, security personnel can monitor access to the source code. Configuration audits are the second management technique making a development organization more secure [10]. Some regulatory agencies require audits for safety critical applications/high reliability applications to provide an independent review of the delivered product. An audit in a high security environment addresses the need to assure the delivered product's software does not expose the organizational assets to risk from either defects or malicious functionality. To increase confidence that the delivered software does not contain defects or malicious functionality, auditors should assure that the test cases provided 100% coverage of the delivered source code. This is particularly important with interpreted programming languages, such as Python or other scripting languages, because a defect can permit the entry of malicious code by a remote use of the software. Auditors could adopt the approach of selectively re-testing configuration items with the unit-test data to assure the results from the re-test match those produced in the verification and validation procedure, and that all of the statements in the code are executed.

Adopting the recommendation for a stronger CM process modifies the typical unit-level development process, illustrated in Figure 1, to a more secure process illustrated in Figure 2. In the more secure process illustrated in Figure 2, a formal change authorization is generated by a defect tracking system or by the version control system's secure change authorization function. Next, a specified developer makes the changes required by the change authorization. After implementing and testing the changes, the developer checks all of the artifacts (source code, test drivers, and results) into the version control system. Checking the artifacts automatically triggers a configuration audit of the developer artifacts. Auditors may accept the developer's changes or create a new work order for additional changes. Unlike the review panel, the auditors may re-test the software to assure adequate coverage and that the test results match those checked in with the source code. Making this change to the development process significantly reduces the exposure to accidental defects or malicious functionality because it verifies the source code deployed in the final product, along with all supporting documentation.

CM should be viewed as important to keeping the development team informed of code changes, to include who made the code changes, and to indicate when those code changes were made [11]. The goal is to inform and challenge developers to merge security and process to prevent the introduction of malware by threat agents through safeguards to strengthen existing processes. CM helps keep the cost of security low-cost with the basics of version control and audit. Management will be pushing the team to deliver and move on to the next project, software developers need something practical that provides structure while not restricting innovation and creativity or impeding the schedule. CM should play a major role in securing the software development environment because it assists with the discovery and prevention of malicious intent by threat agents, whether the agents are hackers, malware coders, or insider threats. Next, we will discuss how to establish QA in the SLDC and use lessons from external threats to emphasize the importance of QA.

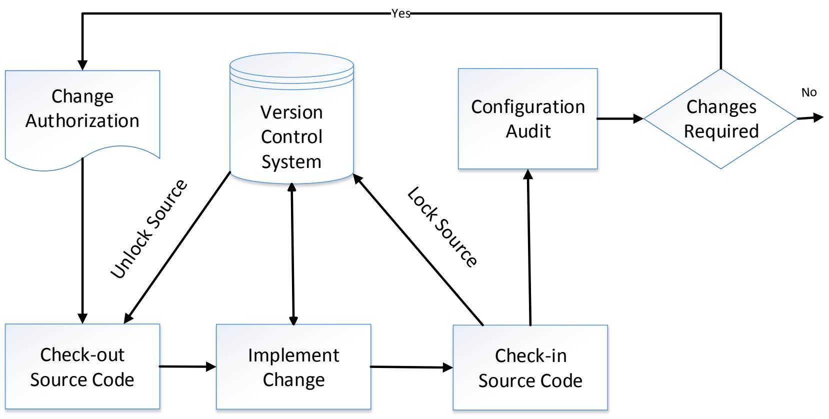


Figure 2. Secure Unit Development Process

Quality Assurance and Lessons from External Threats

Another element of a robust development security policy is QA in the SLDC through the separation of the development and production systems. Developing software in the production environment exposes organizational assets to a number of threats, such as debugging tools or simply writing a program to gain unauthorized access to information stored on the system. A worm implanted on computers or portable flash drives that might eventually be connected to the targeted network, such as the Stuxnet worm. The level of sophistication of the Stuxnet worm could only come from insider knowledge of the computing architecture and daily operations. Another important point about the Stuxnet worm is that it targeted a development tool and the tool introduced the malicious functionality. For a secure development environment, testing must not only look at the vulnerabilities of the past, but also conduct what-if analysis for all of the tools and software being used for the project developers have to think like a hacker because hackers work to reverse engineer an application to make it perform what they want. Since Stuxnet was engineered brilliantly as described by personnel from Kaspersky Lab [12], developers must take preventive action. Code can have a time delay or malware can remain dormant, or malware may be hidden in a library or service routine, all actions an insider could insert for activation after the software is deployed. The Heartbleed Bug was a software defect developers closed in April 2014 with a patch to the OpenSSL code [13]. The OpenSSL software is, as the name implies, open source, a result of many developers coding beginning in 1998 using the C programming language to build crypto services. OpenSSL is used widely both on the Internet and in firmware [13], further delaying the ability of many organizations to implement the patch across all platforms for the HeartBleed bug. There will be damage from the HeartBleed bug because of the types of data possibly extrapolated from websites, i.e. certificates, user names and passwords, messages, emails, and documents. Using open source code does not preclude the development team from checking that the code meets QA standards. Code must perform its intended function and not introduce security vulnerabilities. Identifying all of the potential exposures and creating safeguards provides a significant challenge to the security analysts; but by analyzing the development process, it is possible to identify a number of cost effective safeguards. Configuration management, pairwise coding, and vulnerability checks are all simple security safeguards that could have prevented the HeartBleed bug. Security breaches have made the news headlines and consumed labor hours for patches and corrective actions, a result of attacks on software services by software developers who have become insider threat agents. The insider threat is real, whether it is from former employees who have maintained possession of their identification cards or contracted National Security Agency (NSA) employees having access levels exceeding their authorization, the insider can upend all of a development's team effort. Social engineering in the case of Stuxnet [14] and other worms prompted the government to ban USB drives, and these examples continue to prove that securing the environment is the only option. Give hackers a face; they are thieves with an objective to gain access into a software system for which they do not possess the credentials to access or possess. Locking down the environment means the owners of the SDLC are a step ahead of the criminals because they employed QA for each step in the process. For example, developers have to begin to

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.

Discover more about NAVAIR. Go to **www.navair.navy.mil**.

Equal Opportunity Employer | U.S. Citizenship Required

design tests beyond code issues and look at routine service calls to ensure the code does not introduce a common vulnerability for exploitation upon deployment. Consider whether Bluetooth is enabling cyber espionage. Historical data from the credit card industry has proven Bluetooth as a lucrative technology for cyber thieves and should be monitored to ensure the interfaces do not contain malware. Programs should be designed to perform cyber surveillance on the malware so programs get routine reports on data sent to servers and service routines – trends and analysis can point to unusual activity. Agile takes advantage of the cross-functional teams with both developers and testers to improve the quality of the software [15]. This teaming could be used in any type of development model to encourage collaboration and feedback. The feedback loop and reinforcement of using the QA and CM processes, manual or automated, traces changes and makes changes visible to everyone on the team [8]. Visibility and sharing of information in the Agile Process make it difficult for insiders to bypass the CM process and could even prevent someone from attempting espionage. For example, if a developer wanted to add a logic bomb and deploy it six months later, he or she might reconsider if they knew the code would be peer reviewed [16] [17]. If CM personnel compare lines of code from the previous version to a new version, the records of code submissions are traceable to an individual, and this step could curtail fraudulent activity. Waterfall and its recordkeeping rigor remove the anonymity, but testing at the end still provides the insider opportunities to hide malicious code during the rush to meet the delivery schedule. So how does the development team know the code deployed is the code the team wants to deploy? Adherence to code changes being checked in to CM, and verification, i.e. testing, comparing the number of lines of code to the tested lines of code is critical to documenting the software baseline. Teams must follow the process and not allow a programmer to check in code without an audit trail. Processes remove the human-in-loop factor, i.e. relationships between the developer and the CM Lead, ensuring processes apply to everyone. The process requires buy-in at all levels because security is the responsibility of the development team. Accountability in the software development environment maintains the integrity of the software. If developers follow the CM process, then there is traceability. Development teams must not forfeit traceability and take shortcuts – the process is there to help protect the product. Not following the process opens the software development environment to vulnerabilities. Statistics from the Computer Emergency Response Team/Coordination Center (CERT/CC) indicates well-known vulnerabilities are responsible for 75% of the security breaches; leaving 25% a secure development model will not address [12]. It sounds simple, map to known vulnerabilities to improve your design and let this become a new model to assist the testers, QA teams, and automated tools. Shirazi's proposal lists 25 common vulnerabilities, such as buffer overflow, integer overflow, command injection, SQL injection, cross-site scripting, and illegal pointer values, along with countermeasures to avoid the vulnerabilities [18]. Many developers could view this model as too prescriptive, because it assumes if most programmers followed basic secure programming principles, then the environment would be locked down. Some would also view this model as useful to train inexperienced developers or to standardize knowledge across the organization. Any list can become a useful checklist, but a model is a process that can be repeated to impose consistency on the output. Consider how software development teams moved from Waterfall to Agile; the process selected by the team depends on the environment and business need. Additionally, change has to be significant to encourage and sustain change [14]. Secure software development does not require a new model, but the phases of the existing models must address security and the system architecture in the test and QA phases. Another proposed secure software development model is to build test tools based on ratios of predicted vulnerabilities [15]. Any model relying on reported vulnerabilities or associations attempts to predict future vulnerabilities does provide indicators [19], but until the model's variables are validated, caution must be used. These proposals help to code securely, but still lack in providing a means to locking down the software environment [14]. While past vulnerabilities do assist with inspection and testing [16], [17], this is still a reactive model. Code churn is expected during development, but sound CM practices can manage the change. Additionally, there are many secure development lifecycle models (SDL), for example, Microsoft's SDL is attempting to train new developers. This makes the development team aware of the pitfalls of functional but unsecure software [20], but this still does not secure the environment, and training does ensure the developers understand secure coding. Secure coding does not prevent malicious intent within the environment. As one technology manager stated, developers' mistakes rarely affect them directly because the defects are not discovered until the operations phase. Correcting defects once software is deployed affects the operations budget, not the development budget [21]. Some security professionals attempt to scare the developers with doom and gloom about security and privacy problems, as well as describing how best to ensure compliance to regulations through secure coding [22]. There should be an emphasis on information sharing; inform the development teams and everyone in the company about how much it costs in operations and marketing to clean up the environment if malware is not prevented [23] [24]. Steps such as training, meetings to gain buy-in, rewards to developers who embrace methods that secure the environment, as well as reprisals for those who do not adhere to doing their part to secure the environment are necessary to instill a culture change so everyone values securing the environment. Some notable examples of the threat agent's work were the use of social engineering and development tools in the case of Stuxnet and Heartbleed Bug. Both exploits resulted in havoc and further degraded the public's ability to trust the security of software. Developers must take action by viewing the software development environment as an entity within the SLDC that requires a rigorous auditing method. Development is the stage in the SLDC is when the software is most vulnerable to threat agents so there has to be a focus on gaps in the development process. The company developing software does not discharge its responsibility after its deployment; the creation, however perceived, remains a product that is company's responsibility until the software is retired.

Summary

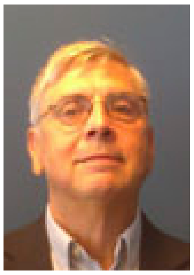
Software development is a business, and it is reasonable to assume that customers want systems that are protected against known vulnerabilities. The insider threat is real, whether it is from former employees who have maintained possession of their identification cards or contracted National Security Agency (NSA) employees having access levels exceeding their authorization, the insider can upend all of a development's team effort. Social engineering in the case of Stuxnet [14] and other worms prompted the government to ban USB drives, and these examples continue to prove that securing the environment is the only option. Software developers can address software vulnerabilities, load the latest virus definitions, and implement all of the security guidelines [7], but if the environment is not secured, the product is still at risk for back doors and malicious intent that lead to security breaches. Security breaches result in damages to customer and developer reputation. Only a comprehensive security policy focusing on personnel, operations and configuration management can provide the safeguards necessary to secure an organization's assets from cyber risk factors [2]. Following all of these recommendations will not guarantee the security of the software development environment. There are always new vulnerabilities and vulnerabilities from social engineering. However, using reoccurring security checks, separating developers from production systems and data, controlling media, training developers, a culture of awareness, and rigorous configuration management practices should make penetration of your information security perimeter more difficult. It is also necessary to conduct a periodic review of development tools and configuration management practices, as well as a review of the security standards because threat agents will adapt to any safeguard that does not adapt to new technology. It makes good business sense to allocate resources proportionately across the development and maintenance phases to prevent malicious intent, versus stopping the current work and re-directing work to address malicious intent. Technology changes, customers' business changes, the market changes, and management changes are all drivers that will influence and determine how the customer will assess the quality of the final software product.

Although this paper does not strive to develop any new methodologies, models, processes, or test tools, each area reviewed in the SLDC emphasized the difference between secure coding and securing the environment with a focus on the effectiveness and simplicity of Configuration Management in keeping the cost of security low-cost with the basics of version control, quality assurance, and auditing. Any security policy focusing on one aspect of the process will not succeed - the software development environment is complex, encompassing people, systems, and software. Since management will be pushing the team to deliver and move on to the next project, software developers need something practical that provides structure while not restricting innovation and creativity or impeding the schedule. In that vein, the authors of this paper hope to have contributed.

ABOUT THE AUTHORS

Madeline Wright has more than 15 years of experience in software testing and evaluation of mission command systems through test planning, resource management, and collecting data to support operational tests She currently serves as senior test manager for the Mission Command Test Directorate (MCTD), US Army Operational Test Command (OTC), a test center for the US Army Test and Evaluation (ATEC) Command. Mrs. Wright gained system of system test experience at U.S. Army Communications-Electronics Command's (CECOM) Whitfill Central Technical Support Facility (CTSF), supporting the Electronic Proving Ground (EPG), to lead the interoperability testing of new software baselines for the Army's Chief Information Office/G-6. Mrs. Wright's MCTD team also has the responsibility to manage testing as system under test (SUT) headquarters for OTC during the Network Integration Evaluation (NIE) events; this includes mission command systems, network capabilities, and radios that are planned for fielding to keep the Warfighter with technology upgrades. Mrs. Wright holds a BA with a major in Computer Information Systems and an MBA from Tarleton State University – Central TX, as well as two Master Certificates from Villanova University in Six Sigma Green Belt and Lean Six Sigma (LSS). Mrs. Wright is Level III certified in Information Technology by the Defense Acquisition University (DAU) and she is a member of the Army Acquisition Corps. Mrs. Wright is working towards her DAU Test and Evaluation certification. Her publications include abstracts/presentations at the 26th and 27th NDIA Conferences. Mrs. Wright lives in Harker Heights, TX with her husband of 27 years, Charles, a retired Army NCO. The Wrights have two daughters, Charlene who serves with the Navy, and Jamilla, who lives in Dallas, TX. Mrs. Wright is pursuing a Computer Science/Software Engineering degree with TAMU-CT prior to pursuing a doctorate as part of her continuing education. Mrs. Wright enjoys family time and traveling. She supports the Killeen Food Bank with food drives and raising funds for scholarships as part of her duties as President of the Delta Mu Delta Zeta Lambda Chapter at Texas A&M University – Central Texas (TAMU-CT).

Phone: 254-286-6300 ext. 6302
E-mail: madeline.b.wright.civ@mail.mil



Carl J. Mueller, Ph.D., CISSP. Dr. Mueller is an assistant professor in the Department of Computer Information Systems, TAMU-Central Texas in Killeen. He received is initial training in computer science from Marine Corps Schools Quantico, and later earned a PhD from Illinois Institute of Technology. Dr. Mueller has over 9 years of teaching experience and more than 35 years of industrial experience specializing in developing and testing safety critical/high reliability applications (medical devices, telephony, and other applications).

Phone: 254-519-5400
E-mail: muellercj@ct.tamus.edu

REFERENCES

1. C. W. Axelrod, Engineering Safe and Secure Software Systems, Norwood, MA: Artech House, 2012, p. 349.
2. M. Mushi and J. Bakari, "Security in In-House Developed Information Systems: THe Case of Tanzania," Systemics, Cybernetics and Informatics, vol. 10, no. 2, pp. 1-5, 10 2012.
3. N. Davis, W. Humphre, S. J. G. Zibulski and G. McGraw, "Processes for Producing Secure Software: Summary of US National Cybersecurity Sumit Subgroup Report," IEEE Security and Privacy, vol. 2, pp. 18-25.
4. S. L. Pfleeger and J. M. Atlee, Software Engineering Theory and Practice, Prentice Hall, 2010.
5. M. Beedle and K. Schwaber, Agile Software Development with Scrum, Upper Saddle River, NJ: Prentice Hall, 2002, pp. 1-158.
6. "SAMATE - Software Assurance Metrics and Tool Evaluation," 2014.
7. H. Shahriar and M. Zulkernine, "Mitigating Program Security Vulnerabilities: Approaches and Challenges," ACM Computing Surveys, vol. 44, no. 3, pp. 11.1-11.46, 6 2012.
8. M. E. Moreira, Adapting Configuration Mangement for Agile Teams: Balancing Sustainability and Speed, Hoboken, NJ: Wiley, 2010, p. 303.
9. A. Leon, A guide to software configuration management, Artech House, Inc, 2000.
10. N. R. Nielsen, "Computers, security, and the audit function," in Proceedngs of the May 19-22, 1975, National Computer Conference and Exposition, Anaheim, CA, 1975.
11. H. R. Berlack, Software Configuration Management, New York: Wiley, 1992.
12. D. Kushner, "The real story of stuxnet," IEEE Spectrum, vol. 50, no. 3, pp. 48-53, 2013.
13. J. Lyne, "How Hearbleed Happened, The SNA and Proof Heartbleed Can Do Real Damage," 2014.
14. I. Porche, S. McKay and J. M. and Sollinger, A Cyberworm That Knows No Boundaries, Santa Monica: Rand Corp, 2011.
15. J. L. Cooke, Everything You Want to Know About Agile: How to Get Agile Results in a Less-than-agile Organization, Ely, Cambridgeshire: IT Governance, 2012.
16. R. L. Jones and A. Rastogi, "Secure Coding: Building Security into the Software Life Cycle," Information Systems Security, pp. 29-39, 11/12 2004.
17. V. O. Safonov, Using Aspect-Oriented Programming for Trustworthy Software Development, Hoboken, NJ: Wiley-Interscience, 2008.
18. H. M. Shirazi, "A New Model for Secure Software Development," International Journal of Intelligent Information Technology Application, vol. 2, no. 3, pp. 136-143, 2009.
19. Y. Shin, A. Meneely, L. William and J. A. Osborne, "Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities," IEEE Transactions on Software Engineering, vol. 37, no. 6, pp. 772-787, 2011.
20. P. C. Jorgensen, Software Testing: A Craftsman's Approach, Boca Raton: Auerbach, 2008.
21. D. Bradbury, "Secure coding from first principles," 8 4 2008. [Online].
22. S. Stirling and J. Rainsberger, JUnit Recipes: Practical Methods for Programmer Testing, Greenwich, CT: Manning, 2005.
23. M. U. A. Khan and M. Zulkernine, "On Selecting Appropriate Development Processes and Requirements Engineering Methods for Secure Software," 33rd Annual IEEE International Computer Software and Applications Conference, pp. 353-358, 2009.
24. M. Dawson, D. N. Burrell, E. Rahim and S. Brewster, "Integrating Software Assurance into the Software Development Life Cylce (SDLC)," Journal of Information Systems Technology & Planning, vol. 3, no. 6, pp. 49-53, 12 2010.