SOAR

State-of-the-Art Report
(SOAR) July 31, 2007

Information Assurance
Technology Analysis Center (IATAC)
Data and Analysis Center for Software (DACS)
Joint endeavor by IATAC with DACS

Software Security Assurance

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
July 31, 2007	State-of-the-Art Report	July 31, 2007

4. TITLE AND SUBTITLE
Software Security Assurance: A State-of-the-Art Report (SOAR)

5a. CONTRACT NUMBER
SPO700-98-D-4002

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Goertzel, Karen Mercedes; Winograd, Theodore; McKinley, Holly Lynne; Oh, Lyndon; Colon, Michael; McGibbon, Thomas; Fedchak, Elaine; Vienneau, Robert

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

IATAC
13200 Woodland Park Road
Herndon, VA 20171

8. PERFORMING ORGANIZATION REPORT

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Technical Information Center
DTIC-I
8725 John J. Kingman Road, Suite 944
Fort Belvior, VA 22060

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT
Distribution Statement A. Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
IATAC is operated by Booz Allen Hamilton, 8283 Greensboro Drive, McLean, VA 22102.

14. ABSTRACT
This Information Assurance Technology Analysis Center (IATAC) State-of-the-Art Report (SOAR) describes the current "state-of-the-art" in software security assurance. It provides an overview of the current state of the environment in which defense and national security software must operate then surveys current and emerging activities and organizations involved in promoting various aspects of software security assurance. The SOAR also describes the variety of techniques and technologies in use in government, industry, and academia for specifying, acquiring, producing, assessing, and deploying software that can, with a justifiable degree of confidence, be said to be secure. Finally, the SOAR presents observations about noteworthy trends in software security assurance as a discipline.

15. SUBJECT TERMS
IATAC Collection, SOAR, software assurance, software security, application security

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Tyler, Gene
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	None	392	19b. TELEPHONE NUMBER *(include area code)* 703-984-0775

Information Assurance Technology Analysis Center (IATAC)
Data and Analysis Center for Software (DACS)

Joint endeavor by IATAC with DACS

Software Security Assurance

State-of-the-Art Report (SOAR)
July 31, 2007

IATAC Authors:
Karen Mercedes Goertzel
Theodore Winograd
Holly Lynne McKinley
Lyndon Oh
Michael Colon

DACS Authors:
Thomas McGibbon
Elaine Fedchak
Robert Vienneau

Coordinating Editor:
Karen Mercedes Goertzel

Copy Editors:
Margo Goldman
Linda Billard
Carolyn Quinn

Creative Directors:
Christina P. McNemar
K. Ahnie Jenkins

Art Director, Cover, and Book Design:
Don Rowe

Production:
Brad Whitford

Illustrations:
Dustin Hurt
Brad Whitford

About the Authors

Karen Mercedes Goertzel
Information Assurance Technology Analysis Center (IATAC)

Karen Mercedes Goertzel is a subject matter expert in software security assurance and information assurance, particularly multilevel secure systems and cross-domain information sharing. She supports the Department of Homeland Security Software Assurance Program and the National Security Agency's Center for Assured Software, and was lead technologist for 3 years on the Defense Information Systems Agency (DISA) Application Security Program. Ms. Goertzel is currently lead author of a report on the state-of-the-art in software security assurance, and has also led in the creation of state-of-the-art reports for the Department of Defense (DoD) on information assurance and computer network defense technologies and research. She was also involved in requirements elicitation and architectural design of several high-assurance trusted guard and trusted server applications for the defense departments of the United States, Canada, and Australia; for the North Atlantic Treaty Organization; and for the US Departments of State and Energy; the Internal Revenue Service; and the Federal Bureau of Investigation. Ms. Goertzel holds a Certified Information Systems Security Professional (CISSP) and a BA from Duke University.

Theodore Winograd (IATAC)

Theodore Winograd supports the Department of Homeland Security's Software Assurance Program, serving as a contributing author to *Security in the Software Lifecycle*. He also supports the National Institute of Standards and Technology's Computer Security Resource Center, for which he has been the lead author of several Special Publications (SP), including SP 800-95, the *Guide to Secure Web Services*. He has provided support to the DISA Application Security Program, where he was a primary author of the *Java 2 Enterprise Edition Security Checklist* and the *Developer's Guide to Secure Web Services*. Mr. Winograd holds a Global Information Assurance Certification (GIAC) Security Essentials Certification (GSEC). He holds a BS in Computer Engineering from the Virginia Polytechnic Institute and State University (Virginia Tech).

Holly Lynne McKinley (IATAC)

Holly Lynne McKinley is certified as a Systems Security Certified Practitioner (SSCP) and has earned the GSEC. Concentrating on software assurance, her client projects have ranged from technical research and guidance development to web and database programming. Ms. McKinley is a member of the Computer Security Institute and has attended conferences and presented seminars on software security, including the Institute for Applied Network Security Mid-Atlantic Forum and the Annual Computer Security Applications Conference. She holds an MS in Information Technology Systems with an Information Security concentration from Johns Hopkins University and a BS in Business Information Technology with an E-Business concentration from Virginia Tech.

Lyndon J. Oh (IATAC)

Lyndon J. Oh covers a broad range of intelligence analysis issues, including counterterrorism. He has previously worked on site in the information operations center of a US Government intelligence agency, where he focused on vulnerabilities in global telecommunications networks. He has previously served as an intern with the US Department of State's Office of Burma, Cambodia, Laos, Thailand, and Vietnam Affairs. He holds an MSc (with merit) in International Relations from the London School of Economics and a BA from Wesleyan University in Middletown, CT.

Michael Colon (IATAC)

Michael Colon has over 16 years of experience providing software engineering solutions to the commercial, aerospace, and defense industries. Mr. Colon specializes in secure systems engineering and has broad knowledge of and experience in object-oriented development, service-oriented architecture, and intelligent agent-based computing. He managed software engineering projects for the Assistant Secretary of Defense for Networks and Information

Integration (ASD NII), US Army Information and Intelligence Warfare Directorate (I2WD), US Army Communication-Electronics Command (CECOM), DISA, National Security Agency (NSA), Farberware, and Hoffritz. Mr. Colon holds a BS in Computer Science and is a member of the Upsilon Phi Epsilon Computer Science Honor Society.

Thomas McGibbon
Data and Analysis Center for Software (DACS)

Thomas McGibbon has served as the Director of DACS since 1994. He has over 30 years of experience in software development, systems development, and software project management. He is author of several DACS state-of-the-art reports on software engineering topics. He holds an MS in Software Engineering from Southern Methodist University and a BS in Mathematics from Clarkson University. He is a Certified Software Development Professional (CSDP).

Elaine Fedchak (DACS)

Elaine Fedchak has most recently worked with the Cyber Operations Group, supporting research in developing tools and systems for network and enterprise defense. Her association with DACS began in 1983. She has been involved in numerous aspects of software engineering evolution and research, from DoD-STD-2167 and the first concepts of software engineering environments, to Ada, to leveraging web technologies in support of DACS functions. She has written several state-of-the-art reports and other technical papers. Ms. Fedchak currently holds a CISSP.

Robert Vienneau (DACS)

Robert Vienneau has assisted in developing a system for measuring the performance of Net-Centric Enterprise Services (NCES) in a tactical environment; an identification and authentication architecture for the Information Management Core Services (IMCS) project; a processor for space-based radar; a real-time, high-resolution synthetic aperture radar image formation system; an automated intrusion detection environment; and systems supporting research in multichannel signal processing simulation and detection. Mr. Vienneau has supported the DACS for more than two decades, including in analyses of software metrics data, in designing web page interfaces to DACS databases, in designing and conducting experiments, and in writing and editing state-of-the-art reports. He holds an MS in software development and management from the Rochester Institute of Technology and a BS in mathematics from Rensselaer Polytechnic Institute. Mr. Vienneau is a member of the Institute of Electrical and Electronic Engineers (IEEE).

About IATAC

The Information Assurance Technology Analysis Center (IATAC) provides the Department of Defense (DoD) with emerging scientific and technical information to support defensive information operations. IATAC's mission is to provide DoD with a central point of access for information on emerging technologies in information assurance (IA). Areas of study include system vulnerabilities, research and development, models, and analysis to support the effective defense against information warfare (IW) attacks. IATAC focuses on all defensive activities related to the use of information, information-based processes, and information systems (IS). One of 20 Information Analysis Centers (IACs) sponsored by DoD, IATAC is managed by the Defense Technical Information Center (DTIC).

IATAC's basic services provide the infrastructure to support defensive information operations. These services include collecting, analyzing, and disseminating IA scientific and technical information; supporting user inquiries; database operations; current awareness activities (*e.g.*, the *IAnewsletter*); and developing critical review and technology assessment and state-of-the-art reports (SOAR).

SOARs provide in-depth analyses of current technologies, evaluate and synthesize the latest information resulting from research and development activities, and provide a comprehensive assessment of IA technologies. Topic areas for SOARs are solicited from the IA community to ensure applicability to emerging warfighter needs.

Inquiries about IATAC capabilities, products, and services may be addressed to:

Gene Tyler, Director
13200 Woodland Park Road, Suite 6031
Herndon, VA 20171
Phone: 703/984-0775
Fax: 703/984-0773
Email: iatac@dtic.mil
URL: http://iac.dtic.mil/iatac
SIPRNET: https://iatac.dtic.smil.mil

Table of Contents

Section 6
Software Assurance Initiatives, Activities, and Organizations 225

Executive Summary

Secure software is software that is able to resist most attacks, tolerate the majority of attacks it cannot resist, and recover quickly with a minimum of damage from the very few attacks it cannot tolerate.

There are three main objectives of attacks on software: they either try to sabotage the software by causing it to fail or otherwise become unavailable, to subvert the software by changing how it operates (by modifying it or by executing malicious logic embedded in it), or to learn more about the software's operation and environment so that the software can be targeted more effectively.

The subversion and sabotage of software always results in the violation of the software's security, as well as some if not all of the software's other required properties. These include such properties as correctness, predictable operation, usability, interoperability, performance, dependability, and safety.

Software assurance has as its goal the ability to provide to software acquirers and users the justifiable confidence that software will consistently exhibit its required properties. Among these properties, *security* is what enables the software to exhibit those properties even when the software comes under attack.

The problem with most software today is that it contains numerous flaws and errors that are often located and exploited by attackers to compromise the software's security and other required properties. Such

exploitable flaws and errors, because they make the software vulnerable to attack, are referred to as *vulnerabilities*.

According to *The National Strategy to Secure Cyberspace*:

> *A…critical area of national exposure is the many flaws that exist in critical infrastructure due to software vulnerabilities. New vulnerabilities emerge daily as use of software reveals flaws that malicious actors can exploit. Currently, approximately 3,500 vulnerabilities are reported annually. Corrections are usually completed by the manufacturer in the form of a patch and made available for distribution to fix the flaws.*
>
> *Many known flaws, for which solutions are available, remain uncorrected for long periods of time. For example, the top ten known vulnerabilities account for the majority of reported incidents of cyber attacks. This happens for multiple reasons. Many system administrators may lack adequate training or may not have time to examine every new patch to determine whether it applies to their system. The software to be patched may affect a complex set of interconnected systems that take a long time to test before a patch can be installed with confidence. If the systems are critical, it could be difficult to shut them down to install the patch.*
>
> *Unpatched software in critical infrastructures makes those infrastructures vulnerable to penetration and exploitation. Software flaws are exploited to propagate "worms" that can result in denial of service, disruption, or other serious damage. Such flaws can be used to gain access to and control over physical infrastructure. Improving the speed, coverage, and effectiveness of remediation of these vulnerabilities is important for both the public and private sector.*

To achieve its main goal, the discipline of software assurance must provide various means by which the number and exposure of vulnerabilities in software are reduced to such a degree that justifiable confidence in the software's security and other required properties can be attained. These means range from to defining new criteria and procedures for how the software is acquired, to changing the processes, methods, and tools used to specify, build, assess, and test the software, to adding anti-attack preventive and reactive countermeasures to the environment in which software is deployed.

This state-of-the-art report (SOAR) identifies the current "state-of-the-art" in software security assurance. It provides an overview of the current state of the environment in which defense and national security software must operate; then provides a survey of current and emerging activities and organizations involved in promoting various aspects of software security assurance; and

describes the variety of techniques and technologies in use in government, industry, and academia for specifying, acquiring, producing, assessing, and deploying software that can, with a justifiable degree of confidence, be said to be secure. Finally, the SOAR presents some observations about noteworthy trends in software security assurance as a discipline.

1

Introduction

The objective of software assurance is to establish a basis for gaining justifiable confidence (trust, if you will) that software will consistently demonstrate one or more desirable properties. These include such properties as quality, reliability, correctness, dependability, usability, interoperability, safety, fault tolerance, and—of most interest for purposes of this document—security. The assurance of security as a property of software is known as *software security assurance* (or simply *software assurance*).

Ideally, secure software will not contain faults or weaknesses that can be exploited either by human attackers or by malicious code. However, all software—even secure software—relies on people, processes, and technologies, all of which can result in vulnerabilities. As a practical matter, to be considered secure, software should be able to resist most attacks and tolerate the majority of those attacks it cannot resist. If neither resistance nor tolerance is possible and the software is compromised, it should be able to isolate itself from the attack source and degrade gracefully, with the damage from the attack contained and minimized to the greatest possible extent. After the compromise, the software should recover as quickly as possible to an acceptable level of operational capability.

The focus of this document is the assurance of security as a consistently demonstrated property in software. This state-of-the-art report (SOAR) describes current and emerging activities, approaches, technologies, and entities that are in some way directly contributing to the discipline of software security, *i.e.*, the art and science of producing secure software.

Secure software is software that is in and of itself robust against attack. This means that software will remain dependable even when that dependability is threatened. Secure software cannot be subverted or sabotaged. In practical terms, this software lacks faults or weaknesses that can be exploited either by human attackers or by malicious code.

Compared with other software properties, security is still not well understood: at no point in the software development life cycle (SDLC) are developers or users able to determine with 100 percent certainty that the software is secure nor, to the extent that it is considered secure, what makes it so. Software security is a dynamic property—software that is secure in a particular environment within a particular threat landscape may no longer be secure if that environment or threat landscape changes or if the software itself changes. In terms of "testability," security is also difficult to gauge. Software testers can run 10,000 hours of testing and ultimately be very confident that the software that passes those tests will operate reliably. The same cannot be said for the software's security. Security testing techniques for software are still immature and collectively represent an incomplete patchwork of coverage of all security issues that need to be tested for.

The state-of-the-art in software security assurance then is much less mature than the state-of-the-art for corollary disciplines of software quality assurance and software safety assurance. This said, the software security discipline has been evolving extremely quickly in the past 10 years—the number of initiatives, standards, resources, methodologies, tools, and techniques available to software practitioners to help them recognize, understand, and begin to mitigate the security issues in their software has increased exponentially.

This state-of-the-art report (SOAR) provides a snapshot of the current status of the software security assurance discipline, not so much to compare it against 10 years past, but to highlight what software practitioners can do to improve software security. It could be very interesting to take a similar snapshot 10 years from now, when software security assurance as a discipline reaches the level of maturity that software quality and safety assurance are at today.

1.1 Background

In 1991, the US National Academies of Science Computer Science and Telecommunications Board published its widely quoted *Computers at Risk*. [1] This book included an entire chapter that described a programming methodology for secure systems. In that chapter, the authors asked the question, "What makes secure software different?" The authors then proceeded to answer that question by offering a set of findings that mirror many of the secure software design, coding, and testing principles and practices repeated a decade or more later in the proliferation of books, articles, and courses on secure software development.

In August 1999, the US Congress General Accounting Office (GAO, now the Government Accountability Office) published a report to the Secretary of Defense entitled *DoD Information Security: Serious Weaknesses Continue to Place Defense Operations at Risk*. [2] In the area of "Application Software Development and Change Controls," GAO reported that—

> *Structured methodologies for designing, developing, and maintaining applications were inadequate or nonexistent. There was no requirement for users to document the planning and review of application changes and to test them to ensure that the system functioned as intended. Also, application programs were not adequately documented with a full description of the purpose and function of each module, which increases the risk that a developer making program changes will unknowingly subvert new or existing application controls…. We found that application programmers, users, and computer operators had direct access to production resources, increasing the risk that unauthorized changes to production programs and data could be made and not detected.*

Additional report findings included—

- Segregation of duties was not enforced—the same individuals were functioning both as programmers and security administrators.
- Security exposures were created by inadequate system software maintenance procedures, such as allowing uncontrolled insertions of code (including malicious code) into privileged system libraries, and not applying security updates and patches.

The report recommended that the US Department of Defense (DoD) accelerate the implementation of its Department-wide information security program. Specific recommendations to mitigate the above problems included—

- Require sensitive data files and critical production programs to be identified and successful and unsuccessful access to them to be monitored.
- Strengthen security software standards in critical areas, such as by preventing the reuse of passwords and ensuring that security software is implemented and maintained in accordance with the standards.
- Determine who is given access to computer systems applications.
- Ensure that locally designed software application program changes are in accordance with prescribed policies and procedures.

Consistent with the nature of these findings, DoD's interest in software security assurance can be said to have emerged from two related but distinct concerns—

- The DoD information assurance (IA) community's concerns about the integrity and availability of the software used in its mission-critical, high assurance, and trusted computing systems
- The DoD application development community's increasing concern about the dependability of DoD applications that, for the first time in DoD history, were being exposed to the constantly proliferating threats present on the Internet [either directly or indirectly *via* "backend" connections between the Internet and the Non-Sensitive Internet Protocol Routed Network (NIPRNet)].

In September 1999, the INFOSEC (information security) Research Council (IRC) [3] published its first *INFOSEC Research Hard Problems List*, [4] which included four hard problems related to software and application security—

- **Security of Foreign and Mobile Code:** Provide users of IT systems with the ability to execute software of unknown or hostile origin without putting sensitive information and resources at risk of disclosure, modification, or destruction.
- **Application Security:** Provide tools and techniques that will support the economical development of IT applications that enforce their own security policies with high assurance.
- **Secure System Composition:** Develop techniques for building highly secure systems in the case where few components or no components at all are designed to achieve a high level of security.
- **High Assurance Development:** Develop and apply techniques for building IT components whose security properties are known with high confidence.

In December of that year, the Defense Science Board (DSB) Task Force on Globalization and Security released its Final Report, [5] which included among its findings—

- Software is the commercial sector upon which DoD is currently most dependent. Commercial software is pervasive, whether embedded within integrated weapons systems, as components or subsystems, or purchased directly by the Department as full-up information systems…. Many of DoD's most critical future systems are based at least partly on commercial software.
- The report described in detail the security risks posed by this reliance on commercial software, and provided recommendations for mitigating that risk. Among its six key recommendations, the report stated—

The Department must act aggressively to ensure the integrity of critical software-intensive systems.

To this end, the report made several recommendations, including—

- The Secretary of Defense should affirm the Assistant Secretary of Defense (ASD) [Command, Control, Communications and Intelligence (C3I)] as responsible for ensuring the pre-operational integrity of essential software systems. In turn, the ASD(C3I) should develop and promulgate an Essential System Software Assurance Program (which the report goes on to describe in detail).
- DoD should enhance security and counter-intelligence programs to deal with the new challenges presented by relying on commercially purchased systems and subsystems of foreign manufacture.

DoD's Software Assurance Initiative (see Section 6.1.1) was formed, at least in part, in response to these recommendations. Prior to that, in December 2001, DoD established the Software Protection Initiative (SPI) to prevent reconnaissance, misuse, and abuse of deployed national security application software by America's adversaries.

The Open Web Application Security Project (OWASP) published its Top Ten Most Critical Web Application Security Vulnerabilities (see Section 3.1.3) at about the same time that the Defense Information Systems Agency (DISA) established the Application Security Project within the Applications Division of its Center for Information Assurance Engineering (CIAE) (see Section 6.1.7). The Application Security Project used the OWASP Top Ten as a baseline for defining the set of application security issues the project needed to address.

In February 2003, the White House published *The National Strategy to Secure Cyberspace*. The second of its five priorities called for establishing a national-scale threat and vulnerability reduction program that would attempt, in part, to "reduce and remediate software vulnerabilities." The *Strategy* specified among its list of actions and recommendations (A/R)—

> *DHS (the Department of Homeland Security) will facilitate a national public-private effort to promulgate best practices and methodologies that promote integrity, security, and reliability in software code development, including processes and procedures that diminish the possibilities of erroneous code, malicious code, or trap doors that could be introduced during development.*

In response to this directive, the DHS Software Assurance Program (see Section 6.1.9) was established. From its inception, the DHS Program has closely coordinated its efforts with the DoD Software Assurance Initiative, with DHS taking the broad view prescribed by the *National Strategy*. DoD focused on those aspects of the software security assurance problem that SPI personnel did not feel were being adequately addressed by the DHS efforts, and also refined the products of DHS-sponsored activities so that they would directly address DoD-specific issues.

In 2005, the President's Information Technology Advisory Committee (PITAC) published its report *Cyber Security: A Crisis of Prioritization*. In this report, PITAC observed that software is a major source of vulnerabilities in US networks and computing systems—

> *Network connectivity provides "door-to-door" transportation for attackers, but vulnerabilities in the software residing in computers substantially compound the cyber security problem…. Today, as with cancer, vulnerable software can be invaded and modified to cause damage to previously healthy software, and infected software can replicate itself and be carried across networks to cause damage in other systems. Like cancer, these damaging processes may be invisible to the lay person even though experts recognize that their threat is growing. And as in cancer, both preventive actions and research are critical, the former to minimize damage today and the latter to establish a foundation of knowledge and capabilities that will assist the cyber security professionals of tomorrow reduce risk and minimize damage for the long term. Vulnerabilities in software that are introduced by mistake or poor practices are a serious problem today. In the future, the Nation may face an even more challenging problem as adversaries—both foreign and domestic—become increasingly sophisticated in their ability to insert malicious code into critical software.*

This statement by the PITAC was adopted as a driving business case by both DoD's and DHS' Software Assurance initiatives to justify expansion of those efforts.

In November 2005, the INFOSEC Research Council (IRC) published its second *Hard Problems List*. [6] In Appendix C of this list, the IRC reported its reflections on its *Hard Problems List* of 1999. With regard to each of the four "hard problems" it had identified in 1999, the 2005 report made observations about the progress of research since 1999 as well as noted key areas in which further research was needed—

1. **Security of Foreign and Mobile Code:**
   - The need for difficult research remains.
   - Proof-carrying code and sandboxing are two important advances.
   - Significant progress has been made in the use of formal methods, which deserve further emphasis.
   - The most fundamental problem is lack of operating systems that can protect themselves against untrustworthy applications.
2. **Application Security**
   - Important progress has been made toward creating intrusion tolerant applications that can function in the presence of flawed components, and which rely less than traditional applications do on the protection of an underlying Trusted Computing Base (TCB).

Research is still needed to make these techniques work in distributed, asynchronous, time-critical environments.
    - There will always be situations in which a TCB is needed. This need compels emphasis on achieving a truly trustworthy TCB that can be used in the creation of scalable secure systems.
3. **Secure System Composition**
    - This remains a crucial hard problem with many difficulties.
    - The inadequacy of simplistic approaches (*e.g.,* just adding firewalls and intrusion detection systems; increasing the length of static passwords) has finally been acknowledged.
    - New Hard Problem Number (No.) 4, "Building Scalable Secure Systems" focuses on finding new approaches to building predictably secure systems. These approaches include use of formal methods and other promising techniques for composing secure systems from trustworthy components.
4. **High-Assurance Development**
    - Some high-assurance development tools now exist, but do not scale.
    - Hard Problem No. 4 includes a proposal for building a Computer Automated Secure Software Engineering Environment (CASSEE) to address the need for scalable tools for high-assurance systems development.

The IRC's observations are being proven accurate in some cases and inaccurate in others through the work of numerous software and security experts involved with the software security assurance initiatives described in Section 6.1, and by the extensive research, in academia, industry, and government, in the United States and abroad (see Sections 6.2 and 6.3).

Section 6.1 expands on the above description of DoD and civilian government software assurance activities.

1.2 Purpose

In the 6 years since the 1999 GAO report, the field of software security assurance (and its associated disciplines) has changed radically. Once a niche specialty of software reliability and information assurance practitioners, software security is now one of the most widely recognized, actively pursued challenges in both the software engineering and information assurance communities.

This SOAR identifies and describes the current state-of-the-art in software security assurance. This SOAR is not intended to be prescriptive; instead it provides a discussion of software security assurance trends in the following areas—

- Techniques that are now being used or are being published (*e.g.*, as standards) to produce—or increase the likelihood of producing—secure software. Examples: process models, life cycle models, methodologies, best practices.

- Technologies that exist or are emerging to address some part of the software security challenge, such as virtualized execution environments, "safe" and secure versions of programming languages and libraries, and tools for assessment and testing of software's security.
- Current activities and organizations in government, industry, and academia, in the United States and abroad, that are devoted to systematic improvement of the security of software.
- Research sector trends—both academic and non-academic, US and non-US—that are intended to further the current activities and state-of-the-art for software security.

Readers of this SOAR should gain the following—

- A better understanding of the issues involved in software security and security-focused software assurance to the extent that they can start to evaluate their own software, processes, tools, *etc.,* with regard to how secure, security enhancing, and security assuring they may or may not be.
- Enough information on security-enhancing software development techniques, tools, and resources to enable them to start recognizing gaps in their own knowledge, processes and practices, and tools. Also, the reader should be enabled to determine which of the existing or emerging security-enhancing techniques, tools, and resources might assist them in making needed improvements to their own software knowledge, processes/practices, and tools.
- Enough information to form the basis for developing criteria for determining whether a given technique, tool, *etc.,* is consistent with the their organization's current knowledge level, processes, and practices, *i.e.*, to determine which techniques, tools, *etc.*, are worth further investigation.
- Information on how to participate in existing software security activities or to establish new ones.
- The basis for developing a roadmap of incremental improvements to the processes, tools, and philosophy by which her organization develops software. This includes a basis for planning a training/education strategy for the organization's software and security practitioners.

1.3 Intended Audience

The intended primary and secondary audiences for this document can be best described in terms of the readers' professional roles.

1.3.1 Primary Audience

The primary audience for this document includes—

- **Software practitioners** involved in the conception, implementation, and assessment of software, especially software used in DoD and other US

Federal Government agencies, or in the improvement of processes by which such software is conceived, implemented, and assessed.

- **Researchers** in academia, industry, and government who are investigating methods, processes, techniques, or technologies for producing software that is secure, or for assuring the security of software during and/or after its creation.

1.3.2 Secondary Audiences

Readers in the following roles are the intended secondary audiences for this document—

- **Systems Engineers and Integrators:** This document should expand the knowledge of these readers, enabling them to broaden and deepen their systems- or architectural-level view of software-intensive systems to include both the recognition and understanding of the security properties, threats, and vulnerabilities to which the individual software components that compose their systems are subject, as well as the impact of those properties, threats, and vulnerabilities on the security of the system as a whole.
- **Information Assurance Practitioners:** These include developers of policy and guidance, risk managers, certifiers and accreditors, auditors, and evaluators. The main objective for such readers is to expand their understanding of information security risks to include a recognition and understanding of the threats and vulnerabilities that are unique to the software components of an information system. Specifically, this audience should be able to understand how vulnerable software can be subverted or sabotaged in order to compromise the confidentiality, integrity, or availability of the information processed by the system.
- **Cyber Security and Network Security Practitioners:** The objective of these readers is to recognize and understand how network operations can be compromised by threats at the application layer—threats not addressed by countermeasures at the network and transport layers. Of particular interest to such readers will be an understanding of application security, a discipline within software security assurance, and also the benefit that software assurance activities, techniques, and tools offer in terms of mitigating the malicious code risk.
- **Acquisition Personnel:** The objectives for readers in the system and software acquisition community are trifold: to obtain a basis for defining security evaluation criteria in solicitations for commercial software applications, components, and systems, and contracted software development services; to identify security evaluation techniques that should be applied to candidate software products and services before acquiring them; to understand the specific security concerns associated

with offshore development of commercial and open source software, and with outsourcing of development services to non-US firms.

- **Managers and Executives in Software Development Organizations and Software User Organizations:** This document should help them recognize and understand the software security issues that they will need to address, and subsequently develop and implement effective plans and allocate adequate resources for dealing with those issues.

1.4 Scope

This SOAR focuses on the numerous techniques, tools, programs, initiatives, *etc.,* that have been demonstrated to successfully—

- Produce secure software, *or*
- Assure that secure software has been produced (whether by a commercial or open source supplier or a custom-developer).

Also covered are techniques, tools, *etc.,* that have been proposed by a respected individual or organization (*e.g.,* a standards body) as being likely to be successful, if adopted, in achieving either of the two objectives above.

Techniques and tools for implementing information security functions (such as authentication, authorization, access control, encryption/decryption, *etc.*) in software-intensive systems will not be discussed, except to the extent that such techniques and tools can be applied, either "as is" or with adaptations or extensions, to secure the software itself rather than the information it process. For example, a tool for digitally signing electronic documents would be considered out of scope unless that tool could also be used for code signing of binary executables before their distribution. Out of scope entirely is how to assure information security functions in software-based systems at certain Common Criteria (CC) Evaluation Assurance Levels (EAL). Techniques, tools, *etc.*, that focus on improving software quality, reliability, or safety are considered in scope only when they are used with the express purpose of improving software security.

To keep this document focused and as concise as possible, we have excluded discussions of techniques that are expressly intended and only used to achieve or assure another property in software (*e.g.,* quality, safety) regardless of the fact that sometimes, as a purely coincidental result, use of such techniques also benefits the software's security.

In short, this document discusses only those methodologies, techniques, and tools that have been conceived for or adapted/reapplied for improving software security. Each SOAR discussion of an adapted/reapplied method, technique, or tool will include—

- A brief overview of the tool/technique as originally conceived, in terms of its nature and original purpose. This overview is intended to provide context so that the reader has a basis for understanding the difference

between the technique/tool as originally conceived and its software security-oriented adaptation.
- A longer description of how the tool/technique has been adapted and can now help the developer achieve one or more software security objectives. This description represents the main focus and content of the discussion.

This SOAR also reports on numerous initiatives, activities, and projects in the public and private sectors who focus on some aspects of software security assurance.

The software addressed in this SOAR is of all types, system-level and application-level, information system and noninformation system, individual components and whole software-intensive systems, embedded and nonembedded.

1.5 Assumptions and Constraints

The state-of-the-art reported in this SOAR reflects the timeframe in which the source information was collected: 2002–2007. The SOAR specifically reports on activities, practices, technologies, tools, and initiatives whose intended benefactors are the developers of software (including requirements analysts, architects, designers, programmers, and testers). The SOAR does not address acquisition issues except to the extent that developers are involved in guiding acquisition decisions. Also excluded is discussion of the physical environment in which software is created or operated. The SOAR also does not address operational and personnel security considerations and nontechnical risk management considerations except to the extent that the developer is involved in software maintenance and patch generation. The impact of budget and balancing priorities for software on its ability to achieve adequate assurance is mentioned but not considered in depth.

Finally, the SOAR does not address how the business purpose or mission for which a software-intensive system has been developed may influence the nature of its threats or its likelihood to be targeted. Regardless of whether the mission is of high consequence—and, thus, high confidence or high assurance—or more routine in nature, the practices, tools, and knowledge required of its developers are presumed to be essentially the same. To the extent that these factors necessarily differ, the SOAR may acknowledge that difference but does not discuss the unique considerations of any particular type of software application or system in any depth.

1.6 Context

This SOAR is the first known effort to provide a truly comprehensive snapshot of the activities of the software security assurance community, the security-enhanced software life cycle processes and methodologies they have described, the secure development practices they espouse, the standards they are striving to define and adopt, the technologies and tools that have emerged to support developers in the

production of secure software, and the research activities underway to continue improving the state-of-the-art for software security and its assurance.

Earlier efforts to capture and describe the state-of-the-art in software security assurance have been much more limited in scope than this SOAR, focusing on only a small subset of available methods, practices, or tools. These efforts have resulted in the following presentations and papers—

- Robert A. Martin, MITRE Corporation, *Software Assurance Programs Overview* (presentation to the Software Assurance Information Session of the Object Management Group's [OMG's] Technical Meeting, Washington, DC, December 7, 2006).
- Mohammad Zulkernine (Queen's University) and Sheikh Iqbal Ahamed, (Marquette University), "Software Security Engineering: Toward Unifying Software Engineering and Security Engineering," Chap. XIV of *Enterprise Information Systems Assurance and System Security: Managerial and Technical Issues,* Merrill Warkentin and Rayford B. Vaughn, eds., Idea Group Publishing (March 2006).
- Noopur Davis, Carnegie Mellon University (CMU) Software Engineering Institute (SEI), *Secure Software Development Life Cycle Processes: A Technology Scouting Report,* technical note CMU/SEI-2005-TN-024 (December 2005).
- K.R. Jayaram and Aditya P. Mathur (Purdue University Center for Education and Research in Information Assurance and Security [CERIAS] and Software Engineering Research Center [SERC]), *Software Engineering for Secure Software—State of the Art: A Survey* CERIAS, tech report 2005-67 (September 19, 2005).
- Software Assurance Initiative, *Software Assurance: Mitigating Software Risks in the Department of Defense (DoD) Information Technology (IT) and National Security Systems (NSS)* (October 6, 2004).
- Samuel T. Redwine, Jr. and Noopur Davis, eds., Vol I of *Processes to Produce Secure Software—Towards More Secure Software, a Report of the National Cyber Security Summit Software Process Subgroup of the Task Force on Security Across the Software Development Lifecycle* (March 2004).

On October 14, 2004, the OASD/NII forwarded to the Committee on National Security Systems (CNSS) its report entitled *Software Assurance: Mitigating Software Risks in Department of Defense (DoD) Information Technology (IT) and National Security Systems (NSS)* specifying reasons why DoD should acknowledge rather than continue to ignore the need for software assurance. This report addressed topics covered in greater depth in this SOAR, including risk management for software-intensive systems, acquisition of software, software life cycle processes, software vulnerabilities, security assessment of software, training of software practitioners, and software assurance research and development.

Draft Version 1.2 of DHS' *Security in the Software Life Cycle* [7] includes descriptions of a number of secure software process models, software methodologies (security-enhanced and not), threat modeling/risk assessment techniques, and security testing techniques, with information current as of fall 2006. However, future versions of the DHS document will omit these descriptions to avoid redundancy with DHS' *Software Assurance Landscape*, report described below, and with this SOAR.

The DHS' *Software Assurance Landscape* and a parallel *Landscape* from the National Security Agency's (NSA) Center for Assured Software (CAS)—are currently underway to address an even broader scope of activities, practices, and technologies associated with software security assurance than those covered in this SOAR. Both Landscapes will differ from this SOAR in that their objective will be to discuss software security assurance within several larger contexts, including information assurance and general software assurance. The Landscapes have also been conceived as "living," and are thus likely to be produced as online knowledge bases rather than "fixed point in time" documents. The SOAR also differs from these Landscapes in its higher level of analysis and commentary on the various activities, practices, and tools.

The managers of both the DHS and NSA CAS Landscape efforts are coordinating their efforts with the authors of this SOAR, with the intent that the SOAR will provide a good basis of information to be included in the Landscape knowledge bases.

1.7 Document Structure

This SOAR comprises nine sections and six appendices, which are described below.

Section 1 Introduction

The Introduction provides the rationale for publishing this SOAR and describes its intended audience and content.

Section 2 Definitions of Software Assurance and Secure Software

Several definitions of "software assurance" are in wide circulation. Section 2 comments on and compares and contrasts those definitions in the context of the definition used in this SOAR. Section 2 also defines "secure software" as referred to in this SOAR.

Section 3 Why Is Software at Risk?

Attacks targeting software have become extremely sophisticated, exploiting unforeseen sequences of multiple, often non-contiguous faults throughout the software. As more exploitable faults, vulnerabilities and weaknesses are discovered in the targeted software, more effective attacks can be crafted. Many resources consulted in developing this SOAR—notably DHS' Security in the Software Life Cycle, agree that threats can manifest themselves at any point in the software life cycle, including development, distribution, deployment, or operation.

Section 4 Secure Systems Engineering

Software is increasingly becoming part of a larger system, requiring system engineers to understand the security issues associated with the software components of a larger secure system. Secure systems engineering is a well-studied field, with many process models available to the systems engineer for developing the system.

Section 5 SDLC Processes and Methods and the Security of Software

Most software assurance research has been geared toward developing software from scratch, as project control can be asserted when the entire process is available. That said, organizations increasingly use turnkey solutions instead of custom-developing software to satisfy a particular organizational need. Purchasing turnkey software is often cheaper and involves less business risk than developing software from scratch. Through regulations and awareness activities, organizations are becoming more aware of the additional costs associated with insecure software. In spite of being a very active research field, no software engineering methodology exists to ensure that security exists in the development of large scale software systems. In addition to the security-enhanced lifecycle processes discussed later in this summary, efforts have focused on developing software-specific risk management methodologies and tools. Most existing risk management techniques are not easily adaptable to software security, and system security risk management is limited by focusing on operational risks.

Section 6 Software Assurance Initiatives, Activities, and Organizations

In the past five years, DoD, NSA, DHS, and the National Institute of Standards and Technology (NIST) have become increasingly active in pursuit of software security assurance and application security objectives. To this end, these agencies have established a number of programs to produce guidance and tools, perform security assessments, and provide other forms of support to software and security practitioners. These organizations also regularly participate in national and international standards activities that address various aspects of software security assurance. The private sector has also become very active, not just in terms of commercial offerings of tools and services, but in establishing consortia to collectively address different software security and application security challenges.

Section 7 Resources

The surge of interest and activity in software security and application security has brought with it a surge of online and print information about these topics, leading to Web sites and journals associated with software security assurance. Universities, domestically and internationally, are offering courses and performing research in

secure software development—along with researching effective ways to deliver knowledge of software security to students. There are also professional training courses and certifications available for those already in the workforce.

Section 8 Observations

Observations made as a result of analysis of the data gathered for this report show that the software security assurance field is still being explored, and has exposed some of the "hard problems" of software security.

Appendix A Abbreviations and Acronyms

Section 9 lists and amplifies all abbreviations and acronyms used in this SOAR.

Appendix B Definitions

Defines key terms used in this SOAR.

Appendix C Types of Software under Threat

This appendix identifies and describes the main types of critical (or high-consequence) software systems the security of which is likely to be intentionally threatened.

Appendix D DoD/FAA Proposed Safety and Security Extensions to iCMM and CMMI

This appendix provides information on the safety and security extensions proposed by the joint Federal Aviation Administration (FAA) and DoD Safety and Security Extension Project Team to add security activities to two integrated capability maturity models (CMM), the FAA's integrated CMM (iCMM) and the Carnegie Mellon University Software Engineering Institute CMM-Integration (CMMI).

Appendix E Security Functionality

This appendix describes some key security functions often implemented in software-intensive information systems.

Appendix F Agile Methods: Issues for Secure Software Development

This appendix augments the brief discussion in Section 5.1.8.1 of agile methods and secure software development with a longer, more detailed discussion of the security issues associated with agile development, as well as some of the security benefits that might accrue from use of agile methods.

Appendix G Comparison of Security-Enhanced SDLC Methodologies

This appendix provides a side-by-side comparison of the activities comprising the different security-enhanced software development life cycle methodologies discussed in Section 5.

Appendix H Software Security Research in Academia

This appendix provides an extensive listing of academic research projects in software security and assurance topics.

1.8 Acknowledgements

This SOAR was planned and executed under the guidance of—

- Dr. Steven King, Associated Director for Information Assurance, Office of the Deputy Under Secretary of Defense (Science and Technology) [DUSD(S&T)], representing the Steering Committee of the Information Assurance Technology Analysis Center (IATAC);
- Mr. Robert Gold, Associated Director for Software and Embedded Systems, Office of the Deputy Under Secretary of Defense (Science & Technology) [DUSD(S&T)], representing the Steering Committee of the Data and Analysis Center for Software (DACS);
- Ms. Nancy Pfeil, Deputy Program Manager for the Information Analysis Center Program Management Office (IAC PMO), and the IATAC Contracting Officers Representative (COR).

This SOAR was provided for review to a number of organizations across DoD and the civil agencies, industry, and academia. We would also like to thank the following individuals for their incisive and extremely helpful comments on the drafts of this document: Mr. Mitchell Komaroff of the Office of the Assistant Secretary of Defense (Network and Information Integration) (ASD/NII); Mr. Joe Jarzombek of the Department of Homeland Security Cyber Security and Communications National Cyber Security Division (DHS CS&C NCSD); Dr. Larry Wagoner of the NSA CAS; Dr. Paul Black of the NIST; Mr. David A. Wheeler of the Institute for Defense Analyses (IDA); Dr. Matt Bishop, University of California at Davis; Dr. Carol Wood (and Dr. Robert Seacord of the Carnegie Mellon University Software Engineering Institute (CMU SEI); Mr. Robert Martin of The MITRE Corporation and software assurance subject-matter experts from Booz Allen Hamilton.

The research methodology and content of this SOAR were presented and discussed at the following conferences and workshops—

- DHS Software Assurance Working Group meetings, Arlington, Virginia (May 15-17, 2007)
- Object Management Group (OMG) Software Assurance Workshop, Fairfax, Virginia (March 5-7, 2007)
- DoD/DHS Software Assurance Forum, Fairfax, Virginia (March 8-9, 2007)
- Institute for Applied Network Security Mid-Atlantic Information Security Forum, Tysons Corner, Virginia (March 5-6, 2007)

- Defense Intelligence Agency Joint Information Operations Technical Working Group Cyber Security Conference, Crystal City, Virginia (February 5-6, 2007)
- Meeting of the Society for Software Quality (SSQ) Washington D.C. Chapter (April 19, 2006).

In addition, the content of this SOAR was described in the following article—

- Goertzel, Karen Mercedes. An IATAC/DACS State-of-the-Art Report on Software Security Assurance. *IAnewsletter*. Spring 2007; 10(1):24-25.

References

1 National Research Council (NRC), *Computers at Risk: Safe Computing In the Information Age* (Washington, DC: National Academy Press, 1991).
Available from: http://books.nap.edu/books/0309043883/html

2 Government Accountability Office (GAO), *DoD Information Security: Serious Weaknesses Continue to Place Defense Operations at Risk*, report no. GAO/AIMD-99-107 (Washington, DC: GAO, August 1999).
Available from: http://www.gao.gov/cgi-bin/getrpt?GAO/AIMD-99-107

3 The IRC's members include several DoD organizations, including the Defense Advanced Research Projects Agency (DARPA); the National Security Agency; the Office of the Secretary of Defense (OSD); and the Departments of the Army, Navy, and Air Force; as well as civilian organizations, including the National Institute of Standards and Technology (NIST), the Department of Energy (DOE), and the Central Intelligence Agency (CIA).

4 IRC, *National Scale INFOSEC Research Hard Problems List, vers. 1.0* (September 21, 1999).
Available from: http://www.infosec-research.org/docs_public/IRC-HPL-as-released-990921.doc

5 Defense Science Board, *Final Report of the Defense Science Board Task Force on Globalization and Security*, (December 1999).
Available from: http://www.acq.osd.mil/dsb/reports/globalization.pdf

6 IRC, *National Scale INFOSEC Research Hard Problems List, vers. 2.0* (November 30, 2005).
Available from: http://www.infosec-research.org/docs_public/20051130-IRC-HPL-FINAL.pdf

7 Karen Mercedes Goertzel, *et al., Security in the Software Lifecycle: Making Software Development Processes—and Software Produced by Them—More Secure*, draft vers. 1.2 (Washington, DC: DHS CS&C NCSD, August 2006).
Available from: https://buildsecurityin.us-cert.gov/daisy/bsi/resources/dhs/87.html

2
Definitions

The readers' comprehension of two terms is key to their understanding of this document: "software assurance" (or, to be more precise, "software security assurance") and "secure software." The first term has been defined in several ways throughout the software assurance community. Section 2.1 discusses these various definitions and compares their key features. The definition of the second term, and that of its closely related construction," software security," are discussed in Section 2.2.

2.1 Definition 1: Software Assurance

Until recently, the term *software assurance* was most commonly relating two software properties: *quality* (*i.e.*, "software assurance" as the short form of "software quality assurance"), and *reliability* (along with reliability's most stringent quality—safety). Only in the past 5 years or so has the term software assurance been adopted to express the idea of the assured security of software (comparable to the assured security of information that is expressed by the term "information assurance").

The discipline of software assurance can be defined in many ways. The most common definitions complement each other but differ slightly in terms of emphasis and approach to the problem of assuring the security of software.

In all cases, all definitions of software assurance convey the thought that software assurance must provide a reasonable level of *justifiable confidence* that the software will function correctly and predictably in a manner consistent with its documented requirements. Additionally, the function of software cannot be compromised either through direct attack or through sabotage by maliciously implanted code to be considered assured. Some definitions of software assurance characterize that assurance in terms of the software's trustworthiness or "high-confidence." Several leading definitions of software assurance are discussed below.

Instead of choosing a single definition of software assurance for this report, we synthesized them into a definition that most closely reflects software security assurance as we wanted it to be understood in the context of this report—

> ***Software security assurance****: The basis for gaining justifiable confidence that software will consistently exhibit all properties required to ensure that the software, in operation, will continue to operate dependably despite the presence of sponsored (intentional) faults. In practical terms, such software must be able to resist most attacks, tolerate as many as possible of those attacks it cannot resist, and contain the damage and recover to a normal level of operation as soon as possible after any attacks it is unable to resist or tolerate.*

2.1.1 CNSS Definition

The ability to establish confidence in the *security* as well as the predictability of software is the focus of the Committee on National Security Systems (CNSS) definitions of software assurance in its National Information Assurance Glossary. [8] The glossary defines software assurance as—

> *The level of confidence that software is free from vulnerabilities, regardless of whether they are intentionally designed into the software or accidentally inserted later in its life cycle, and that the software functions in the intended manner.*

This understanding of software assurance is consistent with the use of the term in connection with information, *i.e.*, information assurance (IA). By adding the term software assurance to its IA glossary, CNSS has acknowledged that software is directly relevant to the ability to achieve information assurance.

The CNSS definition is purely descriptive: it describes what software must *be* to achieve the level of confidence at which its desired characteristics—lack of vulnerabilities and predictable execution—can be said to be assured. The definition does not attempt to prescribe the means by which that assurance can, should, or must be achieved.

2.1.2 DoD Definition

The Department of Defense's (DoD) Software Assurance Initiative's definition is identical in meaning to that of the CNSS, although more succinct—

> *The level of confidence that software functions as intended and is free of vulnerabilities, either intentionally or unintentionally designed or inserted as part of the software.* [9]

2.1.3 NASA Definition

The National Aeronautics and Space Administration (NASA) defines software assurance as—

> *The planned and systematic set of activities that ensure that software processes and products conform to requirements, standards, and procedures.*
>
> The "planned and systematic set of activities" envisioned by NASA include—
> - Requirements specification
> - Testing
> - Validation
> - Reporting.

The application of these functions "during a software development life cycle is called software assurance." [10]

The NASA software assurance definition predates the CNSS definition but similarly reflects the primary concern of its community—in this case, safety. Unlike the CNSS definition, NASA's definition is both descriptive and prescriptive in its emphasis on the importance of a "planned and systematic set of activities." Furthermore, NASA's definition states that assurance must be achieved not only for the software itself but also the processes by which it is developed, operated, and maintained. To be assured, both software *and* processes must "conform to requirements, standards, and procedures."

2.1.3 DHS Definition

Like CNSS, the Department of Homeland Security (DHS) definition of software assurance emphasizes the properties that must be present in the software for it to be considered "assured," *i.e.*—

- Trustworthiness, which DHS defines, like CNSS, in terms of the absence of exploitable vulnerabilities whether maliciously or unintentionally inserted
- Predictable execution, which "provides justifiable confidence that the software, when executed, will function as intended. [11]

Like NASA, DHS's definition explicitly states that "a planned and systematic set of multidisciplinary activities" must be applied to ensure the conformance of both software and processes to "requirements, standards, and procedures." [12]

2.1.4 NIST Definition

The National Institute of Standards and Technology (NIST) defines software assurance in the same terms as NASA, whereas the required properties to be achieved are those included in the DHS definition: trustworthiness and predictable execution. NIST essentially fuses the NASA and DHS definitions into a single definition, thereby clarifying the cause-and-effect relationship between "the planned and systematic set of activities" and the expectation that such activities will achieve software that is trustworthy and predictable in its execution. [13]

2.2 Definition 2: Secure Software

DHS's *Security in the Software Life Cycle* defines secure software in terms that have attempted to incorporate concepts from all of the software assurance definitions discussed in Section 2.1 as well as reflect both narrow-focused and holistic views of what constitutes secure software. The document attempts to provide a "consensus" definition that has, in fact, been vetted across the software security assurance community [or at least that part that participates in meetings of the DHS Software Assurance Working Groups (WG) and DoD/DHS Software Assurance Forums]. According to *Security in the Software Life Cycle*—

> *Secure software cannot be intentionally subverted or forced to fail. It is, in short, software that remains correct and predictable in spite of intentional efforts to compromise that dependability.*

Security in the Software Life Cycle elaborates on this definition—

> *Secure software is designed, implemented, configured, and supported in ways that enable it to:*
> - *Continue operating correctly in the presence of most attacks by either resisting the exploitation of faults or other weaknesses in the software by the attacker, or tolerating the errors and failures that result from such exploits*
> - *Isolate, contain, and limit the damage resulting from any failures caused by attack-triggered faults that the software was unable to resist or tolerate, and recover as quickly as possible from those failures.*

The document then enumerates the different security properties that characterize secure software and clearly associates the means by which software has been developed with its security:

> *Secure software has been developed such that—*
> - *Exploitable faults and other weaknesses are avoided by well-intentioned developers.*
> - *The likelihood is greatly reduced or eliminated that malicious developers can intentionally implant exploitable faults and weaknesses or malicious logic into the software.*
> - *The software will be attack-resistant or attack-tolerant, and attack-resilient.*
> - *The interactions among components within the software-intensive system, and between the system and external entities, do not contain exploitable weaknesses.*

Definitions of other key terms used in this SOAR are provided in Appendix B.

2.3 Software Security *vs.* Application Security

The increased targeting by attackers of vulnerable applications has led to a gradual recognition that the network- and operating system-level protections that are now commonplace for protecting Internet-accessible systems are no longer sufficient for that purpose. This recognition has given use to emerging application security measures to augment system and network security measures. Application security protections and mitigations are specified almost exclusively at the level of the system and network architecture rather than the individual application's software architecture. They are primarily implemented during the application's deployment and operation.

Application security combines system engineering techniques, such as defense-in-depth (DiD) measures (*e.g.*, application layer firewalls, eXtensible Markup Language (XML) security gateways, sandboxing, code signing) and secure configurations, with operational security practices, including patch management and vulnerability management. Application security DiD measures operate predominately by using boundary protections to recognize and block attack patterns, and using constrained execution environments to isolate vulnerable applications, thus minimizing their exposure to attackers and their interaction with more trustworthy components. Operational security measures are focused on reducing the number or exposure of vulnerabilities in the applications (*i.e.*, through patching), and by repeatedly reassessing the number and severity of residual vulnerabilities, and of the threats that may target and exploit them, so that the DiD measures can be adjusted accordingly to maintain their required level of effectiveness.

Application security falls short of providing an adequate basis for software security assurance comparable to the quality and safety assurances that can be achieved through use of established quality assurance practices and fault tolerance techniques respectively.

While application security practitioners acknowledge that the way in which the application is designed and built is significant in terms of reducing the likelihood and number of vulnerabilities the application will contain, these practitioners are not directly concerned with the application's development life cycle. By contrast, software security requires security to be seen as a critical property of the software itself—a property that is best assured if it is specified from the very beginning of the software's development process.

Software security assurance is addressed holistically and systematically, in the same way as quality and safety. In fact, security shares many of the same constituent dependability properties that must be exhibited in software for which safety and quality are to be assured. These properties include correctness, predictability, and fault tolerance (or, in the case of security, attack tolerance). The analogy between safety and security is particularly close. The main difference is that safety-relevant faults are stochastic (*i.e.*, unintentional or

accidental), whereas security-relevant faults are "sponsored," *i.e.*, intentionally created and activated through conscious and intentional human agency.

The ability of software to continue to operate dependably despite the presence of sponsored faults is what makes it secure. This ability is based largely on the software's lack of vulnerability to those sponsored faults, which may be activated by direct attacks that exploit known or suspected vulnerabilities, or by the execution of embedded malicious code. This is why most definitions of secure software emphasize the absence of malicious logic and exploitable vulnerabilities.

2.4 Software Security *vs.* Quality, Reliability, and Safety

The difference between software security and software quality, reliability, and safety is not their objectives: the objective of each is to assure that software will be dependable despite the presence of certain internal and external stimuli, influences, and circumstances. Rather the difference lies in the nature of those stimuli, influences, and circumstances.

These differences can be characterized in terms of threats to whichever property is desired. The main threat to quality is internal, *i.e.*, the presence in the software itself of flaws and defects that threaten its ability to operate correctly and predictably. Because such flaws and defects result from errors in judgment or execution by the software's developer, tester, installer, or operator, they are often termed *unintentional* threats.

The main threats to reliability are internal and external. They include the threats to quality augmented by threats from the software's execution environment when it behaves unpredictably (for whatever reason). The threats to safety are the same as those for reliability, with the distinction that the outcome of those threats, if they are realized, will be catastrophic to human beings, who may be killed, maimed, or suffer significant damage to their health or physical environment as a result.

The faults that unintentionally threaten the quality, reliability, and safety can also be intentionally exploited by human agents (attackers) or malicious software agents (malware). Security threats are differentiated from safety, reliability, and quality threats by their *intentionality*. A flaw in source code that threatens the compiled software's reliability can also make that software vulnerable to an attacker who knows how to exploit that vulnerability to compromise the dependable execution of the software.

Security threats to software are *intentional*. The presence of the vulnerabilities that enable security threats to achieve their objectives may not be intentional, but the *targeting and exploitation* of those vulnerabilities are intentional.

Security threats to software usually manifest either as direct attacks on, or execution of malicious code embedded in, operational software, or as implantations of malicious logic or intentional vulnerabilities in software under development. John McDermott of the Naval Research Laboratory's Center for High Assurance Computer Systems (NRL CHACS) [14] characterizes the

difference between threats to security on the one hand and threats to safety, reliability, quality, *etc.*, on the other. He characterizes the threats in terms of *stochastic* (unintentional) faults in the software *vs*. *sponsored* (intentional) faults. Stochastic faults may cause software to become vulnerable to attacks, but unlike sponsored faults, their existence is not intentional.

References

8 Committee on National Security Systems, *National Information Assurance (IA) Glossary*, CNSS instruction No. 4009 (revised June 2006).
Available from: http://www.cnss.gov/Assets/pdf/cnssi_4009.pdf

9 Mitchell Komaroff (ASD/NII) and Kristin Baldwin (OSD/AT&L), *DoD Software Assurance Initiative* (September 13, 2005).
Available from: https://acc.dau.mil/CommunityBrowser.aspx?id=25749

10 National Aeronautics and Space Administration (NASA), *Software Assurance Standard*, Standard No. NASA-STD-2201-93 (Washington, DC: NASA, November 10, 1992).
Available from: http://satc.gsfc.nasa.gov/assure/assurepage.html

11 National Institute of Standards and Technology, "SAMATE—Software Assurance Metrics and Tool Evaluation" [portal page] (Gaithersburg, MD: NIST).
Available from: http://samate.nist.gov

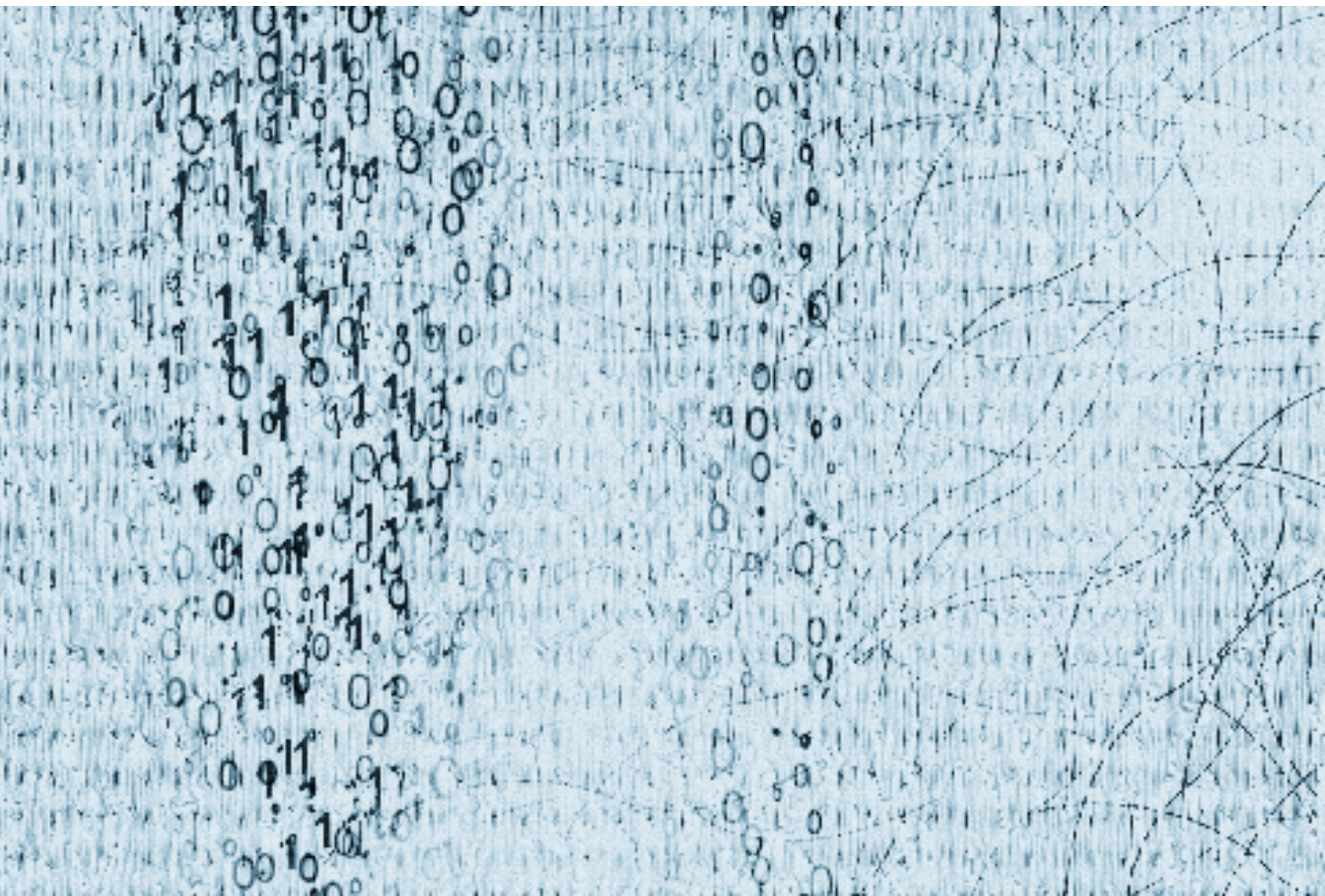
12 US Computer Emergency Response Team, "Build Security In" [portal page] (Washington, DC).
Available from: https://buildsecurityin.us-cert.gov

13 "SAMATE" [portal page] *op. cit.*

14 John McDermott (Naval Research Laboratory Center for High Assurance Computer Systems), "Attack-Potential-based Survivability Modeling for High-Consequence Systems," in *Proceedings of the Third International Information Assurance Workshop*, March 2005, 119–130.
Available from: http://chacs.nrl.navy.mil/publications/CHACS/2005/2005mcdermott-IWIA05preprint.pdf

3

Why is Software at Risk?



Security in the context of software and software-intensive systems [15] usually pertains to the software's functional ability to protect the information it handles. With software as a conduit to sensitive information, risk managers have determined that, to the extent that the software itself might be targeted to access otherwise inaccessible data, the availability and integrity of the software must be preserved and protected against compromise.

In the past, the software considered most likely to be targeted was operating system level software and software that performed critical security functions, such as authenticating users and encrypting sensitive data. Over time, however, the nature of both threats and targets changed, largely as a result of virtually universal Internet connectivity and use of web technologies.

The exponential increase in the exposure of software-intensive systems (*i.e.*, applications together with the middleware, operating systems, and hardware that compose their *platforms*, and the often sensitive data they are intended to manipulate) coincided with the increased recognition by attackers of the potential for a whole new category of attacks that exploited the multitude of bugs, errors, and faults typically present in the vast majority of software. The exploitability of these faults as vulnerabilities was not widely recognized before these software-intensive systems were exposed to the Internet. As noted in *The National Strategy to Secure Cyberspace*—

> *…the infrastructure that makes up cyberspace—software and hardware—is global in its design and development. Because of the global nature of cyberspace, the vulnerabilities that exist are open to the world and available to anyone, anywhere, with sufficient capability to exploit them.*

The *Strategy* further observes—

> *Identified computer security vulnerabilities—faults in software and hardware that could permit unauthorized network access or allow an attacker to cause network damage—increased significantly from 2000 to 2002, with the number of vulnerabilities going from 1,090 to 4,129.*

Statistics like this reflect the events that spurred major customers of commercial software firms, and most notably Microsoft, to start demanding improvements in the security of their software products. This customer demand is what drove Microsoft to undertake its Trustworthy Computing Initiative (see Section 7.2.2).

3.1 What Makes Software Vulnerable?

In his research statement for the University of Cambridge (UK), Andy Ozment, a software security researcher, suggests three primary causes of vulnerabilities and weaknesses in software [16]—

- **Lack of Developer Motivation**: Because consumers reward software vendors for being first to market and adding new features to their products, rather than for producing software that is better or more secure, [17] vendors have no financial incentive to do the latter.
- **Lack of Developer Knowledge**: Software is so complex; it exceeds the human ability to fully comprehend it, or to recognize and learn how to avoid all of its possible faults, vulnerabilities, and weaknesses. This factor, combined with lack of developer motivation, is often used as an excuse for not even teaching developers how to avoid those faults, vulnerabilities, and weaknesses that are within their ability to comprehend and avoid.
- **Lack of Technology**: Current tools are inadequate to assist the developer in producing secure software or even to reliably determine whether the software the developer has produced is secure.

Ozment goes on to state that neither lack of motivation nor lack of knowledge is defensible, and asks—

> *The problems of software insecurity, viruses, and worms are frequently in the headlines. Why does the potential damage to vendors' reputations not motivate them to invest in more secure software?*

Until recently, Ozment's question was valid—reputation was not enough to motivate software vendors because it was not enough to motivate software buyers. As with buyers of used automobiles, buyers of software had no way to ascertain that software actually was secure. They had to accept the vendors'

claims largely on faith. Given this, why should they pay a premium for vendor-claimed security when they had no way to independently evaluate it? With no customer demand, vendors saw no market for more secure software, and were not inclined to invest the additional time and resources required to produce it.

This situation changed to some extent when Microsoft launched its Trustworthy Computing Initiative and adopted its Security Development Lifecycle (SDL, see Section 5.1.3.1). Compelled to acknowledge that the multiplicity of highly publicized security exploits that targeted vulnerabilities in its products had taken its toll, Microsoft—the world's largest commercial software vendor—publicly announced that the security exploits that targeted vulnerabilities in its products, had led some of the firm's most valued customers (*e.g.,* those in the financial sector) to begin voting with their pocketbooks. Valuable customers were rejecting new versions and new products from Microsoft.

Microsoft has always known that reputation matters. As a result, they changed their development practices with the explicit objective of producing software that contained fewer exploitable vulnerabilities and weaknesses. And they widely publicized their efforts. By improving the security of its products, Microsoft began to regain the confidence of its customers. It also influenced the practices of other software vendors, even those whose products' security was not yet being questioned by their customers.

With respect to lack of developer knowledge, Ozment suggests that "most software contains security flaws that its creators were readily capable of preventing." But they lacked the motivation to do so. As the President's Information Technology Advisory Committee (PITAC) report observes—

> *The software development methods that have been the norm fail to provide the high-quality, reliable, and secure software that the IT infrastructure requires. Software development is not yet a science or a rigorous discipline, and the development process by and large is not controlled to minimize the vulnerabilities that attackers exploit.*

The methodology for building secure software is not widely taught in software engineering and computer science programs at the undergraduate or even the postgraduate level (although this is changing, as noted in Section 7.2). This means many university-educated software developers graduate without a clear understanding of the difference between software security functionality and security as a property of software. Developers are unable to recognize the security implications of their design choices and coding errors, and of their neglect of the need to remove debugging hooks and other backdoors from code before it is deployed. They do not understand how the variances in assumptions among components, functions, application programming interfaces (API), and services can result in unintentional vulnerabilities in the software that contains them. Nor do they know how to recognize malicious logic that may have been intentionally introduced into the code base by another developer.

Many software engineering students are never taught about the inherent security inadequacies of popular processing models that they will use (*e.g.,* peer-to-peer, service oriented architecture), programming languages (*e.g.,* C, C++, VisualBasic), programmatic interfaces (*e.g.,* remote procedure calls), communication protocols [*e.g.,* Simple Object Access Protocol (SOAP), Hyper Text Transfer Protocol (HTTP)], technologies (*e.g.,* ColdFusion, web browsers), and tools (*e.g.,* code generators, language libraries). Nor do they learn to recognize the threats to software, how those threats are realized (as attacks) during development or deployment, or how to leverage their understanding of threats and common exploitable weaknesses and faults (*i.e.,* vulnerabilities) when specifying requirements for their software's functionality, functional constraints, and security controls. They often take on faith that software that conforms with a standard for security functionality [such as Secure Socket Layer (SSL) or eXtensible Markup Language Digital Signature (XML Dsig)] will *be* secure in terms of robustness against threats, which is not necessarily the case. Finally, they are not taught secure design principles and implementation techniques that will produce software that can detect attack patterns and contribute to its own ability to resist, tolerate, and recover from attacks.

Finally, vulnerabilities can also originate in the incorrect configuration of the software or its execution environment. An example is the incorrect configuration of the jar files and sandboxes in the Java Virtual Machine (JVM) environment that prevents the software from constraining the execution of untrusted code (*e.g.,* mobile code) as it is intended to. A Java-based system incorrectly configured in this way would be said to have a vulnerability, not because the JVM is inherently weak, but because of the error(s) in its configuration.

3.1.1 The Relationship Between Faults and Vulnerabilities

Read any book or article on software security, and the author will assert that the majority of vulnerabilities in software originate in design defects and coding flaws that manifest, in the compiled software, as exploitable faults. Faults become exploitable, and thus represent vulnerabilities, only if they are accessible by an attacker. The more faults that are exposed, the more potential entry points into the software are available to attackers.

Attacks targeting software faults have become extremely sophisticated. They often exploit unexpected sequences of multiple, often noncontiguous faults (referred to as *byzantine* faults) throughout the software. Reconnaissance attacks provide the attacker with information on the location, nature, and relationships among vulnerabilities and weaknesses in the targeted software. This knowledge then enables the attacker to craft even more effective attacks (see Section 3.2 for more information on the threats and attacks to which software is subject).

Section 3.1.3 describes a number of efforts to systematically characterize and categorize common vulnerabilities and weaknesses in software.

3.1.2 Vulnerability Reporting

It is generally expected that a software vendor's quality assurance process will entail testing for vulnerabilities. Customers who report vulnerabilities back to vendors may be given credit, as is the case when Microsoft receives vulnerability notifications "under responsible disclosure" from its customers. There are also several organizations and initiatives for finding, tracking, and reporting vulnerabilities, both in the private sector and in Federal Government departments and agencies. Examples of the latter include DoD's Information Assurance Vulnerability Alert (IAVA) system.

In the public domain, some noteworthy vulnerability tracking/reporting systems include—

- National Institute of Standards and Technology (NIST) National Vulnerability Database (NVD)
  Available from: http://nvd.nist.gov
- US Computer Emergency Response Team (US-CERT) Vulnerability Notes Database
  Available from: http://www.kb.cert.org/vuls
- Open Source Vulnerability Database
  Available from: http://osvdb.org
- eEye Research Zero-Day Tracker
  Available from: http://research.eeye.com/html/alerts/zeroday/index.html
- MITRE Common Vulnerabilities and Exposures (CVE)
  Available from: http://cve.mitre.org (The CVE is not a vulnerability tracking/reporting system but rather a repository of information about common vulnerabilities tracked and reported by others.)

A number of commercial firms offer vulnerability alert/tracking and management services for a fee. Several of these are frequently cited in the press because of their track record of being the first to discover and report critical vulnerabilities in popular software products. Some of the most notable of such firms are Secunia, Next Generation Security Software (NGSS), Symantec, Talisker, and iDefense Labs.

In 2002, iDefense Labs undertook a new approach to vulnerability detection. The iDefense Vulnerability Contributor Program (VCP) [18] was originally envisioned by iDefense as a 3-year program whereby iDefense would pay private, often anonymous, "researchers" (read: blackhats, *i.e.,* hackers or crackers) for the exclusive rights to advance notifications about as-yet-unpublished Internet-based system vulnerabilities and exploit code. In 2005, iDefense released a set of reverse engineering and vulnerability detection tools for VCP participants to use. iDefense estimates that upwards of 80 percent of its vulnerability reports originated with information it had purchased through the VCP program.

It has been estimated by eEye Digital (iDefense's competitor and operator of the recently established zero-day vulnerability tracker) that the fee paid

by iDefense for each vulnerability report increased from $400 when the VCP started to approximately $3,000 per vulnerability by 2005. The VCP is not without its critics, including eEye Digital. They are mostly concerned that by setting a precedent of paying for vulnerability information, iDefense has increased the likelihood of bidding wars between firms that are competing to obtain such information—information that is also highly valued by cyber criminals and other threatening entities.

The problem of vulnerability disclosure is a widely considered subject in academia, with researchers investigating the ethical, legal, and economic implications. The whitepapers cited below are typical examples of these investigations. In September 2004, the Organization for Internet Safety published Version 2.0 of its *Guidelines for Security Vulnerability Reporting and Response Process*. [19] Similar guidelines for ethical vulnerability reporting have been published by NGSS and other organizations.

For Further Reading

Nizovtsev, Dmitri (Washburn University); Thursby, Marie C. (Georgia Institute of Technology) *To Disclose or Not?: an Analysis of Software User Behavior. Social Science Research Network eLibrary.* 2006 April.
Available from: http://papers.ssrn.com/sol3/papers.cfm?abstract_id=899863

Arora, Ashish; Telang, Rahul; Xu, Hao (CMU). *Optimal Policy for Software Vulnerability Disclosure.* Paper presented at: Center on Employment and Economic Growth Social Science and Technology Seminar; 2006 May 31.
Available from: http://siepr.stanford.edu/programs/SST_Seminars/disclosure05_04.pdf

Wattal, Sunil; Telang, Rahul (CMU): *Effect of Vulnerability Disclosures on Market Value of Software Vendors: an Event Study Analysis*. The 2004 Workshop on Information Systems and Economics; 2004 December 11.
Available from: http://opim.wharton.upenn.edu/wise2004/sat622.pdf

Arora, Ashish Arora (CMU). Release in Haste and Patch at Leisure: *The Economics of Software Vulnerabilities, Patches, and Disclosure. Paper presented at: Center on Employment and Economic Growth Social Science and Technology Seminar*, 2006 May 31.
Available from: http://siepr.stanford.edu/programs/SST_Seminars/MS_forthcoming.pdf

Finland: University of Oulu. *Vulnerability Disclosure Publications and Discussion Tracking.*
Available from: http://www.ee.oulu.fi/research/ouspg/sage/disclosure-tracking/index.html

Ozment, Andy [University of Cambridge (UK)]. *Bug Auctions: Vulnerability Markets Reconsidered. Third Workshop on the Economics of Information Security*, 2004 May 14.
Available from: http://www.cl.cam.ac.uk/~jo262/papers/weis04-ozment-bugauc.pdf

Ozment, Andy [University of Cambridge (UK)]: *The Likelihood of Vulnerability Rediscovery and the Social Utility of Vulnerability Hunting. Fourth Workshop on the Economics of Information Security*, 2005 June 2-3.
Available from: http://www.cl.cam.ac.uk/~jo262/papers/weis05-ozment-vulnrediscovery.pdf

3.1.3 Vulnerability Classifications and Taxonomies

The categorization and classification ("taxonomization") of security vulnerabilities and weaknesses in software and software-intensive systems has been one of the most persistently active areas of software security assurance research. Indeed, this area of research began formally as far back as the 1970s, and the number of research efforts has been significantly greater than

comparable efforts to categorize and classify threats and attack patterns. The remainder of this section reports on some of the early taxonomy efforts as well as more recent taxonomies, then describes some current efforts to define a set of standard software vulnerability definitions, categories, and classifications.

3.1.3.1 Background

Research into defining and classifying software security defects dates back to 1972, with the then National Bureau of Standards' initiation of its Research into Secure Operating Systems (RISOS) project. [20] Conducted by Lawrence Livermore National Laboratory at the University of California, the RISOS project attempted to define a taxonomy of operating system security defects that included seven major categories of flaws, five of which are still cited as software vulnerability categories (the other two were categories of information, rather than software, security vulnerabilities)—

- Inconsistent parameter validation
- Incomplete parameter validation
- Asynchronous validation or inadequate serialization (*i.e.,* race condition)
- Violable prohibition or limit (*i.e.,* failure to handle bounds conditions correctly, or incorrect pointer arithmetic, both of which can lead to buffer overflows)
- Other exploitable logic errors.

Almost two decades after RISOS, researchers of the Naval Research Laboratory's (NRLs) Center for High Assurance Computer System (CHACS) published *A Taxonomy of Computer Program Security Flaws, with Examples.* [21] This document proposed a taxonomy of security flaws that would classify flaws according to how, when, and where they were introduced into the software. This report also thoroughly documented 50 examples of security flaws so that the software development community could benefit and learn the author's research. This research served as the base for much work to come in the area of identifying and documenting software specific flaws.

In 1996, a year after publishing his own *A Taxonomy of Unix System and Network Vulnerabilities* in May 1995, [22] Matthew Bishop and his colleague D. Bailey at the University of California at Davis published *A Critical Analysis of Vulnerability Taxonomies.* [23] This document was one of the first attempts to describe and critique the major vulnerability taxonomy and classification efforts up to that point.

Throughout the 1990s, other researchers at Purdue's Computer Operations, Audit, and Security Technology (COAST) Laboratory continued to pursue the categorization of computer system and software vulnerabilities, culminating in 1998 with the presentation by Wenliang Du and Aditya Mathur of their paper, *Categorization of Software Errors that Led to Security Breaches* [24] at the National Information Systems Security Conference in Arlington, Virginia. In retrospect, the paper is interesting less for the authors' own attempt

to identify and categorize security-relevant software flaws than for its critical descriptions of several previous software vulnerability and software flaw categorization and taxonomy efforts, including Matt Bishop's.

As a natural evolution from the multiplicity of efforts from the 1970s through the 1990s to develop a definitive taxonomy of software (or software system) security vulnerabilities, the 21st century brought a new wave of collaboration initiatives among members of the software security assurance community. Through their discussions, it became apparent that there was still a need for a software vulnerability taxonomy that was not only definitive but also *standard*.

Table 3-1 lists the most noteworthy software vulnerability taxonomies published since 2001.

Table 3-1. Recent Vulnerability Classification and Taxonomy Efforts

Year	Authors	Name	Observations
2002	Open Web Applications Security Project (OWASP)	OWASP Top Ten Most Critical Web Application Security Vulnerabilities 2002 [25]	Published in response to SysAdmin, Audit, Networking, and Security (SANS) Top Ten. [26] Typical of many vulnerability lists in its confusing vulnerabilities, attacks, threats, and outcomes. Makes no distinction between software, system, or data security issues, or technical, policy, or process issues.
2002	Frank Piessens, Catholic University of Leuven (Belgium)	A Taxonomy of Software Vulnerabilities in Internet Software [27]	2-tiered hierarchy mapping to the software development lifecycle. Based on several of the same taxonomies used by MITRE in defining the CVE [28], Carl Landwehr *et al.*, John Viega, and Gary McGraw *et al.* Purpose is to educate developers of Internet-based applications.
2005	John Viega	Comprehensive Lightweight Application Security Process (CLASP) Vulnerability Root Cause Classification [29]	Augments classification axes from Landwehr *et al.* with 2 new axes, Consequence and Problem Type. Unlike Landwehr *et al.*, does not limit number of root causes (or Problem Types) to one per vulnerability.
2005	Michael Howard, David LeBlanc, John Viega	19 Deadly Sins of Software Security [30]	Lists 19 categories of "common, well-understood" coding errors that lead to 95 per cent of software vulnerabilities. Makes no attempt to differentiate between software, system, and information security issues, nor between vulnerabilities, attacks, and outcomes. Over half the "deadly sins" are system-level design issues. Cited by numerous research papers and projects, including MITRE's Common Weakness Enumeration (CWE) project (see Section 3.1.3.2).

Table 3-1. Recent Vulnerability Classification and Taxonomy Efforts - *continued*

Year	Authors	Name	Observations
2005	Katrina Tsipenyuk, Brian Chess, Gary McGraw	Seven Pernicious Kingdoms [31]	Borrows taxonomical terminology from biology (*e.g.,* kingdom, phylum). 7 kingdoms represent 7 categories of exploitable coding errors possible in C++, Java, C#, and ASP, plus an 8th kingdom, Configuration and Environment, unrelated to coding errors. Used as a key reference for the CWE (see Section 3.1.3.2). In *Software Security: Building Security In*, [32] Gary McGraw compares the Seven Pernicious Kingdoms with 19 Deadly Sins and OWASP Top Ten.
2005	Sam Weber, Paul. A Karger, Amit Paradkar (IBM Thomas J. Watson Research Center)	IBM Software Security Flaw Taxonomy [33]	Expressly intended for tool developers. Developed because "existing suitable taxonomies are sadly out-of-date, and do not adequately represent security flaws that are found in modern software." Correlated with available information about current, high priority security threats.
2005	Herbert Thompson, Scott Chase	*Software Vulnerability* Guide Vulnerability Categories [34]	Does not distinguish between software-level and system-level vulnerabilities, but characterizes both from developer's point of view, *i.e.*, in terms of whether developer can avoid them or needs to design around them.
2006	Fortify Software Security Research Group, Gary McGraw	Fortify Taxonomy of Software Security Errors [35]	Describes each vulnerability category in detail, with references to original sources, plus example code excerpts. In essence, the subset of the Seven Pernicious Kingdoms that can be detected by Fortify's source code analysis tools.
2006	Mark Dowd, John McDonald, Justin Schuh	Art of Software Security Assessment [36] Software Vulnerabilities	Describes several categories of exploitable coding errors and system-level vulnerabilities.
2007	OWASP	OWASP Top Ten 2007 [37]	Completed public comment period February 2007; publication anticipated Spring 2007. No longer includes software-specific vulnerabilities. Instead lists 5 attack patterns and 5 system-level vulnerabilities related to access control, identity or location spoofing, or sensitive data disclosure. Preface states: "A secure coding initiative must deal with all stages of a program's lifecycle. Secure web applications are only possible when a secure software development life cycle (SDLC) is used. Secure programs are secure by design, during development, and by default."

For Further Reading

TrustedConsultant: Threat and Vulnerabilities Classification, Taxonomies. Writing Secure Software blog; c2005 December 26.
Available from: http://securesoftware.blogspot.com/2005/12/threat-vulnerabilities-classification.html

Younan, Yves. *An Overview of Common Programming Security Vulnerabilities [thesis].* [Brussels, Belgium]: Vrije University of Brussels; 2003.
Available from: http://www.fort-knox.be/files/thesis.pdf

Seacord, Robert; Householder, A.D. (CMU SEI). *Final Report. A Structured Approach to Classifying Vulnerabilities. CMU SEI; 2005. Report No. CMU/SEI-2005-TN-003.*

Meunier, Pascal. *Wiley Handbook of Science and Technology for Homeland Security. Hoboken:* Wiley and Sons; 2007. Classes of Vulnerabilities and Attacks.

3.1.3.2 MITRE CWE

By 2005, there was a growing recognition among software assurance practitioners and tool vendors that the sheer number of software vulnerability lists and taxonomies was beginning to degrade their usefulness. The users of such lists and taxonomies either had to arbitrarily commit to only one or spend an increasing amount of time and effort to—

- Pinpoint the software-specific vulnerabilities in larger security vulnerabilities lists
- Correlate, rationalize, and fuse the different names, definitions, and classifications assigned to the same vulnerabilities.

The MITRE CVE [38] project had already made significant progress in addressing the second of these needs (one reason why CVE was being increasingly adopted by vulnerability assessment tool vendors and computer and cyber security incident response teams).

Influenced in part by the success of the CVE, Robert Martin, MITRE's project lead for the CVE effort, acknowledged the need to map CVE entries into categories of vulnerability types, thus enabling CVE users to more easily identify those vulnerabilities of interest to them. To address this need, MITRE produced a draft *Preliminary List of Vulnerabilities Examples for Researchers (PLOVER)* [39] which included in its CVE mapping several vulnerability categories of interest to software practitioners and researchers. The directors of the Software Assurance Metrics and Tool Evaluation (SAMATE) program, spearheaded by Department of Homeland Security (DHS) and NIST (see Section 6.1.10) and the DHS Software Assurance Program were so enthusiastic about PLOVER that DHS established and funded the CWE project, [40] which used PLOVER as a starting point for members of the software assurance community to collaboratively expand and refine into a dictionary of software-specific security vulnerability definitions. In addition to PLOVER, CWE derives vulnerability categories from software and application security taxonomies such as the Seven Pernicious Kingdoms and the OWASP Top Ten.

The CWE is intended to provide "a formal list of software weaknesses" that is standardized and definitive, as well as a set of classification trees to help define a taxonomy for categorizing those weaknesses according to software flaw. The CWE taxonomy will include elements that describe and differentiate between the specific effects, behaviors, exploitation mechanisms, and software implementation details associated with the various weakness categories.

The CWE will provide—

- Common language for describing software security weaknesses in architecture, design, and code
- Standard metric for software security tools that target those weaknesses
- Common baseline standard for identification, mitigation, and prevention of weaknesses
- As many real-world examples of the vulnerabilities and weaknesses it defines as possible.

Future products of the CWE project will include a formal schema defining a metadata structure to support other software security assurance-related activities, including software security metrics and measurement, software security tool evaluations and surveys, and methodologies for validating the product claims of software security tool vendors.

Though the CWE project is still in its early stages, given the success of the CVE, it is expected that the CWE will prove similarly useful and will eventually gain comparable exposure and adoption.

For Further Reading

Robert A. Martin (MITRE Corporation), Being Explicit About Security Weaknesses, *CrossTalk: The Journal of Defense Software Engineering,* (March, 2007)
Available from: http://www.stsc.hill.af.mil/CrossTalk/2007/03/0703Martin.pdf

3.1.4 Vulnerability Metadata and Markup Languages

Several vulnerability taxonomists have acknowledged that one of the most significant consumers of their efforts will be developers of automated vulnerability assessment and software security testing tools, as well as other automated tools, such as intrusion detection and prevention systems, application security gateways and firewalls. The need to characterize and exchange vulnerability information in standard, machine-readable syntax and formats had already been widely acknowledged, as demonstrated by the widespread adoption of the CVE by tool vendors, technology developers, and incident response teams.

Perhaps for the first time in the history of software, a technology has emerged for which standards are being defined coincidental with, if not before, implementations emerge—web services. A strong standards-oriented mentality in the web services industry led to the formation of the Organization for the Advancement of Structured Information Standards (OASIS), which is a

standards body devoted extensively to this technology. This means that for each challenge in design or implementation of web services applications, there is likely to be at least one proposed standard (and in some cases, more than one) to address it. The challenge of expressing and exchanging information about web service vulnerabilities among vulnerability scanners and other automated tools resulted in the proposed Application Vulnerability Description Language (AVDL) standard. AVDL, which was limited to expressing and exchanging information about application-level vulnerabilities (*vs.* system- and network-level vulnerabilities), was adopted by a number of commercial vendors; but the OASIS Technical Committee responsible for its adoption was disbanded in January 2006 due to lack of community participation.

The AVDL situation can be attributed, at least in part, to the success of MITRE's competing standard, the Open Vulnerability and Assessment Language (OVAL), described below.

3.1.4.1 OVAL

MITRE Corporation began defining OVAL [41] to provide a standard schema and language for expressing information about the publicly known vulnerabilities and exposures defined and categorized in the CVE. OVAL is now a *de facto* international standard in which vulnerability information is expressed in both XML and Structured Query Language (SQL), using standard CVE names and descriptions. By standardizing a formal XML-based language for capturing vulnerability and flaw information, OVAL eliminates the multiplicity of inconsistent descriptions currently written in plaintext that can lead to multiple redundant analyses and assessments of what often turns out to be the same flaw.

Unlike OASIS' AVDL, OVAL, because it is intended for use in concert with the CVE, is not limited to expressing application-level vulnerability information. Also unlike AVDL, the focus of OVAL is not exchange of such information between automated tools. Finally, OVAL differs from AVDL in a third significant way: it specifies a baseline vulnerability assessment methodology that must be implemented by all OVAL-compliant vulnerability assessment tools and vulnerability scanners (and which may also be used manually). This three-step assessment methodology represents a significant portion of OVAL's intellectual capital. The three OVAL assessment steps are—

- Collection in a predefined OVAL XML schema of the assessment and scanning data about target system configuration (*e.g.*, installed operating system and software applications and their configuration settings, including registry key settings, file system attributes, configuration files
- Inspection and checking for specific OVAL-defined vulnerabilities, configuration issues, and/or patches
- Capture in a predefined OVAL XML schema of reported scan/assessment results.

The assessor then uses the output of the OVAL assessment to identify and obtain appropriate remediation and patch information from the automated vulnerability assessment/scanning tools, the target system's vendor(s), and/or research databases and websites. By defining a standard vulnerability assessment process, OVAL is intended to increase consistency and repeatability of both the process and its report results.

In short, OVAL provides XML-based definitions of how to evaluate whether a specific vulnerability is present or how to check whether a system is configured in a particular manner. In conjunction with XML Configuration Checklist Data Format (XCCDF), OVAL can be used to describe low-level system configuration policy.

DoD has formally adopted both OVAL and CVE, and has mandated that all vulnerability tools acquired by DoD entities must be compatible with both CVE and OVAL.

3.1.4.2 VEDEF and SFDEF

In 2004, the UK's National Infrastructure Security Co-ordination Centre (NISCC) Capability Development and Research (CD&R) Group spun off an IA Metadata Team to develop a Vulnerability and Exploit Description and Exchange Format (VEDEF) [42] as part of a broader initiative to define 11 XML-based IA data exchange formats. VEDEF is also intended to fill what the CD&R Group perceived as a gap in the OVAL technology: lack of a formatting mechanism to enable "the free exchange of information on new vulnerabilities and exploits amongst responsible vendors, Computer Security Incident Response Teams (CSIRT), and their user communities." According to the CD&R Group researchers, OVAL was limited in usefulness to the storage of vulnerability data, but not its active exchange among automated systems; the CD&R Group researchers also claim that OVAL is more appropriately categorized as an exchange format for penetration testing data rather than vulnerability data. As of 2006, MITRE reportedly agreed with NISCC's position and with the need for a technology such as VEDEF.

What VEDEF does have in common with OVAL is the objective of defining a single standard format for expressing vulnerability information, and thereby eliminating the inconsistent proprietary expressions used across the numerous tools and CSIRTs in the European Community (EC). The CD&R Group cites several other European efforts to produce such a standard format; and after further examination, has determined that no obvious convergence path exist between the formats. Additionally, no readily apparent business need exists among the CSIRTs, who are responsible for the development and adoption of the various formats, for such a convergence. Instead, the group has presented two options to the EC Task Force-Computer Security Incident Response Teams (TF-CSIRT) VEDEF WG, to either maintain a mapping between the various formats or produce a common subset derived from across the formats that could then be used by tool vendors. What VEDEF is intended to provide, then, is a metadata interchange format for CVE and OVAL data as well as Common Malware

Enumeration (CME) data (see Section 3.2.3.2), and incident and vulnerability data in several other formats in use by CSIRTs in Europe and the United States.

The NISCC's intention from the start of the VEDEF project was to propose the resulting VEDEF as a standard for adoption by the Internet Engineering Task Force (IETF). The UK Central Sponsor for Information Assurance (CSIA), which funds the NISCC efforts, and the NISSC are also coordinating the VEDEF activities with related efforts by MITRE and NIST in the United States.

Susceptibility and Flaw Definition (SFDEF) is a more recent initiative by the NISCC, to define a metadata interchange format for CWE data, comparable to the VEDEF metadata interchange format for CVE, OVAL, and CME data.

For Further Reading

Ian Bryant (VEDEF WG Co-Chair), *Vulnerability and Exploit Description and Exchange Format (VEDEF) TF-CSIRT Progress Update,* (January 24, 2006).
Available from: http://www.terena.nl/activities/tf-csirt/meeting18/20060526_TF-CSIRT_VEDEF-WG.pdf

Ibid. *VEDEF TF-CSIRT Progress Update,* (May 26, 2006).
Available from: http://www.terena.nl/activities/tf-csirt/meeting17/vedef-bryant.pdf

3.2 Threats to Software

The adversary who finds and exploits a vulnerability in commercial software may be a blackhat seeking to publish a high-profile incident or vulnerability report that will reported in the news. The software company's only recourse is to rush and fix the reported problem, rapidly shipping out a patch. This is a never-ending cycle, because such blackhats appear to be endlessly motivated by the combination of ego and desire for intellectual challenge.

Unfortunately, an even greater number of attacks originate from malicious or criminal (or worse) individuals seeking to harm systems or steal data or money. Most of their efforts go unreported and all too often undetected. Nor is there any easy way for end users to recognize, when their personal information is stolen from their online bank records, that the vulnerability that made the identity theft possible originated in a faulty piece of code in the online banking application.

When software operates in a networked environment, virtually every fault becomes a potential security vulnerability. If the attacker can trigger the particular code path to the fault, that fault becomes a denial of service (DoS) attack waiting to happen—or worse. Attacks targeting or exploiting software bugs have increased exponentially with the coincidental proliferation of software-intensive systems, services, applications, and portals connected to the Internet and wireless-addressable embedded devices, such as cell phones, global positioning systems, and even medical devices. All of these systems are expected to operate continuously, not allowing for interruptions for such inconveniences as downloading of security patches.

Table 3-2 lists four categories of threats to software. Interception is listed separately here, but can also be seen as a form of subversion: one in which the usage of the software, rather than the software's own behavior, deviates from what is intended.

Table 3-2. Categories of Threats to Software

Threat Category	Description	Property Compromised	Objectives
Sabotage	The software's execution is suspended or terminated or its performance is degraded *or* The executable is deleted or destroyed	Availability	▶ DoS
Subversion	The software executable is intentionally modified (tampered with or corrupted) or replaced by an unauthorized party *or* Unauthorized logic (most often malicious code) is inserted into the executable	Integrity	▶ Transformation of the software into a suborned proxy to do the attacker's bidding ▶ Prevent the software from performing its intended functions correctly or predictably (a form of sabotage as well as subversion)
Interception	The software (or a restricted function within it) is accessed by an unauthorized entity	Access Control	▶ Unauthorized execution ▶ Illegal copying or theft of the executable
Disclosure	The software's technological and implementation details are revealed through reverse engineering (*e.g.*, decompilation, disassembly)	Confidentiality	▶ Pre-attack reconnaissance ▶ Obtain knowledge of proprietary intellectual property

The sources of threats to software fall into three general categories—

- external attackers
- malicious insiders who intentionally abuse the software
- nonmalicious insiders who intentionally misuse the software, because they are either frustrated by limitations of correct usage that inhibit their ability to get their jobs done efficiently, or are simply curious about how the software might respond to certain inputs or actions.

Threat sources in the latter two categories are consistent with the malicious, human-made faults described by Avizienis *et al.* [43]

A number of software security practitioners suggest that there is a fourth source of threats—the benign insider who accidentally submits input or performs an action that duplicates an attack pattern. Such accidental incidents usually originate from the user's misunderstanding of the software's functionality and constraints (such misunderstanding is often exacerbated by poorly designed user interfaces, lack of input validation, *etc.*). Because "accidental attack patterns" have human agency they cannot be termed "hazards" in the sense of "acts of God" or other non-human-sponsored events. And though they are unintentional, their outcome is the same as that of intentional attacks. As such, they are considered threats to the system. Fortunately, as long as they are not implemented only at the external boundaries of the system (*e.g.*, where the enterprise network interfaces with the Internet), the security measures that enable software to resist or withstand intentional threats will also enable them to resist or withstand unintentional ones.

The most frequently cited motivations for threats include military advantage, terrorism, activism, criminal gain, blackmail and intimidation, political or economic espionage, competitive advantage, vandalism, and mischief. Several books have been published that discuss the motivations and psychology of cyber attackers, including—

- George Mohay, *et al.*, *Computer and Intrusion Forensics*, (Norwood, MA: Artech House Publishers, 2006).
- Eric Greenberg, *Mission-Critical Security Planner: When Hackers Won't Take No for an Answer*, (Hoboken, NJ: Wiley Publishing, 2003).
- Winn Schwartau, *Cybershock: Surviving Hackers, Phreakers, Identity Thieves, Internet Terrorists and Weapons of Mass Disruption*, (New York, NY: Thunder's Mouth Press, 2000).
- Paul A. Taylor, *Hackers*, (Abingdon Oxford, UK: Routledge, 1999).
- David Icove, Karl Seger, William VonStorch, *Computer Crime: A Crimefighter's Handbook*, (Sebastopol, CA: O'Reilly & Associates, 1995).

NOTE: We have only reviewed those portions of the cited books that address the motivations of attackers and malicious insiders. The overall quality and accuracy of these books should not be inferred from their inclusion here.

Appendix C discusses the types of software that are most likely to come under threat.

Also contributing to the problem is the fact that many organizations lack adequate security programs for their information systems. Deficiencies include poor management, technical, and operational controls over enterprise information technology (IT) operations. Such problems in the Federal Government arena have been repeatedly reported by the Government Accountability Office since at least 1997. While these problems do not pose direct threats to software, they create an environment in which the levels of concern, resources, and commitment to security of software-intensive information systems are insufficient.

3.2.1 Threats From Offshoring and Outsourcing

Commercial software vendors are increasingly outsourcing the development of some or all of their software products to software houses outside of their borders, most often in developing countries in Asia, in Eastern Europe, and in South America. An ACM report, *Globalization and Offshoring of Software*, defines nine different reasons that lead corporations to outsource development, including lower costs of labor or an increased talent pool. In addition, many of these firms suggest that US immigration policy is prohibitive to bringing in talented IT professionals from overseas, and that the demand for IT professionals is increasing so fast that the US labor market cannot keep up with it.

This level of globalization often prevents US software firms from being able to determine or disclose (due to convoluted supply chain) geographical locations or national origins of developers (subcontractors or subsidiaries).

The practice of offshore outsourcing is starting to be recognized for its potential to jeopardize the security of the systems in which the offshore-developed software components are used. [44] But the concern about software of unknown pedigree (SOUP) is not limited to software that is known or suspected to have offshore origins. It is not uncommon for foreign nationals to work on software products within the United States. It is also conceivable that some US citizens working on software projects may be subverted. These concerns are not limited to the US Government—corporations are also concerned with the risks. In the March 2003 issue of *CIO Magazine*, Gary Beach [45] outlined many of the concerns associated with offshoring.

While there are risks associated with offshoring, it is in many ways an extension of the current business model for the US Government and that of many other countries, which have long-standing traditions of outsourcing—*i.e.,* contracting—the development and integration of their software-intensive systems, including security-critical, safety-critical, and mission-critical systems. The contractors that build software-intensive systems for the US Government are usually US Corporations; nevertheless, some of the reduced visibility and difficulty in establishing, verifying, and sustaining security controls that plague offshore outsourcing apply equally to software projects within the United States.

The "use commercial off-the-shelf (COTS)" imperative and increasing acceptance of open source software for use in government projects mean that the software-intensive systems developed under contract to DoD and other government agencies necessarily contain large amounts of SOUP, much of which may have been developed offshore. Many COTS vendors perform some level of off-shoring. Even for a COTS product developed wholly within the United States, the company may subsequently be purchased and managed by a foreign corporation [*e.g.,* the purchase by Lenovo (based in China) of the IBM PC division, or the attempted purchase by Checkpoint (based in Israel) of Sourcefire]. Increasingly, there are cases where foreign developers or corporations are more capable than their US counterparts of producing quality software for a particular market—making a strong case for the benefits of offshoring in those instances.

The SOUP problem is compounded by the fact that it can be difficult to determine the pedigree of a COTS product. Some COTS vendors may not be willing to divulge that information, whereas others may be unaware because some code was acquired through purchasing another company and the software pedigree may have been lost in the shuffle. Some companies, such as Blackduck Software and Palamida, are beginning to offer tools that can determine the pedigree of source code, and in some cases of binary executables, by comparing the code files to a database of code collected from open source projects. The efforts to improve the security of acquired software may result in improved mechanisms for assuring the supply chain.

Similarly, budget and schedule constraints may result in contractor-produced software that has not undergone sufficient (if any) design security reviews, code reviews, or software security (*vs.* functional security) tests. Such constraints are often the result of poor project management—deviation from the initial project plan, reluctance of management to refactor schedule and budget estimates, scope creep, *etc.* As a result, the essential but low-visibility tasks such as security testing and evaluation are performed at a minimal level, if at all—minimized in favor of activities that have more visible and immediate impact, such as implementing new functionality. Because many software firms and integrators operating in the United States hire non-US citizens, [46] DoD and other government agencies that contract with such firms are concerned that software produced by these contractors may be more vulnerable to sabotage by rogue developers.

DoD and other government agencies are realizing that globalization will only accelerate the offshoring trend. Significant cost-savings can be gained from offshoring and, potentially, quality improvements. In fact, in 2003 India had 75 percent of the world's Secure Systems Engineering–Capability Maturity Model (SSE-CMM) Level 5 certified software centers. [47] However, the more complex software supply chain associated with globalization increases software's vulnerability to a rogue developer (either a foreign national or subverted US developer) implanting malicious logic. As noted in the December 1999 *Defense Science Board (DSB) Task Force on Globalization and Security's final report* [48]—

> *The principal risk associated with commercial acquisition is that DoD's necessary, inevitable, and ever-increasing reliance on commercial software—often developed offshore and/or by software engineers who owe little, if any, allegiance to the United States—is likely amplifying DoD vulnerability to information operations against all systems incorporating such software.*
>
> *Commercial software products—within which malicious code can be hidden—are becoming foundations of DoD's future command and control, weapons, logistics, and business operational systems (e.g., contracting and weapon system support). Such malicious code,*

> *which would facilitate system intrusion, would be all but impossible to detect through testing, primarily because of software's extreme and ever-increasing complexity. Moreover, adversaries need not be capable of or resort to implanting malicious code to penetrate commercial software-based DoD systems. They can readily exploit inadvertent vulnerabilities (bugs, flaws) in DoD systems based on commercial software developed by others…. In either case, the trend toward universal networking increases the risk. Inevitably, increased functionality means increased vulnerability…*
>
> *Unfortunately, DoD has little if any market or legal leverage to compel greater security in today's commercial software market.*

Annex IV of the DSB report, entitled *Vulnerability of Critical US Systems Incorporating Commercial Software*, provided a more detailed discussion of the risks associated with commercial software acquisition, and the recommendations for risk mitigation.

Reports from the Government Accountability Office (GAO) in 2005 and 2006 [49] reinforced the DSB's message that the software used in US government systems is increasingly being developed by outsourced entities, including entities in countries whose governments have ambivalent or even adversarial relations with the United States. GAO found that "the combination of technological advances, available human capital, and foreign government policies has created a favorable environment for offshoring." However, GAO also found that analysts have a wide range of views of offshoring, leaving some uncertainty regarding how offshoring affects the US economy. The relevant excerpts of these GAO reports are included in DHS's *Security in the Software Life Cycle*. It is important to note, however, that GAO has identified government computer security and procurement as high-risk issues for more than ten years. Security issues related to the recent offshoring phenomenon can be considered an extension of these risks. According to the January 2007 update of GAO's *High Risk Series*, [50] DoD supply chain management, acquisition, and contract management have been considered high risk since 1990.

In Appendix C of its 2005 *Hard Problems List*, [51] the Infosec Research Council makes the following observations about the offshoring threat—

- Even domestic production of software is being outsourced to firms offshore.
- Even at reputable software companies, insiders can be bought to plant malicious code into key products used by the US Government.
- The focus of security attention for foreign and mobile code seems best shifted to the challenge of developing trustworthy software in the first place, and in conducting extensive static analysis of all critical software—especially foreign and mobile code.

Organizations outside of the US Government are starting to research the threat of rogue developers to understand and mitigate this threat. Many US corporations are concerned about rogue offshore developers stealing intellectual property. In March 2007, the Center for Strategic and International Studies (CSIS) released *Foreign Influence on Software Risks and Responses*. [52] According to the CSIS report, "The global supply chain conflicts with old notions of trust and security. The United States could 'trust' a product that came from US factories and workers. Many would say that we cannot extend the same degree of trust to a product that comes from a foreign and perhaps anonymous source." The CSIS report notes that while this threat is possible, some common software development trends reduce the risks associated with rogue developers, including [53]—

- In a distributed development environment, individual groups do not necessarily know how their code will fit with the rest of the system.
- Companies limit information shared with offshore teams to reduce the likelihood that intellectual property will be stolen.
- Most companies audit any changes made to code and track who made that change. Similarly, most companies use authorization software to limit developers to changing only the code they are responsible for.
- Many companies use some sort of software assurance tool and/or rely on security review teams for code review and/or security testing.

CSIS provides several suggestions for reducing the risks of foreign influence. These suggestions range from improving US leadership in advanced technology to increased R&D to encouraging acquisition efforts to focus on the software's assurance rather than on potential foreign influence. In addition, CSIS recommends improving existing certification processes for software to address software assurance.

In 2006, the Association for Computing Machinery's (ACM) Job Migration Task Force published a report, *Globalization and Offshoring of Software*, [54] which outlines a number of risks magnified and created through offshoring. These findings mirror those of other organizations. Businesses engaging in offshoring increase the risk of intellectual property theft, failures due to longer supply chains, and complexity introduced by conflicting legal arguments. Similarly, US Government participation in offshoring increases risks to national security: the use of offshoring for COTS technologies makes it difficult to determine the pedigree of code, potentially allowing hostile nations or nongovernmental hostile agents (*e.g.*, terrorists and criminals) to compromise these systems.

The report recommends that businesses and nations employ strategies to mitigate these risks—

- Security and data privacy plans certified to meet certain standards
- No outsourcing of services without explicit customer approval
- Careful vetting of offshore provider
- Encrypted data transmissions and tight access controls on databases to minimize inappropriate access by offshore operations

- Stronger privacy policies and investment in research for strong technical methods to secure data and systems
- Nations in particular should implement bilateral and international treaties on correct handling of sensitive data and security compromises.

As shown by the ACM and CSIS, many corporations have concerns similar to those of the US Government in regard to outsourcing and offshoring. In fact, many of the access control measures discussed in Appendix E, Section E.3, are being applied to prevent rogue developers from stealing intellectual property. In particular, limiting developer access to the code base and reviewing all code before it is committed are necessary to adequately protect intellectual property and assure the software security. However, the goal for intellectual property protection is to limit the information that the outsourced developers receive, whereas the goal for security is to ensure that they are not supplying malicious code—so the practices currently employed may need to be modified slightly to provide adequate protection from both threats. Also, ACM, CSIS, and the US Government agree that existing software certification mechanisms are inadequate to assure software security.

A task force of the DSB is preparing a report that is expected to suggest a variety of foreign pedigree detection measures and constraints and countermeasures for SOUP used by the US Government. However, the report will apparently stop short of stating that all software acquired by US defense agencies and military services should originate in the United States.

Offshoring firms are working to alleviate these concerns lest large software vendors stop relying on them. These firms increasingly employ security practices such as International Standards Organization (ISO)-certified security processes (*e.g.,* ISO 17799 and ISO 26001), strict access controls, IPSec virtual private networks for communicating with customers, nondisclosure agreements, and background checks [55]. In general, these strategies are geared more towards preventing intellectual property theft—but this shows that these firms are willing to alter their practices to satisfy their customers. In the future, offshoring firms may implement more stringent protections against rogue developers on their own. In fact, it is possible that enough demand will cause the offshoring market to address the rogue developer concern. Outsource2India, an outsourcing management firm, is certain security will be adequately addressed by Indian firms because "in the instance of a single security breach, the publicity created will not only destroy the reputation of the concerned company, but of India's well established name as the foremost outsourcing destination." [56]

In summary, offshore sourcing of software products and outsourcing of software development services pose the following potential problems—

- Presence of SOUP in software systems built from outsourced components makes assurance of the dependability and security of that software very difficult.

- Software developers in developing countries, such as India, the Philippines, China, and Russia, are frequently subject to government influence, pressure, or direct control.
- It is difficult to obtain verified knowledge of developer identities or to achieve developer accountability; in the case of COTS and open source software (OSS) software, these are often impossible to obtain.

For Further Reading

Ellen Walker (DACS), "Software Development Security: A Risk Management Perspective*". DoD SoftwareTech News 8, no. 2* (July 2, 2005).
Available from: http://www.softwaretechnews.com/stn8-2/walker.html

3.2.2 When are Threats to Software Manifested?

As noted in the DHS's draft *Security in the Software Life Cycle*, threats to software dependability can manifest during the software's development, distribution, deployment, or operation.

The source of threats to software under development is primarily insiders, usually from a rogue developer who wishes to sabotage or subvert the software by tampering with or corrupting one or more of the following—

- Software requirements, design, or architecture
- Source code or binary components (custom-developed or acquired)
- Test plan or test results
- Installation procedures or configuration parameters
- End user documentation
- Tools or processes used in developing the software.

A rogue developer does not necessarily have to belong to the organization that is distributing or deploying the software system. That person could work for one of the commercial software vendors or open source development teams whose components were acquired for use in the software system.

With so much development work being outsourced to smaller or foreign companies or through acquisitions and mergers by large software development firms, it is often difficult, if not impossible, to discover where software was actually built. SOUP is being distributed by well-known commercial software companies and integrators. SOUP is often used in critical software systems in the Federal Government, DoD, and critical infrastructure companies. The insider threat is not limited to rogue developers within the corporation that built the software—the SOUP problem makes it impossible to know exactly who is responsible for creating most of the components used in DoD and other government software systems.

Threats to software during its distribution may originate from the distributor (*e.g.*, inclusion of hidden malicious logic in the executable when it is moved to the distribution/download server or copied to the distribution medium). Or such threats may come from external attackers who intercept and

tamper with download transmissions, or use cross-site scripting to redirect unsuspecting software customers to malicious download sites.

Threats to software during its operation can come from either insiders (authorized users) or external attackers. They are usually targeted at software systems that are accessible *via* a network and that contain known vulnerabilities that attackers understand how to exploit.

Fortunately, it appears that so far, the software industry, though highly globalized and drawing on a geographically diverse workforce, is implementing controls and extraordinary means to protect the integrity of their source code to ensure that it is not tainted with open source or unauthorized components. The design, implementation, and daily execution of global supply chains of many software firms involve a rigorous quality/risk management process.

3.2.3 How are Threats to Software Manifested?

Threats to software are manifested in two basic ways—

- Human attackers who target or exploit vulnerabilities
- Execution of malicious code embedded or inserted in the software.

In the former case, the attacker may directly attack the software that is his or her ultimate target, or may attack something contiguous to that software—either a component of the software's execution environment, or of an external system that interoperates with the targeted software.

In the latter case, the malicious code acts as a kind of proxy on behalf of the attacker, to cause the software to operate in a manner that is inconsistent with its specification and its users' expectations, but which achieves some objective of the attacker who originally planted or delivered the malicious code.

3.2.3.1 Common Attacks and Attack Patterns

Common attacks and attack patterns have been widely studied and well documented for operating systems, network devices, database management systems, web applications, and web services, but have not yet been well established for software-specific based attacks. The root cause for the majority of the application, network, and operating system vulnerabilities exists in flaws inherent in the software code itself. Software security attack patterns are currently being researched and developed, and are being designed to expose exploited code development flaws and to describe the common methods, techniques, and logic that attackers use to exploit software.

Technologically, software attack patterns are a derivative of the software design patterns used by developers when architecting code. Instead of focusing on specifying desirable features or attributes for software, attack patterns attempt to describe the mechanisms used to compromise those features and attributes. The following is an example of an attack pattern for a buffer overflow: [57]

Buffer Overflow Attack Pattern:

Goal: Exploit buffer overflow vulnerability to perform malicious function on target system
Precondition: Attacker can execute certain programs on target system
Attack: *and*

1. Identify executable program on target system susceptible to buffer overflow vulnerability
2. Identify code that will perform malicious function when it executes with program's privilege
3. Construct input value that will force code to be in program's address space
4. Execute program in a way that makes it jump to address at which code resides

Postcondition: Target system performs malicious function

Attack patterns are being used in the education of developers and in the development of risk assessments and threat models for software systems.

The first notable effort to enumerate attack patterns that specifically target software (rather than data or networks) was documented by Greg Hoglund and Gary McGraw in their book, *Exploiting Software: How to Break Code*. [58] The authors identified 48 software-based attack patterns and provided detailed explanations and examples of each. Since this publication, there have been several attempts at categorizing and classifying software based attack patterns, including—

- **CAPEC**: Initiated by the DHS Software Assurance Program in 2006, the Common Attack Pattern Enumeration and Classification (CAPEC) project will define a standard taxonomy of definitions, classifications, and categorizations of software-targeting attack patterns. The first draft CAPEC taxonomy is expected to be released in 2007.
- **WASC Threat Classification**: An effort by members of Web Application Security Consortium (WASC) to classify and describe security threats to web applications. The main objective of the threat classification effort is "to develop and promote industry standard terminology for describing these issues. Application developers, security professionals, software vendors, and compliance auditors will have the ability to access a consistent language for web security related issues."
- **SearchSecurity.com Web Application Attacks Learning Guide:** [59] An informal taxonomy of web application and web service attacks that is tutorial in nature. It is essentially a compendium of attack-related information (including definitions and descriptions of individual categories of attacks, and recommended attack countermeasures; the *Guide* also provides general application security recommendations). Authored by SearchSecurity.com editors and contributors, the *Guide* includes material from numerous other print and online sources.

For Further Reading

US CERT, *Attack Patterns*, (Washington, DC: US CERT)
Available from: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/attack.html

3.2.3.2 Malicious Code

In *Guidance for Addressing Malicious Code Risk*, National Security Agency (NSA) defines malicious code, or malware, as—

> *Software or firmware intended to perform an unauthorized process that will have adverse impact on the confidentiality, integrity, availability or accountability of an information system. Also known as malicious software.*

Significant amounts of activity and effort in the software security assurance community are directed towards defining techniques and developing tools for the detection and eradication of malicious code, during both the development and deployment phases of the software/system life cycle, and during software/system operation.

Recently, the DHS Software Assurance Program has shone a light on the threat of malicious code to the dependability of software and systems by establishing a Malware WG (described in Section 6.1.9).

More recently, MITRE's CME initiative has begun working to define a standard set of identifiers for both common and new malicious code threats. CME is not attempting to replace individual virus scanner vendor and test tool vendor designations for viruses and other forms of malware. Instead, through CME, MITRE seeks to foster the adoption of a shared, neutral indexing capability for malware that will mitigate public confusion arising from malware incidents.

For Further Reading

malware.org v.4.0.
Available from: http://www.malware.org

SANS Institute, *Malware FAQ [frequently asked questions]*, (August 27, 2004).
Available from: http://www.sans.org/resources/malwarefaq

iDefense WebCast, *Rootkits and Other Concealment Techniques in Malicious Code.*
Available from: http://complianceandprivacy.com/AV-Media/iDefense-rootkits-malicious-code-replay.html

Bryan Sullivan (SPI Dynamics), *Malicious Code Injection: It's Not Just for SQL Anymore.*
Available from: http://www.spidynamics.com/spilabs/education/articles/code-injection.html

3.2.3.2.1 Anti-Malware Guidance

Several organizations and individuals have published guidance on how to address the malicious code threat. A great deal of this guidance is directed at system and network administrators, computer security incident response teams, and in some cases, desktop system end users. The focus is primarily or exclusively on

prevention of and incident response to virus and worm "infections" and spyware insertions in operational systems. Typical of such guidance are NIST Special Publication (SP) 800-83, *Guide to Malware Incident Prevention and Handling*, and other anti-malware guidance published by NIST; the NISCC's *Mitigating the Risk of Malicious Software*; the online newsletter *Virus Bulletin*; [60] and Skoudis and Zeltser's book, *Malware: Fighting Malicious Code*.

This said, a small but growing number of guidance publications are intended also or exclusively for developers and/or integrators, and focus on malicious code insertion during the software development process. In June 2006, NSA's Malicious Code Tiger Team (established by the NSA Information Systems Security Engineering organization, with members also invited from the NSA Center for Assured Software) began work on *Guidance for Addressing Malicious Code Risk*. The primary purpose of this document is to provide guidance on which safeguards and assurances should be used throughout the software life cycle to reduce—

- The likelihood the malicious code will be inserted in software under development
- The impact (in terms of extent and intensity of damage) of malicious code that is present in software in deployment.

This guidance is intended, at a minimum, to make an adversary work harder and take more risks to mount a successful malicious code attack. Malicious code for which this guidance is aimed can take many forms, depending on an attacker's motives, accessibility, and the consequences of getting caught. The guidance describes protection mechanisms that may prevent not only current malicious code attacks, but as-yet-undefined future attacks.

The software development life cycle in *Guidance for Addressing Malicious Code Risk* is organized according to ISO/IEC 12207, *Software Life Cycle Processes*. ISO/IEC 12207 and 15288, *System Life Cycle Processes*, are also being used to define the life cycle in DoD's systems assurance guidebook (see Section 6.1.1 for a description), which will provide easy traceability between these two documents.

The document's express audience is DoD software and system development organizations. It provides guidance for mitigating the threat of malicious code in systems developed for use by DoD, including national security systems. While the guidance is applicable for all types of DoD software, the scenarios and associated levels of precaution in the document are limited to security-enforcing and security-relevant [61] software used under a variety of environmental and operational conditions. However, while it is intended primarily for DoD, the document, which was published in April 2007, is available in the public domain through the NSA website (http://www.nsa.gov).

For Further Reading

Gary McGraw, Greg Morrisett, *Attacking Malicious Code: A Report to the Infosec Research Council., final report, (*Infosec Research Council (IRC) Malicious Code Infosec Science and Technology Study Group, September/October, 2000).
Available from: http://www.infosec-research.org/docs_public/ISTSG-MC-report.pdf or http://www.cigital.com/irc/ or http://www.cs.cornell.edu/home/jgm/cs711sp02/maliciouscode.pdf

Edward Skoudis, *Thwarting the Ultimate Inside Job: Malware Introduced in the Software Development Process,* SearchSecurity.com. (*circa* April 1, 2004).
Available from: http://searchappsecurity.techtarget.com/tip/0%2C289483%2Csid92_gci1157960%2C00.html

Rodney L. Brown, (University of Houston at Clear Lake), *Non-Developmental Item Computer Systems and the Malicious Software Threat,* (*circa* April, 1991).
Available from: http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19910016319_1991016319.pdf

Donald J. Reifer, *et al.,* (Reifer Consultants Inc.), *COTS-Based Software Systems,* (Springer; 2005).
Understanding Malicious Content Mitigation for Web Developers. CMU CERT Coordination Center.
Available from: http://www.cert.org/tech_tips/malicious_code_mitigation.html

Sam Nitzberg, et al., (InnovaSafe, Inc.), *Trusting Software: Malicious Code Analyses,* (InnovaSafe, Inc). Available from: http://www.innovasafe.com/doc/Nitzberg.doc

NISSC, *Final Report, The Importance of Code Signing,* Report no. 01089, TN02/2005 (NISCC, December 14, 2005).
Available from: http://www.niscc.gov.uk/niscc/docs/re-20051214-01089.pdf

In addition, *Guidance for Addressing Malicious Code Risk* will be published on the NSA website:
Available from:(http://www.nsa.gov) later in 2007.

3.2.3.2.2 Anti-Malware Education

Academic curricula are also beginning to reflect the significance of malicious software to the security and dependability of software-intensive systems, and particularly network-based systems. Courses on computer, system, information, network, cyber, and Internet security; information assurance; intrusion detection; attacks and countermeasures; *etc.*, routinely include lectures and readings on malicious code, albeit often focusing predominantly (if not exclusively) on the varieties that are typically delivered/replicated over networks (*e.g.,* viruses, worms, and increasingly, spyware).

More recently, courses that focus solely on malicious code have begun to emerge, such as the "Malware Seminar" at University of Virginia, "Computer Virus and Malicious Software" at the Rochester Institute of Technology, "Mobile Code Security" at Syracuse University, "Malicious Code and Forensics" at Portland State University (Oregon), and "Malicious Software" at Capitol College (Laurel, Maryland)—though again, the focus is often predominantly on network delivered/replicating varieties and operational detection, prevention, and recovery (*vs.* detection of malicious logic inserted into software under development).

With tongue in cheek, Scott Craver, an assistant professor in the Department of Electrical and Computing Engineering at Binghamton University (New York) and a specialist in cryptography, information hiding (steganography), digital watermarking, and digital rights management, established what may be the world's only security-related programming contest—the Underhanded C Contest. To date, there have been two Underhanded C Contests, [62]—2005 and 2006—in

which developers were challenged to write C-language programs that appeared to be innocuous, but which in fact implemented malicious behavior. Submissions to the Underhanded C Contest and to a number of other not specifically malicious code-oriented Obfuscated Code Contests demonstrate how much complexity can be intentionally written into source code so that even a skilled code reviewer will not be able to determine the code's true purpose.

Section 7.2 provides a general discussion of academic education and professional training and certification in the software security assurance domain.

3.2.3.2.3 Anti-Malware Research

Researchers in academia and in the antivirus/antispyware industry are investigating more effective, holistic technological solutions for reducing malicious code threats to software in deployment. Stephen Posniak delivered a presentation entitled *Combined Hardware/Software Solutions to Malware and Spam Control* [63] to the 2005 Virus Bulletin Conference that provides a reasonable survey of current solutions in this area. The problem of detection and eradication of malicious code embedded in software during its development is also being investigated by several researchers.

From 2002–2003, researchers in the Princeton [University] Architecture Laboratory for Multimedia and Security published the results of their efforts to develop a hardware-based secure return address stack [64] that would prevent malicious code insertions that resulted from buffer overflows, and a runtime execution monitor [65] that would detect and prevent execution of malicious code embedded in operational software.

More recently, the Function Extraction for Malicious Code (FX/MC) project [66] within the Survivable Systems Engineering group at the Carnegie Mellon University Software Engineering Institute (CMU SEI) has applied formal methods and function theory to the problem of performing automated calculations of program behaviors to define behavior signatures, with the goal of obtaining precise information on structure and functionality of malicious code so that anti-malware strategies can be more effectively tailored. The SEI researchers intend for the core FX technology to be more widely applicable to the analysis of software, *e.g.,* for the detection of errors and vulnerabilities, and for the validation of the "goodness" of authentication, encryption, filtering, and other security functions implemented by software.

Researchers in New Mexico Tech's IA Center of Excellence are researching advanced static analysis techniques for detection of malicious code and analysis of malicious executables.[67]

The Hiding and Finding Malicious Code [68] project at the Johns Hopkins University did not attack the problem of detecting and preventing malicious code head on, but instead investigated techniques for creating and hiding malicious code, with the objective of gaining a better understanding of such techniques in order to enable the creation of more effective malicious code detection measures.

A great deal of malicious code research is being done under the larger umbrella of research into security and trustworthiness of electronic and Internet-based voting.

References

15 We have adapted Ross Anderson's definition of "system" in Ross Anderson, *Security Engineering: a Guide to Building Dependable Systems* (New York: John Wiley and Sons, 2001). "A software-intensive system is a collection of products or components predominantly implemented in software." These products/components may include application, data, communications, middleware, and operating system products/components as well as the firmware and hardware products/components of the physical platforms on which those software components are hosted. According to Anderson, a system also includes the people that interact with, oversee, regulate, or otherwise observe the system, and the processes, procedures, policies that govern and in uence the operations of the system's technological and non-technological elements. For purposes of this document, unless otherwise stated, "system" refers to the technological elements of the system, and excludes the non-technological elements.

16 Andy Ozment, "Research Statement" [web page] (Cambridge, UK: University of Cambridge, ca. 2006). Available from: http://www.cl.cam.ac.uk/~jo262/research.html

17 This is not a new problem. It was documented by Andy Ozment of the University of Cambridge in his research statement, and almost 10 years earlier in virtually identical language by Ed Felten, *et al.* in Edward Felten and Drew Dean (Princeton University), *Secure Mobile Code: Where Do We Go From Here?, in Proceedings of the DARPA Workshop on Foundations for Secure Mobile Code*, March 1997. Felten and Dean wrote, "The market is (for the moment, at least) asking for feature-packed, quickly developed, insecure software. Until the market changes, little progress will be made commercially."

18 Ryan Naraine, "Paying for Flaws Pays Off for iDefense," *eWeek* (March 3, 2005). Available from: http://www.eweek.com/article2/0,1759,1772418,00.asp

19 Organization for Internet Safety (OIS), "Guidelines for Security Vulnerability Reporting and Response, V2.0" [web page] (Houston, TX: OIS). Available from: http://www.oisafety.org/guidelines/

20 Robert P. Abbott, *et al.* (National Bureau of Standards), *The RISOS Project: Security Analysis and Enhancements of Computer Operating Systems*, interagency report no. NBSIR 76-1041 (Gaithersburg, MD: National Bureau of Standards, April 1976).

21 Carl E. Landwehr, *et al., A Taxonomy of Computer Program Security Flaws, With Examples*, report no. NRL/FR/5542-93-9591 (Washington, DC: Naval Research Laboratory, November 19, 1993). Available from: http://chacs.nrl.navy.mil/publications/CHACS/1994/1994landwehr-acmcs.pdf.

Also published in: ACM Computing Surveys 26, no.3 (September 1994). Available from: http://doi.acm.org/10.1145/185403.185412

22 Bishop's paper was published only a few months before Taimur Aslam published his master's dissertation with a nearly identical title: Taimur Aslam, "A Taxonomy of Security Faults in the Unix Operating System" (PhD dissertation, Purdue University, Lafayette, IN, August 1995).

23 Matt Bishop and D.A.. Bailey, *A Critical Analysis of Vulnerability Taxonomies*, tech. report no. CSE-96-11 (Davis, CA: University of California, September 1996). Available from: http://seclab.cs.ucdavis.edu/projects/vulnerabilities/scriv/ucd-ecs-96-11.pdf *or* http://www.cs.ucdavis.edu/research/tech-reports/1996/CSE-96-11.pdf.

In late 2005, the anonymous author of the "Writing Secure Software" blog published a less formal analysis of the various vulnerability taxonomies in circulation. See "Trusted Consultant, Threat and Vulnerabilities Classification, Taxonomies," "Writing Secure Software" blog, December 26, 2006. Available from: http://securesoftware.blogspot.com/2005/12/threat-vulnerabilities-classification.html

24 Wenliang Du and Aditya Mathur, "Categorization of Software Errors That Led to Security Breaches," in *Proceedings of the National Information Systems Security Conference*, 1998.
Available from: http://www.cis.syr.edu/~wedu/Research/paper/nissc98.ps

25 "OWASP Top Ten Project" [web page] (Columbia, MD: Open Web Application Security Project).
Available from: http://www.owasp.org/index.php/OWASP_Top_Ten_Project

26 SANS Institute, *How to Eliminate the Ten Most Critical Internet Security Threats, Version 1.32* (Bethesda, MD: The SANS Institute, January 18, 2001).
Available from: http://www.sans.org/top20/2000/10threats.doc *or*
http://www.sans.org/top20/2000/10threats.rtf.

In 2004, SANS published a revised version: *SANS Top 20 Internet Security Vulnerabilities, Version 5.0* (Bethesda, MD: The SANS Institute, October 8, 2004).
Available from: http://www.sans.org/top20/2004/

27 Frank Piessens (Catholic University of Leuven), *A Taxonomy (With Examples) of Software Vulnerabilities in Internet Software*, report no. CW 346 (Leuven, Belgium: Catholic University of Leuven, 2002).
Available from: http://www.cs.kuleuven.ac.be/publicaties/rapporten/cw/CW346.abs.html

28 The MITRE Corporation, "CVE: Common Vulnerabilities and Exposures" [website] (Bedford, MA: MITRE). Available from: http://cve.mitre.org/

29 Fortify Software Inc., "CLASP: Comprehensive, Light Application Security Process" [web page] (Palo Alto, CA: Fortify Software Inc.)
Available from: http://www.fortifysoftware.com/security-resources/clasp.jsp

30 Michael Howard, David LeBlanc, and John Viega, *19 Deadly Sins of Software Security: Programming Flaws and How to Fix Them*, 1st ed. (Emeryville, CA: McGraw-Hill/Osborne, 2005).

31 Katrina Tsipenyuk, Brian Chess, and Gary McGraw, "Seven Pernicious Kingdoms: a Taxonomy of Software Security Errors," *IEEE Security and Privacy* 6 (November–December 3, 2005).
Available from: http://vulncat.fortifysoftware.com/docs/tcm_taxonomy_submission.pdf

32 Gary McGraw (Digital), "Software Security: Building Security In," [website] (Herndon, VA).
Available from: http://www.swsec.com/

33 Sam Weber, Paul A. Karger, and Amit Paradkar, "A Software Flaw Taxonomy: Aiming Tools at Security," *ACM SIGSOFT Software Engineering Notes* 4 (July 30, 2005).
Available from: http://portal.acm.org/citation.cfm?id=1082983.1083209&coll=GUIDE&dl=GUIDE&CFID=15151515&CFTOKEN=6184618

34 Herbert Thompson and Scott Chase, *The Software Vulnerability Guide* (Boston, MA: Charles River Media, 2005).

35 Fortify Software, "Fortify Taxonomy: Software Security Errors" [web page] (Palo Alto, CA: Fortify Software).
Available from: http://www.fortifysoftware.com/vulncat/

36 Mark Dowd, John McDonald, and Justin Schuh, *The Art of Software Security Assessment, Identifying and Preventing Software Vulnerabilities*, 1st ed. (Boston, MA: Addison-Wesley Professional, 2006).

37 "OWASP Top Ten Project" [web page], *op cit.*

38 "CVE" [website], *op cit.*

39 Steve Christey (MITRE Corporation), *PLOVER: Preliminary List of Vulnerability Examples for Researchers*, version 0.14 (Bedford, MA: The MITRE Corporation, August 2, 2005).
Available from: http://www.cve.mitre.org/docs/plover/

40 "CWE: Common Weakness Enumeration" [website] (Bedford MA: The MITRE Corporation).
Available from: http://cwe.mitre.org/

41 "OVAL: Open Vulnerability Assessment Language" [website] (Bedford MA: The MITRE Corporation).
Available from: http://oval.mitre.org/

42 "VEDEF: Vulnerability and Exploit Description and Exchange Format" [website] (Amsterdam, The Netherlands: Trans European Research and Academic Networks Association [TERENA] Task Force-Computer Security Incident Response Team [TF-CSIRT]).
Available from: http://www.vedef.org/ *or*
http://www.secdef.org/vedef/. (*As of 2 April 2007, neither of these web sites was accessible, apparently due to reconfigurations in progress.*)

43 Algirdas Avizienis, *et al.*, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Transactions on Dependable and Secure Computing* 1, no. 1 (January–March 2004).

44 GAO, *Defense Acquisitions: Knowledge of Software Suppliers Needed to Manage Risk*, report no. GAO-04-678 (Washington, DC: GAO, May 2004).
Available from: http://www.gao.gov/docdblite/summary.php?rptno=GAO-04-678&accno=A10177, *and*

Defense Science Board, *Final Report of the Defense Science Board Task Force on Globalization and Security*, annex IV of *Vulnerability of Essential US Systems Incorporating Commercial Software* (Washington, DC: OUSD/AT&L, December 1999).
Available from: http://www.acq.osd.mil/dsb/reports/globalization.pdf

45 Gary Beach, "Offshore Costs," *CIO Magazine* (March 1, 2003).
Available from: http://www.cio.com/archive/030103/publisher.html

46 Such foreign employees may be holders of permanent resident green cards or of temporary worker [H1-B], business [B-1], student [F-1], or exchange [J-1] visas. Note that Federal procurement rules, except for those covering national security systems, are not allowed to restrict vendors to employing only US citizens. Nonetheless, since September 11, 2001, immigration restrictions have been imposed that have greatly curtailed the number of foreign workers admitted into the United States from countries whose governments are hostile to the United States. As a result, many firms that used to hire such workers have begun complaining about the negative impact such restrictions are having in terms of the available labor pool and reduced productivity.

47 Navyug Mohnat, "Why 'India Inside' Spells Quality," *DataQuest* (October 27, 2003).
Available from: http://www.dqindia.com/content/advantage/103102703.asp

48 Defense Science Board, *Final Report of the Defense Science Board Task Force on Globalization and Security, op cit.*

49 GAO. *Offshoring of Services: an Overview of the Issues*, report no. GAO-06-5 (Washington, DC: GAO, November 2005).
Available from: http://www.gao.gov/docdblite/summary.php?rptno=GAO-06-5&accno=A42097, *and*

GAO, *Offshoring: US Semiconductor and Software Industries Increasingly Produce in China and India*, report no. GAO-06-423 (Washington, DC: GAO, September 2006).
Available from: http://www.gao.gov/new.items/d06423.pdf

50 GAO, *High-Risk Series* (Washington, DC: GAO, January 2007).
Available from: http://www.gao.gov/new.items/d07310.pdf

51 IRC, *Hard Problems List, Version 2.0, op cit.*

52 James A. Lewis, *Foreign Influence on Software Risks and Recourse* (Washington, DC: Center for Strategic and International Studies Press, March 2007).
Available from: http://www.csisbookstore.org/index.asp?PageAction=VIEWPROD&ProdID=166

53 Ibid. 20.

54 William Aspray, Frank Mayadas, and Moshe Y. Vardi, eds., *Globalization and Offshoring of Software: a Report of the ACM Job Migration Task Force*, report no. ACM 0001-0782/06/0200 (New York, NY: Association for Computing Machinery, 2006).
Available from: http://www.acm.org/globalizationreport/

55 Eric Rongley, "Using China for Offshore Software Development," *China Tech News* (January 22, 2007).
Available from: http://www.chinatechnews.com/2007/01/22/4882-using-china-for-offshore-software-development-eric-rongley-ceo-of-bleum/, *and*

"IT Infrastructure and Security" [web page] (North Vancouver, British Columbia, Canada: InnoInco).
Available from: http://www.innoinco.com/how_we_work/infrastructure.html

56 "Data Privacy and Security Concerns in Outsourcing" [web page] "India: Outsource2India."
Available from: http://www.outsource2india.com/why_india/articles/data_privacy.asp

57 Andrew P. Moore, Robert J. Ellison, and Richard C. Linger (Carnegie Mellon University Software Engineering Institute [CMU SEI]), *Attack Modeling for Information Security and Survivability*, technical note no. CMU/SEI-2001-TN-001 (Pittsburgh, PA: CMU SEI, March 2001).
Available from: http://www.sei.cmu.edu/pub/documents/01.reports/pdf/01tn001.pdf

58 Greg Hoglund and Gary McGraw, "Exploiting Software: How to Break Code," Chapter 2, Boston, MA: Addison-Wesley, 2004.

59 Stephen Posniak, "Combined Hardware/Software Solutions to Malware and Spam Control" (paper presented at the Virus Bulletin Conference, October 2003).
Available from: http://csrc.nist.gov/fasp/FASPDocs/network-security/Posniak-VB05.pdf

60 "Virus Bulletin" [website] (Abingdon, Oxfordshire, UK: Virus Bulletin Ltd.).
Available from: http://www.virusbtn.com/

61 "Security relevant" software is a portion of software that (based on system architecture) does not itself function to enforce system security policy but can subvert the enforcement of it.

62 "The Underhanded C Contest" [website] (Binghamton, NY: Binghamton University).
Available from: http://bingweb.binghamton.edu/~scraver/underhanded/ *or*
http://www.brainhz.com/underhanded/

63 Posniak, *Combined Hardware/Software Solutions to Malware and Spam Control, op cit.*

64 Ruby B. Lee, et al., "Enlisting Hardware Architecture to Thwart Malicious Code Injection," in *Proceedings of the International Conference on Security in Pervasive Computing*, March 2003, 237–252.
Available from: http://palms.ee.princeton.edu/PALMSopen/lee03enlisting.pdf

65 A. Murat Fiskiran and Ruby B. Lee, "Runtime Execution Monitoring (REM) to Detect and Prevent Malicious Code Execution," in *Proceedings of the International Conference on Computer Design*, October 2004: 452–457.
Available from: http://palms.ee.princeton.edu/PALMSopen/fiskiran04runtime.pdf

66 CMU SEI (Pittsburgh, PA) "Function Extraction for Malicious Code (FX/MC)" [web page]
Available from: http://www.cert.org/sse/fxmc.html

67 "Malware Analysis and Malicious Code Detection" [web page] (Socorro: New Mexico Tech Computer Science Department).
Available from: http://www.cs.nmt.edu/research.html#malware

68 Lucas Ballard, *et al., Group 2 Report on Hiding Code* (November 11, 2005).
Available from: http://www.cs.jhu.edu/~sdoshi/index_files/Task2.pdf

4

Secure Systems Engineering

This SOAR focuses primarily on current activities, techniques, technologies, standards, and organizations that have as their main objective the production and/or sustainment of secure software. However, software is typically an element or part of a larger system, whether it is a software-intensive system or a system that is composed of both hardware and software elements. The perspective of this section, then, is that of the systems engineer(s) who seeks to understand the security issues associated with the software components of a larger secure system.

As noted in Section 2.2, in defining *system*, we have adapted Ross Anderson's definition by focusing only on the technological elements of a system, *i.e.*, its hardware, software, and communications components.

Within this section, we will summarize typical issues, topics, and techniques used by systems engineers to build secure systems, with a view towards clarifying the relationships between those issues, topics, and techniques and those that pertain specifically to secure software development.

Note: Unless otherwise noted, the source for the information in this section is Anderson's *Security Engineering*.

According to Anderson—

> *Security engineering is about building systems to remain dependable in the face of malice, error, or mischance. As a discipline, it focuses on the tools, processes, and methods needed to design, implement, and test complete systems, and to adapt existing systems as their environment evolves….*
>
> *Security engineering requires cross-disciplinary expertise, ranging from cryptography and computer security, to hardware*

> *tamper-resistance and formal methods, to applied psychology, organizational and audit methods and the law…*
>
> *Security requirements differ greatly from one system to another. One typically needs some combination of user authentication, transaction integrity and accountability, fault-tolerance, message secrecy, and covertness. But many systems fail because their designers protect the wrong things, or protect the right things but in the wrong way.*

To create a context for discussing the development of secure software, Section 4.1 describes the technical processes associated with the development of secure *systems*, and the relationship between the technical aspects of secure systems engineering and those of software development as it occurs within secure systems engineering.

The remaining subsections of Section 4 discuss the technical systems engineering activities involved in building secure systems:

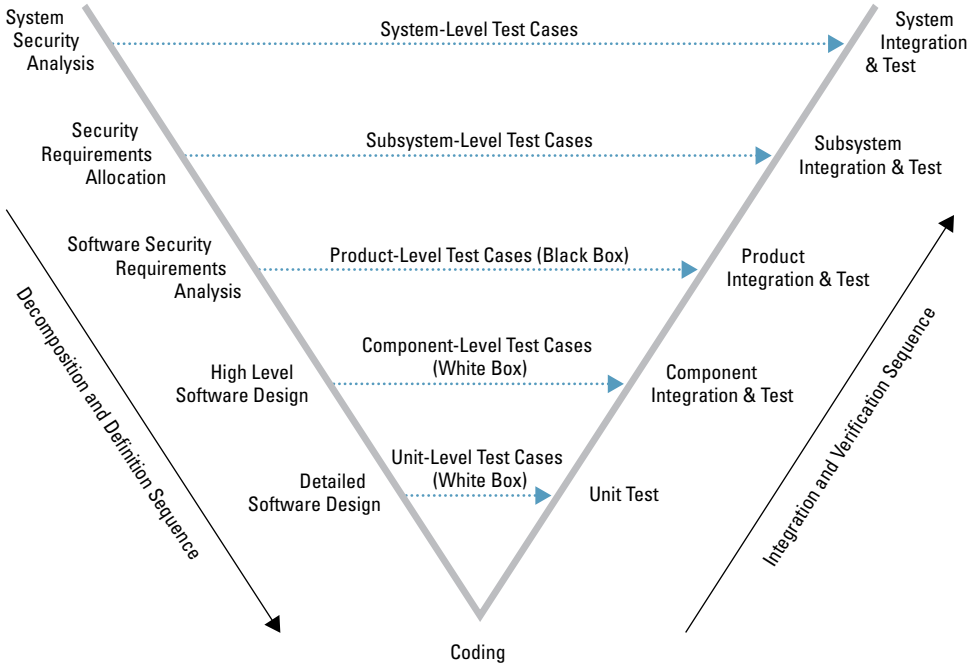
- Section 4.2 addresses the development of systems requirements.
- Section 4.3 discusses the development of secure systems designs.
- Section 4.4 addresses the integration of subsystem components.
- Section 4.5 discusses testing of systems and subsystems.

To continually improve on the process for developing secure systems, one must follow a rigorous and secure systems development process, which is described in Section 4.6 as the Secure Systems Engineering–Capability Maturity Model (SSE-CMM).

4.1 The Relationship Between Systems Engineering Processes and Software Engineering Processes

Many process models are available to the systems engineer for developing the system, such as the waterfall and spiral models. Given that a risk-centric approach is needed for security engineering, a spiral systems development model, which is a risk-driven approach for development of products or systems, is ideal. However, as a mean for understanding the relationships of systems to software engineering and testing and validation, one of the most illustrative systems process models is the "Vee" process model conceived by Forsberg and Mooz, [69] who assert that it depicts the "technical aspect of the project cycle." Figure 4-1 is derived from this Vee model to demonstrate the relationship between secure systems engineering and secure software engineering.

Figure 4-1. "Vee" Process Model Applied to Secure Systems Engineering



In the Vee model, the left downward side depicts the major technical phases (reading from left to right) involved in transforming systems requirements into a functioning software system. The right upward side depicts the technical phases of testing, integration, and validation associated with each phase on the left side. The dotted red arrows show how artifacts of the decomposition and definition phases (*e.g.,* system security requirements analyses, system security design analyses) provide input to the test cases used in the integration and verification phases.

The system security analysis phase represents the system requirements analysis and system design phases of development. System requirements analysis is described elsewhere in this section. Relative to system design, systems engineers are responsible for design of a total systems solution. As discussed in other subsections, to design the system for security, [70] many hardware and software design options are available. Risk-aware tradeoffs need to be performed to achieve the right secure system solutions. Many systems design are at a broad, hierarchical block level.

During the security requirements allocation phase, systems engineers allocate system requirements to hardware, software, *etc.* Concerning software considerations during system design, a recent report by the National Defense Industrial Association (NDIA), *Top Software Engineering Issues within the Department of Defense and Defense Industry* [71], observes that in many organizations, system engineering decisions are made without full participation

of software engineering. This may result in systems solutions, from a security perspective, that do not adequately address software faults that cause security vulnerabilities in the resulting system design.

The requirements allocated to software are then handed-off to software engineering (and requirements allocated to hardware are handed-off to hardware engineering) for further software requirements analysis, design, *etc.* These SDLC activities, as they pertain to the production of secure software, are discussed in Section 5. In the Vee diagram, this represents the phases from software security requirements analysis, down to coding and then up to product integration and test.

Systems engineering then receives finished components from software and, if applicable, hardware. System engineering is responsible for necessary subsystem and system integration and test and finally acceptance testing. These activities are described elsewhere in this section.

4.2 Developing Systems Security Requirements

This section addresses software security requirements engineering. The methodologies described here are also applicable at the system level.

Users may not be totally aware of the security risks, risks to the mission, and vulnerabilities associated with their system. To define requirements, systems engineers may, in conjunction with users, perform a top-down and bottom-up analysis of possible security failures that could cause risk to the organization as well as define requirements to address vulnerabilities.

Fault tree analysis for security (sometimes referred to as threat tree or attack tree analysis) is a top-down approach to identifying vulnerabilities. In a fault tree, the attacker's goal is placed at the top of the tree. Then, the analyst documents possible alternatives for achieving that attacker goal. For each alternative, the analyst may recursively add precursor alternatives for achieving the subgoals that compose the main attacker goal. This process is repeated for each attacker goal. By examining the lowest level nodes of the resulting attack tree, the analyst can then identify all possible techniques for violating the system's security; preventions for these techniques could then be specified as security requirements for the system.

Failure Modes and Effects Analysis (FMEA) is a bottom-up approach for analyzing possible security failures. The consequences of a simultaneous failure of all existing or planned security protection mechanisms are documented, and the impact of each failure on the system's mission and stakeholders is traced.

Other techniques for developing system security requirements include threat modeling and misuse and abuse cases. Both of these techniques are described in Section 5.2.3.1. Requirements may also be derived from system security policy models and system security targets that describe the system's required protection mechanisms [*e.g.,* the Target of Evaluation (TOE) descriptions produced for Common Criteria (CC) evaluations].

Attack tree analyses and FMEAs augment and complement the security requirements derived from the system's threat models, security policy models, and/or security targets. The results of the system security requirements analysis can be used as the basis for security test case scenarios to be used during integration or acceptance testing.

4.3 Secure Systems Design

In the design of secure systems, several key design features must be incorporated to address typical system vulnerabilities: security protocol design, password management design, access control, addressing distributed system issues, concurrency control, fault tolerance, and failure recovery. Appendix E describes security functions that are typically incorporated in secure systems. This is not meant to be an exhaustive list, but rather to provide illustrative examples. The following sections discuss two significant design issues with security implications, which are not directly related to security functionality.

4.3.1 Timing and Concurrency Issues in Distributed Systems

As noted by Anderson, in large distributed systems (*i.e.,* systems of systems), scale-up problems related to security are not linear because there may be a large change in complexity. A systems engineer may not have total control or awareness over all systems that make up a distributed system. This is particularly true when dealing with concurrency, fault tolerance, and recovery. Problems in these areas are magnified when dealing with large distributed systems.

Controlling the concurrency of processes (whereby two or more processes execute simultaneously) presents a security issue in the form of potential for denial of service by an attacker who intentionally exploits the system's concurrency problems to interfere with or lock up processes that run on behalf of other principals. Concurrency design issues may exist at any level of the system, from hardware to application. Some examples of and best practices for dealing with specific concurrency problems, includes—

- **Processes Using Old Data** (*e.g.,* out of date credentials, cookies): Propagating security state changes is a way to address this problem.
- **Con icting Resource Updates**: Locking to prevent inconsistent updates (resulting from two programs simultaneously updating the same resource) is a way to address this.
- **Order of Update in Transaction-Oriented Systems and Databases**: Order of arrival and update needs to be considered in transaction-oriented system designs.
- **System Deadlock**, in which concurrent processes or systems are waiting for each other to act (often one process is waiting for another to release resources): This is a complex issue, especially in dealing with lock hierarchies across multiple systems. However, note that there are four necessary conditions, known as the Coffman conditions (first identified by

E.G. Coffman in 1971)[72] that must be present for a deadlock to occur—mutual exclusion, hold and wait, no preemption, and circular wait.

- **Nonconvergence in Transaction-Oriented Systems**: Transaction-based systems rely on the ACID (atomic, consistent, isolated, and durable) properties of transactions (*e.g.,* the accounting books must balance). Convergence is a state in transaction systems; when the volume of transactions subsides, there will be a consistent state in the system. In practice, when nonconvergence is observed, recovery from failures must be addressed by the systems design.
- **Inconsistent or Inaccurate Time Across the System**: Clock synchronization protocols, such as the Network Time Protocol or Lamport's logical locks, can be run to address this issue.

The above list is merely illustrative. A number of other concurrency issues can arise in software-intensive systems.

4.3.2 Fault Tolerance and Failure Recovery

In spite of all efforts to secure a system, failures may occur because of physical disasters or from security failures. Achieving system resilience through failure recovery and fault tolerance is an important part of a system engineer's job, especially as it relates to recovery from malicious attacks. Fault tolerance and failure recovery make denial of service attacks more difficult and thus less attractive.

As noted by B. Selic [73], dealing with faults involves error detection, damage confinement, error recovery, and fault treatment. Error detection detects that something in the system has failed. Damage confinement isolates the failure. Error recovery removes the effects of the error by restoring the system to a valid state. Fault treatment involves identifying and removing the root cause of the defect.

Failure models of the types of attacks that can be anticipated need to be developed by the systems engineer. Resilience can then be achieved through fail-stop processors and redundancy to protect the integrity of the data on a system and constrain the failure rates.

A fail-stop processor automatically halts in response to any internal failure and before the effects of that failure become visible. [74]

The systems engineer typically applies a combination of the following to achieve redundancy at multiple levels.

- Redundancy at the hardware level, through multiple processors, mirrored disks, multiple server farms, or redundant arrays of independent disks (RAID).
- At the next level up, process redundancy allows software to be run simultaneously on multiple geographically distributed locations, with voting on results. It can prevent attacks where the attacker gets physical control of a machine, inserts unauthorized software, or alters data.

- At the next level is systems backup to unalterable media at regular intervals. For transaction-based systems, transaction journaling can also be performed.
- At the application level, the fallback system is typically a less capable system that can be used if the main system is compromised or unavailable.

Note that while redundancy can improve the speed of recovery from a security incident, none of the techniques described above provide protection against attack or malicious code insertion.

For Further Reading

"IEEE Computer Society Technical Committee on Dependable Computing and Fault-Tolerance" and "IFIP WG 10.4 on Dependable Computing and Fault Tolerance" [portal page]
Available from: http://www.dependability.org

Christ Inacio, (CMU SEI), *Software Fault Tolerance*, (Pittsburgh, PA: CMU SEI, Spring, 1998).
Available from: http://www.ece.cmu.edu/~koopman/des_s99/sw_fault_tolerance/

CMU SEI, *A Conceptual Framework for System Fault Tolerance,* (Gaithersburg, MD: NIST, March 30,1995).
Available from: http://hissa.nist.gov/chissa/SEI_Framework/framework_1.html

4.4 System Integration

To be effectively addressed during the integration phase, system security issues must first be identified during the requirements and design phases. In today's large distributed systems, system components typically interface with outside systems whose security characteristics are uncertain or questionable. The security of an integrated system is built on the behavior and interactions of its components and subsystems. On the basis of a risk analysis of systems components, systems engineers must build in necessary protection mechanisms

As noted in DHS's draft *Security in the Software Lifecycle*:

> *Determining whether the system that contains a given component, module, or program is secure requires an analysis of how that component/module/program is used in the system, and how the system as a whole will mitigate the impact of any compromise of its individual components that may arise from a successful attack on those components or on the interfaces between them. Risks of insecurity can be reduced through:*
>
> 1. *Vetting all acquired, reused, and from-scratch components prior to acceptance and integration into the whole system;*
>
> 2. *Examining interfaces, observation of instances of trust relationships, and implementing wrappers when needed;*
>
> 3. *Security testing of the system as a whole.*

Certain systems design architectures and frameworks (*e.g.,* application frameworks, publish/subscribe architectures) can minimize the likelihood of security problems being introduced through improper integration of application components.

Issues associated with the use of nondevelopmental [*e.g.,* commercial-off-the-shelf (COTS), open source software (OSS), legacy] components are discussed in Sections 3.2.1 and 5.1.1.2. The same issues apply when selecting and integrating components at the whole system level, rather than specifically at the software level.

4.5 System Testing

In *A Practical Guide to Security Engineering and Information Assurance*, [75] Debra Herrmann recommends that because attackers are not biased by knowledge of a systems design or security protection mechanisms, testing of the integrated system by the system's engineers be augmented by independent testing by a disinterested third party.

Tests to discover design defects are difficult to develop. Like the systems engineers developing security designs, the testing group (whether independent or not), will be able to construct test cases based on understanding the psychology of the attackers and knowledge of typical software, hardware, and other system fault types. Additional sources of information for development of test cases and scripts include—

- Misuse and abuse cases
- Threat tree analysis reports
- Threat models
- FMEA reports
- Security policy models
- Security targets
- System security requirements.

At a minimum, testing the resiliency of a system design to attack would include—

- Testing for transient faults, such as an unusual combination or sequence of events, degradation of the operating environment (temporary saturation of the network, power losses, environmental changes), or induced temporary loss of synchronization among components of a system
- Testing for the ability of the system to withstand password guessing, masquerading, *etc.*
- Creative "what if" testing.

Section 4.5 describes a number of security testing techniques that can be applied to software. Some of these techniques are also useful, and in the case of penetration testing, best performed at the system level.

4.6 SSE-CMM

Note: This section will be of the most use to readers already familiar with the Systems Engineering (SE) CMM.

The SSE-CMM process reference model augments project and organizational process areas from the SE CMM with security engineering process areas for improving and assessing the maturity of the security engineering processes used to produce information security products, trusted systems, and security capabilities in information systems. The scope of the processes addressed by the SSE-CMM encompasses all activities of the system security engineering life cycle, including concept definition, requirements analysis, design, development, integration, installation, operation, maintenance, and decommissioning. The SSE-CMM includes requirements for product developers, secure systems developers and integrators, and organizations that provide computer security services and/or computer security engineering, including organizations in the commercial, government, and academic realms.

The SSE-CMM is predicated on the view that security is pervasive across all engineering disciplines (*e.g.,* systems, software, and hardware), and the Common Feature *coordinate security practices* has been defined to address the integration of security with all disciplines and groups involved on a project or within an organization (see Table 4-1). Similarly, the Process Area (PA) *coordinate security* defines the objectives and mechanisms to be used in coordinating the security engineering activities with all other engineering activities and teams.

Table 4-1. SSE-CMM Security Engineering Process Areas and Goals

Security Engineering PA	PA Goals
Administer security controls	Ensure that security controls are properly configured and used.
Assess impact	Reach an understanding of the security risk associated with operating the system within a defined environment.
Assess security risk	Identify system vulnerabilities and determine their potential for exploitation.
Assess threat	Reach an understanding of threats to the security of the system.
Assess vulnerability	Reach an understanding of the system's security vulnerabilities.
Build assurance argument	Ensure that the work artifacts and processes clearly provide the evidence that the customer's security needs have been met.
Coordinate security	Ensure that all members of the project team are aware of and involved with security engineering activities to the extent necessary to perform their functions; coordinate and communicate all decisions and recommendations related to security.

Table 4-1. SSE-CMM Security Engineering Process Areas and Goals - *continued*

Security Engineering PA	PA Goals
Monitor security posture	Detect and track internal and external security-related events; respond to incidents in accordance with policy; identify and handle changes to the operational security posture in accordance with security objectives.
Provide security input	Review all system issues for security implications and resolve those issues in accordance with security goals; ensure that all members of the project team understand security so they can perform their functions; ensure that the solution reflects the provided security input.
Specify security needs	All applicable parties, including the customer, reach a common understanding of security needs.
Verify and validate security	Ensure that the solutions satisfy all of their security requirements and meet the customer's operational security needs.

The SSE-CMM and the method for applying the model (*i.e.*, the appraisal method) are intended to be used as a—

- Tool that enables engineering organizations to evaluate their security engineering practices and define improvements to them
- Method by which security engineering evaluation organizations, such as certifiers and evaluators, can establish confidence in the organizational capability as one input to system or product security assurance
- Standard mechanism for customers to evaluate a provider's security engineering capability.

As long as the users of the SSE-CMM model and appraisal methods thoroughly understand their proper application and inherent limitations, the appraisal techniques can be used in applying the model for self-improvement and in selecting suppliers.

An alternative approach to a secure CMM is described in the Federal Aviation Administration (FAA)/Department of Defense (DoD) *Proposed Safety and Security Extensions to iCMM and CMMI* (see Appendix D).

For Further Reading

Mary Schanken, Charles G. Menk III, James P. Craft, (NSA and United States Agency for International Development), *US Government Use of the Systems Security Engineering Capability Maturity Model*, (SSE-CMM), (presentation at the National Information Systems Security Conference, October 19, 1999). Available from: http://csrc.nist.gov/nissc/1999/program/act10.htm

4.7 System Security C&A and Software Assurance

Certification and accreditation (C&A) processes for government information systems are intended to ensure that before a deployed system becomes operational, the system includes security controls and countermeasures that adequately mitigate identified risks. For a C&A to be truly effective, however, activities to prepare for that C&A should begin early in the system development cycle.

Unfortunately, the Federal Information Security Management Act (FISMA) and the superseded-but-still-used Defense Information Technology Security Certification and Accreditation Process (DITSCAP) initiate C&A during the system's integration testing phase. Moreover, the Security Test and Evaluation (ST&E) activity within the C&A process is primarily concerned with determining the system's level of compliance with management, operational, and technical controls, and involves very little testing of the system's technical security controls. The depth of analysis, particularly of the system's software components, is minimal.

By contrast with FISMA and DITSCAP, DoD Information Assurance Certification and Accreditation Process (DIACAP), Director, Central Intelligence Directive (DCID) 6/3; and National Institute of Standards and Technology (NIST), Special Publication (SP) 800-64, *Security Considerations in the Information System Development Life Cycle* (which maps FISMA-driven C&A activities described in NIST SP 800-37, *Guide for the Security Certification and Accreditation of Federal Information Systems*, to each phase of the system development life cycle) mandate activities that must occur early in the life cycle. For example, DCID 6/3, requires three key activities at the beginning of the system development life cycle—

- Establish protection levels for system components
- Document system security requirements, threats, and countermeasures
- Develop a system security plan and test procedures.

Even these early life cycle activities are limited to what are, in effect, documentation exercises. Reviews and evaluations still begin late in the system life cycle; none of the C&A methodologies cited here include security reviews of the system design or code reviews during system implementation. The code reviews that do occur are solely at the Designated Approving Authority's (DAA) discretion, and performed during the post-integration ST&E phase.

The documentation-oriented approach to C&A tends to influence the approach to system architecture. For example, the *DoD Architecture Framework* (DoDAF) specifies different architectural "views" for capturing different required properties of the system, including security. This may make sense from the point of view of a security certifier, who is only interested in the system's security; but it is not effective for specifying an overall system architecture that can form the basis for a cohesive design with security deeply embedded. Unfortunately, the cost of documentation is already seen as too high, so there

is often little incentive for system development teams to then integrate their separate security architecture views into the overall system architecture view.

Because it is system oriented, C&A reflects one of the weaknesses of system engineering approaches to the software security challenge. Because individual system components are seen as "black boxes," activities within the software development life cycle are often treated as "black boxes" as well, with regard to how they map to the overall system life cycle. In DoD in particular, the problem is compounded by the frequent use of the *Life Cycle Framework* view described in the *Defense Acquisition Guidebook* as the basis for structuring the activities and artifacts of the actual system development process. The problem with this is the fact that this framework view was intended as an abstraction of system development that would be meaningful for acquisition managers, not as a practical methodology for structuring system or software development projects.

During C&A, the holistic view of the system and the consideration of individual components as black boxes mean that the certifier focuses mainly on each component's external interactions with its host platform, connected network, and users, and to a lesser extent with the other black boxes that compose the system. The behaviors and security deficiencies within each black box component are not examined. Nor do system C&A artifacts and reviews provide sufficiently detailed information about the individual components to enable the certifier to trace system security vulnerabilities to the faults and weaknesses of individual software components. [76]

Finally, lack of component-level detail in system C&A documentation renders such documentation inadequate as a template after which software security assurance cases might be modeled.

4.8 CC Evaluations and Software Assurance

In the United States, a significant portion of the software security assurance community (including its initiatives, research, and tools) originated not in the software safety/reliability arena, but in the information assurance/cyber security arena. In the United States, CC artifacts are frequently suggested for use as a basis for defining software assurance cases.

Information system security assurance cases for certain types of information systems components were defined even earlier than safety cases. In pursuit of Trusted Computer System Evaluation Criteria (TCSEC) or CC evaluations or Federal Information Processing Standard (FIPS) 140-1 or 140-2 certifications for their security-enforcing IT products, vendors are required not only to submit assurance claims for those products to the independent evaluation or certification facility but to provide complete assurance cases that provide a sufficient basis for the facility to verify those assurance claims.

The inadequacies of the TCSEC have been perpetuated in the CC, in that the CC does not provide a meaningful basis for documentation of assurance cases that can be used to verify security as a property of software (*vs.* the

correctness of the security functionality provided by system components). Also perpetuated in the CC are the inadequacies of the evaluation processes [*e.g.,* the National Information Assurance Partnership (NIAP) CC Evaluation Program] with regard to their omission of direct vulnerability testing of the software components of systems under evaluation. Moreover, the vulnerability analyses that are done focus solely on system-level vulnerabilities in security functions of the system, rather than on the types of software vulnerabilities that may be exploited to compromise the system's overall dependendability. Fortunately, these (and other) inadequacies of the CC and its associated evaluation schemes have been widely acknowledged, not least by The National Strategy to Secure Cyberspace which, among its actions and recommendations, stated:

> *…the Federal Government will be conducting a comprehensive review of the National Information Assurance Partnership (NIAP), to determine the extent to which it is adequately addressing the continuing problem of security flaws in commercial software products.*

In 2005, the DHS CS&C NCSD contracted the Institute for Defense Analyses (IDA) to undertake a major review of the NIAP, [77] and to recommend improvements that would make NIAP's CC evaluations more timely, effective, and relevant.

The most significant issues are the lack of criteria pertaining to the characteristics of software that are considered essential to its security, *i.e.,* lack of exploitable vulnerabilities and weaknesses and ability to continue operating dependably when under attack. The current CC and its evaluation process take a system level, rather than software component level view. Moreover, CC evaluation assurance levels (EAL) indicate only the degree of confidence that can be achieved concerning the claims the product's vendor makes in the security target (ST) document about the conformance of the product's (or target of evaluation's) security-enforcing and security-relevant functionality with security policy rules defined for the TOE. STs say nothing about the robustness against attacks of, and lack of vulnerabilities in, the software that implements those security functions. With this as its objective, it is not surprising that the CC evaluation does not include analysis or testing of the TOE for implementation faults that may lead to vulnerabilities.

Unlike safety cases, ST documents (which are, in essence, security cases) are not intended to demonstrate the TOE's own inherent security robustness, but only its conformance to security policy. For this reason, CC STs are considered by those who are developing standards for software security assurance cases (see Section 5.1.4) to provide only a limited amount of meaningful security evidence for supporting security claims made in the context of software security assurance cases.

References

69 Kevin Forsberg and Harold Mooz, "The Relationship of System Engineering to the Project Cycle," in *Proceedings of the First Annual Symposium of National Council on System Engineering*, October 1991: 57–65.

70 Besides security, systems engineers also need to balance the designs to achieve needed reliability, maintainability, usability, supportability, producibility, affordability, disposability, and other "-ilities."

71 National Defense Industrial Association (NDIA) Systems Engineering Division, *Top Software Engineering Issues Within Department of Defense and Defense Industry*, draft vers. 5a. (NDIA, September 26, 2006).
Available from: http://www.ndia.org/Content/ContentGroups/Divisions1/Systems_Engineering/PDFs18/NDIA_Top_SW_Issues_2006_Report_v5a_final.pdf

72 E.G. Coffman, M.J. Elphick, and A. Shoshani, "System Deadlocks," *ACM Computing Surveys* 3, no.2 (1971): 67–78.
Available from: http://www.cs.umass.edu/~mcorner/courses/691J/papers/TS/coffman_deadlocks/coffman_deadlocks.pdf

73 Brian Selic, "Fault Tolerance Techniques for Distributed Systems," *IBM DeveloperWorks* (July 27, 2004).
Available from: http://www-128.ibm.com/developerworks/rational/library/114.html

74 Elisabeth A. Strunk, John C. Knight, and M. Anthony Aiello, "Assured Reconfiguration of Fail-Stop Systems," in *Proceedings of the International Conference on Dependable Systems and Networks*, 2005.
Available from: http://www.cs.virginia.edu/papers/01467774.pdf

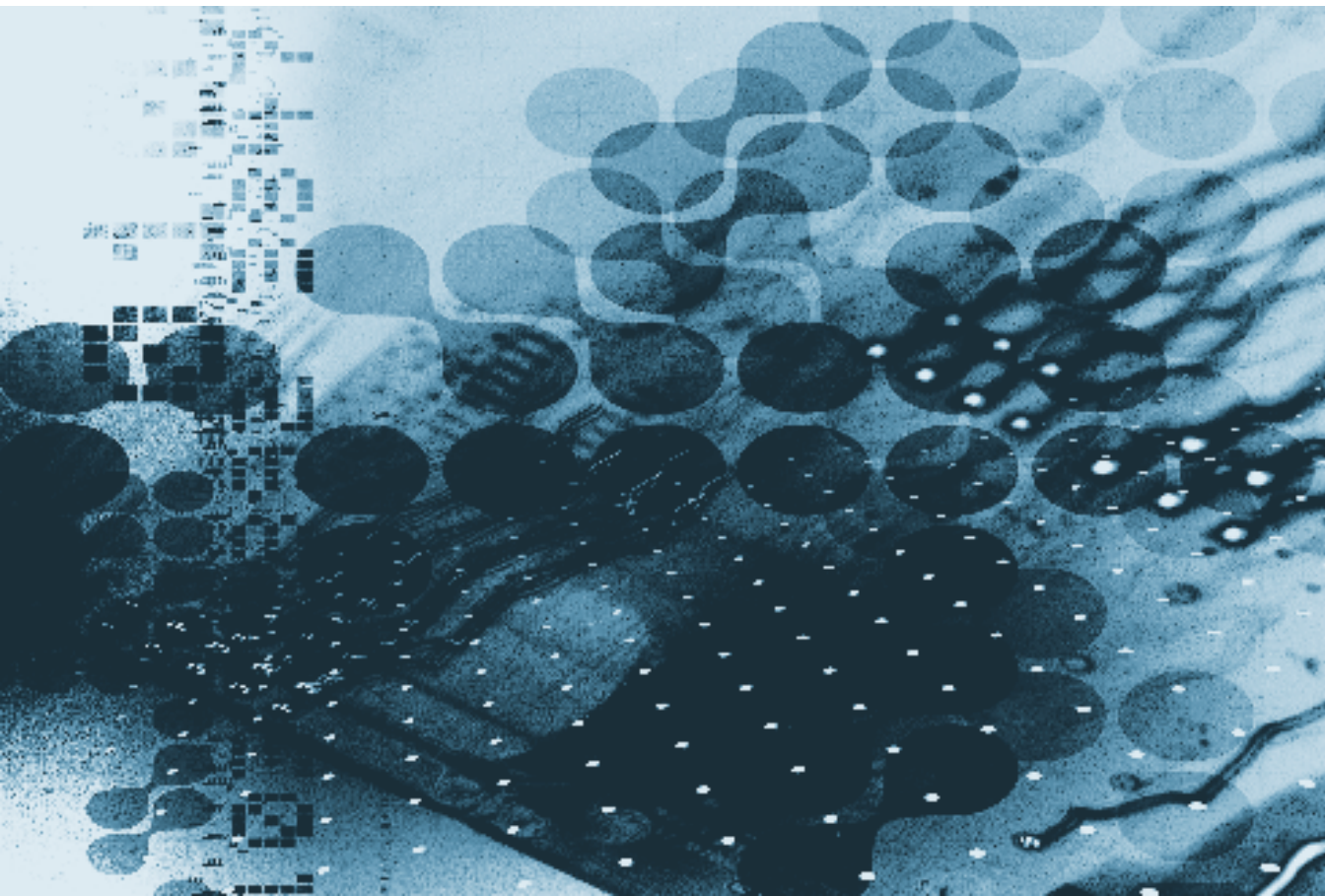
75 Debra Herrmann, A Practical Guide to Security Engineering and Information Assurance, (Auerbach Publications, 2002)

76 Thus the accreditor is forced to accept risks without fully understanding why they exist, why many conventional system-level countermeasures do not adequately mitigate those risks, or what alternative countermeasures might be possible. Again, the "too late in the life cycle" problem means that even if the accreditor does understand that the only effective mitigation for a particular risk may entail reengineering a software component, the cost of implementing such a change when the risk is discovered so late in the system life cycle is usually prohibitive. For this reason, it would behoove accreditors to become better informed about the threats, vulnerabilities, and mitigations that are unique to the software components of the systems they evaluate, and to mandate inclusion of secure development practices and security reviews starting in the earliest stages of the life cycle so that he/she can put pressure on the system developer and increase the likelihood that through security engineering, the system, by the time it enters formal C&A evaluation, will be inherently more secure, with fewer residual risks for the accreditor to accept and manage.

77 Edward Schneider and William. A Simpson, "Comprehensive Review of the National Information Assurance Partnership" (paper presented at the Annual Computer Security Applications Conference, 2005).

5

SDLC Processes and Methods and the Security of Software



Integrating security activities into software engineering life cycle processes involves adding security practices (*e.g.*, penetration testing) and principles (*e.g.,* a design that enforces least privilege) to the activities and practices (such as requirements engineering, modeling, and model-based testing) in each phase of the traditional software development life cycle (SDLC).

As noted by Mouratidis and Giorgini, [78] in spite of active research, no software engineering methodology exists to ensure that security exists in the development of large-scale software systems. No matter what life cycle model is followed, current research indicates that security should be considered from the early stages of the software life cycle. Section 5.1 discusses security considerations that affect the whole SDLC (or at least multiple SDLC phases), including the SDLC methodology used and how risk management, security measurement, quality assurance, and configuration management are performed.

Security requirements should be considered simultaneously with other requirements, including those pertaining to functionality, performance, usability, *etc.* As Charles Haley *et al*. [79] have observed, "Security requirements often con ict with each other, as well as with other requirements," and indeed their research, along with that of several others, focuses on the combination (or composition) of a software system's security requirements with its functional as well as other nonfunctional requirements in an effort to minimize such con icts in the resulting requirements specification. This research is discussed in Section 5.2.

Many security weaknesses in software-intensive systems arise from inadequate architectures and poor design choices. Section 5.3 discusses the concepts, techniques, and tools in use for developing and verifying secure software architectures and designs.

Security concerns also need to be addressed as part of the software team's and project's choice of programming languages, coding practices, and implementation tools. Secure implementation issues, techniques, and tools are discussed in Section 5.3, followed by a discussion of software security assessment and testing techniques and tools in Section 5.4.

5.1 Whole Life Cycle Security Concerns

Unlike the remainder of Section 5, Section 5.1 focuses on software security concerns, influences, and activities that span the whole SDLC or multiple phases of the SDLC. Specifically, this section discusses—

- Security implications of the various ways in which software enters the enterprise
- Security benefits and limitations of formal methods
- Security concerns associated with agile methods
- Risk management for secure software
- Software security assurance cases, metrics, and measurement
- Secure software configuration management
- Quality assurance for secure software
- Security-enhanced SDLC methodologies.

5.1.1 Software Security and How Software Enters the Enterprise

Software-intensive systems come into existence in four different ways:

- **Acquisition**—Acquisition refers to purchase, licensing, or leasing of nondevelopmental software packages and components [80] produced by entities other than the acquirer. Such nondevelopmental software may be used "as is," or may be integrated or reengineered by the acquirer (or by a third party under contract to the acquirer). Nondevelopmental software includes shrink-wrapped commercial off-the-shelf (COTS), government off-the-shelf (GOTS), and MOTS (modified off-the-shelf) software packages, and open source software (OSS), shareware, and freeware components. For purposes of this SOAR, obtaining and reusing legacy components is also considered "acquisition" of non developmental software, even though it does not involve acquisition as defined by the Federal Acquisition Regulation (FAR) and Defense FAR Supplement (DFARS).
- **Integration or Assembly**—If software items must be combined to achieve the desired functionality of the system, that system comes into existence through integration or through assembly. The software items to be combined may be nondevelopmental or customized, or a combination

of the two. In some cases, integration may entail custom development of code to implement interfaces between software items. In other cases, nondevelopmental items may need to be modified or extended through reengineering (see below). If the software items to be combined are *components* (*i.e.,* self-contained with standard interfaces), the integration process is referred to as *component assembly*.

- **Custom Development**—Custom-developed software is purpose-built for the specific system in which it will be used. It is written by the same organization that will use it or by another organization under contractor to the user organization. Very few large information systems are completely custom developed; most are the result of integration and include at least some nondevelopmental components.
- **Software Reengineering**—Existing software is modified so that one or more of its components can be modified/extended, replaced, or eliminated.

Each of the above approaches to software conception and implementation has its own advantages and disadvantages with regard to the software's cost, support, and technical effectiveness—which are usually the driving factors organizations use when deciding which approach to take. Most development organizations do not consider security assurance to be a driving factor; security is often seen as something to be considered only as the application is being prepared for deployment or for approval to operate. Too often, software security assurance is not even considered until a security incident occurs that can be directly associated with a vulnerability in the software.

5.1.1.1 Security Issues Associated With Acquired Nondevelopmental Software

Many organizations increasingly turn to turnkey systems and applications rather than custom-developing software for a particular organizational need. Purchasing turnkey software is often cheaper and involves less business risk than developing software from scratch. When purchasing COTS software, "a large portion of the design and test work has already been done by the vendors, and that cost is spread among all those who license the software" [81] rather than a single organization. Additionally, COTS software is the result of a successful software development project; in 2003, research by the Standish Group found that 34 percent of corporate software projects were successful, [82] which indicates it is very likely that custom-built software often exceeds budget, and the development project not meet its deadline. With COTS software, the cost is clearer.

Nevertheless, the properties and capabilities delivered in acquired software do not always map directly to the requirements of the organization acquiring that software, particularly with regard to security requirements. Many commercial developers have admitted that security is not considered a major requirement because of the current practices in acquisition to accept software

that satisfies functionality with little regard for achieving and assuring security properties. Few organizations ask "how do you know the product is secure?" so vendors do not perceive a demand for secure products. According to Jeremy Epstein of webMethods—

> *Despite the failure of users to ask, vendors are actually quite willing, able, and eager in many cases to provide and demonstrate the improved security in their products, the problem is that they are not offered incentives to do so, because purchasers have not expressly stated requirements for security. In other words, there is adequate supply. The problem is that there is insufficient demand, at least as expressed in buying decisions.* [83]

Efforts inside and outside the Federal Government are underway to evolve current acquisition practices to include security concerns. While purchasing secure software requires more upfront costs than traditional COTS software, organizations are becoming more aware of the additional costs associated with insecure software: according to Mark Graff and Ken van Wyk, [84] the cost of lost time and resources to maintain a vulnerable software component can be as much as three times the initial purchase of secure software. In response, many organizations are working to develop the necessary language, regulations, policies, and tools to improve the security of acquired software.

In response to the increasing costs of patching vulnerable software—and the dwindling "mean time between security breaches" of unpatched systems connected to the Internet—the US Air Force contracted with Microsoft to supply securely configured software. In June 2005, the Air Force began testing the securely configured software that would be distributed service-wide. [85] By requiring Microsoft to supply securely configured software, the Air Force could potentially save $100 million by streamlining the patching process.

On September 30, 2005, FAR 7.105(b)(17) was modified to include the following language: "For information technology acquisitions, discuss how agency information security requirements will be met." [86] This change was seen by many as a step in the right direction. The security discussions now mandated by the FAR can include software assurance strategies based on initial software assurance risk assessment and software assurance requirements.

On October 3, 2005, the Department of Homeland Security (DHS), Cyber Security and Communications (CS&C), National Cyber Security Division, (NCSD) and the Department of Defense (DoD) Office of the Chief Information Officer for Information Management and Technology began jointly sponsoring the software acquisition working group (WG), which is addressing how to leverage the procurement process to ensure the safety, security, reliability and dependability of software. This WG is developing a document entitled *Software Assurance (SwA) in Acquisition* [87] (see Section 6.1.9).

Nongovernmental efforts to improve the security of acquired software are gaining steam as well. In 2006, the Open Web Application Security Project (OWASP) Legal Project began supporting the *OWASP Secure Software Contract Annex*, which helps "software developers and their clients negotiate and capture important contractual terms and conditions related to the security of the software to be developed or delivered." [88] The Contract Annex provides a sample contract that defines the life cycle activities, security requirements, development environment, assurance, and other aspects of software acquisition and development to result in a more secure COTS product.

An important aspect for acquiring software is certification and testing. Several efforts are aimed at improving the certification and testing environment for software:

- International Organization for Standardization/International Electrotechnical Commission (ISO/IEC) 15408, *The Common Criteria*, provides a framework for evaluating how well software has met a set of security requirements.
- The National Institute of Standards and Technology (NIST) Cryptographic Module Verification Program certifies products against Federal Information Processing Standard (FIPS) 140-1 and 140-2 requirements for cryptographic software.
- Federal, DoD, and Intelligence Community certification and accreditation (C&A) processes, ensure that information systems conform to mandated security policies [*e.g.*, DoD Directive (DoDD) 8500.1, Federal Information Security Management Act (FISMA), Director of Central Intelligence Directive (DCID) 6/3].

While these certifications do not focus on software security, organizations are beginning to include software assurance requirements in the assessment process used by these and other certification processes.

5.1.1.2 Security Issues Associated With Component-Based Software Engineering

For many organizations, turnkey software applications do not provide the necessary functionality or flexibility to support their mission. Under pressure to produce systems more quickly using state-of-the-art software products and technologies, software engineers are forced to use third-party components about whose underlying security properties they have little or no knowledge. Software engineers generally lack confidence in COTS (and to a lesser extent open source) components because they cannot assess the compatibility between the components' security properties and the security requirements of their own applications.

In component-based software engineering, a software-intensive system is composed of several stand-alone software components acquired from

COTS or open source suppliers. In a component-based system, the developer needs to achieve both the security compatibility between pairs of interacting components and the security objectives of the entire system.

Software components range from individual procedure and object libraries to turnkey applications that may be composed of other, smaller components. Regardless, as they are considered for assembly into a new component-based system, each component's security properties need to be defined to be sufficient and meaningful to the other components to which those properties will be exposed, *i.e.,* the other components with which the component will interact.

Over its life time, the same component may play a variety of roles in a variety of systems running in a variety of environments. The security properties exhibited by the component will seldom satisfy the security requirements for all these possible combinations; the component may be proved secure in one application in a particular operating environment, but insecure when used in a different application and/or environment. For example, a component considered reasonably secure in an application for a car manufacturing plant may not be secure when used in an air traffic control system because although the component's functionality provided by the component remains same for both applications the use contexts and security requirements for the applications differ.

In a well-designed software application, it should be possible to isolate various components and identify those that need to be examined in depth. For example, a component used to perform authentication of users, such as a UNIX pluggable authentication module (PAM), should have strictly defined inputs and outputs while interfacing with only a small portion of the application. In this case, in-depth security testing can be performed on the component itself in place of the entire login system used by the application. If a system is not designed to separate components, the entire system must be examined—which is often infeasible. By using a strictly defined interface, it is possible to generate a test component that will test how the application interfaces with the component, providing assurance that the integrated system will function as expected. However, complex component interfaces or multipurpose components may be necessary for a particular system, limiting the effectiveness of testing individual components or how the application interfaces with the component.

Similarly, components should communicate using open standards. According to David A. Wheeler of the Institute for Defense Analyses (IDA), open standards "create economic conditions necessary for creating secure components." [89] Similarly, Jerome Saltzer and Michael Schroeder identified the need for openness in 1973: "This decoupling of protection mechanisms from protection keys permits the mechanisms to be examined by many reviewers without concern that the review may itself compromise the safeguards." [90] Using open standards is beneficial for security for several reasons: multiple

components can be tested and compared using the same tests, and a vulnerable component may be replaced with another component easily. Finally, the ability to replace components can prove immeasurably useful when responding to vulnerabilities because it introduces diversity into the system.

For Further Reading

(DHS), *"BuildSecurityIn Portal", "Assembly, Integration, and Evolution",* (DHS).
Available from: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/assembly.html

Arlene F. Minkiewicz, "Security in a COTS-Based Software System", *CrossTalk,* (November).
Available from: http://www.stsc.hill.af.mil/crosstalk/2005/11/0511Minkiewicz.html

Davide Balzarotti, Mattia Monda, and Sabrina Sicari (Milan Polytechnic, University of Milan and University of Catania), "Assessing the Risk of Using Vulnerable Components", in *Springer, Quality of Protection: Security Measurements and Metrics, Advances in Information Security;* 2006.
Available from: http://homes.dico.unimi.it/~monga/lib/qop.pdf

5.1.1.2.1 Assuring the Security of Component-Based Systems

Even if the security of all of the system's individual components can be established, this will not be enough to predict whether their secure behavior will continue to be exhibited when they interact with other components in a larger component-based system. Nor will it help predict the overall security of the system assembled from those components. A security claim for a single component (*e.g.,* a CC ST) in a system assembled from multiple components is of little help in determining the assurance of that system.

Individual component security evaluations focus on the component in isolation (and CC evaluations, as noted in Section 4.7, focus solely on the correctness of the component's security functionality rather than its continued dependability in the face of threats). Examining a component in isolation cannot reveal the security conflicts that will arise as a result of interactions between components. Such conflicts, often referred to as security mismatches, usually originate in an inconsistency between the security assumptions one component has about another's security properties, functionality, policy rules, constraints, *etc.,* and those that the second component actually exhibits. The problem is complicated by the need to periodically add or change the functionality of individual components or the system as a whole, often necessitating changes to the assembly's design, as well as to the individual components.

Research into assuring systems assembled from components dates as far back as 1973 with the Stanford Research Institute (SRI) Computer Science Lab's (CSL) Provably Secure Operating System (PSOS). [91] The PSOS project demonstrated how a complex system of small modular components could be predictably composed and analyzed using formal specifications and examining dependencies. According to Peter Neumann, principal scientist at SRI's CSL and one of the researchers involved with PSOS, "One of the biggest problems (associated with trustworthy systems) is the composition problem—how do we combine components into systems that predictably enhance trustworthiness." [92]

Neumann and the CSL, through the Defense Advanced Research Projects Agency (DARPA) Composable High-Assurance Trustworthy Systems (CHATS) project, published guidelines for composing a secure system in December 2004. [93]

With some of the same objectives, the Carnegie Mellon University (CMU) Software Engineering Institute (SEI) sponsors the Predictable Assembly from Certifiable Components (PACC) project, which aims to develop methods for predicting the behavior of a component-based system prior to implementation. [94] In addition the US Naval Research Laboratory (NRL) sponsors the Center for High Assurance Computer Systems (CHACS), which is investigating techniques for developing highly-assured building blocks (*i.e.,* components) from which trustworthy systems can be assembled. [95]

Each of these projects is developing techniques for identifying the effects of individual components on the system as a whole. Most of this research is difficult; success stories, such as PSOS, tend to rely on highly assured custom-developed components rather than on commodity COTS components.

Assured component assembly is becoming an increasingly popular research topic for survivable systems engineering. Robert Ellison of the CMU CERT provides an overview of the composition problem with respect to systems engineering on the DHS BuildSecurityIn portal. The CMU CERT Survivable Systems Engineering team is sponsoring a number of research activities related to assured component assembly, in particular the automated component composition for developing reliable systems project, which aims to define methods for automatically calculating the composite behavior of components of a system.

Testing and certification are becoming an important factor in assessing the security of component assemblies—both to ensure the security of components and to ensure their interoperability.

The DoD Software Assurance Tiger Team (see Section 6.1) is developing a DoD *Software Assurance Concept of Operations (CONOPS)* explaining how to securely integrate low assurance COTS components into DoD systems to minimize the amount of high assurance software that needs be developed. This activity is in its early phases, as of December 2006, the 180 implementation planning phase was closing out, and the pilot phase was under way. The *Software Assurance CONOPS* will begin research into Engineering in Depth (EiD), which minimizes the number of critical components in a system and manages the risk inherent in using less assured products. Critical components will be supplied by assured suppliers, who will provide DoD with sufficient documentation to document how much risk—such as foreign control of suppliers—is inherent in the supply chain. DoD will leverage this information when awarding contracts for critical high assurance components. [96]

To complement the CONOPS, the National Defense Industrial Association (NDIA), under sponsorship of the DoD Software Assurance Program, is developing a System Assurance Guidebook that will provide guidance to NDIA

members and partners in government, industry, and academia. The guidance in the *Systems Engineering Guidebook* is intended to supplement:

- ISO/IEC 15288, *System Life Cycle Processes*
- *DoD Acquisition Guidebook*
- IEEE STD 1220, *Application and Management of the Systems Engineering Process*.

The *Guidebook* will describe the activities necessary to address concerns for maliciousness, reducing uncertainty and providing a basis for justified confidence in the resulting system. The *Guidebook* should be released in early fiscal year 2007 (FY07).

The SEI's website and the DHS BuildSecurityIn portal provide a wealth of information discussing component assembly. [97] The Assembly, Integration, and Evolution section of the DHS BuildSecurityIn portal provides insight into the security issues associated with assembly of component-based software systems.

For Further Reading

Arlene F. Minkiewicz, "Security in a COTS-Based Software System", *CrossTalk*, (November, 2005).
Available from: http://www.stsc.hill.af.mil/crosstalk/2005/11/0511Minkiewicz.html

Peter G. Neumann (SRI), *Principled Assuredly Trustworthy Composable Architectures* (December, 2004).
Available from: http://www.csl.sri.com/users/Neumann/chats4.html

Peter G. Neumann and Richard J. Feiertag, "PSOS Revisited", in Proceedings of the Annual Computer Security Applications Conference (ACSAC 2003), (December, 2003),
Available from: http://www.csl.sri.com/users/neumann/psos03.pdf

"NRL CHACS" [website].
Available from: http://chacs.nrl.navy.mil

Robert J. Ellison (CMU SEI), *Trustworthy Composition: The System is Not Always the Sum of Its Parts* (September 2003).
Available from: https://buildsecurityin.us-cert.gov/daisy/bsi/50.html?branch=1&language=1

5.1.1.2.2 Research Into Engineering of Secure Component-Based Software

Significant research has been done since the late 1990s to address the problems associated with achieving security in component-based software systems. This research has focused on the following challenges:

- How to expose component security properties in a way that is usable by other components
- How to reconcile one component's expectations of the security functions (and associated data outputs and formats) it needs from another component versus the security functionality (and associated data/formats) actually provided by the second component
- How to predict and measure the security properties and assurance levels of individual components and the impact on those properties and measurements

- How to predict and measure the security properties and assurance levels of a system assembled from components based on the security properties, measurements, and assurance levels of the system's individual components as they interact to achieve their required functionality
- How to engineer component-based systems in ways that minimize the exposure and impact of individual components' vulnerabilities and intercomponent security mismatches.

Much of this research has used information security criteria as the basis for designating an individual component or component-based system secure. The assumption is that component-based systems are most likely to be information systems (most recently, web services). Therefore, the security properties that must be exhibited are information security properties: confidentiality, integrity, and availability of information, and accountability of users.

To the extent that requirements for integrity and availability extend to the software system that handles the information, they may be seen as software security properties. However the problems most often examined and the examples most often given in the research literature revolve around how to achieve, across a system composed of components with different security properties or assurance levels, the secure, cohesive implementation of functionality for user identification and authentication (I&A), trust establishment among components (based on authenticated identity and authorization attributes, as in the WS-Trust standard), and access control and confidential exchange of data. Unsurprisingly, Common Criteria (CC), Evaluation Assurance Levels (EAL), are often suggested as the basis for defining component-level and system-level assurance.

Although such research may yield useful concepts for addressing software security challenges in component-based systems, the specific techniques, technologies, and tools produced by the researchers may not be directly useful or easily adaptable. This said, the following papers provide a representative sampling of the research that has been done, and is being done, in this area—

- Scott Hissam and Daniel Plakosh, (CMU SEI) *COTS in the Real World: a Case Study in Risk Discovery and Repair,* technical note number CMU/SEI-99-TN-003, (Pittsburgh PA): CMU SEI, (June, 1999).
  Available from: http://www.sei.cmu.edu/publications/documents/99.reports/99tn003/99tn003abstract.html
- Ulf Lindqvist and Erland Jonsson (Chalmers University of Technology), "A Map of Security Risks Associated with Using COTS", *IEEE Computer*, 31(1998) 60–66.
  Available from: http://www.windowsecurity.com/uplarticle/1/cots98.pdf
- Khaled M. Khan, (University of Western Sydney), Jun Han (Swinburne University of Technology), "Assessing Security Properties of Software Components: a Software Engineer's Perspective", in *Proceedings of the*

Australian Software Engineering Conference, Sydney, Australia, April 18–21 2006.

- Ibid; "Security Characterisation of Software Components and Their Composition", in *Proceedings of the 36th IEEE International Conference on Technology* of *Object-Oriented Languages and Systems*, Xi'an, China, October 30–November 4 2000.
  Available from: http://www.it.swin.edu.au/personal/jhan/jhanPub.html#security
- Manasi Kelkar, Rob Perry, Todd Gamble and Amit Walvekar (University of Tulsa), "The Impact of Certification Criteria on Integrated COTS-based Systems," in *Proceedings of the Sixth International Conference on COTS-Based Software Systems*. Banff, Alberta, Canada, February 26–March 2 2007.
  Available from: http://www.seat.utulsa.edu/papers/ICCBSS07-Kelkar.pdf

5.1.1.3 Security Issues Associated With Custom Developed Software

Rather than acquire a COTS or OSS product, or assemble a system out of existing components, many organizations develop software from scratch. This provides the organization with software that meets its exact needs while also enabling tight ongoing control of the software during its operation and maintenance/support phases. To this point, most software assurance research and the knowledge base has been in the area of custom-built software. This is primarily because when an organization is developing custom software, it can directly control all aspects of the SDLC.

At the beginning of the SDLC, an initial system-level risk assessment focusing on the business assets, threats, likelihood of risks, and their potential business impacts can provide input to the requirements specification and define the context for the security aspects of the software's architecture and design. The software's architectural risk assessment then refines the system risk assessment by analyzing how well the software addresses the system risks, suggesting mitigation strategies, and identifying additional risks that are added by the software architecture. Applied iteratively through the development life cycle phases, these methods can help refine the understanding of risk with increasing degrees of detail and granularity. Several software risk assessment (*i.e.*, threat modeling) methodologies have been developed and used within the software development industry. These methodologies are described in Section 5.2.3.1.

In addition to threat modeling and other security risk analysis methodologies, testing tools are available for developers of custom-built software. Section 5.5 describes software security testing methodologies.

For the implementation phase of the SDLC, many organizations have adopted programming languages, development tools, and execution environment protections to increase the security of their software. These are discussed in Section 5.4.

To take full advantage of the security tools available throughout the SDLC, several organizations have developed security enhancements to existing methodologies or new security-focused methodologies. These are discussed in Section 5.1.3.

In addition to security-enhanced life cycle methodologies, attempts have been made to define a process capability maturity model (CMM) that includes security activities and checkpoints to increase the likelihood that secure software or systems will be engineered under those processes. The most noteworthy examples of security-enhanced CMMs are—

- **Systems Security Engineering Capability Maturity Model (SSE-CMM)**—[98] Originally defined by the National Security Agency (NSA), and now an international standard (ISO/IEC 21827), SSE-CMM enables a system development organization to add security practices into their systems engineering (SE) CMM process. SSE-CMM augments the process areas in the SE CMM by adding security engineering process areas that address various aspects of security engineering. See Section 4.6 for a description of SSE-CMM.
- **Federal Aviation Administration (FAA)/DoD Proposed Safety and Security Extensions to integrated Capability Maturity Model (iCMM) and Capability Maturity Model Integration (CMMI)**—The security extensions add security activities to iCMM and CMMI process areas in the same way that the SSE-CMM adds security activities to the process areas of the SE-CMM. (See Appendix D for a description of the proposed safety and security extensions.)

Although there are a wealth of tools and methodologies available to improve the security of custom-built software, there is little empirical evidence showing the return on investment (ROI) or success of these various techniques. Organizations like SEI, DHS, and DoD are looking into methods to estimate the ROI of the various techniques to aid organizations in deciding what security methodologies should be embraced to secure their custom-built software.

5.1.1.4 Security Issues Associated With Software Reengineering

The reengineering of software was described by E.J. Chikofsky and A.H. Cross in 1990. [99] Software reengineering is the modification of existing software so that components can be changed, replaced, or removed to make the software more effective or efficient. According to Chikofsky and Cross, reengineering usually involves some level of reverse engineering, along with forward engineering. Often, portions of the software system may need to be reverse engineered to give an organization a more abstract understanding of the existing system. The reverse engineering is generally followed by forward engineering to reconstitute components or to add functionality to the system.

Reengineered software brings with it many of the risks associated with acquired software, custom-built software, and assembled software. The new or modified functionality may be provided by COTS or custom-built software; regardless, some modifications or custom-developed code may be required to integrate the new functionality with the unmodified portions of the software.

Additionally, reengineering software may introduce new vulnerabilities into a system because of an incomplete understanding of the original software design. For example, updating a library to provide enhanced or improved features may inadvertently affect the software using the library. Some portions of the software may somehow depend on the original implementation of the library; the software may modify the library results as a result of a defect in the library. Without a full understanding of the original system, the software may modify the now-correct output with unexpected results.

Similarly, reengineering software may introduce new vulnerabilities by increasing the complexity of the system. By adding a component to the software, its library dependencies may result in unexpected behavior from other components, or it may not be aware of the entire range of output or input supported by the other components. Any unexpected behavior in the overall system may manifest itself as a security vulnerability.

Although little security research activity is performed in the specific realm of software reengineering, practitioners benefit from security research in software composition, acquisition, and custom development, as well as reverse engineering. Another factor affecting the security research activity in this field is that much of the research is in developing structured reengineering processes, such as those being developed by the CMU SEI and the University of Queensland (Australia).

For Further Reading

"Legacy Systems" [website].
Available from: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/legacy.html

IEEE Computer Society Technical Council on Software Engineering, Reengineering Bibliography.
Available from : http://www-static.cc.gatech.edu/reverse/bibliography

CMU SEI, *Reengineering*.
Available from: http://www.sei.cmu.edu/reengineering

"Reengineering Forum" [website].
Available from: http://www.reengineer.org

5.1.1.5 Why Good Software Engineering Does Not Guarantee Secure Software

Good software engineering is essential for producing secure software, but it is not sufficient. This is because most software engineering is oriented toward functionality, quality, or reliability. It can be argued that for software, security is an essential aspect of quality and reliability—but this has not always been considered the case, so most quality- and reliability-oriented software processes

omit many of the activities necessary for security. If the software is 100 percent reliable but relies on Data Encryption Standard (DES) for encryption, it must still be considered insecure.

Multiple quality- and reliability-focused system and software engineering techniques are available. ISO/IEC 15288, CMM, and CMMI are examples of system engineering processes that focus on the generic processes that an organization performs when developing software. These methodologies provide frameworks through which organizations can define repeatable processes that can potentially be measured and improved to improve the quality and cost of the development process. If security is provided as a requirement up front, it is possible for organizations to use these models to improve the security of their software while improving the quality of software. However, many of the activities required to improve the security of a system are separate from those routinely performed to assess the quality of a system. Consequently many organizations have proposed extending ISO/IEC 15288 and the SEI CMMI to address security activities.

Similarly, the developers [100] of an application may not have made appropriate assumptions about security. Because many universities do not teach the importance of security—or the activities necessary to develop secure software—many developers are untrained in security. To remedy this, DHS is sponsoring the development of *Software Assurance: A Guide to the Common Body of Knowledge to Produce, Acquire, and Sustain Secure Software* (often referred to simply as the CBK) to serve as input in curricula for university and college courses. DHS is also developing a comparable Essential Body of Knowledge (EBK) as the basis for developing or enhancing professional training courses for software programmers, testers, and others.

Note: Academic and industry concerns about the DHS CBK and EBK are discussed in Section 7.2.2.

Developers trained in security may still make inappropriate assumptions. One common assumption is that security is dealt with at a different level of the system or in a different phase of development. Related assumptions include—

- The operating system supports mandatory access control, so the software does not have to be secure.
- The software's host is on a classified network, so the software does not have to be secure.
- The software is a prototype; security will be addressed later.
- The software encrypts data before transmitting it, so the software is adequately secure.
- The software is written in a type-safe language (*e.g.,* Java or C#); therefore, buffer overflows are impossible and security has been addressed.

For a small subset of software, such assumptions may be valid. However, it is likely that the assumptions may become invalid by the end of the SDLC: mandatory access controls may be disabled on the production system, the trusted network may

begin to allow untrusted entities to access it, the prototype may so impress the client that it enters production, the encryption may be strong but attackers could bypass it by attacking the user interface instead, or the Java software may have a command-injection vulnerability. Therefore, it is important to periodically review the software in its current state to determine whether the original security assumptions have been invalidated and if so, to make the necessary adjustments to ensure that the assumption mismatches (between the software-as-designed and the software-as-operated) have not introduced new security vulnerabilities into the system.

5.1.2 Using Formal Methods to Achieve Secure Software

Formal methods are an adaptation to software development of certain aspects of mathematics developed in 19th and 20th century mathematics. A formal system consists of four elements:

1. A set of symbols.
2. Rules for constructing well-formed formulas in the language.
3. Axioms for formulae postulated to be true.
4. Inference rules, expressed in a metalanguage. Each inference rule states that a formula, called a consequent, can be inferred from other formulae, called premises.

Formal methods are not just disciplined methods, but rather the incorporation of mathematically based techniques for the specification, development, and verification of software. Inasmuch as vulnerabilities can result from functionally incorrect implementations, formal methods, in general, improve software security (at a cost).

An example of a successful implementation of formal methods to further software security is type checking. An integral feature of modern programming languages like Java, C#, and Ada95, type checking is a particularly successful and familiar implementation of formal methods. Type checking increases the detection rate of many types of faults and weaknesses at compile time and runtime. The "specification" contains the type information programmers provide when declaring variables. The "verification" of the specification is achieved through use of algorithms (such as Robin Milner [101]) to infer types elsewhere in the code, and to ensure that overall typing is internally consistent. The outcome of the verification is "assurance" (contingent on an absence of extrinsic interventions) of—

- The integrity of how raw bits are interpreted in software as abstract values.
- The integrity of the access pathways to those values. Type checking affects all software engineering practices using modern languages.

It remains a research challenge to develop formal methods for the specification and verification of non-trace security properties in software, such as non-subvertability of processes and predictability of software behavior under unexpectedly changing environment conditions associated with malicious input, malicious code insertions, or intentional faults.

At a high level, the main uses for formal methods in the SDLC include—

- Writing the software's formal specification
- Proving properties about the software's formal specification
- Constructing the software program through mathematical manipulation of its formal specification
- Using mathematical arguments and proofs to verify the properties of a program.

Note: The majority of formal methods focus either on formal construction or on after-the-fact verification, but not both.

Formal methods have applications throughout the SDLC. Figure 5-1 maps possible uses of formal methods to each phase of the SDLC. Because formal methods can be used for correctness, independently of security concerns, the life cycle phases are not labeled in terms of security concerns.

Figure 5-1. Formal Methods in the SDLC

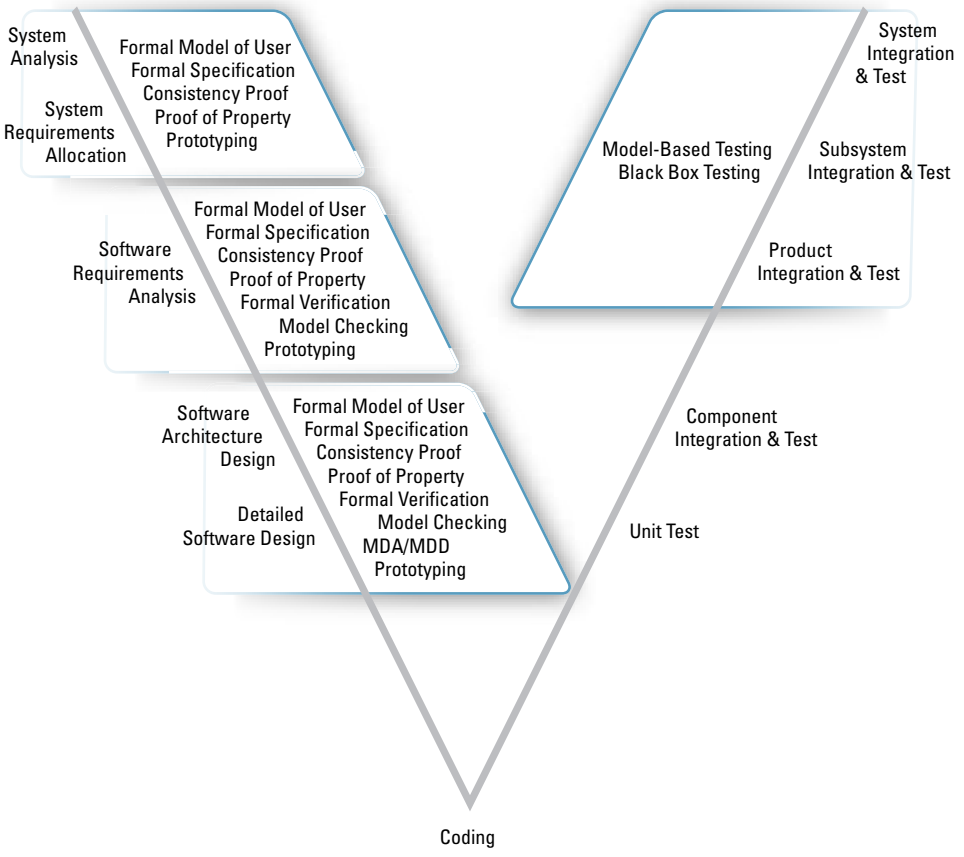


Table 5-1 describes each of the formal methods activities in the diagram, indicating the SDLC phases to which each activity pertains.

Table 5-1. Formal Methods Activities and SDLC Phases

Formal Method Activity Description	SDLC Phases in Which Undertaken
Formal models of user behavior: often describe sequences in which users invoke the functionality of a system. For example, decision tables, finite state machines, Markov models, or Petri nets can characterize user actions.	▶ system analysis ▶ system requirements allocation ▶ software requirements *also useful in generating test cases during:* • system integration and testing • subsystem integration and testing • product integration and testing
Formal specifications: rigorously describe the functionality of a system or system component. Languages used, such as in the Vienna Development Method (VDM) and Z, often involve extensions of a mixture of predicate logic and set theory.	▶ system analysis ▶ system requirements allocation ▶ software requirements ▶ architecture design
Consistency proofs: examine the components of a system in a formal specification developed at a single level of abstraction. They are useful at every phase in which a formal model is developed.	▶ system analysis ▶ system requirements allocation ▶ software requirements ▶ software architecture design ▶ software detailed design ▶ coding
Proofs of properties: prove that some proposition regarding states or combinations of states in the system is always maintained as true. For example, a formal method for safety might include the proof that some state never arises in a system.	▶ system analysis ▶ system requirements allocation ▶ software requirements ▶ software architecture design ▶ software detailed design ▶ coding
Model checking: a practical technique for automated formal verification. Model checking tools use symbolic expressions in propositional logic to explore a large state space. Model checking can be used in the same phases in which formal verification is used. With some analysis, it is possible to determine whether a model checking result is trustworthy enough to form the basis for positive assurance; however, such a determination is not intrinsic to the technique.	▶ software requirements ▶ software architecture design ▶ software detailed design ▶ coding
Prototyping: not necessarily a formal method. However some formal method tools can be used to generate a prototype, particularly if an operational semantics is used. (Prototyping can be accomplished without as high a degree of automation and formality.)	▶ system analysis ▶ system requirements allocation ▶ software requirements ▶ software architecture design ▶ software detailed design

Table 5-1. Formal Methods Activities and SDLC Phases - *continued*

Formal Method Activity Description	SDLC Phases in Which Undertaken
Model-driven architecture (MDA): automatic generation of an architecture from a Unified Modeling Language (UML) specification of a system.	▶ software architecture design
Model-driven development (MDD): supports the construction of a system or system component by transforming a formal or semi-formal model into an implementation.	▶ software detailed design ▶ coding
Black box testing: entails the development of test cases based on specifications of the system or system component being tested, as opposed to the development of test cases based on knowledge of internal implementation of the system or component. Because the specification is formal, formal techniques can be used in generating black box test cases.	▶ system integration and testing ▶ subsystem integration and testing ▶ product integration and test
Model-based testing: the automatic generation of efficient test cases from models of requirements and functionality, given a formal model of the user developed in the corresponding requirements phase.	▶ system integration and testing ▶ subsystem integration and testing ▶ product integration and test

For Further Reading

Constance Heitmeyer (NRL), *Applying Practical Formal Methods to the Specification and Analysis of Security Properties,* (May, 2001).
Available from: http://chacs.nrl.navy.mil/publications/CHACS/2001/2001heitmeyer-MMM-ACNS.pdf

Jeannette M. Wing (CMU SEI), *A Symbiotic Relationship Between Formal Methods and Security,* CMU-CS-98-188, (December, 1998).
Available from: http://reports-archive.adm.cs.cmu.edu/anon/1998/abstracts/98-188.html

D. Richard Kuhn, Ramaswamy Chandramouli, and Ricky W. Butler (NIST, NASA Langley Research Center), "*Cost Effective Use of Formal Methods in Verification and Validation*", in *Proceedings of the Foundations 02 Workshop on Verification and Validation,* (October, 2002).
Available from: http://csrc.nist.gov/staff/kuhn/kuhn-chandramouli-butler-02.pdf

Formal Methods Virtual Library.
Available from: http://vl.fmnet.info

Michael Huth and Mark Ryan, *Logic in Computer Science: Modelling and Reasoning About Systems,* 2nd ed, (Cambridge University Press, 2004).
Available from: http://www.cambridge.org/us/catalogue/catalogue.asp?isbn=052154310x

Jonathan P. Bowen and Michael G. Hinchey, "The Use of Industrial-Strength Formal Methods," in: *Proceedings of the 21st International Computer Software and Applications Conference (COMPSAC 97).* 1997. Available from: http://www.jpbowen.com/pub/compsac97.pdf

Robert L. Vienneau, (Data and Analysis Center for Software), *A Review of Formal Methods,* (May 26 1993).
Available from: http://www.dacs.dtic.mil/techs/fmreview/title.html

5.1.2.1 Limitations of Formal Methods for Assuring Software Security

Formal methods have limitations of scale, training, and applicability in principle. To compensate for the limitations of scale, formal methods have been applied to selected parts or properties of a software project, in contrast to applying them to the entire system. As for training limitations, it may be difficult to find developers with the needed expertise in formal logic, the range of appropriate formal methods for an application, or appropriate automated software development tools for implementing formal methods. Finally, not all formal methods are equally applicable on all systems. Formal languages without modularization capabilities and scope-delimiting rules are difficult to use on large systems at any but the highest level of abstraction.

Formal methods also have limitations in principle. A formal verification can prove that an abstract description of an implementation satisfies a formal specification or that some formal property is satisfied in the implementation. However a formal method cannot prove that a formal specification captures a user's intuitive understanding of a system and furthermore cannot prove that an implementation runs correctly on every physical machine. As restated by Barry W. Boehm, [102] formal methods are sometimes useful for verifying a system, but they cannot be used in validating a system. Validation shows that a system will satisfy its operational mission. Verification shows that each step in the development satisfies the requirements imposed by previous steps.

DHS' *Security in the Software Life Cycle* observes that because software security properties are often expressed in negative terms (*i.e.,* what the software must *not* do) it is particularly difficult to specify requirements for those properties (formally or informally), and then to mathematically prove that those requirements have been correctly satisfied in the implemented software. The same is true of both security and safety properties, which are both a form of universal statement that "nothing bad will happen."

Formal methods for design have been mandated for software that must meet high assurance levels, for example, at CC EAL 7 and above. Formal methods have also proven successful in specifying and checking small, well structured systems such as embedded systems, cryptographic algorithms, operating system reference models, and security protocols.

5.1.3 Security Risk Management in the SDLC

The term software risk management is generally used with regard to management of project risk or risk to software quality. This is the how the term is understood in ISO/IEC 16085:2004, *Systems and software engineering—Life cycle processes—Risk management*. Most software project and quality risk management methodologies and tools are not easily adaptable to security risk concerns.

System security risk management methods and tools would seem to be more directly useful for software security risk management, but are limited in

the same way their component risk assessment methods are limited when used for software security risk assessments. The view they tend to take is system-level/architectural, with the focus on operational risks. They address software-specific risks, especially those that emerge in the software's development process rather than after deployment.

Only recently has management of security risks throughout the software life cycle become a topic of widespread discussion, with techniques and tools emerging to support it, and the integration of project and security risk management for software development projects. This is an area in which the software security and reliability consulting firm Cigital has been active for a number of years. In the chapter "Managing Software Security Risk" in *Building Secure Software*, [103] Cigital's Gary McGraw and John Viega suggest the following activities as the components of a security risk management process within the SDLC (versus the software operational life cycle):

- Security requirements derivation/elicitation and specification (Section 5.2)
- Security risk assessment (Section 5.2.3.1)
- Secure architecture and design (Section 5.3)
- Secure implementation (Section 5.4)
- Security testing (Section 5.5)
- Security assurance (Section 5.5.4).

Cigital's viewpoint is, of course, not the only one. Like Cigital, Marco Morana of Foundstone suggests an activity-driven approach. However, his approach combines both long-term, holistic software security approaches to mitigating risk with short-term, issue-specific application security approaches. He also recommends considering software security and information security risks in tandem, rather than separately.

Morana outlines a set of risk management activities that map directly to the main phases of the SDLC: [104]

1. **Requirements**—Security requirements engineering, setting of compliance goals, application of industry/organizational standards, specification of technical security requirements, threat modeling, and security measurements (Section 5.2)
2. **Architecture and Design**—Threat modeling, architecture and design security patterns, security test planning, architecture and design security reviews, and security measurements (Section 5.3)
3. **Development**—Code reviews, use of security patterns, flaw and bug mitigation, unit security testing, update of threat models, and security measurements (Section 5.4)
4. **Testing**—Use of attack patterns, automated black box and white box testing, regression testing, stress testing, third-party security assessments, updating of threat models, and security measurements (Section 5.5)

5. **Deployment**—Deployment and operational security measures, patch management, incident management, update of threat models, and security measurements (Section 5.6).

Across all of these life cycle phases, policy, training, tools, and metrics are applied. Morana also recommends the use of security-enhancing life cycle process models, such as Comprehensive Lightweight Application Security Process (CLASP) or Microsoft's Security Development Lifecycle (SDL), Gary McGraw's Seven Touch Points, SEI's TSP-Secure, as well as security best practices, and automated tools.

Yet another vulnerability-oriented viewpoint is presented by Charles Le Grand in "Managing Software Risk," [105] identifying four key activities for software security risk management:

1. **Risk Assessment**—Determination of the extent of vulnerabilities; estimation of the probability of losses caused by exploits; includes intrusion detection/prevention, risk/attack assessment of network-facing systems, and cost-benefit analysis of countermeasures
2. **Vulnerability Management**—Identification, measurement and remediation of specific vulnerabilities
3. **Adherence to Security Standards and Policies for Development and Deployment**—Prevents the introduction of vulnerabilities
4. **Assessment, Monitoring, and Assurance**—Ongoing audits and monitoring of risk levels to ensure that they remain within acceptable thresholds; also determine the effectiveness of software security risk management measures in achieving legal, regulatory, and security policy compliance.

Le Grand also emphasizes the executive's role in security risk management because "any enterprise-wide program for managing software risk requires executive-level sponsorship and leadership."

For Further Reading

"Risk Management".
Available from: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/risk.html

Idongesit Mkpong-Ruffin and David A. Umphress, PhD (Auburn University), "High-Leverage Techniques for Software Security", *CrossTalk*, (March, 2007).
Available from: http://www.stsc.hill.af.mil/CrossTalk/2007/03/0703RuffinUmphress.html

5.1.3.1 Risk Management Frameworks

The use of risk management frameworks (RMF) for managing enterprise/organizational risk has been prevalent for years. More recently, the framework approach has been applied to security risk management, and even more recently, specifically to software/application security risk management.

In 2004, Microsoft described its concept of security risk management, including its own Security RMF, [106] which maps to its larger Microsoft Solutions

Framework (MSF). The MSF maps risk management activities to the phases of a life cycle process it calls the MSF Process Model, as depicted in Table 5-2.

Table 5-2. Microsoft Security Risk Management Framework

MSF Process Phase	Risk Management Framework Activities
Initiation	Initiation of project definition: All parties involved in security management must define goals, assumptions, and constraints. Outcome: Approval of project vision and scope
Planning	Assessment and analysis of security management processes: includes organizational assessment, asset valuation, threat identification, vulnerability assessment, security risk assessment, countermeasure/ security remediation planning. Outcome: Approval of project plan
Building	Development of security remediations: development, unit testing, and quality validation of countermeasures. Outcome: Completion of scoping
Stabilizing	Security remediation testing and resource functionality testing: involves detection and severity rating of security bugs. Outcome: Approval of release readiness
Deploying	Security policy and countermeasure deployment: includes enterprise-wide, centralized, and site-specific policies, countermeasures and security components. Outcome: Completion of deployment and re-entry into beginning of life cycle, with conveyance of security risk management knowledge and lessons learned

Like many of Microsoft's security processes (*e.g.*, SDL), its security risk management framework is strongly oriented toward turnkey software product development, rather than software integration, so it would have to be expressly adapted for reuse by integrators of noncommercial software systems.

The software security and reliability consulting firm Cigital has used its proprietary Cigital RMF for more than a decade. Under contract to DHS, Cigital developed the BuildSecurityIn (BSI) RMF, a condensed version of the Cigital RMF. The BSI RMF consists of five-phases of risk management activities:

1. Understanding the business context in which software risk management will occur.
2. Identifying and linking the business and technical risks within the business context to clarify and quantify the likelihood that certain events will directly affect business goals. This activity includes analyses of software and development artifacts.
3. Synthesizing and ranking the risks.
4. Defining a cost-effective risk mitigation strategy.
5. Carrying out and validating the identified fixes for security risks.

Surrounding all of these activities is a sixth, pervasive activity, Measurement and Reporting. According to Gary McGraw, whose article on the BSI RMF appears on the DHS BuildSecurityIn portal: [107]

As we converge on and describe software risk management activities in a consistent manner, the basis for measurement and common metrics emerges. Such metrics are sorely needed and should allow organizations to better manage business and technical risks given particular quality goals; make more informed, objective business decisions regarding software (e.g., whether an application is ready to release); and improve internal software development processes so that they in turn better manage software risks.

According to McGraw, typical risk metrics include, but are not limited to, risk likelihood, risk impact, risk severity, and the number of risks that emerge and are mitigated over time.

Like McGraw, Morana of Foundstone is also a proponent of what he terms *Software Security Frameworks*, such as that depicted in Figure 5-2.

Figure 5-2. Notional Software Security Framework [108]

SDLC Phases	Requirements		Design	Development	Testing	Deployment and Operations	
Secure Software Best Practices	Preliminary Software Risk Analysis	Security Requirements Engineering	Security Risk-Driven Design	Secure Code Implementation	Security Tests	Security Configuration & Deployment	Secure Operations
Ongoing S-SDLC Activities Metrics and Measurements, Training, and Awareness							
S-SDLC Activities	Define Use & Misuse Cases	Define Security Requirements	Secure Architecture & Design Patterns Threat Modeling Security Test Planning Security Architecture Review	Peer Code Review Automated Static and Dynamic Code Review Security Unit Tests	Functional Test Risk Driven Tests Systems Tests White Box Testing Black Box Testing	Secure Configuration Secure Deployment	
Other Disciplines	High-Level Risk Assessments		Technical Risk Assessment				Incident Management Patch Management
Other On Going Disciplines Information Risk Management, Detect Management, Change Management, Vulnerability Management							

5.1.3.2 Tools for Software Security Risk Management

In his paper, *Security Risks: Management and Mitigation in the Software Life Cycle*, [109] David Gilliam of The National Aeronautics and Space Administrations (NASA) Jet Propulsion Laboratory (JPL), describes a formal approach to managing and mitigating security risks in the SDLC that requires integration of a software security checklist and assessment tools with a security risk management and mitigation tool, and used iteratively throughout the software life cycle. The tools Gilliam describes using at JPL are the Software Security Assessment Instrument (SSAI), developed by the NASA Reducing Software Security Risk (RSSR) program, and JPL's Defect Detection and Prevention (DDP) tool. [110]

Since the development of the SSAI and DDP, several commercial tools vendors have introduced integrated, centrally managed software and/or application security risk management tool suites (*e.g.,* Radware's APSolute Application Security solution suite; SPI Dynamics' Assessment Management Platform) that include tools that support capabilities including code review, vulnerability assessment, black box and penetration testing, attack/intrusion monitoring and detection, security compliance testing/verification, security policy management, application firewalls, encryption, vulnerability management, and even patch management. Examples include those described below.

5.1.4 Software Security Assurance Cases

As noted in Section 2.1.3, according to DHS, software assurance provides "a basis for justified confidence" in a required property of software (or of a software-intensive system). This basis for justified confidence may take the form of an assurance case. T. Scott Ankrum and Charles Howell [111] define an assurance case as—

> *A documented body of evidence that provides a convincing and valid argument that a specified set of critical claims regarding a system's properties are adequately justified for a given application in a given environment.*

An assurance case documents assurance arguments and assurance claims about a software component, product, or system, and provides the necessary evidence of the validity of those arguments and claims sufficient to reduce uncertainty to an acceptable level, thus providing the grounds for justified confidence that the software exhibits all of its required properties. For this document, the required property of interest is security.

The assurance case should be developed alongside the software component or system itself. The information added to the assurance case at each life cycle phase will vary based on the level of assurance that is sought.

There is an increasing emphasis, in the software assurance community, on defining standards for the content and evaluation of security assurance cases

for software. The only claim that can be realistically made for software security assurance cases at this point is that they will provide a useful mechanism for communicating information about software risk. To date, there has been little if any empirical evidence that assurance cases can or will improve the security of software or increase the level of trust between users and software suppliers.

The most mature of the emerging software security assurance case standards, SafSec (see Section 5.1.4.2.1), has only been tested in two case studies. As of April 2007, the next stage of SafSec evaluation, which involves applying the methodology on a system under development, was still underway. It would seem that so far, assurance case proponents have based their expectation of the effectiveness of security assurance cases on an extrapolation from the success of safety cases (see Section 5.1.4.1) and to a lesser extent from CC STs (discussed in Section 4.8).

For Further Reading

Assurance Cases.
Available from: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/assurance.html

Elisabeth A. Strunk and John C. Knight, "The Essential Synthesis of Problem Frames and Assurance Cases", in *Proceedings of the Second International Workshop on Applications and Advances in Problem Frames,* May 23 2006.

CMU SEI, *Assurance Case and Plan Preparation.*
Available from: http://www.sei.cmu.edu/pcs/acprep.html

J. McDermott, "Abuse Case-Based Assurance Arguments", in *Proceedings of the 17th Annual Computer Security Applications Conference,* December 2001: 366-374.
Available from: http://www.acsa-admin.org/2001/abstracts/thu-1530-b-mcdermott.html

5.1.4.1 Safety Cases as a Basis for Security Assurance Cases

The concept of an assurance case originated with the safety case. In response to several significant catastrophes that were traced to the failure of certain physical systems (*e.g.,* space shuttles, off-shore petroleum drilling facilities), multiple safety standards and regulations emerged in the 1990s [*e.g.,* Radio Technical Commission for Aeronautics (RTCA) DO-178B *Software Considerations in Airborne Systems and Equipment Certification*, ISO 14971 *Application of Risk Management to Medical Devices*, MIL-STD-882D DoD *Standard Practice for System Safety*, UK Defence Standard 00-56 *Safety Management Requirements for Defence Systems*] specifying requirements for the verification of reliability and fault-tolerance in safety-critical systems.

Safety-critical systems constitute physical systems or devices (*e.g.,* airplanes, nuclear power stations, medical devices), including their software-based monitoring and control components. These software components are either embedded within or networked to the physical system they monitor or control. [An example of the former is an electric flight control system; an example of the latter is software in an air traffic control system or the

Supervisory Control and Data Acquisition (SCADA, see Appendix C) software for climate monitoring and control in nuclear power plants].

To satisfy the mandates of the new safety standards and regulations, safety cases were conceived [112] to provide a formal documentary basis that regulators, auditors, *etc.,* needed to verify with high levels of justified confidence the reliability and fault-tolerance of safety-critical systems. A new term was coined for systems and software that required the exhibition of critical properties (such as reliability, safety, and survivability) [113] to be verified with a high level of confidence: high confidence (as in High Confidence Software and Systems).

In addition to regulatory, standards, or policy compliance, safety cases can be a prerequisite of third-party certification, approvals to operate, licensing, and other situations in which a compelling case needs to be made that the system satisfies its critical properties for use in specific contexts, such as healthcare or avionics.

There is a growing academic trend in Europe whereby software safety research is being extended to include software security. Consistent with this trend, most research by into software security assurance case methodologies and tools is being done at European institutions that have long been engaged in software safety research. (Appendix H lists specific examples of such research)

Just as many other practices from the software safety and information assurance communities have proven adaptable for purposes of assuring security (*e.g.,* formal methods, fault injection), so the notion of assurance cases for establishing confidence in the *security* of software has emerged.

Most currently defined software security assurance cases include—

- One or more claims about the required security attributes of the software
- A body of evidence supporting those claims
- Arguments that clearly link the evidence to the claims.

Depending on the nature of its evidence and the persuasiveness of its arguments, an assurance case can reduce uncertainty and lead to justified confidence in the software's security, or it may provide grounds for a rational lack of confidence.

The body of evidence that supports each part of the assurance case can come in many forms. This evidence may reflect either direct analysis of the software (*e.g.,* test results, code review results, mathematical proofs from formal methods) or review of indirect indicators that the claims are likely to be true, such as the nature of the development process used, the reputation of the development organization, and the trustworthiness and expertise of the individual developers.

5.1.4.2 Software Assurance Case Standards

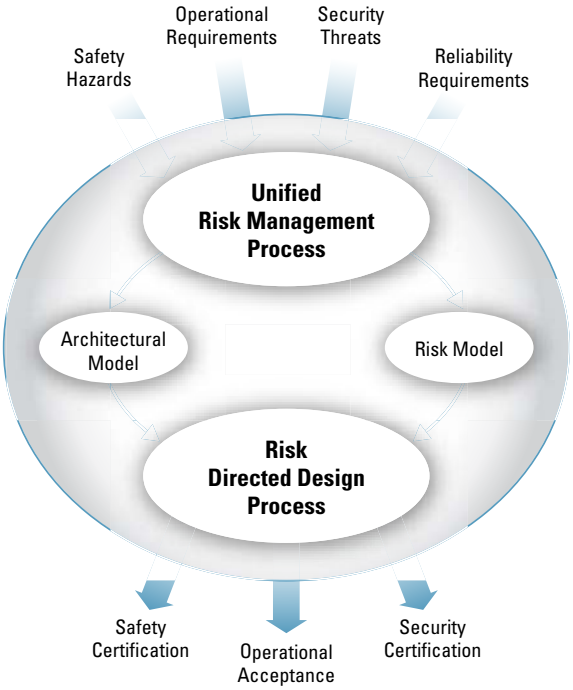
Some significant efforts are underway in the research community to develop such tools, or to extend, adapt, and refine safety case tools to also accommodate the particular requirements of security assurance cases. Efforts are also underway to standardize the software assurance process and its relationship to the software and

system development processes, as well as the required content and structure of assurance case artifacts. The most significant of these efforts are described below.

5.1.4.2.1 UK Ministry of Defence SafSec

SafSec [114] is an assurance methodology developed by Praxis High Integrity Systems in response to the requests of its sponsors in the UK Ministry of Defence (MOD) for a program that would "reduce the cost and effort of safety certification and security accreditation for future military Avionics systems." Praxis' response was to develop the SafSec standard and guidance documents, which define a standard structure for producing and evaluating a combined assurance case for software safety/reliability and security. In this way, SafSec provides an integrated view of assurance: not only do safety, reliability, and security converge in the C&A domain, they converge in the development domain, as illustrated in Figure 5-3.

Figure 5-3. SafSec Convergence of Safety, Reliability, and Security Assurance



The SafSec standard seeks to overcome the following problems associated with other software assurance processes:

- It ensures completeness.
- It minimizes overlap and duplication of evidence, and thus reduces the evaluation effort and the associated costs. Evidence that supports both safety/reliability and security assurance claims needs to be generated only once.

- It provides a single methodology and framework that supports both safety and security certification and accreditation of both products and systems, including highly modular systems.

The SafSec assurance case can be said to be an integrated dependability case in which safety and security are handled in parallel. The inclusion of reliability and maintainability concerns in the assurance argument and evidence ensures that all aspects of dependability are addressed. SafSec emphasizes the need to concentrate on the product rather than the processes. Justification of the means or processes by which the product was produced is less important than ensuring that the system itself is assured. For this reason, SafSec uses the assurance case as a structure for evidence about the product alone.

SafSec's implementation process incorporates three phases:

1. **Unified Risk Management**—The risk model is developed, taking into account safety hazards, security threats, and operational requirements of the target system.
2. **Risk-Directed Design**—The system's architectural design is produced and then used in conjunction with the risk model to define the required dependability properties and functionalities for all system modules.
3. **Modular Certification**—Each module's dependability properties and functionalities (documented in clear specifications) are used as the basis for building the supporting assurance arguments and evidence, to justify the certification/accreditation of the module. Based on the evaluation of the arguments and evidence, a set of safety and security certificates is produced.

SafSec assurance activities are initiated during the earliest phases of the software/system development life cycle, and are predicated on the collaboration of representatives of the safety, security, and program management domains.

5.1.4.2.2 ISO/IEC and IEEE 15026

In 2001, ISO/IEC began revising ISO/IEC 15026, System and Software Engineering—System and Software Assurance. The revised standard was to incorporate the concept of an "assurance case" for justifying confidence that the system/software exhibits all of its required critical properties (*e.g.*, security, safety, reliability). Because the assurance case is considered a life cycle artifact, the revised 15026 also specified how it should be defined, maintained, and revised throughout the system/software life cycle.

Unfortunately, by 2006, the ISO/IEC project reached its deadline without producing a 15026 revision considered ready for balloting. At the same time, The Institute of Electrical and Electronic Engineers (IEEE) initiated project P15026 [115] to take over work on the 15026 revision, although it is not clear whether IEEE's project has replaced the ISO/IEC effort or is being run parallel

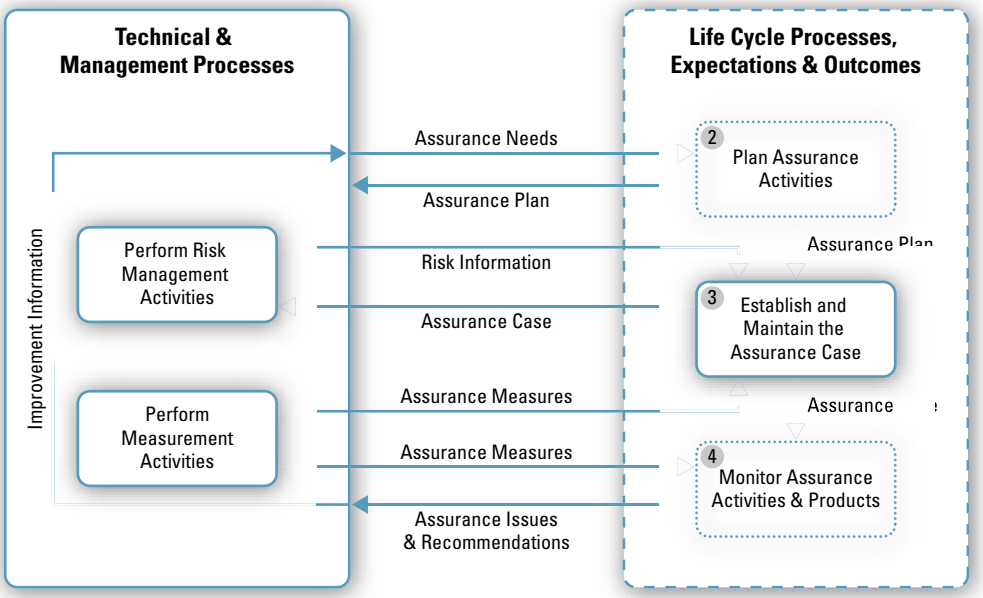
to it. The latest ISO/IEC draft of the revised standard defines the following life cycle process expectations and their associated outcomes:

1. Plan assurance activities
2. Establish and maintain the assurance case
3. Monitor and control assurance activities and products.

Figure 5-4 illustrates the relationship of the proposed new 15026 activities and those of the ISO/IEC life cycle and risk management activities to which 15026 is meant to provide assurance support.

Note: This diagram is the latest version available in the IEEE P15026 Working Document Revision 6, which is the most recent version of the proposed revision.

Figure 5-4. IEEE P15026



5.1.4.3 Workshops on Assurance Cases for Security

The Workshop on Assurance Cases for Security (hosted by the CMU SEI) grew out of the recognition by the previous Workshop on Assurance Cases (in June 2004), Best Practices, Possible Obstacles, and Future Opportunities, that security assurance cases presented a challenge that deserved further consideration and discussion.

The 2005 workshop brought together practitioners and researchers from the safety, reliability, and security communities. Participants came from government, academic, and industry organizations from several countries. Their objective was to visualize assurance cases for security, to explore the challenges they presented, and to propose viable technical approaches to realize them. A report entitled *Assurance Cases for Security* [116] was produced from technical output of the workshop.

The 2005 Workshop was followed up in March 2006 with the Workshop on Assurance Cases for Security: Communicating Risks in Infrastructures, which brought together the core group of attendees from the two previous workshops with experts from the risk assessment and network communications industries, including a representative of a critical UK nuclear infrastructure organization who was responsible for justifying the security of critical information and communications technology systems.

The most important conclusion from this workshop was the recognition of the need to support communication of risks between stakeholders involved in critical infrastructures: assurance cases appear to be a workable solution because they can be applied to the different elements of the infrastructure, including individual components, whole systems, processes, and organizations. The assurance cases for different types of elements might have different objectives, forms, and sources of evidence. The workshop participants, however, came to the conclusion that it should be possible to develop a single theoretical and methodological basis for all types of assurance cases.

The next workshop on Assurance Cases for Security is subtitled "The Metrics Challenge" and will be held in conjunction with the International Conference on Dependable Systems and Networks (DSN 2007) [117] in Edinburgh, Scotland. For more information on past Assurance Cases for Security workshops, see Robin E. Bloomfield, *et al.*, "International Working Group on Assurance Cases (for Security)" in the May/June 2006 issue of IEEE *Security & Privacy*.

5.1.4.4 Inherent Problems With Current Assurance Cases

There are acknowledged problems with the current state of assurance argument development, evidence gathering, and assurance case evaluation. According to T. Scott Ankrum and Charles Howell, [118] these problems have numerous sources, including (but not limited to)—

- The volume and nature of evidence to be considered.
- The lack of explicit relationships among assurance claims, assurance arguments, and supporting evidence.
- The lack of support for structuring the information. Most assurance case information is presented in free text, making it tedious to review, and difficult to discern linkages and patterns, or to locate key results within the sheer volume of evidence presented.
- The lack of a standard set of "rules of evidence."
- Exclusive emphasis in current guidance for assurance case development on the format of the information (often described in intricate detail).
- The lack of guidance on how to gather, merge, and review arguments and evidence, requiring both developers and evaluators of assurance cases to develop their own ad hoc criteria.
- The lack of explicit guidance for weighing conflicting or inconsistent evidence.

- Difficulty in comprehending the often-complex impacts of changes because of the immense volume of information to be considered. Changes to assurance cases may be triggered by invalidity of claims and/or evidence, or the availability of new evidence, thus enabling new claims. In either case, such changes can render existing assurance arguments invalid, requiring revalidation (with significant associated cost).

It has been suggested (primarily by researchers and vendors who are developing such tools) that at least some these problems can be mitigated by improving the tools that support software assurance case development and assessment activities.

5.1.5 Software Security Metrics and Measurement

Software security and application security have become big business. Advocates of different security-enhanced software processes, software security best practices, and a variety of supporting techniques and tools all suggest that those who adopt them will reap great benefits in terms of more secure (or at least less vulnerable) software. However, the fact is that there are few concrete metrics by which to precisely and objectively measure the effectiveness (innate and comparative) of all these different processes, practices, techniques, and tools. Moreover, there is active debate underway in the metrics and measurement community, which is attempting to define such metrics, regarding exactly what can and should be measured as a meaningful indicator that software is actually secure (or not vulnerable).

In August 2006, the first-ever conference devoted to security metrics, Metricon 1.0, [119] was held in Vancouver, British Columbia, Canada. Steve Bellovin, one of the "greybeards" of the information and network security community, summed up the problem nicely in his keynote address to Metricon. He argued that for software, meaningful security metrics are not yet possible:

> *Safes are rated for how long they'll resist attack under given circumstances. Can we do the same for software?…It's well known that any piece of software can be buggy, including security software…This means that whatever the defense, a single well-placed blow can shatter it. We can layer defenses, but once a layer is broken the next layer is exposed; it, of course, has the same problem…The strength of each layer approximates zero; adding these together doesn't help. We need layers of assured strength; we don't have them. I thus very reluctantly conclude that security metrics are chimeras for the foreseeable future. We can develop probabilities of vulnerability, based on things like Microsoft's Relative Attack Surface Quotient, the effort expended in code audits, and the like, but we cannot measure strength until we overcome brittleness.*

By Bellovin's criteria metrics for software security are impossible because 100 percent security of software is not possible, *i.e.,* one cannot measure what cannot possible exist. During the software security metrics track that followed Bellovin's keynote (and the very fact that there was such a track implicitly refuted Bellovin's argument), Jeremy Epstein of webMethods implicitly agreed with Bellovin that absolute security of software is probably not possible, [120] but disagreed that it was impossible to collect some combination of statistics about software—measurements that are already being taken—and then to determine which of these metrics (alone or in combination with others) actually says something meaningful about the security of the software. In short, given a statistic such as number of faults detected in source code, is it possible to extrapolate something about the influence of that statistic on the security of software compiled from that source code, *i.e.,* do fewer faults in source code mean software that is less vulnerable? (Incidentally, this is the premise upon which the whole source code analysis tools industry is based.)

Of course one can reasonably argue, as Bellovin has, that even a single implementation fault, if exploited, can compromise the software. Even were there no implementation faults, the software could exhibit overall weakness due to inadequacies in its design and architecture. This type of inadequacy is much harder to pinpoint, *let alone* to measure.

Researchers involved in defining metrics for software security do not pretend they can define measurements of absolute security. The best they can hope for is to measure different characteristics and properties of software that can be interpreted in aggregate as indicating the *relative* security of that software, when compared either with itself operating under different conditions, or with other comparable software (operating under the same or different conditions).

Along the lines of Epstein's suggestion, *i.e.,* to gather statistics that one knows can be gathered, then to consider them in terms of their indications for software security, quite a bit of the software security metrics work to date has, in fact, involved investigating already-defined information security and software quality and reliability metrics, to determine whether any of these can be applied to the problem of measuring software security assurance (and, if so, which metrics). This is the approach of the DHS Software Assurance Program's Measurement WG, for example (see Section 6.1.9.1). The WG's approach is typical in its attempt to "leverage" metrics from the software quality arena (*e.g.,* CMMI) and the information security arena (*e.g.,* CC, SSE-CMM, [121] NIST Special Publication (SP) 800-55, ISO/IEC 27004).

One software security metric that is already in use is Microsoft's Relative Attack Surface Quotient (RASQ), referred to by Steve Bellovin in his Metricon address. Developed with the assistance of CMU to compensate for the lack of common standards for software security metrics, RASQ measures the "attackability" of a system, *i.e.,* the likelihood that an attack on the system will occur and be successful. A RASQ score is calculated by finding the *root attack vectors*,

which are features of the targeted system that positively or negatively affect its security. Each root attack vector has an associated *attack bias* value between 0 and 1, indicating the level of risk that a compromise will be achieved by the attack *vector*, and an *effective attack surface*, indicating the number of attack surfaces within the root attack vector. The final RASQ score for a system is the product of the sum of all effective attack surfaces multiplied by the root vector's attack bias.

According to a study by Ernst & Young, [122] the RASQ for an out-of-the-box Windows 2000 Server running Internet Information Server (IIS) is 341.20, a high attackability rating (based on the number of vulnerabilities found in Windows 2000 Server since its release). By contrast, the RASQ for Windows Server 2003 running IIS was significantly lower—156.60, providing evidence that Microsoft has addressed many of the security shortfalls in the earlier Windows Server version.

The Ernst & Young study notes that RASQ's usefulness is limited only to comparing relative attack surface rates between Microsoft operating system versions, because RASQ relies heavily on parameters that are only meaningful within those operating systems. In addition, the study stressed that RASQ does not measure a system's vulnerability to attack or its overall level of security risk. Nevertheless, building and configuring a system to lower its RASQ score will reduce the number of potentially vulnerable attack surfaces, thereby reducing its overall risk level.

In addition to RASQ, several other software security metrics have been proposed, and are under development, by researchers in industry, government, and academia. The following are some examples (this is by no means a comprehensive list):

- **Relative Vulnerability Metric**—[123] Developed by Crispin Cowan of Novell, Inc., this metric compares the calculated ratio of exploitable vulnerabilities detected in a system's software components when an intrusion prevention system (IPS) is present, against the same ratio calculated when the IPS is not present.
- **Static Analysis Tool Effectiveness Metric**—[124] Devised by Katrina Tsipenyuk and Brian Chess of Fortify Software, this metric combines the actual number of flaws (true positive rate) with the tool's false positive and false negative rates, and then weights the result according to the intended audience for the resulting measurements, *i.e.,* tool vendors wishing to improve the accuracy of the tool, or the auditors attempting to avoid false negatives, or software developers trying to minimize false positives.
- **Relative Attack Surface Metric**—[125] Under development by Pratyusa K. Manadhata and Jeannette M. Wing of CMU, this metric extends CMU's work on the Microsoft RASQ to define a metric that will indicate whether the size of a system's attack surface is proportional to size of the system overall, *i.e.,* if A > B, is the attack surface of A larger than the attack surface of B? The metric will define a mathematical model for calculating the attack surface of a system based on an *entry point and*

exit point framework for defining the individual entry and exit points of a system. These entry and exit points contribute to the attack surface according to their accessibility, attack weight, damage potential, effort, and attackability. The CMU metric is more generic than RASQ and thus applicable to a wider range of software types. In their paper, Manadhata and Wing calculate the attack surface for two versions of a hypothetical e-mail server. However, the CMU attack surface metric is also significantly more complex than RASQ and requires further development before it will be ready for practical use.

- **Predictive Undiscovered Vulnerability Density Metric**—[126] O.H. Alhazmi, Y.K. Malaiya, and I. Ray at Colorado State University are adapting quantitative reliability metrics to the problem of predicting the vulnerability density of future software releases. By analyzing data on vulnerabilities found in popular operating systems, the researchers have attempted to determine whether "vulnerability density" is a useful metric at all, then whether it is possible to pinpoint the fraction of overall software defects with security implications (*i.e.,* those that are vulnerabilities). From this analysis, they produced a "vulnerability discovery rate" metric. Based on this metric, *i.e.,* the quantity of discovered vulnerabilities, they are now attempting to extrapolate a metric for estimating the number of undiscovered (*i.e.,* hypothetical) vulnerabilities.
- **Flaw Severity and Severity-to-Complexity Metric—**[127] Pravir Chandra of Foundstone (formerly of Secure Software Inc.) is researching a set of metrics for: (1) rating reported software flaws as critical, high, medium, or low severity; (2) determining whether flaw reports in general affect a product's market share, and if so whether reporting of low severity flaws reduce market share less than reporting of high severity flaws; and (3) determining whether it is it possible to make a direct correlation between the number and severity of detected vulnerabilities and bugs and the complexity of the code that contains them.
- **Security Scoring Vector (S-vector) for Web Applications**—[128] Under development by a team of researchers from Pennsylvania State University, Polytechnic University, and SAP as a "a means to compare the security of different applications, and the basis for assessing if an application meets a set of prescribed security requirements." The S-vector metric will be used rate a web application's implementation against its requirements for: (1) technical capabilities (*i.e.,* security functions), (2) structural protection (*i.e.,* security properties), and (3) procedural methods (*i.e.,* processes used in developing, validating, and deploying/configuring the application) in order to produce an overall security score (*i.e.,* the S-vector) for the application.
- **Practical Security Measurement (PSM) for Software and Systems**—[129] In February 2004, the PSM Technical WG on Safety and Security began work to tailor the ISO/IEC 15939 PSM framework to accommodate the

measurement of several aspects of software-intensive system security, including (1) compliance with policy, standards, best practices, *etc.*; (2) management considerations (resources/costs, schedule, project progress); (3) security engineering concerns (*e.g.,* conformance with requirements, constraints, security properties, stability, architectural and design security, security of functionality and components, verification and test results); (4) outcome (in terms of security performance, risk reduction, customer satisfaction); (5) risk management considerations (*e.g.,* threat modeling, attack surface, vulnerability assessment, countermeasure design/implementation, and trade-offs); and (7) assurance (in support of assurance cases, independent product evaluations, *etc.*).

- **Measuring Framework for Software Security Properties**—[130] Another framework, this one proposed by the DistriNet research team at Catholic University of Leuven (Belgium). Starting with two lists of security principles and practices—M. Graff and K. van Wyk's *Secure Coding: Principles and Practices* (O'Reilly, 2003) and NIST SP 800-27, *Engineering Principles for Information Technology Security*—the researchers produced an initial short list of five properties that could realistically be measured: (1) smallness and simplicity; (2) separation of concerns; (3) defense in depth; (4) minimization of critical functions/components; and (5) accountability. In future research, the team plans to identify more measurable properties and to identify meaningful metrics that can be used for such measurements, *e.g.,* the Goal Question Metric. [131]
- **Metrics Associated With Security Patterns**—[132] Thomas Heyman and Christophe Huygens, also of Catholic University of Leuven (Belgium), are investigating ways in which security metrics can be directly associated with software security patterns in order to measure the effectiveness of those patterns in securing the software system. In this case, as discussed in Section 5.3.3, the security patterns they are considering are those that describe software security functionality, so it is likely that the metrics the team defines will measure effectiveness of those security functions in terms of policy enforcement or intrusion/compromise prevention.
- **Quantitative Attack-potential-based Survivability Modeling for High-consequence Systems**—In 2005, John McDermott of the NRL CHACS published an extensive paper [133] on his team's work on methodology for using Performance Evaluation Process Algebra (PEPA) to mathematically model and quantify the survivability of a software-based system to what he terms "human sponsored" (rather than stochastic) faults. Quantification is achieved using *mean time to discovery of a vulnerability* [134] as the metric, *i.e.,* the longer a vulnerability goes undiscovered due to lack of activation of the associated fault, the longer the software system is considered to have survived in the undetected presence of that fault.

In October 2006, the Second ACM Workshop on Quality of Protection was held in Alexandria, Virginia; it included a session devoted to software security metrics during which other research into software security metrics was presented. In June 2007, the third Workshop on Assurance Cases for Security will focus on security assurance metrics, including metrics for software security assurance. See Section 5.1.4.3 for more information on these workshops.

To date the efforts of the NIST Software Assurance Metrics and Tools Evaluation (SAMATE) program (see Section 6.1.10) have focused almost exclusively on tools evaluation, although it is expected that they will become more active in pursuing the metrics and measurement portion of their charter.

For Further Reading

"Measurement."
Available from: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/measurement.html

"NIST Information Technology Laboratory Software Diagnostics and Conformance Testing: Metrics and Measures."
Available from: http://samate.nist.gov/index.php/Metrics_and_Measures

Andy Ozment and Stuart E. Schechter (Massachusetts Institute of Technology), "Milk or Wine: Does Software Security Improve with Age?", in *Proceedings of the 15th Usenix Security Symposium,* July 31–August 4 2006.
Available from: http://www.cl.cam.ac.uk/~jo262/papers/Ozment_and_Schechter-Milk_Or_Wine-Usenix06.pdf

Andy Ozment (University of Cambridge [UK]), "Software Security Growth Modeling: Examining Vulnerabilities with Reliability Growth Models", in: *Quality of Protection: Security Measurements and Metrics*, Dieter Gollman, Fabio Massacci and Yautsiukhin, Artsiom.
Available from: http://www.cl.cam.ac.uk/~jo262/papers/qop2005-ozment-security_growth_modeling.pdf

MITRE Corporation, "Making Security Measurable" [portal page], This portal provides a "collection of information security community standardization activities and initiatives" provides a portal to all the different MITRE security that are guided by the informal mission statement on the portal page: "*MITRE's approach to improving the measurability of security is through enumerating baseline security data, providing standardized languages as means for accurately communicating the information, and encouraging the sharing of the information with users by developing repositories.*"
Available from: http://makingsecuritymeasurable.mitre.org/ *or* http://measurablesecurity.mitre.org/.

5.1.6 Secure Software Configuration Management

Because uncontrolled software development activities make it easier for those with malicious intent to tamper with specifications or source code, potentially inserting malicious code into source code or binary executables, all software development artifacts should be kept under configuration management (CM) control. Under strict version control, it becomes difficult for developers, testers, or external attackers to tamper with the source code or executables. CM activities place additional control over a project, intentionally separating the role for managing configurable items from the developer or tester.

Security for software configuration management (SCM) practices and systems has been the subject of a number of papers and guidelines dating at least as far back as 1988, when the NSA's National Computer Security Center (NCSC) published *A Guide to Understanding Configuration Management in*

Trusted Systems (NCSC-TG-006, also known as the "Amber Book"). [135] To date, many of the recommendations in that guide are still valid in general, although the specific technical approaches suggested may have become obsolete. Both the Amber Book and Section B.2 of NIST's SP-800-64, *Security Considerations in the Information System Development Life Cycle* (2004) suggest that SCM requirements include those for methods that preserve the security of software. These methods include—

- Increasing developer accountability for software development artifacts by increasing the traceability of software development activities
- Ongoing impact analyses and control of changes to software development artifacts
- Minimization of undesirable changes that may affect the security of the software.

Since the Amber Book was published, advances in SCM technology and techniques have enabled configuration managers to improve upon the paper-trail-based SCM methods described in the Amber Book. Further improvements are still being researched, recommended, and implemented.

DHS's *Security in the Software Life Cycle* includes a significant discussion of the current state-of-the-art in secure SCM practices, as well as summary of "security enhancements" that can be added to current SCM practices, such as—

- Access control for development artifacts, including but not limited to threat models, and use/misuse/abuse cases; requirements, architecture, and design specifications; source code, binary executables; test plans/scenarios/reports/oracles, code review findings, and vulnerability assessment results; installation/configuration guides, scripts, and tools; administrator and end user documentation; Independent Verification and Validation (IV&V) documents (*e.g.,* C&A documents, CC ST); security patches and other fixes
- Time stamping and digital signature of all configuration items upon check-in to the SCM system
- Baselining of all configuration items before they are checked out for review or testing
- Storage of a digitally signed copy of the configuration item with its configuration item progress verification report
- Separation of roles/access privileges, and least privilege enforcement, for SCM system users
- Separation of roles and duties (developing, testing, *etc.*) within the software development team
- Authentication of developers and other users before granting access to the SCM system
- Audit of all SCM system access attempts, check-ins, check-outs, configuration changes, traceability between related components as they evolve, and details of other work done.

Some other capabilities have been suggested in other sources as necessary for SCM to be truly secure. These include—

- Flexible but carefully controlled delegation [136] of SCM administrator privileges
- No remote access, or remote access only *via* encrypted, authenticated interfaces [137]
- Reporting of differences between security aspects of previous and subsequent versions and releases.

In *Software Configuration Management Handbook*, [138] Alexis Leon identifies security criteria that should be applied to the selection of development artifacts that should, at a minimum, be placed under configuration manager's control as configuration items. These include—

- Items that are mission critical, security critical, safety critical, or high risk
- Items that, if they failed or malfunctioned, would adversely affect security, human safety, or mission accomplishment, or would have a significant financial impact
- Items for which an exact configuration and status of changes must be known at all times.

In practical terms, Leon is suggesting that, at a minimum, the development artifacts of high-consequence software should always be designated as configuration items.

5.1.6.1 Secure SCM Systems

The interest in secure SCM has led to the emergence of secure software version control systems and repositories, such as MKS' (formerly Mortice Kern Systems) MKS Integrity [139] and the Oracle Developer Suite 10g Software Configuration Manager [140]. In addition, the Information Systems Security Operation research team at Sparta, Inc. is working to move secure SCM technology forward with its prototype Secure Protected Development Repository. [141]

As part of the Better SCM Initiative, Schlomi Fish, an Israeli open source programmer, compared the features, capabilities, and technical characteristics of 16 different open source SCM systems. [142] Two of the aspects he compared were directly relevant to the systems' ability to support secure SCM:

1. Ability to assign access permissions to users and to restrict access to the repository based on those permission assignments
2. Ability to limit read and write accesses (check-ins and check-outs) to a single directory.

5.1.6.2 SCM and Nondevelopmental Components

CM of software systems that include acquired or reused components presents a complex challenge. The schedules and frequency of new releases, updates, and security (and nonsecurity) patches, and response times for technical support by acquired or reused software suppliers, are beyond the control of both developers and the configuration manager.

In the case of security patches, developers can never be sure when or even if the supplier of a particular software component will release a needed security patch for a reported vulnerability that might render a selected component otherwise unacceptable for use in the software system. Nor can the developer predict whether a particular security patch may invalidate the security assumptions that other components in the component-based system have about the component to be patched.

Given five COTS components, all on different release schedules, all with vulnerabilities reported and patches released at different times, the ability to "freeze" the component-based software system at an acceptable baseline may confound even the most flexible development team. Developers may have to sacrifice the freedom to adopt every new version of every nondevelopmental component and may, in some cases, have to replace components for which security fixes are not forthcoming with more secure alternatives from other suppliers.

Security enhancements and patches announced by suppliers should be investigated and evaluated by developers as early in the software life cycle as possible to allow sufficient time for risk assessment and impact analysis. This is particularly important for new versions of or replacements for software components that perform security functions or other critical trusted function, because such new versions and replacements will also have implications for system recertification and reaccreditation.

If a particular nondevelopmental component is not kept up to date according to the supplier's release schedule, the developer and configuration manager need to keep track of the minutiae of the supplier's support agreement or contract to determine whether there is a point in time at which a non-updated/non-patched version of the software becomes "no longer supportable." The risks associated with using unsupported software have to be weighed against the risks of adopting new versions or applying patches that have significant impacts on the system's security assumptions (or of adopting an alternative product from a different supplier). A supplier's willingness to support older versions for a fee may be worth negotiating during the product's acquisition, as are custom modifications by the supplier to counteract security vulnerabilities that might not be deemed significant enough, by the supplier, to warrant issuing a standard patch.

Vulnerability reports issued by the United States Computer Emergency Readiness Team (US-CERT) and DoD's Information Assurance Vulnerability Alert (IAVA) program and entries in the NIST National Vulnerability Database (NVD) and the Common Vulnerabilities and Exposures (CVE) represent reliable sources

of information about software product vulnerabilities. The software configuration manager should monitor those sources and download all necessary patches indicated by in the vulnerability reports, and then work with the developers to determine the impact of adopting those patches and the risk of not adopting them.

It is extremely critical for configuration managers to determine and understand how the security of the component-based system may be affected by new behaviors or interfaces introduced by patches applied to individual components. Suppliers often include other features that have no vulnerability-mitigating purpose in security patches, using the patch as a chance to introduce features that will later appear in their next full release of the software. Unfortunately, these features are seldom documented or even announced when delivered with the patch, making the need for impact analysis of such features impossible to recognize.

Secure SCM should track all fixes, patches, updates, and new releases by the suppliers of COTS and OSS components. It will be a challenge to both project and configuration managers to define a release schedule for systems that contain several such components so that as many of those components as possible can be brought "up to date" in terms of security (and other) patches prior to the system's release, in hopes of reducing the amount of patching that will be needed soon after the system has been deployed.

The patch management solution typically used for post-deployment patching may not be flexible enough for patching during the development of component-based software. The patch management solution used during development needs to support the types of impact analyses that must be performed prior to assembly of patched components or integration of software products, including analysis of the patch's impact on other components' security assumptions.

If the system is shipped with custom-developed installation scripts, the developer needs to verify that these scripts do not overwrite security patches already installed on the target hosts. For example, when developing installation scripts for hosts running Microsoft operating systems, the developer can run Microsoft's Baseline Security Analyzer to ensure that the script will not overwrite patches already applied in the intended target environment.

For Further Reading

Klaus Keus and Thomas Gast, "Configuration Management in Security related Software Engineering Processes", in: *Proceedings of the 1996 National Information Systems Security Conference*, 1996.
Available from: http://csrc.nist.gov/nissc/1996/papers/NISSC96/paper035/scm_kk96.pdf

Premkumar T. Devanbu, M. Gertz, and Stuart Stubblebine, "Security for Automated, Distributed Configuration Management", in: *Proceedings of the Workshop on Software Engineering over the Internet at the 21st International Conference on Software Engineering, 1999.*
Available from: http://www.stubblebine.com/99icse-workshop-stubblebine.pdf

Bob Aiello, "Behaviorally Speaking: Systems Security—CM is the Missing Link!!!" *CM//Crossroads.* June 1, 2003.

David A. Wheeler, *Software Configuration Management (SCM) Security,* (May 6, 2005).

Mark Curphey and Rudolph Araujo, "Do Configuration Management During Design and Development", in: *Software Mag: The IT Software Journa,* (October, 2005).
Available from: http://www.softwaremag.com/L.cfm?doc=2005-10/2005-10-config-man

Tom Olzak, "Web Application Security: Application Denial of Service and Insecure Configuration Management", podcast, August 2006.
Available from: http://adventuresinsecurity.com/Podcasts/AISSeries/ShowNotes/AdventuresinSecurity_Episode_37.pdf

5.1.7 Software Security and Quality Assurance

It has become a truism in the software security assurance community that to the extent that software quality (or, more specifically, its constituent properties of correctness and predictability) is a prerequisite of software security, "vanilla" software quality assurance (QA) practices can be expected to aid in the assurance of software security.

Within both the software security assurance and QA communities, the relationship between security assurance and quality assurance for software is being considered from two directions:

- **Addition of Security Risk Management to QA of Secure Software**—Which security risk management practices should be added to the QA process, and how should those additions be accomplished?
- **QA of Secure SDLC Practices**—Which QA practices will be the most useful for assuring the quality of secure software life cycle practices?

Addition of security testing to quality-oriented software testing is a frequent theme of QA-for-secure-software discussions.

These are questions that are being discussed at the highest executive levels of organizations in the software security, software development, and software quality arenas. For example, in September 2006, Ed Adams, Chief Executive Officer of Security Innovations (a software security firm) and founder of the Application Security Industry Consortium, gave a keynote speech at the International Conference on Practical Software Quality and Testing in Minneapolis entitled *What Does Security Mean to My Business: the Quest for Security Testing and ROI.* Six months later, in April 2007, Dr. Sachar Paulus, Chief Security Officer of the SAP Group, gave a keynote address on *Measuring Security* at the Software and Systems Quality Conferences International 2007 in Zurich (Switzerland).

The *Standard of Good Practice,* [143] published by the Information Security Forum, is typical in its approach to QA of secure software and system development practices. The *Standard* states that—

> *Quality assurance of key security activities should be performed during the development life cycle…to provide assurance that security requirements are defined adequately, agreed security controls are developed, and security requirements are met.*

Section SD1.3.2 of the Standard lists key security activities that should be subjected to QA reviews and controls during the SDLC. These activities are—

1. Assessment of development risks (*i.e.,* those related to running a development project, which would typically include risks associated with business requirements, benefits, technology, technical performance, costing, and timescale)
2. Ensuring that security requirements have been defined adequately
3. Ensuring that security controls agreed to during the risk assessment process (*e.g.,* policies, methods, procedures, devices or programmed mechanisms intended to protect the confidentiality, integrity or availability of information) have been developed
4. Determining whether security requirements are being met effectively.

Section SD1.3.3 of the standard goes on to specify how QA of key security activities should be performed. This includes making sure QA starts early in the SDLC and is documented and reviewed at all key stages of the SDLC. Section SD1.3.4 goes on to state that security risk should be minimized through the revision of project plans (including schedule, budget, and staffing) and resources whenever it is discovered that security requirements are not being effectively satisfied, to the extent that development activities should be cancelled if security requirements still cannot be satisfied after such adjustments are made.

In Best Practices on Incorporating Quality Assurance into Your Software Development Life Cycle, [144] Katya Sadovsky *et al.* identify a number of security risk management measures to be included among broader QA activities in the different phases of the SDLC, as shown in Table 5-3.

Table 5-3. Security-Relevant QA Activities Throughout the SDLC

Life Cycle Phase	QA Activities
Requirements	Identification of acceptable levels of down time and data loss Identification of security requirements
Design	Identification and provision of countermeasures to vulnerabilities Design to reflect needs of forensics and disaster recovery activities, and the ability to test for both
Implementation	Automation of nightly code scans, application and database vulnerability scans, and network and configuration scans Documentation and use of manual security test procedures
Testing	Addition of penetration testing and black box testing to functional, compatibility, and regression testing
Deployment	Security training of help desk, system administration, and support staff Identification of security policy issues Establishment of schedule and procedures for system and data backups, and disaster recovery
Operations/Maintenance	Repetition of "routine" security reviews and vulnerability scans Secure change control
Decommissioning	Sanitization of media Proper disposal of hardware and software

For Further Reading

John D. McGregor (Clemson University), *Secure Software. Journal of Object Technology,* (May 2005). Available from: http://www.jot.fm/issues/issue_2005_05/column3

Mark Willoughby, "Quality Software Means More Secure Software", *Computerworld* (March 17 2004). Available from: http://www.computerworld.com/securitytopics/security/story/0,10801,91316,00.html

Ryan English (SPI Dynamics), "Incorporating Web Application Security Testing Into Your Quality Assurance Process", IISresources.com, July 26 2006. Available from: http://www.iis-resources.com/modules/AMS/article.php?storyid=586

5.1.8 Software Life Cycle Models and Methods

Over the years, various attempts have been made to define the most effective model for organizing the technical, management, and organizational activities of the SDLC.

Table 5-4 lists all the major SDLC models, with examples of each.

Table 5-4. SDLC Models

Model	Implementation Examples
Waterfall (Linear Sequential)	Winston Royce, [145] DoD-STD-2167A [146]
Iterative and incremental	Joint Application Design (JAD), [147] MIL-STD-498 [148]
Evolutionary [149]	Tom Gilb's original Evolutionary Life Cycle (which has evolved into Evolutionary Systems Delivery, or Evo), [150] Rapid Iterative Production Prototyping (RIPP), Rapid Application Development (RAD), [151] Genova [152]
Spiral	Barry Boehm [153]
Concurrent Release	Cascade Model or Reparenting Model [154]
Unified Process [155]	Rational Unified Process (RUP), [156] Agile Unified Process (AUP), [157] Enterprise Unified Process (EUP) [158]
Agile	*See Appendix F*

For Further Reading

David Russo (University of Texas at Dallas), *Software Process Planning and Management, Process Models.* Available from: http://www.utdallas.edu/~dtr021000/cse4381/processOverView.ppt

John Petlicki (De Paul University), *Software Development Life Cycle (SDLC).* Available from: http://condor.depaul.edu/~jpetlick/extra/394/Session2.ppt

5.1.8.1 Agile Methods and Secure Software Development

Agile methods are characterized by achievement of customer satisfaction through early and frequent delivery of workable and usable software (*i.e.*, short iterations), iterative development, acceptance of late requirements changes, integration of the customer and business people in the development environment, parallel development of multiple releases, and self-organizing teams.

According to Philippe Kruchten and Konstantin Beznosov, [159] agile methods are iterative in nature; they do not follow the traditional linear development method of requirements development, design, implementation and testing phases. Instead, agile methods repeat the "traditional" development sequence many times. In this way, agile methods can be likened to the spiral model. However, agile methods also emphasize an evolutionary approach to software production ("build a little, test a little, field a little"), with many life cycle activities occurring concurrently.

As characterized by the Agile Manifesto, all agile methods have a single overriding goal: to produce functionally correct software as quickly as possible. For this reason, agile methods avoid life cycle activities that—

- Do not directly involve the production of software, (*i.e.*, other artifacts such as documentation, which is needed for most security evaluations and validations).
- Cannot be performed by members of the software team, and concurrently with other life cycle activities. For example, agile methods do not accommodate IV&V.
- Require specialist expertise beyond that expected from the developers in the software team. Agile methods do not allow for the inclusion of security experts or other non-developer personnel on software teams.
- Focus on any objective other than producing correct software quickly. Agile projects have difficulty incorporating other nonfunctional objectives, such as safety, dependability, and security.
- Must be performed in an environment with constraints on who works on the project, in what role, and under what working conditions. Agile projects do not include or easily accommodate concepts such as separation of roles and separation of duties, least privilege, and role-based access control of development artifacts.

Much discussion and debate has occurred regarding whether it is possible for software projects using agile methods to produce secure software. Appendix F discusses the security issues frequently cited in connection with agile development, as well as counterarguments for how agile methods can benefit software security. Appendix F also describes some efforts to define "adapted" agile methods that are more security supportive.

For Further Reading

Rocky Heckman, *Is Agile Development Secure?*, CNET Builder.au., August 8. 2005. Available from: http://www.builderau.com.au/manage/project/soa/Is_Agile_development_secure_/0,39024668,39202460,00.htm *or* http://www.builderau.com.au/architect/sdi/soa/Is_Agile_development_secure_/0,39024602,39202460,00.htm

M. Siponen, R. Baskerville and T. Kuivalainen, *Extending Security in Agile Software Development Methods, In: Integrating Security and Software Engineering: Advances and Future Visions,* Idea (Group Publishing, 2007).

X. Ge, R.F. Paige, F.A.C. Polack, H. Chivers and P.J. Brooke, "Agile Development of Secure Web Applications", in *Proceedings of the ACM International Conference on Web Engineering,* 2006.

L. Williams, R.R. Kessler, W. Cunningham and E. Jeffries, "Strengthening the Case for Pair-Programming", in *IEEE Software 17, no.4* (July-August 2000): 19-25.

5.1.8.2 Security-Enhanced Development Methodologies

A security-enhanced software development methodology provides an integrated framework, or in some instances, phase-by-phase guidance for promoting security-enhanced development of software throughout the life cycle phases. The methodologies described here either modify traditional SDLC activities, or insert new activities into the SDLC, with the objective

of reducing the number of weaknesses and vulnerabilities in software and increasing software's dependency in the face of threats.

The methodologies described in Sections 5.1.8.2.1 through 5.1.8.2.5 have been used successfully in either multiple real-world development projects or multiple academic pilots. Appendix G provides an overview of how the main security enhancements in these five methodologies map into a standard life cycle model.

Section 5.1.8.2.6 describes additional methodologies that have been developed by researchers, but that have not yet had extensive enough use in pilots or real-world development projects to justify confidence in the researchers' claims for their effectiveness.

For Further Reading

Noopur Davis (CMU SEI), *Secure Software Development Life Cycle Processes,* (Washington, DC US CERT, July 5, 2006).
Available from: https://buildsecurityin.us-cert.gov/daisy/bsi/326.html?branch=1&language=1

5.1.8.2.1 Microsoft Trustworthy Computing SDL

Microsoft's formally established its security-enhanced software development process, the SDL, during its "security pushes" of 2002 [160] as a means of modifying its traditional software development processes by integrating tasks and checkpoints expressly intended to improved the security of the software produced by those processes. SDL's goals are twofold:

1. To reduce the number of security-related design and coding defects in Microsoft software
2. To reduce the severity of the impact of any residual defects.

Microsoft has stated that its software developed under the SDL process initially demonstrated a 50-percent reduction in security bulletins on its major products compared with versions of the same products developed prior to SDL; more recent Microsoft estimates claim up to an 87-percent reduction in security bulletins.

The SDL method proceeds along phases, mapping security-relevant tasks and deliverables into the existing SDLC. The following describes each of the phases:

1. **Requirements**—The team reviews how security will be integrated into the development process, identifies critical objectives, and considers how security features will affect or be affected by other software likely to be used concurrently.
2. **Design**—Key participants—architects, developers, and designers—perform feature design, including design specification that details the technical elements of implementation.
3. **Implementation/Development**—The product team codes, tests, and integrates the software. Developers use security tools, security checklists, and secure coding best practices.

4. **Verification**—The software is functionally complete and enters beta testing. The product team may conduct penetration testing at this point to provide additional assurance that the software will be resistant to threats after its release.
5. **Release**—Software is subject to a final security review. The central security team evaluates the software before shipping and may ask safety check questions on a questionnaire. The final security review (FSR) also tests the software's ability to withstand newly reported vulnerabilities affecting similar software.
6. **Support and Servicing**—The product team conducts any necessary evaluations post-production, reporting vulnerabilities and taking action as necessary, such as updating the SDL process, education and tools usage.

Microsoft has published extensive and detailed information on SDL for the benefit of other software organizations that may want to adopt its approach. Questions have been raised among systems integrators, particularly in government, regarding whether SDL is well suited for noncommercial software projects, given its focus on development, distribution, patching, maintenance, and customer support for turnkey products. SDL does not include activities related to installation and deployment, operation, or disposal, which are key phases in DoD and other government system life cycles. It may, however, be possible to follow SDL in the development of individual components, while following a more integration-oriented system life cycle process for the system as a whole.

Microsoft itself has acknowledged that education and trust of its developers are both key to the success of SDL-guided projects. This points to the need for adding and enforcing criteria to government request for proposals (RFP) and statement of work (SOW) related to individual developer knowledge, expertise, and education and contractor commitment to ongoing developer education and training.

Finally, SDL does not explicitly address business and other nontechnical inputs to the software process, such as inputs driven by Federal, DoD, or individual combatant command, service, or agency policy. The SDL view of security risk management is technically focused, so provisions would have to be made to ensure that security risk management within the SDL-guided software project dovetailed smoothly with less technical risk management methods used in many government software projects, which are defined mainly in terms of C&A concerns.

Even if SDL does not prove to be adaptable for use in government software projects, however, the fact that the methodology has been so well documented provides the acquisition officer with unprecedented insight into the software process followed by one of the government's major software suppliers. This visibility into Microsoft's development process provides exactly the type of assurance evidence acquisition officers should seek to attain from all of their suppliers of critical software.

For Further Reading

Michael Howard and Steve Lipner, *The Security Development Lifecycle,* (Redmond, WA Microsoft Press, 2006).

Michael Howard (Microsoft Corp.), "How Do They Do It?: A Look Inside the Security Development Lifecycle at Microsoft" *MSDN Magazine,* (November, 2005).
Available from: http://msdn.microsoft.com/msdnmag/issues/05/11/SDL/default.aspx

Steve Lipner and Michael Howard (Microsoft Corp.), "The Trustworthy Computing Security Development Lifecycle", [web page], (Redmond, WA: Microsoft Corporation).
Available from: http://msdn.microsoft.com/security/sdl *or*
http://msdn.microsoft.com/security/default.aspx?pull=/library/en-us/dnsecure/html/sdl.asp

Steve Lipner (Microsoft Corp.), "Practical Assurance: Evolution of a Security Development Lifecycle," [web page] in *Proceedings of the 20th Annual Computer Security Applications Conference,* 2004 December 6-10; Tucson, AZ.
Available from: http://www.acsa-admin.org/2004/papers/Lipner.pdf

Microsoft Corp, "Application Security Best Practices at Microsoft: The Microsoft IT Group Shares Its Experiences", January 2003.
Available from: http://download.microsoft.com/download/0/d/3/0d30736a-a537-480c-bfce-5c884a2fff6c/AppSecurityWhitePaper.doc

5.1.8.2.2 Oracle Software Security Assurance Process [161]

Like Microsoft, Oracle Corporation claims to have adopted a secure development process in which all developers are required to follow secure coding standards and use standard libraries of security functions (authentication, cryptography), and to perform extensive security testing that includes penetration testing, automated vulnerability scanning, validations against security checklists, and third-party (government and industry) security IV&V.

As does Microsoft, Oracle claims to ensure that all of its developers are "security aware" throughout the development process (presumably, through training). Oracle products are shipped with secure configuration guidelines, and the company's vulnerability management practices include critical patch deliveries along with fixes to the main code base, both of which are then used to inform revisions to Oracle's security standards in order to reflect their lessons learned from vulnerability discoveries and security incident reports. Indeed, the main emphasis of the Software Security Assurance Process appears to be on security patch distribution.

By contrast with Microsoft's SDL, however, Oracle does not appear to consider its Software Security Assurance Process to be widely usable or adaptable by other software firms, and especially not by software development teams not involved in commercial software product development. It provides some guidance for consumers of its products, primarily in support of database security configuration and vulnerability assessment, and patch management. Guidance for developers of applications or systems that incorporate Oracle databases is not provided.

In contrast with the more than 350-page book by Microsoft's Michael Howard and Steve Lipner detailing their SDL, and dozens of pages of SDL resources on Microsoft's various portals, websites, and blogs, Oracle has published a 12-page whitepaper, along with other information and resources on the Software Security Assurance Process.

5.1.8.2.3 CLASP

Developed by software security expert John Viega, chief security architect and vice-president of McAfee, Inc., CLASP [162] is designed to insert security methodologies into each life cycle phase. CLASP has been released under an open source license. Its core feature is a set of 30 security-focused activities that can be integrated into any software development process. CLASP offers a prescriptive approach and provides ongoing documentation of activities that organizations should perform to enhance security. Some of the 30 key activities include the following:

- Monitor security metrics
- Identify user roles and requirements
- Research and assess security solutions
- Perform security analysis of system design
- Identify and implement security tests.

While the above activities are designed to cover the entire software development cycle, they also enable an organization to skip tasks that are not appropriate to their development efforts. The program also provides users a detailed implementation guide to help determine which activities are the most appropriate including—

- **Activity Assessment**—CLASP Activity Assessment lessens the burden on a project manager and his/her process engineering team by giving guidance to help assess the appropriateness of CLASP activities. The Assessment provides the following information for each activity: information on activity applicability; information on risks of omitting the activity; implementation costs in terms of frequency of activity, calendar time and staff-hours per iteration.
- **Vulnerability Catalog**—CLASP contains a comprehensive vulnerability catalog. It helps development teams avoid or remediate specific design or coding errors that can lead to exploitation. The basis of the catalog is a highly flexible classification structure that enables development teams to quickly locate information from many perspectives: problem types, categories of problem types, exposure periods, avoidance and mitigation periods, consequences of exploited vulnerabilities, affected platforms and programming languages, and risk assessment.
- **CLASP and RUP**—CLASP is available as a plug-in to the RUP development methodology, or as a reference guide to a stand alone development process. In 2005, IBM/Rational released a CLASP plug-in to RUP that included a notation language for diagramming system architectures, and a suggested set of UML extensions for describing system security elements.

For Further Reading

John Viega, "Security in the Software Development Lifecycle", *IBM DeveloperWorks*, (October 15, 2004) Available from: http://www-128.ibm.com/developerworks/rational/library/content/RationalEdge/oct04/viega/#N100AF

5.1.8.2.4 Seven Touchpoints for Software Security

Part II of Gary McGraw's *Software Security: Building Security In* (Addison-Wesley, 2006) is devoted to describing "Seven Touchpoints for Software Security" (see Figure 5-5), which are "lightweight" best practices to be applied to various software development artifacts. (He uses artifacts, rather than life cycle phases, as the basis for his touchpoints to ensure that they are as process-agnostic as possible.) He numbers these practices—or touchpoints—according to what he perceives to be their effectiveness and importance.

Figure 5-5. Seven Touchpoints for Software Security

1. **Static Analysis/Review of Source Code**—Entails use of static analysis tools to detect common vulnerabilities.
2. **Risk Analysis of Architecture and Design**—Includes documenting assumptions and identifying possible attacks, and uncovering and ranking architectural flaws for mitigation. Recurrent risk tracking, monitoring, and analysis should be ongoing throughout the life cycle.
3. **Penetration Testing**—With the architectural risk analysis results driving the selection and implementation of "canned" black-box tests offered by automated application security and penetration testing tools.
4. **Risk-Based Security Testing**—Includes testing of security functionality using standard functional testing techniques, and risk-based security testing of the software as a whole, with test scenarios based on attack patterns.
5. **Building Abuse Cases**—Describe the system's behavior when under attack to clarify what areas and components of the software-based system need to be protected, from which threats, and for how long.
6. **Security Requirements Specification**—Covering both overt functional security (e.g., use of applied cryptography) and emergent security properties (as revealed by the abuse cases and attack patterns).
7. **Security Operations**—After deployment, this includes monitoring the behavior of the fielded software system for indications of attacks and exploits against the software. Knowledge gained through monitoring attacks and exploits should be cycled back into the other touchpoints.

In addition to the seven touchpoints, McGraw identifies an eighth, "bonus" touchpoint: External Analysis, in which analysts from outside the software's development team perform independent security reviews, assessments, and/or tests of the software's design and implementation.

As McGraw repeatedly reminds his readers, none of the 7+1 touchpoints are sufficient on their own to achieve secure software. They are intended to be used collectively.

5.1.8.2.5 TSP-Secure

CMU's SEI and CERT Coordination Center (CERT/CC) developed the Team Software Process for Secure Software Development (TSP-Secure). [163] TSP-Secure's goals are to reduce or eliminate software vulnerabilities that result from software design and implementation mistakes, and to provide the capability to predict the likelihood of vulnerabilities in delivered software. Built on the SEI's TSP, TSP-Secure's core philosophy incorporates two core values:

1. Engineers and managers need to establish and maintain an effective teamwork environment. TSP's operational processes help create engineering teams and foster a team-oriented environment.
2. TSP is designed to guide engineers through the engineering process, reducing the likelihood that they will inadvertently skip steps, organize steps in an unproductive order, or spend unnecessary time figuring out the next move.

The TSP includes a systematic way to train software developers and managers to introduce the methods into an organization. TSP is well-established and in use by several organizations, with observed metrics for quality improvement published in SEI reports. TSP-Secure inserts security practices throughout the SDLC and provides techniques and practices for the following:

- Establishment of operational procedures, organizational policies, management oversight, resource allocation, training, and project planning and tracking, all in support of secure software production
- Vulnerability analysis by defect type
- Establishment of security-related predictive process metrics, checkpoints, and measurement
- Risk management and feedback, including asset identification, development of abuse/misuse cases, and threat modeling
- Secure design process, that includes conformance to security design principles, use of design patterns for avoiding common vulnerabilities, and design security reviews
- Quality management for secure programming, including use of secure language subsets and coding standards, and code reviews using static and dynamic analysis tools
- Security review, inspection, and verification processes, that include development of security test plans, white and black box testing, and test defect reviews/vulnerability analyses by defect type
- Removal of vulnerabilities from legacy software.

TSP-Secure adopters attend an SEI workshop in which they are introduced to the common causes of vulnerabilities and to practices that will enable them to avoid or mitigate vulnerabilities. After training, the team is ready to plan its software development work. Along with business and feature goals, the team

defines the security goals for the software system and then measures and tracks those security goals throughout the development life cycle. One team member assumes the role of security manager and is responsible for ensuring that the team addresses security requirements and concerns through all of its development activities. In a series of proofs-of-concept and pilot projects, researchers at the SEI observed that, through use of TSP-Secure, they were able to produce software that was nearly free of defects.

5.1.8.2.6 Research Models

The following models, developed by academic researchers, have undergone only limited testing in pilot projects. The researchers who have defined these models do not consider them complete and are still working on modifications, refinements, and further piloting.

Appropriate and Effective Guidance In Information Security

Appropriate and Effective Guidance in Information Security (AEGIS) [164] is a software engineering method developed by researchers at University College London. AEGIS first emerged within the software development community as developers observed the multiple complexities of achieving secure design throughout the life cycle. Complications arose as developers undertook projects with conflicting requirements, such as functionality, usability, efficiency, and simplicity. Each of those attributes tends to compete with the others, and with the system's security goals.

AEGIS uses context regeneration based on contextual design and risk analysis. It is aimed at supporting developers in addressing security and usability requirements in system design. The process involves stakeholders in the high-level risk analysis and selection of countermeasures. AEGIS is UML-based, providing a uniform basis on which to discuss and bind the separate areas of usability, risk management, and technical design.

AEGIS uses a spiral model of software development to integrate security and usability with UML. It ensures usability by relying on contextual regeneration, while maintaining functions in asset modeling and risk analysis.

AEGIS' core value is to base all security decisions on knowledge of assets in the system. Some studies have suggested that AEGIS can take place over a series of four design sessions between developers and stakeholders. Depending on the level of security needed and experience of the developers, security experts can assist with the identification of threats and selection/design of countermeasures.

AEGIS proceeds along design sessions, described as follows:

1. **Identifying Assets and Security Requirements**—Using the model of the assets, scenarios are devised in which properties of a security asset are compromised—the resulting analysis defines security requirements.

2. **Risk Analysis and Security Design**—This design session focuses on clarifying the asset model of the system and security requirements.
3. **Identifying the Risks, Vulnerabilities, and Threats to the System**—AEGIS suggests the use of a lightweight risk analysis method that allows the rapid assessment of human and technical risks and threats. The design process involves determining social and technical vulnerabilities, assessing costs and likelihood of attacks, and the selection of countermeasures.

The final output of the design sessions is a design document detailing the architecture of the system together with all the countermeasures that have been agreed on, including training and motivation of personnel. This documentation can serve as the foundation for future iterations.

University College London researchers report having performed a case study that demonstrated AEGIS' effectiveness in the development of software for the European Grid of Solar Observations.

Rational Unified Process-Secure

Rational Unified Process-Secure (RUPSec) was developed by researchers at the Amirkabir University of Technology (Tehran Polytechnic) to build security extensions to the highly popular and successful RUP methodology. The security extensions are aimed at adding and integrating activities, roles, and artifacts to RUP for capturing, modeling, and documenting the threats to and security requirements of a software system, and to ensure that those security requirements are addressed in all subsequent development phases (*i.e.,* design, implementation, and testing).

To accomplish these goals, RUPSec proposes new artifacts and activities that build on the use case-driven approach of RUP. RUPsec proposes adding a misuse case model extension to RUP to help developers through the iterative process of defining or reusing misuse cases and then developing solutions to counter each threat identified in those misuse cases.

Specifically, RUPSec adds security extensions to the following RUP activities:

- **Maintain Business Rules**
- **Find Business Actors and Use Cases**—document security aspects and use cases
- **Find Business Workers and Entities**—define access level of workers to entities
- **Define Automation Requirement**—capture business security requirements from security policy
- **Detail Software Requirements**—refine security requirements.

For Further Reading

Pooya Jaferian et al, (Amirkabir University of Technology/Tehran Polytechnic), "RUPSec: Extending Business Modeling and Requirements Disciplines of RUP for Developing Secure Systems", *presentation at the 31st IEEE EuroMicro Conference on* Software Engineering and Advanced Applications, August 3–September 2005, 232-239.
Available from: http://ce.aut.ac.ir/~jaferian/files/publications/WECRUPSec.doc ***or*** http://ieeexplore.ieee.org/xpls/abs_all.jsp?tp=&arnumber=1517747&isnumber=32500

Mohammad Reza Ayatollahzadeh Shirazi et al, (Amirkabir University of Technology/Tehran Polytechnic), "RUPSec: An Extension on RUP for Developing Secure Systems", *World Enformatika Society Transactions on Engineering, Computing, and Technology,* (February 4, 2005)
Available from: http://www.enformatika.org/data/v4/v4-51.pdf

Secure Software Development Model

Secure Software Development Model (SSDM) was defined by Simon Adesina Sodiya, a researcher at the Nigerian University of Agriculture [165] in response to security problems in the development processes by software organizations. SSDM integrates security engineering into the software development process through use of a unified model that comprises a number of existing techniques for producing secure software.

SSDM defines a secure four-phase workflow within the software engineering process:

1. **Security Training**—Provide stakeholders adequate security education in software development. Training concepts include security awareness, knowledge of attackers on previous related applications, understanding of attackers' interests, and knowledge of secure development practices.
2. **Threat Modeling**—Provide users an understanding of the attributes of the software, identify attackers within the given operating environment and their goals and techniques, and identify possible future patterns and behaviors. The threat model is used to construct an attack profile. Vulnerabilities are then identified based on the attack history and threat model results.
3. **Security Specification**—Lists attacks, defines potential protection measures against issues such as development errors, guides security implementation, assists monitoring of security postures, and helps make system adaptable to the changing landscape of security.
4. **Review of Security Specification**—Ensures design content of the software is in accordance with its security specification. This phase also includes penetration testing that initiates all identified attacks and future attack patterns online into the software.

SSDM has been used in real-world applications. An accounting program in Nigeria known as "Standard Accounting" successfully implemented SSDM engineering principles. The components of Standard Accounting are general ledger, sales and purchase ledger, and payroll. The SSDM team identified 129 security breaches and classified them under three categories according to the

security properties compromised: confidentiality, integrity, and availability. The old package was then replaced with a system developed according to SSDM principles. In the year since that replacement occurred, no security incidents have been recorded. The Nigerian University of Agriculture researchers are currently working to further improve the SSDM methodology.

Waterfall-Based Software Security Engineering Process Model

In their paper *Software Security Engineering*, [166] Mohammad Zulkernine and Sheikh Ahamed describe a waterfall-based software security engineering process model based on the standard waterfall model first suggested by Dr. Winston W. Royce, [167] and later modified by Barry Boehm in his groundbreaking book *Software Engineering Economics*. The Zulkernine/Ahamed model is based on the original five-phase version of the waterfall in which some phases are renamed, and the implied recursions between phases suggested by both Royce and Boehm are eliminated (*i.e.,* the simplified waterfall flows in only one direction) as are the verifications, tests, and validations suggested by Royce and integrated into each of the life cycle phases in Boehm and in DoD-STD-2167A (possibly the most widely used specific waterfall model). Zulkernine and Ahamed do not explain the reason for their renamings and omissions.

Zulkernine and Ahamed suggest that security engineering activities and artifacts that should be added to the functionality-focused development activities at each phase of their adapted waterfall model. The security engineering activities they suggest are identified in Table 5-5.

Table 5-5. Waterfall-Based Software Security Engineering Process Model

Life Cycle Phase	Added Security Engineering Activities
System Engineering	▶ Analyze security threats ▶ Define security needs and constraints of software elements of the system ▶ Produce informal security requirements
Requirements Specification	▶ Identify attack scenarios ▶ Produce formal security specification ▶ Reconcile and integrate security and functional requirements ▶ Produce combined formal software and security requirements specification
Software Design	▶ Define scenario-based compositional architecture, including attack scenarios

Table 5-5. Waterfall-Based Software Security Engineering Process Model - *continued*

Life Cycle Phase	Added Security Engineering Activities
Program Implementation	▶ Identify security flaws ▶ Refactor code to address security flaws and add self-protection ability to control vulnerabilities against attacks
System Operation	▶ Monitor and detect unexpected behaviors, failures, and intrusions ▶ Generate new attack specifications ▶ Reconcile and integrate new attack specifications into formal requirements specification

With the exception of "Identify security flaws" in the Program Implementation phase (implying the need for code review and/or security testing) and "Detect of unexpected behaviors, failures, and intrusions" in the final System Operation phase, Zulkernine and Ahamed not only exclude testing, verification, or validation from their waterfall model but from their list of security engineering activities. This omission is interesting because security reviews, tests, verifications, and validations (starting early in the software life cycle) are integral to most other "security-enhanced" methodologies.

Proposed Security Extensions to MBASE [168]

In 1999, the University of California's (USC) Center for Software Engineering introduced Model-Based Architecting and Software Engineering (MBASE), an extension of the spiral life cycle model also introduced by USC in 1983. [169] MBASE is intended to be a comprehensive, risk-driven methodology that combines four models of software-intensive systems—a property model, a process model, a product model, and a success model—and demonstrates their feasibility and compatibility. MBASE is a documentation-based process that comprises two phases:

1. Inception and Elaboration (IE)
2. Construction, Transition, and Support (CTS).

A 2004 case study by USC researchers in which MBASE was evaluated in terms of its effectiveness for addressing software security risk revealed its shortcomings when used in this context. As a result, the researchers proposed adding a set of security extensions to the documents produced during the first (IE) phase of MBASE. (It was felt that security extensions to the CTS phase documentation would not be helpful.)

The two major documentation milestones produced during the IE phase are a set of documents that define the system's Life Cycle Objectives (LCO) (produced during Inception), and elaborations, refinements, and risk mitigations of those documents to produce the Life Cycle Architecture (LCA).

The security extensions to MBASE, then, take the form of security extensions to the LCO and LCA documents.

It does not appear that the proposed MBASE extensions have been adopted by USC; no new "security-enhanced" version of MBASE has yet been published.

Secure Software Engineering

Secure Software engineering (S2e)[170] is a process-based methodology developed by reverse engineering expert Dr. Thorsten Schneider, founder and managing director of the International Institute for Training, Assessment, and Certification (IITAC), and of Dissection Labs. S2e leverages existing whitehat and blackhat knowledge areas and methodologies, while also incorporating management and business considerations into what Schneider calls "development philosophy-independent," adaptable set of processes to be integrated into an organization's existing SDLC process. The objective of S2e is to significantly reduce the number of vulnerabilities in the software that results. S2e is also intended to benchmarked using a CMM such as ISO/IEC 21827 SSE-CMM.

In defining the secure software processes to be added to the SDLC, S2e considers four viewpoints:

1. **Software Engineering Viewpoint**—Regardless of what life cycle model or methodology (what Schneider terms "philosophy") the organization uses (*e.g.,* waterfall, spiral, incremental-iterative, agile, *etc.*), every software project can be reduced to four basic activities (or phases): (1) requirements; (2) design; (3) implementation; (4) testing. Security cannot be addressed at any single phase, but must be addressed at all phases.
2. **Management Viewpoint**—Management needs to expend more effort on achieving one goal: the security optimization of the software to mitigate its potential vulnerabilities over the longest possible period of time.
3. **Whitehat Viewpoint**—By applying known attack patterns whitehats are able to optimize their protection processes. This includes integrating blackhat processes into their own defined and managed processes, which include—
    a. Threat modeling
    b. Security modeling
    c. Malware analysis
    d. Secure design testing
    e. Secure code construction
    f. Secure code testing
    g. Integration and development of protections
    h. Secure software syllogisms
    i. Secure software patterns.
4. **Blackhat viewpoint**—Given that most recent developments in attack techniques for compromising software systems originated from blackhats, it makes sense to direct the constructive, legitimate use of such techniques toward the production of software that will be

more robust against those techniques. Blackhat techniques to be standardized into the S2e process include—

- Analysis and management of blackhat information
- Attack modeling
- Penetration and attack engineering
- Exploitation
- Reverse code engineering
- Shell code development
- Malicious logic.

According to Schneider, the S2e processes are being evaluated in a small-scale software development project that involves server-side development. In addition, subsets of specific processes are being evaluated by an organization of more than 700 developers. Both evaluations are still in progress, so case studies or empirical data regarding the effectiveness of S2e are not yet available. However, Schneider has such confidence in the methodology that the IITAC has set up a Secure Software Engineering portal http://www.secure-software-engineering.com as well as the *Secure Software Engineering Journal*, which began publication this year (2007) to promote it, at least indirectly.

For Further Reading

Thorsten Schneider, "Secure Software Engineering Processes: Improving the Software Development Life Cycle to Combat Vulnerability", *Software Quality Professional* 8, no. 1 (December 2006). Available from: http://www.asq.org/pub/sqp/past/vol9_issue1/sqpv9i1schneider.pdf

5.2 Requirements for Secure Software

Requirements for security functionality in software-intensive systems are often confused with requirements for secure software. The first category includes functions that implement a security policy, such as an information security policy in a software-intensive information system. These are the functional areas of access control, identification, authentication and authorization, and the functions that perform encryption, decryption, and key management These functions prevent the violation of the security properties of the system or the information it processes, such as unauthorized access, modification, denial of service, disclosure, *etc.* They are also known as security service requirements.

The second category of requirements directly affects the likelihood that the software itself will be secure. These are the nonfunctional—or property—requirements that collectively ensure that the system will remain dependable even when that dependability is threatened. These requirements are often directed toward reducing or eliminating vulnerabilities in the software. They are more closely tied to process, to the software development plan, and to project management direction. These requirements deal with things like input validation, exception handling, sandboxing, *etc.*

Microsoft calls the second category *Security Objectives* and recommends that developers define security objectives and security requirements early in the process. Security objectives are goals and constraints that affect the confidentiality, integrity, and availability of the data and the application software. [171]

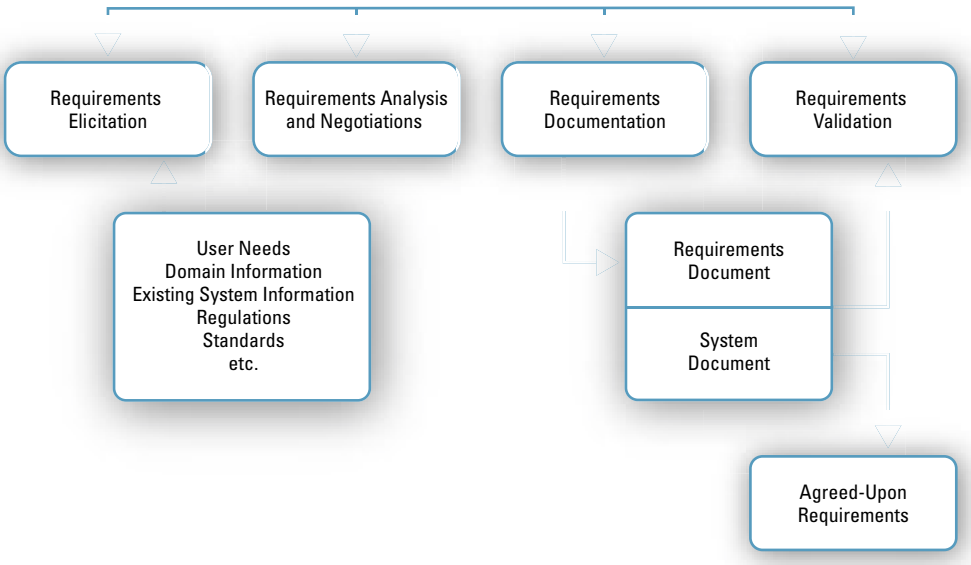
Significantly influencing both categories of requirements is an understanding of the threat environment, calculations of risk, and identification of mitigation strategies.

5.2.1 Software Requirements Engineering

The software requirements phase covers everything that needs to occur before design begins. Inputs may include the system specification, with functions allocated to software, and a software development/project plan. Inputs come from many sources, including the customer, users, management, QA and test groups, and as systems requirements allocated to software. The main output of the requirements phase is the requirements specification that defines both functional and nonfunctional aspects of the software. The end of phase is marked by acceptance of the specification by the customer and other stakeholders. The requirements may be modified in the course of the development effort as needs for changes are identified.

Figure 5-6 shows a high-level view of the tasks and artifacts involved in the requirements phase of a software development effort.

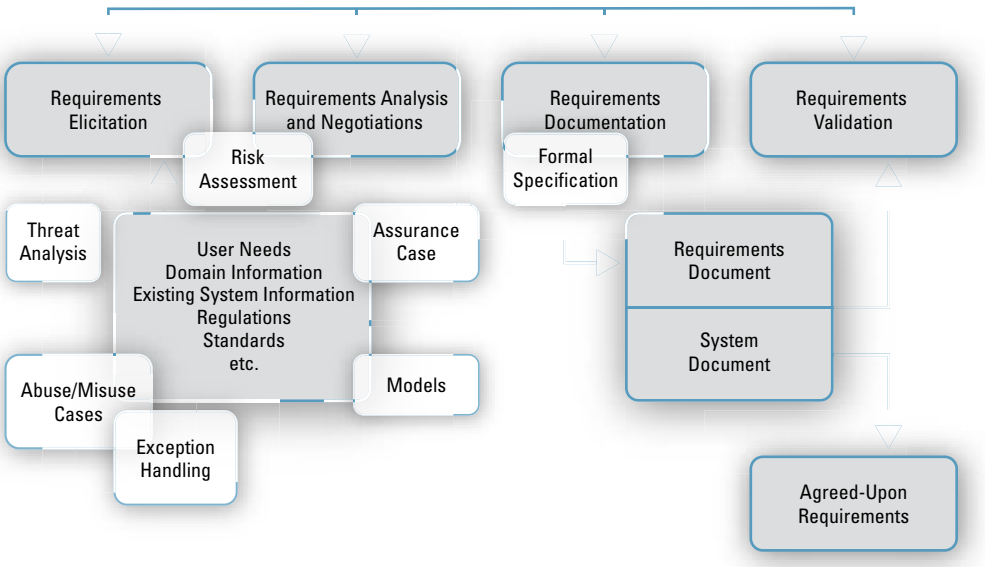
Figure 5-6. Requirements Engineering for Survivable Systems [172]



There is a rich body of work and results on requirements engineering, as well as tools and techniques to support the processes. Unfortunately, most of this work does not explicitly consider security. Work that does is usually concerned with security requirements in the sense of requirements engineering for the security functionality in a system, such as access controls. Our concern, however, is the engineering of requirements for security as an emergent property of a software system. Although the implementation of security functionality may coincidentally satisfy many requirements for security as a software property, different analyses will be needed to attain these different objectives.

The purpose of this section is to highlight differences between requirements phase activities as they are normally performed and how they could be adapted and augmented to increase the security of the software under development. Figure 5-7 shows examples of additional activities and artifacts for increasing software security overlaid on the generic requirements process.

Figure 5-7. Secure Software Additions to Requirements Engineering Process



For Further Reading

"Requirements Engineering" [web page], (Washington DC: US CERT)
Available from: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/requirements.html

5.2.1.1 Traditional Requirements Methods and Secure Software Specification

To begin with, the source of security requirements problems are not limited to the security domain. Many software security problems originate in the inadequate or inaccurate specification of the requirements for the software or from the mismatch between the interpretation of the requirements during

development and their actual intent. [173] Requirements engineering is hard to do, and harder to do well. The number of requirements needed to adequately define even a small software project can be very large. That number grows exponentially with the effects of large size, complexity, and integration into evolving operational environments.

Some of the security-specific issues involved in requirements engineering that have been cited by various authors include—

- The people involved are not likely to know or care (in a conscious sense) about nonfunctional requirements. They will not bring them up on their own. Stakeholders have a tendency to take for granted nonfunctional security needs.
- Traditional techniques and guidelines tend to be more focused on functional requirements.
- Security controls are perceived to limit functionality or interfere with usability.
- It is more difficult to specify what a system should not do than what it should do.
- Stakeholders must understand the threats facing a system in order to build defenses against them, but the threat environment faced by the delivered system will be different from the threat environment existing at the time of requirements development because the threats are evolving.
- The users who help define the system are not typically the abusers from whom the system must be protected.
- It requires creativity, experience, and a different mindset to define the bad things that could happen.
- There may be security issues involving development team personnel, including access to security-competent development or integration contract services. Parts of the development may be outsourced or offshored. Risk assessment and/or vulnerability information may be classified, precluding access by developers without clearances.
- There may be lack of understanding of how a component fits into the larger system, *i.e.,* a component that may behave securely when operating in stand alone made may not behave securely when assembled with other components (see Section 5.1.1.2 for more information on security issues of component-based development).
- There are questions of who will be held accountable (professionally and/or contractually/legally) for software that turns out to be insecure. Currently, developers bear (at least some) responsibility for failed functionality (*i.e.,* functional requirements that are not satisfied in the implemented system). Can this already-accepted accountability be expanded to include comparable failures in security (*i.e.,* security was specified, but not achieved in the implemented system)? What about responsibility for not specifying security requirements in the first place,

even when the intended execution environment for the software is known to be high-risk? If accountability can be enforced, who should be held responsible? The developer? The accreditor?

Secure software requirements engineering can usefully draw on work in related disciplines, such as—

- Software safety, *e.g.,* medical devices, nuclear power
- Software survivability (also referred to as fault tolerance), *e.g.,* telephone systems
- Embedded systems, *e.g.,* space applications, weapons systems
- Software reliability (unreliable software contains defects that become vulnerabilities)
- Information systems (in particular, requirements for availability and integrity, also accountability and nonrepudiation).

The techniques and solutions in these disciplines are not sufficient for software security as is, because they tend to defend against natural dangers, *i.e.,* hazards, whereas software must defend against malicious users and hostile attacks, *i.e.,* threats. The threat landscape is more dynamic and much less predictable than hazards are.

Security has generally been included with the other nonfunctional requirements for software, such as performance and quality requirements. During the requirements phase, nonfunctional security requirements are captured and defined as attributes of the software, but the process does not end there. The nonfunctional security requirements need to be mapped to functional requirements so that they can be built into the software and tested appropriately. For example, a requirement such as "The software must not be susceptible to buffer overflows" could be mapped to functional requirements for input validation and for use of only type-safe/memory-safe programming languages. Some of the use case adaptations to abuse and misuse cases (see Section 5.2.3.2.1) make this mapping step explicit.

By mapping the nonfunctional to functional requirements, the security requirements become a part of the overall requirements analysis process. Potential conflicts with other functional requirements can be identified (and resolved). Not all of the nonfunctional requirements may be addressable as software functions, and so may need to be allocated back to the system level. Certain types of pervasive security requirements may be best addressed as part of the system's security policy, *e.g.,* "All input from users must be validated to ensure its conformance with the expected format and length of that type of input."

5.2.1.2 Security Experts on the Requirements Team

Although not all of the roles listed below will have the same level of involvement, all should have input and should be able to review the results. Unlike the conventional requirements team members—stakeholders, users, developers, project managers, software architects, quality assurance and testing—these additional roles serve more as consultants and advisors:

- **Security Engineers**—Security engineers can serve as liaisons to the systems-level, to resolve issues of what security can be incorporated into software and what is outside.
- **Risk Analysts**—Risk assessment and threat analysis are key sources for requirements that affect the security of the software, especially for later in the analysis and prioritization of security.
- **Certifiers and Accreditors**—Software that is to be certified or accredited must meet the requirements for the particular certification's standard or assurance level.
- **Information Assurance (IA) Experts**—IA experts with experience in defending and repairing existing software can provide insight on how to define and build new software.

5.2.2 "Good" Requirements and Good Security Requirements

Table 5-6 contains a generally-accepted set of characteristics that good requirements possess, coupled with how those characteristics may be expanded or expressed in requirements for secure software.

Table 5-6. Characteristics of Effective Requirement Statements

	Conventional concepts of goodness	Software assurance community concepts of goodness
Correctness (Does It Say What It Means?)	▶ The functionality to be delivered is accurately described ▶ There are no conflicts with other requirements	▶ Requirement describes how it should behave when it fails—exception handling ▶ Constraints on other requirements may be identified
Feasibility (Can It Do What It Says?)	▶ Each requirement can be implemented within known capabilities and limitations of the system and its environment	▶ Threats are included in the list of known limitations
Necessary (Is It Needed?)	▶ Requirement documents something that customers really need or is required for conformance	▶ Functionality is required for secure software

Table 5-6. Characteristics of Effective Requirement Statements - *continued*

	Conventional concepts of goodness	Software assurance community concepts of goodness
Supportable (Why is the Requirement Important?)	▶ Rationale to support the need/ purpose for the requirement is documented	▶ The justifications are important for building the assurance case ▶ Documenting the security rationale helps ensure the requirement is not assigned too low a priority to be implemented
Prioritized (Is It More or Less Important Than Others?)	▶ How essential is each requirement, feature, or use case to a specific product release is defined ▶ Typical prioritization: HIGH = next product release; MEDIUM = defer to future release; LOW = nice to have, but not a need	▶ Threat analysis and risk assessments will affect the prioritization
Unambiguous (Can It Be Interpreted in Only One Way?)	▶ Only one interpretation can be drawn	
Verifiable (Can It Be Tested, Traced and Measured?)	▶ Ability to test or inspect that each requirement is properly implemented ▶ Ability to trace each requirement between phases is implemented ▶ Ability to measure/demonstrate that each requirement has been met is implemented	▶ Tests are needed to demonstrate that certain behaviors are not implemented, that x doesn't happen (in the face of y) ▶ Tests need to demonstrate that constraints are also met.

Software requirements by and large are requirements for functionality, and in some cases, they are requirements for performance constraints (*e.g.*, "the function must be completed within *n* microseconds"); they tend to be expressed in positive terms, *e.g.*, "the system must…"

By contrast, security requirements, particularly in software security, tend to be either constraints on functionality or a statement of a needed property (or attribute), that will be manifested by a software behavior. At least initially during requirements capture, they will often be stated in negative terms.

As with all software, the requirements process for secure software may require multiple iterations of elicitation and analysis. Requirements engineers should not feel constrained by conventions that avoid the statement of negative requirements or the inclusion of non-actionable requirements. These need to be captured and then analyzed and converted to actionable, positive, functional requirements. For example, the requirement that "Software must not be susceptible to buffer

overflows," is both negative and non-actionable, but is necessary for software to be secure. It would therefore need to be expressed as requirements for functionality that prevents buffer overflows from happening, *i.e.*, validation of input, good memory allocation and management, exception handling functionality, *etc.*

5.2.2.1 Where Do Software Security Requirements Originate?

DHS's *Software Assurance* (CBK) describes several categories of needs that should be mined for security requirements:

- **Stakeholders' Security-Related Needs—**
  - Asset protection needs
  - Needs arising from the threat environment
  - Needs arising from interfaces
  - Usability needs
  - Reliability needs
  - Availability needs
- **Attack Resistance and Survivability Needs—**
  - Sustainability needs
  - Validation, verification, and evaluation needs
  - Deception needs
  - Certification needs
  - Accreditation needs
  - Auditing needs.

A need is not a requirement. Once identified, needs must be analyzed, prioritized, and specified before they become requirements.

Table 5-7 shows how conventional methods for requirements elicitation and analysis are being used to support specification of software security requirements.

Table 5-7. Extensions to Conventional Requirements Methods

Conventional Method	Security Extension/Example
CONOPS	"Development Work on a CONOPS for Security" [174]
Quality Function Deployment	*(no corollary)*
Functional Decomposition	Identification of threats along data flows
Object-Oriented Decomposition	Aspect-oriented methods
Use Case Development	Misuse and abuse cases
Trade Studies	Identification of threats and attacks unique to the application domain (such as web application)

Table 5-7. Extensions to Conventional Requirements Methods - *continued*

Conventional Method	Security Extension/Example
Simulations	"Simulation-Based Acquisition" is a major DoD thrust [175]
Modeling	Threat modeling, also aspect-oriented modeling
Prototyping	Data-oriented methods
(no corollary)	Threat/attack trees [MS TR-2005]

In 2003, A. Rashid [176] *et al.* proposed the use of aspect-oriented software development for mapping one or more functional requirements to each nonfunctional requirement that affects the functional requirements, with security treated as just one of several types of nonfunctional requirement.

Axel van Lamsweerde *et al.* [177] proposed the KAOS method for modeling security and safety requirements through the use of anti-goals (the converse of goals, such as availability and integrity). Anti-goals describe the vulnerabilities that make satisfaction of the system's stated goals impossible. The resulting security requirements are expressed in terms of "avoiding" anti-goals in order to eliminate the vulnerabilities that would prevent the system from achieving its goals.

Ian Alexander [178] takes a similar approach of avoiding vulnerabilities, but relies on misuse cases instead of "anti-goals," as do G. Sindre and A.L. Opdahl, [179] while John McDermott [180] substitutes abuse cases for misuse cases. By contrast, H. In and Barry Boehm have adapted the Win-Win quality management framework [181] to include security requirements. C.L. Heitmeyer made similar adaptations to NRL's Software Cost Reduction (SCR) methodology. [182]

Charles Haley *et al.* [183] propose representing security requirements as crosscutting threat descriptions, which then aid in the composition of these requirements with the system's functional requirements; the resulting specification defines a set of constraints on the functional requirements. These constraints are the "security requirements." Haley's work is in response to what he sees as several problems with the other approaches to integrating security requirements into overall system requirements:

1. There is no single definition of what is meant by "security requirement." In some cases, the term "security requirements" refers to requirements for security functionality. In other cases, it refers to "constraints" on functionality. In yet other cases, it pertains to the need for functionality to operate consistent with a governing security policy. In addition, security requirements are increasingly being seen as "anti-requirements" or "anti-patterns," defined in terms of avoiding a vulnerability that would prevent the system from satisfying its other, nonsecurity requirements. This profusion of definitions of security requirement makes it difficult to determine which requirements engineering approach is best suited for a particular requirements engineering problem.

2. Security requirements, when they are defined, are inconsistent, and the criteria for determining whether they have been satisfied are difficult to understand.
3. There is still no clear path for deriving security requirements from business goals or needs.

John Wilander and Jens Gustavsson [184] agree with Haley that the "current practice in security requirements is poor." They further observe that "Security is mainly treated as a functional aspect composed of security features such as login, backup, and access control. Requirements on how to secure systems through assurance measures are left out." Wilander and Gustavsson elaborate on this observation by identifying some key defects they have observed in the majority of security requirements they studied:

- Most security requirements are either absent or poorly specified because of an inadequate understanding by most requirements engineers of security in general, and security as a property in particular.
- The selection of security requirements is inconsistent, with the majority being functional requirements. Across those functional security requirements, there is further inconsistency, because some requirements are included while other requirements, often requirements on which those included directly depend, are omitted.
- The level of detail across requirements statements is inconsistent.
- Nonreliance on security standards results in unnecessary custom solutions that are usually limited in effectiveness because of the inadequate security expertise of the developers.

In later work, [185] Haley describes a security requirements framework that addresses these problems by combining what he identifies as a requirements engineering approach with a security engineering approach. In the framework, security goals are defined that express the need to protect system assets from harm by threats. These goals are "operationalized" into security requirements, which are, in essence, constraints on functional requirements that result in the necessary level of protection. Finally, the framework specifies the need to develop "satisfaction arguments" that demonstrate the system's ability to function consistent with its security requirements.

Two types of documentation are produced in the requirements phase:

1. The specification, which is the end-product of the work.
2. The justifications that document the process, and the decisions that led to the specification. The justifications are used in building the assurance case, but are also useful throughout the development whenever requirements changes are proposed.

Requirements remain important throughout the development life cycle. Table 5-8 shows the security interpretation of post-requirements phase requirements-related activities and concerns.

Table 5-8. Requirements Throughout the Life cycle

Requirements Concern	Security Interpretation
Traceability, *i.e.*, making sure all the elements of the design and code are derived from some requirement(s)	Prevents delivered software from having unspecified functions
Verification that the design and code implement all the requirements	Ensures the delivered software will have all the security features and properties that were specified
Change control and management	Analyzes how proposed changes may affect security, directly or *via* a ripple effect
Update of documentation to reflect changes	Keeps the threat analysis updated to reflect changes in threat environment

For Further Reading

Paco Hope, Gary McGraw and Annie I. Anton, *Misuse and Abuse Cases: Getting Past the Positive.* IEEE Security & Privacy, (March-April 2004) 2(3).
Available from: http://www.cigital.com/papers/download/bsi2-misuse.pdf

5.2.2.1.1 Policies and Standards as a Source of Software Security Requirements

Software security requirements may be driven by existing information security, system security, or software policies, some of which may be mandated by legislation. For DoD software, such policies include DoD Directive 8500.1 and DoD Instruction 8500.2, DCID 6/3, DoD's policies on use of mobile code and open source software, and policies mandated by Federal laws, such as the Sarbanes-Oxley Act, the FISMA, and the Healthcare Information Portability and Accountability Act (HIPAA). (Other legislation that may be relevant is listed in Section 6.4.)

These policies and laws vary in scope and purpose, but each provides vital statements regarding what is or is not permitted, and any exceptions or mitigations that may be applicable. Each policy or law provides statements that can be translated into explicit or implicit requirements for the functions, controls, and properties of the software elements of information systems.

The quantity and scope of requirements that can be derived from policy or law depend on the scope and purpose of that policy or law. The language of such mandates is usually very limited in terms of explicit discussion of or reference to secure software. It is more likely that language pertaining to secure information and/or information systems will need to be interpreted and adapted in order to establish secure software requirements.

For example, there are three items in DoD Instruction (DoDI) 8500.2 that are directly relevant to software security:

- Security Design and Configuration Integrity: DCSQ-1, Software Quality
- Security Design and Configuration Integrity: DCSS-2, System State Changes
- Security Design and Configuration Integrity: DCMC-1, Mobile Code.

DCID 6/3 has a slightly larger number of software security relevant statements (*e.g.,* "Mobile Code and Executable Content," "Protected Hardware, Software and Firmware"). The document correlates requirements to system protection levels (PL) and then categorizes policy statements according to their applicability at each PL.

In the NIST FIPS Publication 200, *Minimum Security Requirements for Federal Information and Information Systems*, the following items can be construed as relevant to software assurance:

- **System and Services Acquisition**—"Organizations must…(ii) employ system development life cycle processes that incorporate information security considerations; (iii) employ software usage and installation restrictions; and (iv) ensure that third-party providers employ adequate security measures to protect information, applications, and/or services outsourced from the organization."
- **System and Communications Protection**—"Organizations must…(ii) employ architectural designs, software development techniques, and systems engineering principles that promote effective information security within organizational information systems."
- **System and Information Integrity**—"Organizations must: (i) identify, report, and correct information and information system flaws in a timely manner; (ii) provide protection from malicious code at appropriate locations within organizational information systems."

NIST SP 800-53, *Recommended Security Controls for Federal Information Systems*, elaborates on the FIPS 200 requirements, but it does not represent a policy or mandated standard.

At this point, more extensive, useful language is likely to be found in published guidance documents, such as NIST SP 800-53 and the Defense Information Systems Agency (DISA) Application Security Project's *Application Security Checklist* [186] and *Reference Set of Application Security Requirements*.

As with FIPS 200, other information security standards, most notably the CC, [187] represent another potential source for software security requirements, though only althrough careful interpretation. (Refer to the Section 5.1.4 discussion of the shortcomings of the CC with regard their lack of language pertaining to software security assurance.)

5.2.3 Methods, Techniques, and Tools for Secure Software Requirements Engineering

This section describes methods that are being used in real-world development projects or very successful research pilots (and that are therefore considered ready for technology transfer), to address security in software requirements, along with tools that implement or support the techniques.

Looking critically, many of these techniques seek the same information. They may perform similarities, but use different names, or have different emphases. It is not a matter of choosing one in favor the others. Different techniques will result in different views of the problem and complement each other. For example, one needs to know about threats in order to define defenses. Paco Hope *et al.*[188] suggest using attack patterns to drive the search for misuse cases. Some techniques focus more on elicitation, others on analysis, and still others on specification, documentation, verification, or management of requirements across the life cycle.

The DHS BuildSecurityIn portal includes results of work (including case studies) done to define a method for choosing among the available techniques. Selection criteria include such elements as the learning curve for the technique and the availability of computer-aided software engineering (CASE) tool support. [189]

The adoption of one technique over another may depend on the software methodology used. Some methodologies require techniques that produce artifacts that are more easily reused or transitioned into what is needed for later phases, *e.g.,* as input to some CASE tool. In some cases, the essential process defined in the methodology is the same, but the specific implementation, terminology, notation, or degree of rigor required is different.

5.2.3.1 Threat, Attack, and Vulnerability Modeling and Assessment

Modeling is a well-known approach for discovering and learning about the requirements for software. It provides a way to envision the workings and interactions of the proposed software within its intended environment. The more closely the model reflects the intended environment, the more useful the modeling approach becomes. Therefore, secure software development benefits from modeling that explicitly incorporates security threats.

The primary issues in modeling are—

1. Doing it well
2. Doing it thoroughly enough
3. Knowing what to do with the results, *e.g.,* how to transform the analysis into a metric and/or otherwise usable decision point.

There are several threat, attack, and vulnerability modeling tools and techniques. Microsoft, in particular, has emphasized this type of modeling in its secure software initiative. In addition, multiple methodologies have emerged to enable developers to conduct threat modeling and risk assessment of software. These are described below.

For Further Reading

Suvda Myagmar, Adam J. Lee and William Yurcik, "Threat Modeling as a Basis for Security Requirements", (presented at the symposium *etc.*) Symposium on Requirements Engineering for Information Security, August 29, 2005.
Available from: http://www.projects.ncassr.org/sift/papers/sreis05.pdf

5.2.3.1.1 Microsoft Threat Modeling

The core element of Microsoft's program is the threat model—a detailed textual description and graphical depiction of significant threats to the software system being modeled. The threat model captures the ways in which the software's functions and architecture may be targeted and identifies the potential threat agents, *i.e.,* vectors for delivering threat-associated attacks.

Version 1: "STRIDE/DREAD" Model

To help define threat scenarios in its Version 1, the acronym *STRIDE* helps the user envision potential threat scenario from an attacker's perspective: Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, Elevation of privilege. To complement STRIDE, a risk calculation methodology known as *DREAD*, named for the acronym that encapsulates answers to potential question about risk: Damage potential, Reproducibility, Exploitability, Affected users, Discoverability. DREAD helps rate threats and prioritizes the importance of their countermeasures and mitigations. Once a threat's attributes are ranked, a mean of the five attributes is taken, with the resulting value representing the overall perceived risk associated with the threat. This process is repeated for all identified threats, which are then prioritized by descending order of overall risk value.

For Further Reading

Frank Swiderski and Window Snyder, *Threat Modeling,* (Microsoft Press, 2004).

P. Torr (Microsoft Corporation), *Guerrilla Threat Modeling.*
Available from: http://blogs.msdn.com/ptorr/archive/2005/02/22/GuerillaThreatModelling.aspx

Version 2: Microsoft Threat Analysis and Modeling

Microsoft's revision of its Threat Modeling methodology, now named Microsoft Threat Analysis and Modeling, released by Microsoft in March 2006, provides two key features:

1. A new threat modeling methodology and process intended to be more user-friendly for software developers, architects, and other stakeholders who are not security experts to understand and execute
2. A completely reengineered Threat Modeling application tool.

To make threat modeling more user-friendly, Microsoft eliminated the STRIDE and DREAD tools from Threat Modeling Version 1 (v1) and shifted the perspective from the attacker to the defender. The user identifies closely with threats, rather than attacks, reflecting Microsoft's belief that the defender can better understand threats to the system than the attacker.

As with Threat Modeling v1, the Threat Modeling and Analysis process is iterative process, adding layers of detail to an initial high-level threat model as the design progresses into subsequent phases of the life cycle. However, it more strictly defines a threat as an event that results in negative business or mission impact. The new threat model attempts to clarify the distinction between threats, attacks, and vulnerabilities. Microsoft Threat Analysis and Modeling also incorporates predefined attack libraries describing effective mitigations to each attack type associated with each threat, auto-generating threat models based on a defined application context. The model then maps those threat models to relevant countermeasures.

For Further Reading

"Threat Modeling", MSDN Developer Center.
Available from: http://msdn2.microsoft.com/en-us/security/aa570411.aspx

"Microsoft Application Threat Modeling" [weblog].
Available from: http://blogs.msdn.com/threatmodeling/

"Microsoft Threat Analysis & Modeling" [download page].
Available from: http://www.microsoft.com/downloads/details.aspx?familyid=59888078-9daf-4e96-b7d1-944703479451&displaylang=en

5.2.3.1.2 PTA Practical Threat Analysis Calculative Threat Modeling Methodology

Practical Threat Analysis (PTA) Technologies developed Calculative Threat Modeling Methodology (CTMM), a risk management methodology aimed at refining and expanding on Microsoft Threat Modeling v1. PTA Technologies identifies the following as limitations:

- No support for relating threats to financial losses caused by attacks
- No ranking or prioritization of countermeasures according to their effectiveness in reducing risk

- Reliance on "predefined" cases, making the tool difficult to adapt for modeling other threat scenarios
- No support for a complete system view for threat analysis or risk management
- Limited reporting and collaboration capabilities.

Note: Microsoft Threat Analysis and Modeling may render the need for CTMM's enhancements to Threat Modeling v1 unnecessary.

To address shortcomings in Microsoft Threat Modeling v1, CTMM builds a body of knowledge through iterative interaction between threat analysts and software developers. It enables analysts to maintain a growing database of threats, create documentation for security reviews, and produce reports showing the importance of various threats and the priorities of corresponding countermeasures. PTA automatically recalculates those threats and countermeasure priorities, and provides decision-makers with an updated action item list that reflects changes in the threat landscape.

For Further Reading

PTA Technologies, *Practical Threat Analysis for Securing Computerized Systems.* Available from: http://www.ptatechnologies.com/

5.2.3.1.3 Threat Modeling Based on Attacking Path

USC's Threat Modeling based on Attacking Path analysis (T-MAP) is a risk management approach that quantifies total severity weights of relevant attacking paths for COTS-based systems. T-MAP's strengths lie in its ability to maintain sensitivity to an organization's business value priorities and Information Technology (IT) environment, to prioritize and estimate security investment effectiveness and evaluate performance, and to communicate executive-friendly vulnerability details as threat profiles to help evaluate cost efficiency.

The T-MAP framework is value driven, utilizing an attack path concept to characterize possible scenarios in which an attacker could jeopardize organizational values. It maintains two key assumptions:

1. The more security holes left open for an (IT) system, the less secure it is.
2. Different IT servers might have different levels of importance in terms of supporting the business' core values.

With its awareness of a value's relative importance, T-MAP calculates the severity weight of each attack path based on both technical severity and value impacts. T-MAP then quantifies the IT system threat with the total weight of all possible attacking paths.

T-MAP uses a graph analysis to define evaluate and attack scenario. The attack is based on Bruce Schneier's "attack path" approach [190] and incorporates

a classic IT risk management framework consisting of Attacker, Asset, Vulnerability, and Impact. Attack tree nodes are structured into five layers:

1. Stakeholder values, *e.g.,* productivity, privacy, reputation
2. IT hosts that uphold stakeholder values
3. COTS software installed on IT hosts
4. Vulnerabilities in the COTS software
5. Possible attackers, *e.g.,* malicious insiders, external hackers, *etc.*

T-MAP defines a set of threat-relevant attributes for each of the above layers or nodes. These attributes can be classified as either probability-relevant, size-of-loss relevant, or descriptive. These class attributes are primarily derived from NIST SP 800-30, *Risk Management Guide for Information Technology Systems*, and the Common Vulnerability Scoring System (CVSS). [191]

T-MAP assigns estimated values to various attacker groups based on attributes such as skill level, group size, and motivation. T-MAP can then apply those attribute values to score the severity of the attack path with a numeric weight. Based on the classic risk calculation formula

*Risk = Probability * Size of Loss*

the user can calculate the weight of each attack path by multiplying its relevant attributes ratings together. This quantitative ranking enables security managers to prioritize the allocation of measures based on his/her ranking of vulnerabilities. Furthermore, the amount of manual effort required can be greatly reduced through use of the automated Tiramisu ranking tool.

T-MAP defines a formal framework to measure COTS system security based on attack path weights. Its strength lies in its three key features: distillation of technical details of published software vulnerabilities into executive-friendly information, providing an automated ranking system, and generating prioritized outcomes. Note, however, that for maximum impact, the T-MAP requires comprehensive, accurate, and up-to-date vulnerability information. Furthermore, it quantifies security threats only for published software vulnerabilities; the system is not sensitive to those that are unpublished.

For Further Reading

Yue Chen (University of Southern California), "Stakeholder Value Driven Threat Modeling for Off the Shelf Based Systems: 2006", Presentation at the ACM International Conference on Software Engineering, December 11, 2006.
Available from: http://sunset.usc.edu/csse/TECHRPTS/2006/usccse2006-620/usccse2006-620.pdf

Yue Chen, Barry Boehm and Luke Sheppard (University of Southern California), "Measuring Security Investment Benefit for COTS Based Systems—A Stakeholder Value Driven Approach", 2006. Presentation at the ACM International Conference on Software Engineering, September 8, 2006.
Available from: http://sunset.usc.edu/csse/TECHRPTS/2006/usccse2006-609/usccse2006-609.pdf

5.2.3.1.4 Trike

Trike is an open source conceptual framework, methodology, and toolset designed to autogenerate repeatable threat models. Its methodology enables the risk analyst to accurately and completely describe the security characteristics of the system, from high-level architecture to low-level implementation of details. Its consistent conceptual framework provides a standard language enabling communication among members of a security analysis team and between the security team other system stakeholders. It features three key tools:

1. **Threat-Model Generation**—Trike generates a threat model using the Trike toolset. The input to the threat model includes two additional Trike-generated models, a requirements model and an implementation model, along with notes on system risk and workflows. The tool also provides threat and attack graphs.
2. **Automation**—Trike provides high levels of automation. Unlike most offensive-based threat methodologies, it is based on a defensive perspective, imposing a greater degree of formalism on the threat modeling process.
3. **Unified Conceptual Framework**—Trike is a unified conceptual framework for security auditing. The unity of the framework enables team members to communicate with one another fluidly.

For Further Reading

"Trike: A Conceptual Framework for Threat Modeling".
Available from: http://dymaxion.org/trike

Demo Versions of Trike.
Available from: http://www.octotrike.org

5.2.3.1.5 Consultative Objective Risk Analysis System

The European Union (EU)-funded Consultative Objective Risk Analysis System (CORAS) project established an iterative framework for developing customizable, component-based roadmaps to aid the early discovery of security vulnerabilities, inconsistencies, and redundancies. It integrated existing risk assessment methods to yield six methodological results,

1. **Model-Based Risk Assessment**—The CORAS methodology for model-based risk assessment applies standard modeling technique ML to form input models to risk analysis methods used in a risk management process. The process is based on the Australian/New Zealand Standard (AS/NZS) 4360:1999, and is aimed at assessment of security-critical systems. The CORAS model has been tested successfully on telemedicine and e-commerce systems.

2. **UML Profile for Security Assessment**—The CORAS UML profile allows nonexpert users to understand UML diagrams and preserve the well-defined nature of UML. The profile also provides rules and constraints for risk assessment relevant system documentation.
3. **Library of Reusable Experiences Packages**—The CORAS project documented existing risk analysis processes to create a library of best practices.The library enables the user to recapture general practices from "experience elements" built into the library; examples of these practices include UML diagrams, checklists, and patterns. The experience elements also contain guidelines and recommendations derived from practices.
4. **CORAS Integration Platform**—The CORAS integration platform is the main computerized component of the CORAS framework. It stores results from ongoing and completed security analyses in two repositories: the Assessment Repository for analysis results and the Reusable Elements Repository for the reusable elements. The platform provides the end user with administrative functionality, such as creating new security analysis projects and applying the reusable elements and experience packages toward their own security goals.
5. **CORAS Mark-Up for Security Assessment**—The XML mark-up addressed the absence of a standardized meta-data format. Meta-data descriptions of core risk assessments can be used for consistency checking between different items on repositories provided by the CORAS integration platform. The mark-up also facilitates integration of risk analysis tools with the CORAS integration platform.
6. **Vulnerability Assessment Report**—The CORAS Vulnerability Assessment Report Format aims to standardize data reporting formats for describing network vulnerabilities. The format addresses existing reporting differences on currently available tools.

Secure Information Systems

The Research Council of Norway's model-driven development and analysis for Secure Information Systems (SECURIS) project aimed to establish computerized methodology for the development of secure IT systems targeting security from an overall business perspective, emphasizing the organizational and business context to the same extent as the actual technology. It is a trial-driven, iterative process, building on the established results of the CORAS project. Its own results will attempt to develop prototype tools that will produce—

- Capture and formalization of security requirements
- Model-driven specification and implementation of security policies
- Model-driven specification and development of security architectures
- Model-driven security assessment.

The project will also attempt to develop a methodology handbook.

For Further Reading

CORAS: A Tool-Supported Methodology for Model-Based Risk Analysis of Security Critical Systems.
Available from: http://heim.ifi.uio.no/~ketils/coras/

The CORAS Project.
Available from: http://coras.sourceforge.net/

CORAS: A Platform for Risk Analysis of Security Critical Systems.
Available from: http://www2.nr.no/coras/

SECURIS: Model-Driven Development and Analysis of Secure Information System.
Available from: http://www.sintef.no/content/page1_____1824.aspx

The SECURIS Project: Model-Driven Development and Analysis of Secure Information Systems.
Available from: http://heim.ifi.uio.no/~ketils/securis/index.htm

5.2.3.1.7 Attack Trees, Threat Trees, and Attack Graphs

Attack trees, threat trees, and attack graphs are visual representations of possible attacks against given targets that help visualize more complex series of events (attack patterns) that may be combined to cause a security compromise (see Section 3.2 on threats to software). Attack and threat trees and attack graphs provide alternative formats for capturing abuse case and misuse case information. The holistic view provided by the trees and graphs also clarifies the dependencies between attack patterns that exploit software vulnerabilities and those that target vulnerabilities at other levels (*e.g.,* system, network, personnel, procedure). This view can increase the understanding of risk managers so that they can better target and coordinate the countermeasures to these vulnerabilities across all layers of the system.

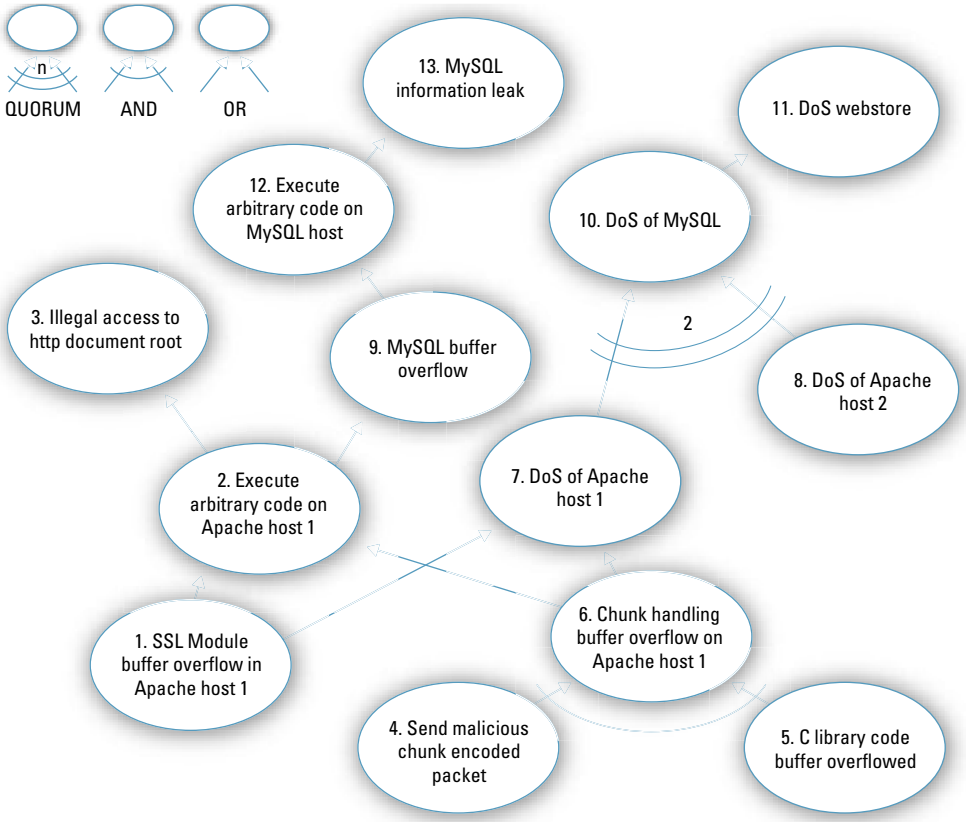
The following is an example of a non-graphical version of an attack tree [192] that includes several attack patterns that exploit software vulnerabilities:

Goal—Fake Reservation

1. Persuade an employee to add a reservation.
   - 1.1 Blackmail an employee.
   - 1.2 Threaten an employee.
2. Access and modify the flight database.
   - 2.1 Perform SQL injection from the web page (V1).
   - 2.2 Log into the database.
     - 2.2.1 Guess the password.
     - 2.2.2 Sniff the password (V7).
     - 2.2.3 Steal the password from the web server (AND).
       - 2.2.3.1 Get an account on the web server.
         - 2.2.3.1.1 Exploit a buffer overflow (V2).
         - 2.2.3.1.2 Get access to an employee account.
       - 2.2.3.2 Exploit a race condition to access a protected file (V3).

Figure 5-8 similarly illustrates an attack graph [193] that includes several attack patterns that exploit software vulnerabilities.

Figure 5-8. Example of an Attack Graph



Not all attack and threat trees and attack graphs lend themselves to manual analysis because of their sheer size and complexity. Some trees and graphs have been generated to depict real-world distributed attacks targeting large networked systems; these trees and graphs have included hundreds and even thousands of simultaneous different branches and paths leading to the completion of the attack.

According to some who have attempted to use attack and threat trees, they are difficult if not impossible to use by anyone who is not a security expert. Because a tree is a "list of security-related preconditions," it is unrealistic to expect a non-expert to accurately generate an appropriate list of security-related preconditions. Microsoft is one software development organization that has discovered that painstakingly generated security-related preconditions (attack or threat trees) actually form *patterns* that can then be standardized as reusable attack patterns (or, in Microsoft parlance, "threat tree patterns") that are easily comprehensible by non-security-expert developers. This pattern-based approach is rapidly supplanting use of attack and threat trees and graphs as a preferred threat modeling approach in many development organizations. See Section 3.2.3.1 for a discussion of the ways in which attack patterns can and are being used throughout the software life cycle.

For Further Reading

"Attack Trees", (Washington, DC: US CERT).
Available from: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/requirements/236.html

Bruce Schneier, "Attack Trees—Modeling Security Threats", *Dr. Dobb's Journal,* (December, 1999).
Available from: http://www.schneier.com/paper-attacktrees-ddj-ft.html

Oleg Sheyner, *et al.*, Automated Generation and Analysis of Attack Graphs: 2002, Presentation at the IEEE Symposium on Security and Privacy, (May 2002).
Available from: http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/calder/www/sp02.html

Oleg Sheyner and Jeannette M. Wing, "Tools for Generating and Analyzing Attack Graphs: 2003" Presentation at the Workshop on Formal Methods for Components and Objects, 2004: 344–371.
Available from: http://www-2.cs.cmu.edu/~wing/publications/SheynerWing04.pdf

5.2.3.1.8 System Risk Assessment Methods and Software-Intensive Systems

Even though system risk assessment methods cover a broad range of information security concerns, they also usually include adequate mechanisms and techniques for incorporating detailed software security assessments and threat modeling scenarios into a larger system security risk assessment. The challenge is that system risk assessment is in and of itself such a demanding, time-consuming activity that there are often no time, resources, or motivation to extend the risk assessment to the lower level of granularity required to assess individual software components of the system.

Systems risk analysts, then, generally proceed without consulting the software engineers who build those components. The result is that several security threats, vulnerabilities, and countermeasures that are unique to software components and best understood by software developers are not reflected in the system risk assessment. The result is, at best, an inaccurate report of the actual risks to the software intensive system.

This said, until recently, the absence of software-specific risk assessment and threat modeling techniques and tools left only one option to software teams: attempt to apply system-level risk assessment techniques (and supporting tools) to software components (and development projects). The following are the most popular, well-established of the system security risk assessment techniques. All of these are have been used in software security risk assessments.

- **Automated Security Self-Evaluation Tool (ASSET)**—[194] NIST's ASSET provides automated self-assessments against its own established criteria for security controls. ASSET is, in essence, an automated version of the questionnaire in NIST SP 800-26, *Guide for Information Security Program Assessments and System Reporting Form*. ASSET differs from other tools in that it is aimed only at facilitating Federal agency compliance with security assurance provisions mandated by law (*e.g.*, FISMA), policy (*e.g.*, DoDD 8500.1), and standards (*e.g.*, FIPS 200).

- **Central Communication and Telecommunication Agency (CCTA) Risk Analysis and Management Method (CRAMM)**—[195] Developed by Siemens Corporation and Insight Consulting, offers both quantitative and qualitative measurements of risk. Its distinguishing feature is its Controls Database of more than 3,000 security controls defined by a variety of security agencies and standard bodies. For each control, the database describes where the control would be appropriate and ascertains its cost and effectiveness against a variety of security breaches. The program delineates costs to the user and enables the user to rank or prioritize controls.
- **Mission Oriented Risk and Design Analysis (MORDA)**—[196] The MORDA methodology was developed by the NSA to provide a state-of-the-art quantitative risk assessment method. MORDA employs a variety of tools to execute a comprehensive risk management approach: attack tree models, IA models, and multiple-objective decision analysis. Each model yields mathematical outputs capturing figures such as estimated losses from attacks, predicted attack frequency, and effectiveness of countermeasures. Those quantitative outputs inform investment decisions aimed at enhancing security controls, reengineering, and upgrades or downgrades of existing system features. MORDA is the methodology that has been mandated for use in risk assessment of DoD Global Information Grid (GIG) applications and enterprise services.
- **Operationally Critical Threat, Asset, and Vulnerability Evaluation (OCTAVE)**—[197] The SEI's OCTAVE differs from MORDA in that it offers a qualitative approach. OCTAVE inventories critical assets, threats to those assets, vulnerabilities associated with those assets, and risk levels. The user is guided through a step-by-step process to catalog risk and generate relevant "Threat Profiles," which map a critical asset to its sources of potential risk through flowcharts. Analysts from all parts of the organization are involved in the assessment process.

5.2.3.2 Requirements Specification and Modeling Methods

Because requirements specifications are a bridge between multiple audiences, it may be necessary to prepare (and maintain) multiple versions of them. A version that is understandable to the end user would not need the same level of detail as one that is useful for the developers. Neither version would be understood by automated tools. The CONOPS document has been suggested as particularly helpful for the user. Specifications in formal notation are useable by verification tools, conversely, specifications that may be understandable to development tools, may not be readable. The CONOPS is also helpful for the developer in that it fleshes out requirements to achieve the level of understanding needed to avoid being surprised by the system that is ultimately developed from those requirements. Spiral development methods are also helpful in eliminating the

"surprise factor." Requirements are too abstract for developers to be able to conceptualize the usage of the system based on them. The problem is similar to the one a car buyer would have if, based on a parts list, he had to decide whether he wanted to drive the car once it was built. Thus, the CONOPS enables developers to get a better understanding of what the end user will be doing with the system than can be extrapolated from requirements alone.

Consistency then becomes an issue (*i.e.,* does the formal notation faithfully contain everything in the CONOPS?) and remains an issue throughout the SDLC (*i.e.,* are the changes made to one view of the specification replicated in all the views?)

Many software engineering toolsets include the concept of a repository, *i.e.,* a database or similar system that contains artifacts from the development process. A requirements repository would contain each requirement (that has been elicited, analyzed, prioritized, and specified), as an individually retrievable object. The repository will usefully contain additional information related to the actual requirements, such as the justifications, needed to build the assurance case, and documentation of the analysis and prioritization tasks that led to the specified requirements. It is especially important to keep this information for those requirements that are related to the security of the software and that grew out of the threat analyses and risk management work.

A key to making the repository useful is to ensure that it stores information at the individual requirement level so that the individual requirements can be entered, iterated, approved, placed under CM, published, managed, and traced. Then, by using an appropriate tool, it is possible to automatically, easily, and inexpensively generate various types of high-quality requirements specifications that are tailored to meet the individual needs of their various audiences. [198]

5.2.3.2.1 Use Cases, Misuse Cases, and Abuse Cases

A very promising set of approaches to developing requirements that address software security is the adaptation of use cases. These approaches have been given several names to distinguish them from standard use cases: abuse cases, misuse cases, hostile use cases, and dependability cases (these focus on exceptions). What they have in common is that they view the software from the perspective of an adversary. There are differences in their details, but for purposes of this document, they are the same.

In the same way that use cases are used successfully for eliciting requirements, misuse cases are used to identify potential threats from which it becomes possible to elicit security requirements or security use cases. The authorized user interaction with the system is diagrammed simultaneously with the hostile user's interactions. Where the connections between actor and actions in a use case are labeled with terms like "extends" and "includes," the connections in a misuse case are labeled "threatens" and "mitigates." Misuse

cases form the basis for constructing a set of security use cases to counter each of the threats. As with their use case counterparts, each misuse case drives a requirement and corresponding test scenario for the software. Much of the work in building misuse cases is therefore in developing requirements for security functionality. The analysis technique, however, provides a way to elicit nonfunctional requirements for protecting the software itself.

A good introduction to and motivation for misuse cases is found in *Misuse and Abuse Cases: Getting Past the Positive* [199] by Paco Hope *et al*. The article makes a distinction between misuse cases describing unintentional misuse and abuse cases describing intentional (hostile) misuse. The distinction is similarly to that between hazards and threats, but that is not a standard distinction. The article also suggests that attack patterns should be employed to help identify misuse cases. This task should be performed by teams involving both software developers (subject matter experts) and security and reliability experts.

A concise description of the technique is contained in Meledath Damodaran's article *Secure Software Development Using Use Cases and Misuse Cases* [200], which is excerpted here:

> *Essential description of how misuse cases are constructed: For each use case, brainstorm and identify how negative agents would attempt to defeat its purpose or thwart some of the steps in the use case description; this leads to the major misuse cases. During the brainstorm sessions the focus should be to identify as many ways an attacker could cause harm in the service provided by the use case in focus; details of such attacks may be determined later. Each of these modes of attacks becomes a candidate misuse case.*
>
> *The goal is to identify security threats against each of the functions, areas, processes, data, and transactions involved in the use case from different potential risks such as unauthorized access from within and without, denial of service attacks, privacy violations, confidentiality and integrity violations, and malicious hacking attacks. In addition to modes of attacks, the process should also try to uncover possible user mistakes and the system responses to them. Often these mistakes could cause serious issues in the functioning or security of the system. By identifying all inappropriate actions that could be taken, we would capture all actions of abnormal system use—by genuine users in terms of accidental or careless mistakes and by attackers trying to break or harm the system function.*

Some of the abuse case work has been derived from the domain of software safety, where safety requirements are elicited from safety cases.

For Further Reading

Ian Alexander, "Misuse Cases Help to Elicit Non-Functional Requirements", *The IET (Institution of Engineering and Technology) Computing and Control Engineering,* 14, no 1 (February, 2003): 40-45. Available at: http://easyweb.easynet.co.uk/~iany/consultancy/misuse_cases/misuse_cases.htm or http://swt.cs.tu-berlin.de/lehre/saswt/ws0304/unterlagen/MisuseCasesHelptoNon-Functional.pdf

Ian Alexander, "Misuse Cases: Use Cases with Hostile Intent", *IEEE Software.* (January/February 2003): 58-66. Available at: http://easyweb.easynet.co.uk/~iany/consultancy/misuse_cases_hostile_intent/misuse_cases_hostile_intent.htm

G. Sindre and A. L. Opdahl, "Eliciting Security Requirements by Misuse Cases: 2000", (Presentation at the International Conference on Technology of Object-Oriented Languages and Systems, 2000): 120-131. Available at: http://www.nik.no/2001/21-sindre.pdf

Donald G. Firesmith, "Security Use Cases", *Journal of Object-Technology.* 2, no.3 (May 2003): 53-64. Available at: http://www.jot.fm/issues/issue_2003_05/column6.pdf

5.2.3.2.2 Goal-Oriented Requirements Engineering

In goal-oriented requirements engineering, a goal is an objective that the software under consideration should achieve. Goal formulations thus refer to intended properties to be ensured. Goals may be formulated at different levels of abstraction. Goals also cover different types of concerns: functional concerns associated with the services to be provided, and nonfunctional concerns associated with quality of service—such as safety, security, accuracy, performance, and so forth. Therefore, goal-oriented approaches can be useful in developing requirements for secure software.

Goal-oriented methods can be applied to all activities within the requirements phase, including elicitation, analysis, prioritization, and specification. For example, goal modeling and specification can be used to support some form of goal-based reasoning for requirements-related subprocesses such as requirements elaboration, consistency and completeness checking, alternative selection, and change management. The nonfunctional requirements (NFR) framework provides qualitative reasoning procedures for soft goals, including security. This procedure determines the degree to which a goal is either satisfied or denied by lower-level goals or requirements. The general idea is to propagate satisfied links from the bottom-up, from lower-level nodes (*i.e.,* requirements) to higher-level nodes (*i.e.,* goals).

The NFR framework is an attempt to define a concrete framework for integrating nonfunctional requirements into the software development process. The authors of this approach recognize that it needs much more research to be theoretically sound, but also that, in the interim, it offers an adoptable solution that can be put to immediate use in an area that is in great need of concepts, methodologies, and tools. Since the NFR framework was proposed, it has been referenced and built upon extensively within the research community, as in the example above, where it provides structure for the goal-oriented approach.

The Méthodes d'Ingenierie de Logicels Securisés (MILOS, *i.e.,* Secure Software Engineering Methods) [201] project at the Catholic University of

Leuven (Belgium) is developing methodologies for goal-oriented security requirements engineering. These methodologies range from "application-specific security requirements engineering to secure architecture design to secure programming platform implementation."

The MILOS security model starts with a set of generic specification patterns that map to the desired security properties of information systems: confidentiality, integrity, availability, privacy, authentication, authorization, and nonrepudiation. These security patterns are then abstracted as goals, and a correlated "anti-model" is developed that captures a set of attacker "anti-goals" that, if achieved, would prevent achievement of the system security goals. MILOS' researchers claim their approach is unique among goal-oriented requirements methods in its modeling of requirements from the point of view of the attacker.

For Further Reading

Pierre-Jean Fontaine, *Goal-Oriented Elaboration of Security Requirements* (c2001).
Available from: http://citeseer.ist.psu.edu/fontaine01goaloriented.html

Xiaomeng Su, Damiano Bolzoni, Pascal van Eck and Roel Wieringa, *Understanding and Specifying Information Security Requirements*, (c2006).
Available from: http://arxiv.org/PS_cache/cs/pdf/0603/0603129.pdf

5.2.3.2.3 Security Quality Requirements Engineering and SCR

The CERT program at CMU is working on a Security Quality Requirements Engineering (SQUARE) process that is aimed at providing methods, tools, case study results, *etc.* that consider and address security for every aspect of the requirements process. [202] Table 5-9 is a summary of the process defined so far. The evolution of this work can followed on the DHS BuildSecurityIn portal.

Table 5-9. Steps in the SQUARE Process

Step #/ Activity	Input	Techniques	Participants	Output
1 Agree on definitions	Candidate definitions from IEEE and other standards	Structured interviews, focus groups	Stakeholders, requirements team	Agreed-to definitions
2 Identify security goals	Definitions, candidate goals, business drivers, policies and procedures, and examples	Facilitated work sessions, surveys, interviews	Stakeholders, requirements engineer	Goals

Table 5-9. Steps in the SQUARE Process - *continued*

Step #/ Activity	Input	Techniques	Participants	Output
3 Develop artifacts to support security requirements definition	Potential artifacts (*e.g.*, scenarios, misuse cases, templates, forms)	Work session	Requirements engineer	Needed artifacts: scenarios, misuse cases, models, templates, and forms
4 Perform risk assessment	Misuse cases, scenarios, and security goals	Risk assessment method, analysis of anticipated risk against organizational risk tolerance, including threat analysis	Requirements engineer, risk expert, stakeholders	Risk assessment results
5 Select elicitation techniques	Goals, definitions, candidate techniques, expertise of stakeholders, organizational style, culture, level of security needed, cost benefit analysis, *etc.*	Work session	Requirements engineer	Selected elicitation techniques
6 Elicit security requirements	Artifacts, risk assessment results, and selected techniques	Joint Application Development (JAD), interviews, surveys, model-based analysis, checklists, lists of reusable requirements types, document reviews	Stakeholders facilitated by requirements engineer	Initial cut at security requirements

Table 5-9. Steps in the SQUARE Process - *continued*

Step #/ Activity	Input	Techniques	Participants	Output
7 Categorize requirements by level (system, software, *etc.*) and whether they are requirements or other kinds of constraints	Initial requirements, and architecture	Work session using a standard set of categories	Requirements engineer, other specialists as needed	Categorized requirements
8 Prioritize requirements	Categorized requirements and risk assessment results	Prioritization methods such as Triage, Win-Win	Stakeholders facilitated by requirements engineer	Prioritized requirements
9 Requirements inspection	Prioritized requirements, and candidate formal inspection technique	Inspection method such as Fagan, peer reviews	Inspection team	Initial selected requirements, and documentation of decision-making process and rationale

The NRL SCR [203] model was originally developed in 1978 as a requirements methodology to be used for building high assurance control systems. Since then, SCR has been widely accepted and used by many software projects to develop software requirements in systems ranging from telephone networks and SCADA systems to military and commercial flight control systems. SCR was designed to be understandable by all participants in a software project, from engineers to high-level managers. The methodology is based in formal methods, enabling SCR-derived requirements to be mathematically proved *via* consistency and model-checking analyses. SCR is supported by a suite of tools, including a requirements specification editor, a dependency graph browser, a simulator for validating the specification, and verification tools that check that the requirements specification satisfies a defined set of critical design properties. Though not originally intended for software security requirements engineering, SCR has been used effectively for many years for specifying high assurance systems. Like secure software, such system have requirements for integrity and availability as well as dependability (which may include fault tolerance and safety).

5.2.3.2.4 Object-Oriented and Aspect-Oriented Modeling

Object-oriented analysis and modeling has benefited from the introduction of a standard notation, UML. [204] What had been a number of conflicting techniques has become a common practice. The widespread use of UML has also increased awareness of how valuable modeling is for dealing with complexity. In MDA and MDD, UML models are the primary artifacts of the software architecture and design phases. Automated tools are then used to transform the models into code (and back).

It has been observed that one weakness of object-oriented modeling is that it focuses on security properties only as they relate to specific software functionalities, but cannot efficiently capture crosscutting security properties that hold true across the system. For example, UML does not include such a construct as a misuse case or an abuse case, which makes it difficult to use UML to capture such properties as "resilience after a successful attack" or "reliability in the face of an intentional fault or weakness," although it is well-suited to capturing functionality-related properties such as "self-correction capability" or "detection of unintentional faults and weaknesses."

MDA is being used in developing secure software in the commercial sector as well. Object Security offers SecureMDA, which provides a consistent architecture for modeling, generating, and enforcing security policies *via* MDA. For modeling and application development, SecureMDA integrates with SecureMiddleware, an open source secure Common Object Request Broker Architecture (CORBA) implementation sponsored by Object Security. For security policy enforcement, SecureMDA integrates with OpenPMF, a central policy management framework offered by Object Security. Security policies generated by SecureMDA are enforced at the middleware level, preventing any malicious or improper components from affecting the rest of the distributed system. Although SecureMDA only enforces security models at the middleware layer, it is an important step in using modeling to develop high assurance systems.

By contrast with other UML-based object-oriented modeling methods such as MDA and RUP, Aspect Oriented Modeling (AOM—the first phase of Aspect Oriented Software Development [AOSD]) is a MDD technique that specifically addresses the difficulty of capturing and depicting crosscutting properties such as security. AOM separates the design for the nonfunctional concerns from the main functionality design and then composes the separate models together to create a comprehensive solution model. One typically has several models, each built for a specific purpose. A primary model embodies a solution that meets the most important functional requirements. One or more aspect models each embodies a solution that meets a nonfunctional requirement, such as a security or fault tolerance concern. The aspect models are composed with the primary model, according to a set of composition rules, to produce a more comprehensive solution model.

The following are UML security profiles that are available in the public domain:

- **UMLSec**—[205] The emphasis of the Security and Safety in Software Engineering WG lies in the methodological development of security-critical systems, including the use of formal methods under the aspect of official certification. This group, at the Technical University of Munich, developed both UMLSec and an extension of the AutoFocus CASE tool that adds security information. The AutoFocus extension allows the seamless consideration of security aspects in the development process with support of modeling, simulation, consistence checking, code generation, verification, and testing. The formal specifications in both UMLSec and AutoFocus can be used to verify security requirements. This allows certification on the highest degree, *i.e.,* CC, EAL 7. UMLSec is intended to encourage developers to consider system security requirements starting at the outset of the architectural design phase. The modeling language enables the developer to evaluate UML specifications for security vulnerabilities in the system design based on established rules of secure engineering encapsulated in a checklist. Because it is based on standard UML, UMLSec is intended to be useful to developers who are not specialists in secure systems development. Used in conjunction with AOM, UMLSec can be used to add semantic meaning to aspects. Integrity, confidentiality, and authentication are examples given by UMLSec's inventor of security properties that can be expressed in UMLSec extensions, and therefore used as specification elements by AOM security models.
- **SecureUML**—[206] Conceived by Torsten Lodderstedt, David Basin, and Jürgen Doser at the Institute for Computer Science at the University of Freiburg (Germany), and applied practically by Foundstone in designing secure authorization systems, SecureUML is a UML-based modeling language for expressing role-based access control (RBAC) and authorization constraints in the overall design of software systems. Although UML lends itself to modeling the many types of software security properties such as integrity, availability, non-compromisability, non-exploitability, and resilience, SecureUML focuses primarily on authorization and access control, and thus is unlikely to be helpful for expressing security extensions in the modeling of software dependability properties.
- **CORAS UML Profile**—Unlike UMLsec and SecureUML, the CORAS UML Profile for security assessment is directly relevant to modeling software security properties (*versus* system security functions that happen to be implemented in software). Developed by the CORAS Project (see Section 5.2.3.2.4), the CORAS UML profile provides a meta-model that defines an abstract language for modeling threats such as buffer overflow exploits, as well as associated "treatments" (countermeasures)

for those threats. The profile maps classes in the meta-model to modeling elements in the Object Management Group (OMG) UML standard. The CORAS UML profile was accepted as a recommended standard by the OMG in November 2003 and is being finalized.

Increasingly AOM is being adopted for capturing software security requirements. Aspects are a modeling construct that captures system behaviors and other nonfunctional characteristics, referred to in AOM terminology as "crosscutting properties," which cannot be represented effectively in purely functionality-oriented models. Some frequently-cited examples of such aspects include synchronization, component interaction, persistence, fault tolerance, quality of service, dependability, and security. AOM is intended to be used in conjunction with functionally-oriented UML (with or without extensions such as UMLSec or SecureUML), with AOM used to map crosscutting aspects to transformational model entities. Research into the use of AOM for modeling software dependability requirements and design aspects, and to model security policies for software-based systems has been underway at the University of Colorado since the early 2000s in the university's Model-Based Software Development Group. [207]

For Further Reading

Jan Jürjens publications page.
Available from: http://www4.in.tum.de/~juerjens/publications.html

C. B. Haley, R. C. Laney and B. Nuseibeh, "Deriving Security Requirements from Crosscutting Threat Descriptions: 2004", (Presentation at the 3rd International Conference on Aspect-Oriented Software Development, 2004): 112-121.
Available from: http://portal.acm.org/citation.cfm?id=976270.976285

Robert B. France *et al.*, (Colorado State University), "Evaluating Competing Dependability Concern Realizations in an Aspect-Oriented Modeling Framework: 2003", Presentation at the 14th IEEE International Symposium on Software Reliability Engineering: (November 17-20, 2003).
Available from: http://www.chillarege.com/fastabstracts/issre2003/150-FA-2003.pdf

5.2.3.2.5 Formal Methods and Requirements Engineering

Formal methods can be used to precisely specify requirements such that one can later prove an implementation meets those requirements. Formal languages used in requirements, such as Z (pronounced "Zed"), the specification language in the Vienna Development Method, and Larch, often come with tools. For requirements analysis, the attraction of formal methods is that the outputs are amenable to manipulation by machine and analysis tools. Praxis High Integrity Systems' Correctness by Construction [208] method, for example (which has been in use for more than 15 years in the UK) focuses on the accurate specification of mathematically verifiable requirements.

The emphasis of Correctness by Construction in the requirements phase is to make a clear distinction between user requirements, system specifications, and domain knowledge. "Satisfaction arguments" are then used

to demonstrate that each user requirement can be satisfied by an appropriate combination of system specification and domain knowledge. Praxis asserts that this emphasis on domain knowledge is key: it cites studies showing that half of all requirements errors are related to domain, and then asserts that the vast majority of requirements processes do not explicitly address domain issues. This is one of several ways in which Correctness by Construction differs from other requirements elicitation, specification, and verification methods. In the architecture and design stages, mathematical (or formal) methods and notations are used to precisely define the behavior of the software, consistent with its requirements, and to model its properties and attributes.

One obvious drawback to formal notation is that, being mathematically based, it is not communicable to the vast majority of stakeholders and other developers. This necessitates the generation of a parallel version in human-readable form, which must be kept synchronized with the formally notated version. A second drawback is in the level of effort needed to translate the requirements into the formalisms. Thus formal methods, while promising, are only feasible for small projects or small, critical subsets of projects, or when mandated, *i.e.,* for software that must meet higher assurance levels.

There is promise, because the benefits of formal methods make them attractive to researchers, that more usable tools and techniques for "lightweight" formal methods will emerge, and will enable more practitioners to reap their benefits at a reasonable cost. The use of formal specifications can result in fewer requirements errors. This benefit may exist, as Bertrand Meyer demonstrates, [209] even when the final specification is expressed in a natural language (*e.g.,* English) rather than a formal language.

Security concerns are sometimes introduced into formal specifications in a manner similar to the introduction of safety concerns. For example, a formal specification may mandate that some system state, as defined by values of state variables, can never arise.

5.2.3.2.6 Security Requirements Patterns

Research into security requirements patterns arose in parallel with research into security design patterns (see Section 5.3.3), which share the same basic foundation and premise. As of today, however, there are no standards or best practices for specification or use of security requirements patterns. Also in common with security design patterns, the research in security requirements patterns has focused on patterns for information security functionality (the work by Miroslav Kis, cited under "For Further Reading" below, is typical), rather than security for preserving dependability properties. This said, Sascha Konrad *et al.* [210] have done some promising work with their definition and analysis of security patterns for requirements and design, which includes introducing several new fields of security requirements, including behavior, constraints, and supported security principles, into the base design pattern template.

For Further Reading

Miroslav Kis, "Information Security Antipatterns in Software Requirements Engineering, 2002", (Presentation at the 9th Conference on Pattern Language of Programs, September 8-12, 2002). Available from: http://jerry.cs.uiuc.edu/~plop/plop2002/final/mkis_plop_2002.pdf

5.2.3.2.7 Security-Aware Tropos

Tropos [211] is a full life cycle methodology for development agent-based software systems. Created by researchers at the University of Trento (Italy), Tropos uses Goal-oriented Requirements Language (GRL) to model the required aspects of the agent-based system, and then proceeds through design, implementation, and testing phases, during which the system's specified functionality is refined and ultimately mapped back to the early requirements captured in GRL. Although Tropos was not originally conceived for capturing security requirements, researchers at the University of Trento and the University of Sheffield (UK) have defined security extensions that enable Tropos to be used for modeling both functional and nonfunctional security requirements and designing security aspects for both security functionality and security properties, such as those associated with trust management. [212]

5.3 Software Architecture and Design

Architecture design, sometimes called preliminary or high-level design, allocates requirements to components identified in the design phase. An architecture describes components at an abstract level, while leaving their implementation details unspecified. Some components may be modeled, prototyped, or elaborated at lower levels of abstraction, if, for example, analysis reveals a risk that the functions to be performed by such elaborated components are infeasible within performance, safety, or security constraints. High-level design activities might also include the design of interfaces among components in the architecture and a database design. Documents produced during the architectural design phase can include—

- Documentation of models, prototypes, and simulations
- Preliminary user's manual
- Preliminary test requirements
- Documentation of feasibility analyses
- Documentation of analyses of the consistency of components and of different representations of the architecture
- Documentation of the traceability of requirements to the architecture design.

Detailed design activities define data structures, algorithms, and control information for each component in a software system. Details of the interfaces between components are identified.

Attention to security issues, such as those captured in the threat models and analyses developed during the requirements and early architecture phases, can decrease the likelihood that the software's design will contain weaknesses that will make the software implemented from that design vulnerable to attack. [213]

A decrease in the number of vulnerabilities is a side effect of verifiable correctness. Correctness can be largely ensured by adopting rigorous formal methods, but at high cost. Semiformal modeling is a lower cost analysis method for increasing the likelihood of correctness.

Whether vulnerabilities exist often depends on the implementation details of the components identified in the design. Furthermore, exploits often attempt to change the environment and inputs to a system in ways that are difficult to formally model and predict. Nevertheless, some general principles for secure design can decrease the probability that exploits will exist in those components and that any remaining vulnerabilities will be exploitable throughout the system. Furthermore, a few design patterns at the architecture design level identify components—particularly components for input validation and escaping output—that ensure the absence of today's most common vulnerabilities.

The composition of systems (at least partially) from existing components presents particular challenges to secure software architecture design. A reused component may be exposed to inputs with which it has not been previously been tested. Thus, it may introduce vulnerabilities into the new system. In one case study, vulnerabilities were minimized by designing a "system (that) places no reliance on COTS correctness for critical security properties." [214]

Figures 5-9 and 5-10 illustrate the architectural and detailed design phases of a standard software life cycle (in this case, that depicted in IEEE Standard 1074-2006) with security assurance activities and artifacts added.

Figure 5-9. Inputs and Outputs of Architecture Design Activities With Assurance Concerns

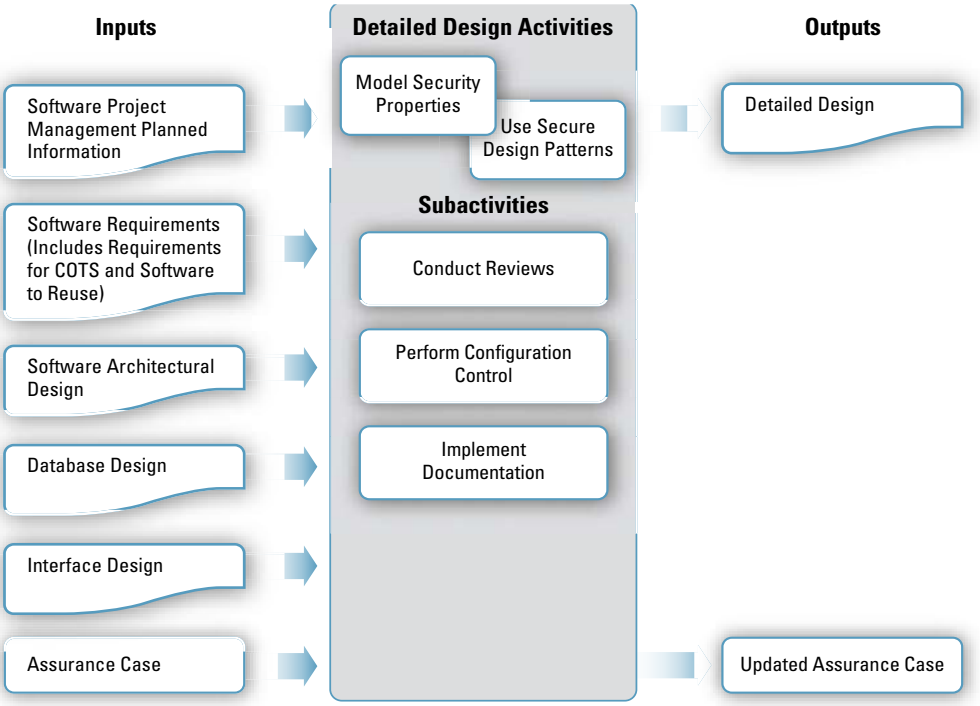
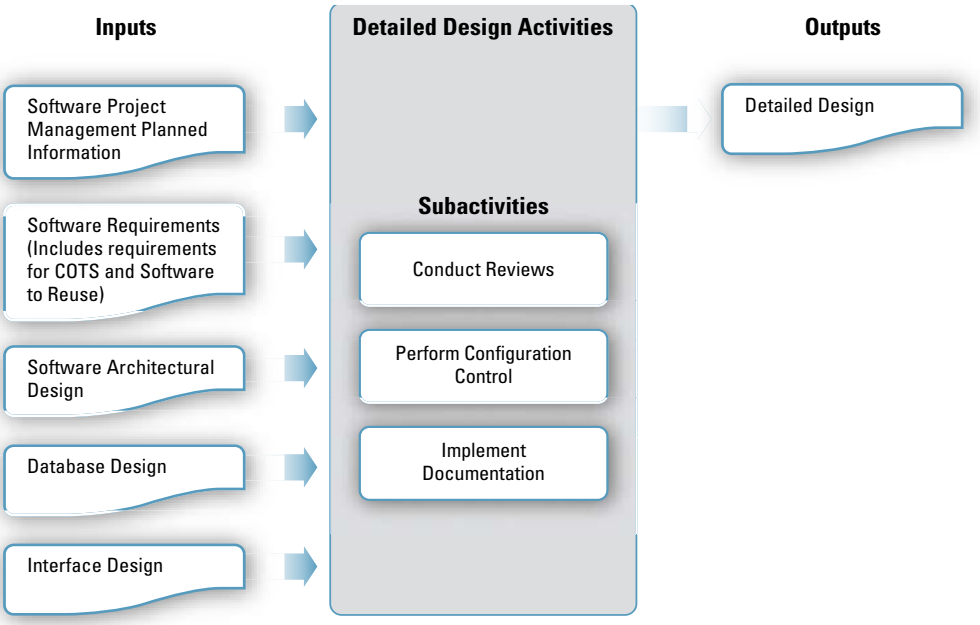


Figure 5-10. Inputs and Outputs of Detailed Design Activities With Assurance Concerns



5.3.1 Design Principles and Objectives for Secure Software

To resist attack, software needs to be expressly designed according to secure design principles. The number of books, white papers, articles, and websites that suggest and describe general design principles for secure software (including advice on secure software architecture) is proliferating.

Saltzer and Schroeder's *The Protection of Information in Computer Systems* [215] identified a set of access control and other protection principles that should be applied when designing a secure system. In DHS' Secure Software, [216] the Saltzer and Schroeder principles were analyzed, and the following subset was identified as directly relevant to software security assurance:

- **Least Privilege**—Least privilege is a principle whereby each principal (user, process, or device) is granted the most restrictive set of privileges needed for the performance of each of its authorized tasks. A more recent corollary is least-authorization, which is less transactional.
- **Complete Mediation**—Before granting access to a resource or execution of a process, the user or process requesting that access or execution should be checked for proper authorization, and the request should be rejected if the requestor is not appropriately authorized.
- **Fail-Safe Defaults**—This principle calls for defaulting, in security-relevant decision-making, toward denial rather than permission. Thus, the default situation will be to deny all requests that do not explicitly conform to the conditions under which such requests are permitted.
- **Separation of Privilege**—A high-assurance or otherwise high-consequence function should require two keys to unlock it (*i.e.,* to cause its execution). By requiring two keys, no single accident, deception, or breach of trust will be sufficient to compromise the trusted function. In practical application, code signatures provide this type of separation: code cannot be executed by one entity unless it has been signed using a certificate from a valid Certificate Authority, and moreover, that signature can be validated by a second entity (*i.e.,* a certificate validation service).
- **Open Design**—Security should not depend on security-through-obscurity, *i.e.,* the ignorance of potential attackers, but rather on assurance of dependability and/or the possession by users of specific, easily protected, authorization credentials. This permits the software to be examined by a number of reviewers without concern that the review may itself compromise the software's security. The practice of openly exposing one's design to scrutiny is not universally accepted. The notion that security should not depend on attacker ignorance is generally accepted, [217] but some would argue that obfuscation and hiding of both design and implementation has advantages: they raise the cost to the attacker of compromising the system.
- **Recording of Compromises**—If the software behaves suspiciously or is compromised, trails of evidence can aid in determining whether the

behavior or compromise resulted from an intentional attack, and if so, in understanding the attack patterns so as to better resist or tolerate it. After compromises, evidence trails also aid resilience, *i.e.,* recovery, diagnosis and repair, forensics, and accountability. Records of legitimate behaviors also have value, because they provide a "normal" baseline against which anomalous behavior can be compared.

- **Defense in Depth**—Defense-in-depth is a strategy in which the human, technological, and operational capabilities that comprise a system are integrated to establish variable protective barriers across multiple layers and dimensions of that system. This principle ensures that an attacker must penetrate more than one element of the overall system to successfully compromise that system. Diversity of mechanisms can make the attacker's efforts even more difficult. The increased cost of an attack may dissuade an attacker from continuing the attack. Note that composition of multiple less expensive but weak mechanisms is subject to the same composability issues as other software: combining them is unlikely to create a barrier that is stronger than the least secure of its individual parts, let alone the sum of its parts.
- **Work Factor**—The cost of a countermeasure or mitigation for a vulnerability or weakness (or of an overall increase in the level of security, which should eliminate a number of vulnerabilities and weaknesses) should be commensurate with the cost of the loss that would result were an attack to successfully exploit that vulnerability or weakness.
- **Economy of Mechanism**—"Keep the design as simple and small as possible" applies to any aspect of a system, but it deserves emphasis for trusted software. This principle minimizes the likelihood of errors in the code and directly aids its analyzability.
- **Analyzability**—Systems whose behavior is analyzable from their engineering descriptions such as design specifications and code have a greater chance of performing correctly because relevant aspects of their behavior can be predicted in advance.

In addition to the principles identified by Schroeder and Saltzer, software should also provide—

- **Security-Aware Error and Exception Handling**—Software needs to correctly handle exceptions so that active faults that are triggered by attack patterns will not result in a software crash (denial of service). The software needs to validate all user input to ensure it is not too large for the memory buffer allocated to receive it and does not contain segments of executable code.
- **Mutual Suspicion**—Components should not trust other components except when they are explicitly intended to. Each component in an interacting pair should always be prepared to protect itself against an attack from the other.

- **Isolation and Constraint of Untrusted Processes**—Sandboxing, virtual machines, trusted processor modules, and access control configurations can be used to isolate untrusted processes so that their misbehavior does not affect trusted processes or grant access to restricted memory areas to malicious code or human attackers.
- **Isolation of Trusted/High Consequence Processes**—In high-risk environments especially, it may be sensible to also isolate trusted and other high-consequence processes, to protect their integrity and availability from potential threats posed by untrusted processes and external entities. Trusted Processor Modules (TPM) were expressly conceived for this purpose, *i.e.,* for hosting trusted processes (such as cryptographic functions) in a physically separate execution environment with only tightly controlled interfaces to the rest of the software application or system.

In addition to the principles and objectives listed above, Morrie Gasser [218] suggested two additional secure design principles for software-intensive computer systems.

1. The system design should anticipate future security requirements.
2. The developer should consider security from the start of the design process.

DHS' *Software Assurance* (CBK) identifies a number of other secure software architecture and design objectives and design principles gleaned from several sources. These include:

- The architectural design should ease creation and maintenance of an assurance case.
- The architectural design should ease traceability, verification, validation, and evaluation.
- The architecture should eliminate possibilities for violations.
- The architectural design should help ensure certification and accreditation of the operational system.
- The architecture should provide predictable execution behavior.
- The design should avoid and work around any security-endangering weaknesses in the environment or development tools.
- The number of components to be trusted should be minimized.
- The system should be designed to do only what the specification calls for and nothing else.
- The system should be designed to tolerate security violations.
- The designer should make weak assumptions.
- The system should not cause security problems for other systems in the environment.
- The system should be designed for survivability.

Most books and other guidance on secure coding or secure programming include lists of secure design principles as well as secure implementation principles without making a clear distinction between the two.

For Further Reading

Brian Snow (NSA), "We Need Assurance!" (Presentation at the: Annual Computer Security Applications Conference; 2005).
Available from: http://www.acsac.org/2005/papers/Snow.pdf

Design Guidelines, (Washington, DC: US CERT).
Available from: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/guidelines.html

Software Security Principles, (Washington, DC: US CERT).
Available from: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/principles.html

Thomas Hrustka, *Safe C++ Design Principles,* (CubicleSoft, c2006).
Available from: http://www.cubiclesoft.com/SafeCPPDesign/

5.3.2 Architecture Modeling

A number of notations and techniques are available for modeling various characteristics of software architectures. UML and related technologies provide a currently popular approach to modeling. In modeling, one describes the architecture from the viewpoint of different stakeholders and their concerns. Different viewpoints require different descriptions. UML 2.0 supports this need by providing 13 different graphical representations of a system.

Some UML diagrams are useful in the analysis of the security of an architecture. Use cases can be used to define interactions with a system, a useful step in defining the capabilities of system users and in understanding possible attacks. Some find previous modeling approaches insufficient for a comprehensive security analysis of an architecture. Jie Ren and Richard Taylor at University of California at Irvine have defined xADL (eXtensible Architecture Description Language) [219] for modeling subjects, resources, privileges, safeguards, and policies. xADL models are also intended to facilitate the detection of architectural vulnerabilities. Michael Shin, a researcher at Texas Tech University, approaches the modeling of security requirements in terms of connectors in the software architecture. [220] In Shin's approach, security requirements are captured and refined separately from functional requirements. (See Section 5.2.3.2.4 for information on UML security profiles designed to model various elements of secure software-intensive systems.)

For Further Reading

Eduardo B. Fernandez, "A Methodology for Secure Software Design: 2004", (Presentation at the International Symposium on Web Services and Applications, June 21–24, 2004)
Available from: http://www.cse.fau.edu/~ed/EFLVSecSysDes1.pdf

Christian Damsgaard Jensen, "Secure Software Architectures: 1998", (Presentation at the 8th Nordic Workshop on Programming Environment Research, August 1998).
Available from: http://citeseer.ist.psu.edu/cache/papers/cs/8957/http:zSzzSzwww.ifi.uib.nozSzkonfzSznwper98zSzpaperszSzjensen.pdf/secure-software-architectures.pdf

Dianxiang Xu and Joshua J. Pauli, Publications on threat-driven design and misuse case-based secure software architectures.
Available from: http://www.homepages.dsu.edu/paulij/pubs/ or http://cs.ndsu.edu/~dxu/publications

5.3.3 Security (*vs.* Secure) Design Patterns

As noted by the authors of the DHS BuildSecurityIn portal's Attack Pattern Glossary, "In software engineering, a design pattern is a general repeatable solution to a recurring software engineering problem." [221] As is typical in the area of security design patterns, *Security Patterns* [222] by Markus Schumacher *et al.* contains no design patterns that are expressly intended to contribute to a less vulnerable, more attack-resistant/attack-tolerant software design. Currently, security design patterns are limited to security functions and controls at the system, communication, and information (data) levels.

In 2001, 4 years before *Security Patterns* was published, researchers Darrell Kienzle, Ph.D. MITRE Corporation, Matthew Elder, Ph.D. University of Virginia's Software Engineering Research Group, James Edwards-Hewitt of Surety, Inc., David Tyree of the University of South Florida, and James Croall of McAfee undertook the development of a repository of 29 web application security design patterns, [223] under a project sponsored by DARPA. Although the majority of the security design patterns included in their repository focused on system security functionality and information protection, there were a number of key patterns the objective whose was the protection of the application software itself, rather than its data or communications paths. These patterns included—

- **Hidden Implementation**, which hides the internal workings of the software application as a countermeasure to reverse-engineering attacks
- **Partitioned Application**, which splits a large, complex application into smaller, simpler components in order to assign privileges at the lowest possible level of granularity, thus enforcing least privilege for application processes and components
- **Secure Assertion**, which disseminates security assumption checks throughout the application to continually monitor the program for correct behavior and detect evidence of attack patterns and misuse
- **Server Sandbox**, which implements a virtual machine sandboxing mechanism to protect the web server in order to constrain damage resulting from an undetected vulnerability or fault in the server software.

Moreover, unlike any of the other widely recognized security design pattern sources, the DARPA-funded repository included a set of Procedural Patterns whose objective is to promote secure system development practices. Noteworthy among these are—

- **Choose the Right Stuff**, which entails using security as a criterion in the selection of nondevelopmental (COTS, OSS, *etc.*) components (including determination of whether a custom-developed component would be preferable), programming languages, and development tools
- **Build the Server From the Ground Up**, which ensures that unnecessary services are removed from the server that hosts the application, and vulnerable services are identified for ongoing risk management
- **Document the Server Configuration**, both the initial configuration of the server, including all applications hosted on it, and any changes to those configurations
- **Patch Proactively** by applying patches when they become available, not waiting until exploits occur
- **Red Team the Design**, performing an independent security evaluation of the design before implementing the application.

The final report of the DARPA project [224] not only described these patterns in more detail, as well as the outcomes of using the patterns in a web application proof-of-concept, it lists several other security pattern efforts predating the DARPA effort, including the Visual Network Rating Methodology (NRM) project at the NRL CHACS, which defined and applied "argument patterns" written in Category Abstract Machine Language (CAML) to the formal verification of security-critical software. To date, the DARPA-funded repository appears to be the only security design patterns effort that gives equal consideration to software security and system, information, and communication security needs.

The Open Group's *Technical Guide: Security Design Patterns* [225] is clear in its objective to describe system-level security design patterns. This said, the "System of Security Patterns" by which the Patterns Catalog in the document is organized is noteworthy in its inclusion in the two categories of patterns it defines, of a category devoted to availability ("Available System Patterns"), albeit at the whole-system level. The availability patterns defined (Checkpointed System, Standby, Comparator-Checked Fault-Tolerant System, Replicated System, Error Detection/Correction) are standard mechanisms that will directly contribute to attack-resilience in a software-intensive system. The other category, "Protection System Patterns," is concerned exclusively with access control, security policy enforcement, subject (user/process) authentication, association of security attributes (*e.g.,* privileges) with subjects, secure communications and message protection, and the establishment of security activities and contexts (privileges, *etc.*) of proxies acting on behalf of subjects—in short, the types of security functions that are the typical focus of security

design patterns. [Note also that the Open Group design patterns appear to be derived, in large part, from the CORBA security model.]

In their book *Core Security Patterns*, [226] Christopher Steel *et al*. include a catalogue of what they term "Core Security Patterns" for designing web services that run in the Java 2 Enterprise Edition (J2EE) environment. While the majority of the design patterns they describe are concerned with implementing system, communication, and information security functions, such as user-to-service or service-to-service authentication and authorization, session, message, and "pipe" encryption, event auditing, *etc.,* the book's catalog does include one pattern, labeled the Intercepting Validator pattern. This pattern performs input validation to prevent attacks that attempt to leverage the web service's lack of input parameter checking, *e.g.,* lack of checks for buffer overflows, SQL injections, *etc.*

In their paper, *A Practical Evaluation of Security Patterns*, [227] Spyros Halkidis *et al*. apply the STRIDE threat modeling criteria to the evaluation of two web applications, one of which was developed without use of the security design patterns, and another that was developed with a subset of the security design patterns described in *Core Security Patterns*. Using various security testing tools to launch attack patterns at each of the applications, the evaluators observed whether the number of vulnerabilities in the first application was significantly reduced, and its attack resistance enhanced, when security design patterns were applied to it. The specific attack patterns the evaluators employed were—

- Structured Query Language (SQL) injection
- Cross-site scripting
- Race conditions for servlet member variables
- Hypertext Transfer Protocol (HTTP) response splitting (exploits inadequate input validation)
- Eavesdropping.

Their findings revealed that—

> *…proper use of the security patterns leads to the remediation of all major security flaws. The flaws that are not confronted are…unencrypted SSL parameters…and servlet member variable race conditions…existing security patterns do not confront these kind of problems.*

The design pattern that appeared to have the most significant effect on the robustness of the application against the above attacks was, predictably, the Intercepting Validator pattern. The Intercepting Validator, in some sense, generalizes the Input Validation pattern, a defense against SQL injection, to standardize and to increase the maintainability of input validation. Using STRIDE (in essence a simple taxonomy of system-level and information security threats, *i.e., S*poofing of identity, *T*ampering with data, *R*epudiability, *I*nformation disclosure, *D*enial of service, *E*levation of privilege) as the basis for their evaluation necessarily weighted the findings of Halkidis *et al*. toward

the benefits of security design patterns which are expressly intended to address those types of threats. However, the evaluators' actual methodology focused more on software security problems than their claimed use of STRIDE would suggest, so their findings are significant as evidence of the effectiveness of security patterns such as the Intercepting Validator pattern in reducing the vulnerability of software to certain types of attacks.

In September 2007, the first International Workshop on Secure Systems Methodologies Using Patterns (SPattern '07) [228] will be held in Regensburg, Germany. The workshop Call for Papers states that it will focus on secure software methodologies, with papers sought that describe individual security patterns, new methodologies and new aspects of existing methodologies, pattern languages to use in the methodologies, reference architectures and blueprints, and related aspects, and especially experiences in applying these methodologies in real-world development projects.

For Further Reading

Eduardo B. Fernandez and Maria M. Larrondo-Petrie (Florida Atlantic University), "A Methodology to Build Secure Systems Using Patterns: 2006", (Presentation at the 22nd Annual Computer Security Applications Conference, December 11-15, 2006).
Available from: http://www.acsac.org/2006/wip/ACSAC-WiP06-03-Fernandez-EF-ACSAC06.pdf

5.3.4 Formal Methods and Software Design

In the design phase, formal methods are used to build and refine the software's formal design specification. Because the specification is expressed in mathematical syntax and semantics, it is precise (by contrast with nonformal and even semiformal specifications, which are open to reinterpretation). The correctness of the specification's syntax and semantics can be achieved independently of the use of tools, and its consistency and completeness can be verified through mathematical proofs.

Formal specification languages, of which there are dozens—many of which are limited in scope to the specification of a particular type of software or system (*e.g.*, security protocols, communications protocols, encryption algorithms)—fall into three classes:

- **Model-Oriented**—Support the specification of a system by constructing a mathematical model of it. *Examples*: Z, VDM.
- **Logical**—Close to (or identical with) logical languages not originally intended for specifying information systems. The use of these languages reflects the belief sometimes held that formal specification is special only in its use of formal notations, not in the kinds of logic or mathematics that it employs. *Example*; Z.
- **Constructive**—Constructive logical systems (usually type theories) are particularly concerned with the ability to realize (in an informal as well as a technical sense). Whereas in classical mathematics the notion of a function is very broad and includes many functions that could never be

evaluated by a computer, constructive mathematics is concerned only with functions that can be effectively computed. *Examples*: Program/Proof Refinement Logic (PRL).

- **Algebraic/Property-Oriented**—Specify information systems using methods derived from abstract algebra or category theory. *Examples*: Larch, Common Algebraic Specification Language (CASL), OBJ.
- **Process Model**—Are used for describing concurrent systems. These languages are sometimes implicitly based on a specific (though perhaps nonexplicit) model for concurrency. *Examples*: Calculus of Communicating Systems (CCS), Communicating Sequential Processes (CSP), ϖ-calculus.
- **Broad Spectrum**—Suitable for use at all stages in the development of an information system from conception through specification, design, implementation, and verification. *Examples*: Rigorous Approach to Industrial Software Engineering (RAISE) Specification Language (RSL), LOTOS (language for specifying communications protocols).

For Further Reading

M. G. Hinchey and J. P. Bown, *High-Integrity System Specification and Design,* (Springer, 1999).

Markus Roggenbach, *Formal Methods in Software Design,* (c2001).
Available from: http://www.informatik.uni-bremen.de/mmiss/TEKS/formalMethods.pdf

R. B. Jones, *Formal Specification Languages,* (c1996).
Available from: http://www.rbjones.com/rbjpub/cs/csfm02.htm

5.3.4.1 Formal Methods and Architectural Design

Formal methods can be used in the architecture phase—

- Specify architectures, including security aspects of an architectural design
- Verify that an architecture satisfies the specification produced during the previous phase, if that specification itself is in a formal language
- Establish that an architectural design is internally consistent
- Automatically generate prototypes
- Automatically generate a platform-dependent architecture.

The literature provides some examples of uses of formal methods in architecture design. Because IA applications frequently must meet mandatory assurance requirements, examples are easier to find of the use of formal methods for IA applications than for many other types of applications. Formal methods used in assuring IA applications, however, have wider applications in assuring correctness for those willing to incur the costs. In IA applications, formal methods have been used to prove correctness of security functionalities (*e.g.,* authentication, secure input/output, mandatory access control) and security-related trace properties (*e.g.,* secrecy). It is more difficult to prove non-trace security properties.

A variety of automated tools are available to assist developers adopting formal methods. Theorem provers are used to construct or check proofs. The latter task is easier to implement in a tool, but the former is more useful. Theorem provers differ in how much the user can direct them in constructing proofs. Model checkers are a recent class of theorem provers that has extended the practicality of formal methods. Another range of automated tools are associated with MDA and MDD (which are considered semiformal rather than formal methods).

In *Correctness by Construction* [229] Anthony Hall and Roderick Chapman [230] describe the development of a secure Certificate Authority, an IA application. The formal top-level specification (architecture design) was derived from the functionality defined in the user requirements, constraints identified in the formal security policy model, and results from the prototype user interface. Praxis used a type checker to automatically verify the syntax in the formal top-level specification and reviews to check the top-level specification against the requirements. The formal security policy model and the formal top-level specification are written in Z, a formal specification language, while the detailed design derived from the top-level specification is written in CSP.

In *E-Process Design and Assurance Using Model Checking* [231] W. Wang *et al.* describe the application of Verisoft and Spin, two model checkers, to verify an E-commerce application. The specification uses temporal logic, while the implementation uses C or Promela, depending on which model-checker is being used. Model checking, in this case, identified a specification flaw and a vulnerability that made the implementation open to a denial-of-service attack.

In *Modeling and Analysis of Security Protocols* [232] Peter Ryan *et al.* describe their use of Failure Divergence Refinement (FDR), a model-checking tool available from Formal Systems Ltd., the Caspar compiler, and CSP. They use these tools to model and analyze Yahalom (a protocol for distributing the symmetric shared keys used by trusted servers and for mutual entity authentication).

Further applications of formal methods are mentioned in *Security in the Software Life Cycle*. These include applications by Kestrel and Praxis. Technology described includes SLAM (Software specification, Language, Analysis, and Model-checking, which is Microsoft's model checking tool), the Standard Annotation Language (SAL), and Fugue.

5.3.4.2 Formal Methods and Detailed Design

Typically, the formal methods (see Section 5.1.2) used in detailed design and implementation differ from those used in system engineering, software requirements, and software architecture. Formal methods adopted during earlier phases support the specification of systems and system components and the verification of high-level designs. For example, the previous section mentions the use in architecture design of model checkers, VDM, and formal specification languages such as Z. Formal methods commonly used in detailed design and implementation are typically older methods, such as Edsger

Dijkstra's predicate transformers [233] and Harlan Mill's functional specification approach. [234] C.A.R. Hoare's CSP [235] might be used in both detailed design and in previous phases.

These formal methods for detailed design are most appropriate for—

- Verifying the functionality specified formally in the architecture design phase is correctly implemented in the detailed design or implementation phases
- Documenting detailed designs and source code.

For example, under Dijkstra's approach, one would document a function by specifying pre- and post-conditions. Preconditions and post-conditions are predicates such that if the precondition correctly characterizes a program's state on entry to the function, the post-condition is established upon exiting. An invariant is another important concept from this early work on formal methods. An invariant is a predicate whose truth is maintained by each execution of a loop or for all uses of a data structure. A possible approach to documentation includes stating invariants for loops and abstract data types.

Without explicit and executable identification of preconditions, post-conditions, and invariants for modules, formal methods in detailed design are most appropriate for verifying correctness when the interaction between system components is predefined and well-understood. In a Service-Oriented Architecture (SOA), [236] by contrast, the order and interactions of components vary dynamically. Such software-intensive systems impose new challenges on the use of formal methods to verify the correctness of a design. [237]

5.3.4.2.1 Design by Contract

Design by Contract (DbC) is an approach to providing components that can be used and reused in dynamically varying contexts. In DbC, classes have executable preconditions, post-conditions, and invariants. Exceptions are raised when any of these predicates are violated in an execution. In principle, some assurance of correct behavior is provided by incomplete contracts. DbC was first developed by Bertrand Meyer [238] for the Eiffel [239] programming language. Since then, tools have been developed to provide wrappers to support DbC in other programming languages.

For example, Stephen Edwards *et al.* [240] describe an approach for C++. Yves Le Traon *et al.* [241] report on a recent measure of the impact of DbC on (1) vigilance (*i.e.,* the probability that system contracts dynamically detect erroneous states that if left undetected would have provoked a failure), and (2) the ability to diagnose (*i.e.,* the effort needed to locate a fault in a system that is known to have caused a failure in that system). Le Troan *et al.* also illustrate the use of DbC with the Object Constraint Language (OCL), a formal language related to UML. Jean-Marc Jezequel and Bertrand Meyer [242] present a particularly striking case where DbC could have detected a $500 million error. They argue that the

failure of an Ariane 5 rocket launcher was not caused by defective management, processes, designs, implementation, or testing, but rather a reuse specification error. Although presumably, DbC helps prevent hackers from exploiting vulnerabilities, no research or case studies was found justifying this claim.

DHS's CBK describes the use of DbC-like properties for dependability and suggests SafSec. (See Section 5.1.4.2.1 as an example of such usage.)

5.3.5 Design Review Activities

Verification activities are typically conducted during the design phases at a number of types of reviews:

- Structured inspections, [243] conducted on parts or views of the high-level design throughout the phase
- IV&V reviews
- A preliminary design review conducted at the end of the architecture design phase and before entry into the detailed design phase
- A critical design review conducted at the end of the detailed design phase and before entry into the coding and unit testing phase.

Some IEEE standards treat specific review activities individually, including—

- IEEE Std. 730-2002, *Software quality assurance*
- IEEE Std. 828-2005, *Software configuration management*
- IEEE Standard 829-1998, *Software test documentation*
- IEEE Std. 1008-1987, *Software unit testing*
- IEEE Std. 1012-2004, *Software verification and validation*
- IEEE Std. 1028-1988, *Software reviews and audits*.

IEEE/EIA 12207.0 section 6.3, Quality assurance, 6.4, Verification, 6.5, Validation, and 6.6 Joint review processes, apply across life cycle phases. Section 6.4 of the standard, in addition to providing a general description of verification activities, provides criteria specific to design verification:

Design Verification–The design shall be verified considering the criteria listed below:

- The design is correct and consistent with and traceable to requirements.
- The design implements proper sequence of events, inputs, outputs, interfaces, logic flow, allocation of timing and sizing budgets, and error definition, isolation, and recovery.
- The selected design can be derived from requirements.
- The design implements safety, security, and other critical requirements correctly as shown by suitably rigorous methods.

Which reviews will be conducted and their definition is decided with the design of life cycle processes (see section A.5 of IEEE Std. 1074-2006,

Standard for Developing a Software Project Life Cycle Process). Such definitions typically include entry criteria, exit criteria, the roles of participants, the process to be followed, and data to be collected during each review. The choice of reviews, particularly those performed as part of IV and V, is partly guided by the evaluation requirements at the CC EAL being sought (if any). The processes for reviewing the architecture and detailed design should also accommodate later reviews of architecture and design modifications.

For Further Reading

Architectural Risk Analysis, (Washington, DC: US CERT).
Available from: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/architecture.html

Gary McGraw, *The Role of Architectural Risk Analysis in Software Security,* (Addison-Wesley Professional, c2006).
Available from: http://www.awprofessional.com/articles/article.asp?p=446451&seqNum=1&rl=1

Michael Charles Gegick, "*Analyzing Security Attacks to Generate Signatures from Vulnerable Architectural Patterns*" (thesis, North Carolina State University, 2004).
Available from: http://www.lib.ncsu.edu/theses/available/etd-08202004-171053/unrestricted/etd.pdf

Michael Gegick and Laurie Williams (North Carolina State University), "Matching Attack Patterns to Security Vulnerabilities in Software-Intensive System Designs: 2005," (Presentation at the Workshop on Software Engineering for Secure Systems at the ACM International Conference on Software Engineering; 2005.)
Available from: http://portal.acm.org/citation.cfm?id=1083200.1083211

5.3.6 Assurance Case Input in the Architecture and Design Phase

Material that might be included in the assurance case during the software's architecture and design phase includes—

- A modular, layered, and simple architecture
- A formal or semiformal presentation of the architectural design
- A formal or semiformal demonstration, including a requirements traceability analysis, that the architectural design fulfills the requirements, including the security policy
- A formal or semiformal presentation of the low-level design
- A formal or semiformal demonstration, including a requirements traceability analysis, that the detailed design implements the architecture design, including the security policy
- Evidence of a developer search for vulnerabilities within the architecture, including a search for covert channels
- Evidence of the continued maintenance of environment development controls, including physical security and controls on the staffing of designers
- Evidence of the continued use of a configuration management process, helping to ensure unauthorized modifications, additions, or deletions to the design.

Section 5.1.4.2, which discusses software security assurance cases, describes the limitations of the CC as a basis for defining assurance levels

for software security, as well as recent work to define an assurances case methodology that overcomes these limitations.

5.4 Secure Coding

Coding transforms the functions and modules of the detailed design into executable software. The resultant code is tested to show that it works, and that it implements the design. The other primary output of this phase is documentation, of both the code and the test results.

The implementation phase of the software life cycle is a concept, rather than a particular slice of time. Depending on the software development process or method in use, the project schedule, and the work breakdown for design and development, coding, and testing may be done multiple times, may be done at different times for different parts of the software, or may be done simultaneously with activities from other phases.

Security issues for the coding activities of the software implementation phase include—

- Language choice
- Compiler, library, and execution environment choices
- Coding conventions and rules
- Comments
- Documentation of security-sensitive code, constructs, and implementation decisions
- Integration of non-developmental software
- Need for filters and wrappers.

In virtually all software methods, testing is integrated into the coding phase, at a minimum debugging and unit testing. There are security issues to be considered at all levels and in all types of testing. For this SOAR, however, these are discussed separately in Section 5.5.

The software implementation phase will also include some mechanism for iteration and feedback. Often, coding or testing reveals a flaw from earlier in the development life cycle. The security of the software depends on how such flaws are remedied, including updating the previous artifacts: requirements specifications, design documents, test cases, traceability matrices, *etc.,* to reflect any changes made.

5.4.1 Secure Coding Principles and Practices

There is a lot of information published on specific techniques for writing secure code. Some of it is organized by language or platform. A lot of it aimed at a mass audience and does not presume any knowledge of software engineering. As a result, many of the guidelines include design or project-level suggestions, such as compartmentalization, in which security-sensitive code is separated from other functionality.

Virtually every article or book on secure coding, secure programming, secure application development, *etc.,* includes its own list of secure coding principles and/or practices. Some of these are language, technology, and environment neutral, while others are language, technology, and/or environment specific, although many of the principles and practices described in the latter are broadly relevant.

The following is a representative sampling of lists of secure coding principles and practices:

- **Language and Environment Neutral**:
  - Appendix I:G.4 of DHS's *Security in the Software Life Cycle* (Draft Version 1.2)
  - Chapter 4 "Implementation," in Mark G. Graff and Kenneth R. Van Wyk: *Secure Coding: Principles and Practices*
  - Part II "Secure Coding Techniques," in Michael Howard and David LeBlanc: *Writing Secure Code*, Second Edition
  - OWASP: *Secure Coding Principles*.
    Available from: http://www.owasp.org/index.php/Secure_Coding_Principles
  - Victor A. Rodriguez, Institute for Security and Open Methodologies: *Secure Programming Standards Methodology Manual*, Version 0.5.1, May 2002.
    Available from: http://www.isecom.org/projects/spsmm.shtml.
- **Language-Specific**:
  - Sun Microsystems: *Java Security Code Guidelines*.
    Available from: http://java.sun.com/security/seccodeguide.html
  - Apple Computer: *Secure Coding Guide*.
    Available from: http://developer.apple.com/documentation/Security/Conceptual/SecureCodingGuide/
  - Microsoft Corp.: *Secure Coding Guidelines for the .NET Framework*.
    Available from: http://msdn2.microsoft.com/en-us/library/aa302372.aspx
  - *FreeBSD [Free Berkeley System Distribution]* [244] *Developers' Handbook*. Chapter 3, Secure Programming.
    Available from: http://www.freebsd.org/doc/en_US.ISO8859-1/books/developers-handbook/secure.html
  - Robert Seacord: *Secure Coding in C and C++*
  - Thomas Oertli: Secure Programming in PHP (30 January 2002).
    Available from: http://www.cgisecurity.com/lib/php-secure-coding.html.
- **Environment-Specific**:
  - David A. Wheeler: Secure Programming for Linux and Unix HOWTO.
    Available from: http://www.dwheeler.com/secure-programs/ Secure-Programs-HOWTO/index.html or http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO.pdf.

Many of these and other "secure coding" and "secure programming" resources do not, however, make a clear-cut distinction between principles and practices to be used in the design of the software, and those to be used during the implementation of the software. Many secure coding lists include principles such as "Practice defense-in-depth" or "Compartmentalize" (cited by McGraw and Viega [245] and Cinnabar Networks' tutorial on secure programming and development practices, [246] among numerous other sources), both of which are principles achieved at the architecture and design levels rather than the coding level. Then, if the system is coded and integrated strictly according to its design, the resulting implementation should reflect these and any other secure design principles incorporated into that design.

A number of other secure coding principles are, in fact, system security and information security principles, such as "secure the weakest link," "promote privacy," "hiding secrets is hard," and "be reluctant to trust" (all cited in Viega and McGraw's *Building Secure Software*), and the numerous principles related to handling of passwords, and to user authentication and access control functions included in many secure programming guides.

In all cases, the authors seldom make clear whether their lists of secure coding principles are, in fact, most effectively addressed at the implementation level or whether, in fact, they are better addressed at the design or architecture level, or in some cases the secure deployment configuration level.

Table 5-10 attempts to extract those principles cited by most if not all of the above lists that are, in fact, secure *coding vs*. secure architecture/design or secure configuration principles.

Table 5-10. Generally Accepted Secure Coding Principles

Principle	Amplification
Input validation	The program should validate the length, format, correct termination, and characters (allowed *vs*. disallowed) of all input data, environment variables, and other externally sourced data (*e.g.*, filenames, URLs) and reject or filter those that do not pass validation
Least privilege	All processes should run with the minimal possible privileges and should retain those privileges for the minimal amount of time possible
Segregation of trusted from untrusted processes	The software should not invoke untrusted processes from within trusted processes
Small trusted processes	Trusted processes should be as simple and small as possible
No hard-coded credentials	Do not include authentication credentials or other sensitive data in the program's source code
No publicly accessible temp files	The program should write all of its data, configuration, and temporary files to non-public locations

Table 5-10. Generally Accepted Secure Coding Principles - *continued*

Principle	Amplification
Frequent purging of temporary data	Cache and other temporary data should be purged frequently, ideally immediately after it has been used by the program
Never escape to system	Never call or escape to the system command line or shell from within an application program
Avoid unsafe coding constructs	Do not use commands, library routines, or coding constructs that are known to cause security problems
Security-aware error and exception handling	Implement error and exception handling so that all conceivable events are explicitly handled in ways consistent with the attack resistance, attack tolerance, and fail-secure requirements of the software
Fail-secure	If the software must fail, it should do so gracefully and securely. It should not simply crash or hang, and its failure should not result in a core dump, nor expose system resources, sensitive data, or trusted code to any type of access (read, write, execute, or delete) by users or external processes
Non-informative error messages	To deter reconnaissance attacks, error messages should report only the fact that an error occurred, not the nature or cause of the error
Smallness and Simplicity	Every process should be as small and simple as possible, ideally with a single entry point and as few exit points as possible. Smallness and simplicity makes it easier to analyze the implemented code to determine whether any flaws or vulnerabilities are present
Use safe/secure languages and libraries	Use type-safe languages or security measures that reduce the risks associated with nontype-safe languages, such as compiler checking. Use alternatives to library functions in nontype-safe languages that are known to have vulnerabilities (*e.g.*, *printf* in C)
Safe memory allocation and management	Programs should self-limit their own resource consumption (*e.g.*, memory, processing time). Initial values for buffers (and for all other variables) should be set correctly

Along with principles for creating the code, proper documentation contributes to the security of software. Thorough documentation of the code includes data dictionaries that fully define both allowable inputs and outputs for functions and allowable ranges and types of values for all variables. [247]

Special attention needs to be paid to comments surrounding any code that is included or implemented specifically for security reasons, so that protections and checks that may have been implemented originally are not inadvertently removed or changed when the code is revised.

For Further Reading

Coding Practices, (Washington, DC: US CERT).
Available from: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/coding.html

5.4.1.1 Secure Coding Standards

The use and enforcement of well-documented coding standards is increasingly seen as an essential element of secure software development. Coding standards are intended to encourage programmers to follow a uniform set of rules and guidelines determined by the requirements of the project and organization, rather than by the programmer's familiarity or preference. Once established, these standards can be used as a metric to evaluate source code (using manual or automated processes) to determine compliance with the standard. At one extreme, a secure coding standard is developed for a particular release of a compiler from a particular vendor. At the other extreme, a standard may be both compiler- and language-independent.

The secure coding standards proposed by the CMU CERT are based on documented standard language versions as defined by official or de facto standards organizations, as well as on applicable technical corrigenda and documented language extensions, such as the ISO/IEC TR 24731 extensions to the C library. To date, the CERT has published secure coding standards for C (ISO/IEC 9899:1999) and C++ (ISO/IEC 9899:1999), [248] with plans to publish additional standards for Sun Microsystems' Java2 Platform Standard Edition 5.0 API Specification and Microsoft's C# programming language (ISO/IEC 23270:2003).

For Further Reading

Coding Rules, (Washington, DC: US CERT).
Available from: https://buildsecurityin.us-cert.gov/daisy/bsi/76.html

5.4.1.2 Secure Programming Languages and Coding Tools

Assuming the coder has a say in what programming language(s) will be used, the choice of language can have an effect on the security of code. Some languages are inherently safer than others. C and C++ are notoriously vulnerable to memory corruption attacks, while others, such as Ada, Standard ML [249] (SML), Pascal, Java, and C# exhibit the properties of type safety (*i.e.,* operations in these languages can handle only data types deemed compatible with those operations) and memory safety (*i.e.,* operations in those languages can write data only to those memory locations that have explicitly been allocated for those operations, once authorized, to write to).

Some languages (most notably Java and C#) are also tightly coupled with their own execution environments [*e.g.,* the Java Virtual Machine (JVM)] which provide software security mechanisms, such as code signing (to verify the authenticity of code before executing it) and sandboxing of untrusted code (to isolate it from other code and data, in case of its misbehavior during execution).

A lot of research has gone into the development of safe languages or safe versions of existing languages. Many (although not all) of these are variants on C and C++ that add type- and memory-safety (as well as other safe and secure execution features in some cases). Some of these include—

- MISRA C [250]
- Safe-C [251]
- CCured [252]
- Cyclone [253]
- Vault [254]
- Control-C [255]
- Fail-Safe C [256]
- SaferC [257]
- SafeJava [258]
- SPARKAda [259]
- Hermes [260]
- E [261]
- Oz-E [262]
- Clay [263]

One way that safe languages work is by using runtime checks to provide memory safety. Such checks are made for many different properties such as array bounds checking, null pointer references, and type conversions. Some of them also rely on runtime garbage collection to ensure the safety of pointer references. Safe languages, however, have some limitations. First, the language does not prevent all vulnerabilities. It is still possible to write insecure code in a safe language. For example, one could write a process that accepts user input without checking its validity. Second, the safety properties often cause performance, flexibility, or other constraints that require trade-offs with other requirements. This is particularly true for real-time applications, and embedded control systems. Secure coding practices as a whole need to be traded off against other requirements.

In addition to safe, secure languages, "safe" versions of C and C++ libraries are available (*e.g.*, the Safe C String Library, [264] Libsafe, [265] and the safe libraries in Microsoft's Visual Studio 2005) from which library calls associated with issues such as buffer overflows have been removed and replaced with less risky alternatives. Use of template-based collections of type-safe C and C++ pointers [266] can also help minimize memory-related vulnerabilities. Another problem noted by Les Hatton, Chair of Forensic Software Engineering at the University of Kingston (UK), [267] is that many of the rules in coding standards or secure language subsets have no justification in terms of quantitative data (in many cases they are simply stylistic rules instead of rules that counter actual security problems), and often raise false warnings in code that is actually safe.

Compilers and compiler extensions are also available that check code for type, memory, and/or synchronization issues. Safe-Secure C/C++, [268] the Safe C Compiler, [269] and the Memory Safe C Compiler, [270] for example, are "software component(s) that can be integrated into compilers and software analysis tools to detect and prevent buffer overflows and other common security vulnerabilities in C and C++ programs." Stack canaries and operating system-level stack randomization (available in gnu/Linux and Windows Vista) are aimed at making it more difficult

for attackers to exploit buffer overflows. In addition, the x86_64 instruction set supports marking portions of memory as non-executable, which implements heap and stack overflow prevention at the processor level rather than at the language or compiler level. The x86_64 instruction set is supported by gnu/Linux, BSD Unix, Mac OS X, and Windows XP Service Pack 2 and later.

The NIST SAMATE Project has identified classes of tools that aid the developer in following good and safe software development practices or in finding and highlighting errors in the code as it is being written. In addition to classes of tools that support model checking, code review and analysis, and software security analysis and testing (code analysis and testing tool categories are listed in Table 5-13 and 5-14), SAMATE has also identified four classes of tools that aid in producing secure code or remediating (*vs*. detecting) code-level vulnerabilities:

- Error-checking compilers
- Safety-enforcing compilers
- Wrappers
- Constructive approaches.

A variety of these secure coding tools (both for analysis and remediation) are now offered as plug-ins to compilers or integrated development environments. For example, Microsoft's Visual Studio 2005 now includes *PRE*fast (a static source code analysis tool), *PRE*fix (a dynamic analysis tool), FxCop (a design analysis tool), AppVerifier (a runtime verification tool), and Safe CRT (safe C/C++ run-time) libraries. For Java, many security tools are available as plug-ins to the open source Eclipse development environment.

For Further Reading

Les Hatton, *Safer Subsets.*
Available from: http://www.leshatton.org/index_SA.html

5.5 Software Security Analysis and Testing

Security analyses and tests, including code review and vulnerability assessment, collectively represent the most widespread of best practices for software security assurance. The number of software code analysis and security testing tools vendors has increased exponentially in the past decade and continues to grow, as do the number of open source tools, both independently developed and released as "teasers" by commercial tool vendors. Security testing services are becoming a standard offering of firms that specialize not only in software security or application security, but in software QA; software reliability; software development; IA; network, Internet, and cyber security; and IT services in general. Moreover, several firms have emerged that specialize in nothing but third-party independent security testing. Books on how to test for the security of software are proliferating as well: four titles were published in 2006 alone. [271] Software security test techniques and tools have become regular topics in software testing publications and at testing conferences. Professional training for software security testers is widely available.

5.5.1 What Are Software Security Analysis and Testing?

Unlike functional correctness testing of security function software, security analysis and testing of software is performed regardless of the type of functionality that software implements. Its function is to assess the security properties and behaviors of that software as it interacts with external entities (human users, its environment, other software), and as its own components interact with each other. The main objective of software security analysis and testing is the verification that the software exhibits the following properties and behaviors:

1. Its behavior is predictable and secure.
2. It exposes no vulnerabilities or weaknesses (ideally it *contains* no vulnerabilities or weaknesses, exposed or not).
3. Its error and exception handling routines enable it to maintain a secure state when confronted by attack patterns or intentional faults.
4. It satisfies all of its specified and implicit nonfunctional security requirements.
5. It does not violate any specified security constraints.
6. As much of its runtime-interpretable source code and byte code as possible has been obscured or obfuscated to deter reverse engineering.

Note that the Testing section of DHS' *Security in the Software Life Cycle* lists some key indicators of root causes for software's inability to exhibit the above properties and behaviors during testing.

To yield meaningful results, software security test plans should include a combination of techniques and scenarios sufficient to collectively determine (to whatever level of assurance is desired) whether the software does indeed exhibit the properties and behaviors listed above. The security test plan should be included in the overall software test plan, and should define—

- Security test cases or scenarios (based on misuse and abuse cases)
- Test data, including attack patterns (see Section 3.2)
- Test oracle (if one is to be used)
- Test tools (white box and black box, static and dynamic)
- Analyses to be performed to interpret, correlate, and synthesize the results from the various tests and outputs from the various tools.

The security test plan should acknowledge that the security assumptions that were valid when the software's requirements were specified will probably have changed by the time the software is deployed. The threat environment in which the software will actually operate is unlikely to have remained static. New threats and attack patterns are continually emerging. Also emerging are new versions of nondevelopmental components and patches to those components. These changes all have the potential to invalidate at least some of the security assumptions under which the original requirements were specified.

For Further Reading

Security Testing, Washington (DC): US CERT.
Available from: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/testing.html

Brian Chess and Jacob West, *Improving Software Security Using Static Source Code Analysis,* (Addison-Wesley Professional, 2007).

Maura van der Linden, *Testing Code Security,* (Auerbach Publishers, 2007).

Chris Wysopal, Lucas Nelson, Dino Dai Zovi and Elfriede Dustin, *The Art of Software Security Testing,* (Addison Wesley/Symantec Press, 2007).

Mike Andrews and James A. Whittaker, *How to Break Web Software: Functional and Security Testing of Web Applications and Web Services,* (Addison-Wesley Professional, 2006).

Mark Dowd, John McDonald and Justin Schuh, *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities,* (Addison-Wesley Professional, 2006).

Greg Hoglund and Gary McGraw, *Exploiting Software: How to Break Code,* (Addison-Wesley, 2004).

Irfan A. Chaudhry, *et al.,* *Web Application Security Assessment,* (Microsoft Press, 2003).

James A. Whittaker and Herbert H. Thompson, *How to Break Software Security,* (Addison Wesley, 2003).

5.5.1.1 When to Perform Analyses and Tests

It is safe to assert that all software security practitioners would agree that the common practice of postponing security analyses and tests until after the software has been implemented and integrated, and even until after it has been deployed (*i.e.,* during its acceptance phase), makes it extremely difficult to address in a cost-effective, timely manner any vulnerabilities and weaknesses discovered during the analysis and testing. The experiences of software security analysts and testers mirror those of their counterparts in the software safety community: a far greater number of the most significant security faults in software originate in the inadequate specification of its requirements and flaws in its architecture and design, rather than from errors in its coding or configuration. As the software progresses through its development life cycle, these early-life cycle security problems propagate and expand, becoming broadly and deeply embedded in the very fabric of the software. As a result, they have a great impact on the security assumptions under which later development phases are performed. Inadequate security requirements lead to deficient architectures, deficient architectures to defective designs, and so on. To avoid this progression, software security analysis and testing need to begin very early in the software's life cycle. To increase the likelihood that security problems will be caught as early as possible, developers should include security tests in their regime of daily builds and smoke testing.

This is not to say that late-stage "tiger team" or "red team" type security reviews and tests are not useful, particularly as an assurance validation measure, whether in the context of the Security Test and Evaluation (ST&T) phase of a government C&A, a third-party IV&V, or a commercial "security push."

As illustrated in Table 5-11, a range of security reviews, analyses, and tests can be mapped to the different software life cycle phases starting with the requirements phase.

Table 5-11. Security Reviews and Tests throughout the SDLC

Life Cycle Phase	Reviews/tests
Requirements	Security review of requirements and abuse/misuse cases
Architecture/Product Design	Architectural risk analysis (including external reviews)
Detailed Design	Security review of design. Development of test plans, including security tests.
Coding/Unit Testing	Code review (static and dynamic analysis), white box testing
Assembly/Integration Testing	Black box testing (fault injection, fuzz testing)
System Testing	Black box testing, vulnerability scanning
Distribution/ Deployment	Penetration testing (by software testing expert), vulnerability scanning, impact analysis of patches
Maintenance/support	(Feedback loop into previous phases), impact analysis of patches and updates

In preparation for analysis and testing, the software, test plan, test data, and test oracles (if used) are migrated from the development environment into a separate, isolated test environment. All security test cases should be run to ensure the adherence of the assembled and integrated system to all of its security requirements (including those for security properties and attributes, secure behaviors, self-protecting characteristics, and not just security functionality). Particular attention should be paid to the security of interfaces within the software system, between peer components (or peer services in a SOA), and between the system and external (environment and user) entities.

5.5.2 Security Analysis and Test Techniques

The following sections describe security analysis and testing techniques used to verify the security (or non-vulnerability) of software and software-intensive systems. Not discussed here are techniques and tools for review and verification of requirements, architecture, and design specifications, which were discussed in Sections 5.2 and 5.3.

Note: While the automated source code scanners and application vulnerability scanners described in Sections 5.5.2.1 and 5.5.2.2 are able to review very large programs in a short time and also to report metrics (*e.g.*, how many of each type of vulnerability has been located), the findings of such tools are necessarily only as complete as the set of patterns they have been programmed (or configured) to seek. When such tools are relied upon, there is a potential for a number of security vulnerabilities and weaknesses to go undetected. This is due, in large part, to the fact that such tools implement pattern matching. Pattern matching is effective for detecting simple implementation faults (and in the case of application vulnerability scanners, configuration vulnerabilities). It is not effective at finding architectural and design weaknesses, or byzantine implementation faults.

Nor will all of the patterns flagged by the automated scanner necessarily be vulnerabilities. Automated scanners can produce high rates of "false positives," i.e., patterns that appear to be vulnerabilities but which, in the context of the actual program, are not. The usual approach to reducing the false positive rate is for the tester to configure the scanner to look for fewer patterns. The problem with this approach is that it increases the scanner's "false negative" rate, i.e., not finding vulnerabilities that exist, but whose patterns have been "turned off" in the tool. In all cases, as with all tools, it is up to the tester to interpret the results to determine whether each finding is, in fact, indicative of a real vulnerability.

5.5.2.1 "White Box" Techniques

"White box" tests and analyses, by contrast with "black box" tests and analyses, are performed on the source code. Specific types of white box analyses and tests include—

- **Static Analysis**: Also referred to as "code review," static analysis analyses source code before it is compiled, to detect coding errors, insecure coding constructs, and other indicators of security vulnerabilities or weaknesses that are detectable at the source code level. Static analyses can be manual or automated. In a manual analysis, the reviewer inspects the source code without the assistance of tools. In an automated analysis, a tool (or tools) is used to scan the code to locate specific "problem" patterns (text strings) defined to it by the analyst *via* programming or configuration, which the tool then highlights or flags. This enables the reviewer to narrow the focus of his/her manual code inspection to those areas of the code in which the patterns highlighted or flagged in the scanner's output appear.
- **Direct Code Analysis**: Direct code analysis extends static analysis by using tools that focus not on finding individual errors but on verifying the code's overall conformance to a set of predefined properties, which can include security properties such as noninterference and separability, *persistent_BNDC*, noninference, forward-correctability, and nondeductibility of outputs.
- **Property-Based Testing**: [272] The purpose of property-based testing is to establish formal validation results through testing. To validate that a program satisfies a property, the property must hold whenever the program is executed. Property-based testing assumes that the specified property captures everything of interest in the program and assumes that the completeness of testing can be measured structurally in terms of source code. The testing only validates the specified property, using the property's specification to guide dynamic analysis of the program. Information derived from the specification determines which points in the program need to be tested and whether each test execution is

correct. A metric known as Iterative Contexts Coverage uses these test execution points to determine when testing is complete. Checking the correctness of each execution together with a description of all the relevant executions results in the validation of the program with respect to the property being tested, thus validating that the final product is free of any flaws specific to that property.

- **Source Code Fault Injection**: A form of dynamic analysis in which the source code is "instrumented" by inserting changes, then compiling and executing the instrumented code to observe the changes in state and behavior that emerge when the instrumented portions of code are executed. In this way, the tester can determine and even quantify how the software reacts when it is forced into anomalous states, such as those triggered by intentional faults. This technique has proved particularly useful for detecting the incorrect use of pointers and arrays, and the presence of dangerous calls and race conditions. Fault injection is a complex testing process and thus tends to be limited to code that requires very high assurance.
- **Fault Propagation Analysis**: This involves two techniques for fault injection of source code: extended propagation analysis and interface propagation analysis. The objective is not only to observe individual state changes that result from a given fault, but to trace how those state changes propagate throughout a fault tree that has been generated from the program's source code. Extended propagation analysis entails injecting a fault into the fault tree and then tracing how the fault propagates through the tree. The tester then extrapolates outward to predict the impact a particular fault may have on the behavior of the software module or component, and ultimately the system, as a whole. In interface propagation analysis, the tester perturbs the states that propagate *via* the interfaces between the module or component and its environment. To do this, the tester injects anomalies into the data feeds between the two levels of components and then watches to see how the resulting faults propagate and whether any new anomalies result. Interface propagation analysis enables the tester to determine how a failure in one component may affect its neighboring components.
- **Pedigree Analysis**: While not a security testing technique in itself, the detection of pedigree indicators in open source code can be helpful in drawing attention to the presence of components that have known vulnerabilities, pinpointing them as high-risk targets in need of additional testing. This is a fairly new area of code analysis that was sparked by concerns regarding open source licensing and intellectual property violations.
- **Dynamic Analysis of Source Code**: Dynamic analysis involves both the source code and the binary executable generated from the source code. The compiled executable is run and "fed" a set of sample inputs while

the reviewer monitors and analyzes the data (variables) the program produces as a result. With this better understanding of how the program behaves, the analyst can use a binary-to-source map to trace the location in the source code that corresponds with each point of execution in the running program, and more effectively locate faults, failures, and vulnerabilities. In *The Concept of Dynamic Analysis*, [273] T. Ball describes two analyses:

1. Coverage concept analysis
2. Frequency spectrum analysis.

Coverage concept analysis attempts to produce "dynamic control flow invariants" for a set of executions, which can be compared with statically derived invariants in order to identify desirable changes to the test suite that will enable it to produce better test results. Frequency spectrum analysis counts the number of executions of each path through each function during a single run of the program. The reviewer can then compare and contrast these separate program parts in terms of higher *versus* lower frequency, similarity of frequencies, or specific frequencies. The first analysis reveals any interactions between different parts of the program, while the second analysis reveals any dependencies between the program parts. The third analysis allows the developer to look for specific patterns in the program's execution, such as uncaught exceptions, *assert* failures, dynamic memory errors, and security problems. A number of dynamic analysis tools have been built to elicit or verify system-specific properties in source code, including call sequences and data invariants.

For Further Reading

Code Analysis, (Washington, DC: US CERT).
Available from: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/code.html

White Box Testing, (Washington, DC: US CERT).
Available from: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/white-box.html

Michael Howard (Microsoft Corporation), "A Process for Performing Security Code Reviews", IEEE Security & Privacy, 4, no.4 (July/August 2006): 74-79.
Available from: http://doi.ieeecomputersociety.org/10.1109/MSP.2006.84

Shostack, Adam, *Security Code Review Guidelines*.
Available from: http://www.homeport.org/~adam/review.html

5.5.2.2 "Black Box" Security Analysis and Test Techniques

"Black box" analyses and tests are performed directly on compiled binary executables, see Section 5.5.2.4). With the exception of static analysis of binaries, black box tests are performed on executing software and use a variety of input types to simulate the behaviors of attackers and other misusers and abusers of the software. The tests provide a view of the software from its outside, revealing the behaviors and outputs that result from the test inputs.

Black box techniques are the only techniques available for analyzing and testing nondevelopmental binary executables without first decompiling or disassembling them. Black box tests are not limited in utility to COTS and other executable packages: they are equally valuable for testing compiled custom-developed and open source code, enabling the tester to observe the software's actual behaviors during execution and compare them with behaviors that could only be speculated upon based on extrapolation from indicators in the source code. Black box testing also allows for examination of the software's interactions with external entities (environment, users, attackers)—a type of examination that is impossible in white box analyses and tests. One exception is the detection of malicious code. On the other hand, because black box testing can only observe the software as it runs and "from the outside in," it also provides an incomplete picture.

For this reason, both white and black box testing should be used together, the former during the coding and unit testing phase to eliminate as many problems as possible from the source code before it is compiled, and the latter later in the integration and assembly and system testing phases to detect the types of byzantine faults and complex vulnerabilities that only emerge as a result of runtime interactions of components with external entities. Specific types of black box tests include—

- **Binary Security Analysis**: This technique examines the binary machine code of an application for vulnerabilities. Binary security analysis tools usually occur in one of two forms. In the first form, the analysis tool monitors the binary as it executes, and may inject malicious input to simulate attack patterns intended to subvert or sabotage the binary's execution, in order to determine from the software's response whether the attack pattern was successful. This form of binary analysis is commonly used by web application vulnerability scanners.The second form of binary analysis tool models the binary executable (or some aspect of it) and then scans the model for potential vulnerabilities. For example, the tool may model the data flow of an application to determine whether it validates input before processing it and returning a result. This second form of binary analysis tool is most often used in Java bytecode scanners to generate a structured format of the Java program that is often easier to analyze than the original Java source code. [274]
- **Software Penetration Testing**: Applies a testing technique long used in network security testing to the software components of the system or to the software-intensive system as a whole. Just as network penetration testing requires testers to have extensive network security expertise, software penetration testing requires testers who are experts in the security of software and applications. The focus is on determining whether intra-or inter-component vulnerabilities are exposed to external access, and whether they can be exploited to compromise the software, its data, or its environment and resources. Penetration testing can be more extensive in its coverage and also test

for more complex problems, than other, less sophisticated (and less costly) black box security tests, such as fault injection, fuzzing, and vulnerability scanning. The penetration tester acts, in essence, as an "ethical hacker." As with network penetration testing, the effectiveness of software penetration tests is necessarily constrained by the amount of time, resources, stamina, and imagination available to the testers.

- **Fault Injection of Binary Executables**: This technique was originally developed by the software safety community to reveal safety-threatening faults undetectable through traditional testing techniques. Safety fault injection induces stresses in the software, creates interoperability problems among components, and simulates faults in the execution environment. Security fault injection uses a similar approach to simulate the types of faults and anomalies that would result from attack patterns or execution of malicious logic, and from unintentional faults that make the software vulnerable. Fault injection as an adjunct to penetration testing enables the tester to focus in more detail on the software's specific behaviors in response to attack patterns. Runtime fault injection involves data perturbation. The tester modifies the data passed by the execution environment to the software, or by one software component to another. Environment faults in particular have proven useful to simulate because they are the most likely to reflect real-world attack scenarios. However, injected faults should not be limited to those that simulate real-world attacks. To get the most complete understanding of all of the software's possible behaviors and states, the tester should also inject faults that simulate highly unlikely, even "impossible," conditions. As noted earlier, because of the complexity of the fault injection testing process, it tends to be used only for software that requires very high confidence or assurance.
- **Fuzz Testing**: Like fault injection, fuzz testing involves the input of invalid data *via* the software's environment or an external process. In the case of fuzz testing, however, the input data is random (to the extent that computer-generated data can be truly random): it is generated by tools called fuzzers, which usually work by copying and corrupting valid input data. Many fuzzers are written to be used on specific programs or applications and are not easily adaptable. Their specificity to a single target, however, enables them to help reveal security vulnerabilities that more generic tools cannot.
- **Byte Code, Assembler Code, and Binary Code Scanning**: This is comparable to source code scanning but targets the software's uninterpreted byte code, assembler code, or compiled binary executable before it is installed and executed. There are no security-specific byte code or binary code scanners. However, a handful of such tools do include searches for certain security-relevant errors and defects; see http://samate.nist.gov/index.php/Byte_Code_Scanners for a fairly comprehensive listing.

- **Automated Vulnerability Scanning**: Automated vulnerability scanning of operating system and application level software involves use of commercial or open source scanning tools that observe executing software systems for behaviors associated with attack patterns that target specific known vulnerabilities. Like virus scanners, vulnerability scanners rely on a repository of "signatures," in this case indicating recognizable vulnerabilities. Like automated code review tools, although many vulnerability scanners attempt to provide some mechanism for aggregating vulnerabilities, they are still unable to detect complex vulnerabilities or vulnerabilities exposed only as a result of unpredictable (combinations of) attack patterns. In addition to signature-based scanning, most vulnerability scanners attempt to simulate the reconnaissance attack patterns used by attackers to "probe" software for exposed, exploitable vulnerabilities.

Vulnerability scanners can be either network-based or host-based. Network-based scanners target the software from a remote platform across the network, while host-based scanners must be installed on the same host as the target. Host-based scanners generally perform more sophisticated analyses, such as verification of secure configurations, while network-based scanners more accurately simulate attacks that originate outside of the targeted system (*i.e.,* the majority of attacks in most environments).

Vulnerability scanning is fully automated, and the tools typically have the high false positive rates that typify most pattern-matching tools, as well as the high false-negative rates that plague other signature-based tools. It is up to the tester to both configure and calibrate the scanner to minimize both false positives and false negatives to the greatest possible extent, and to meaningfully interpret the results to identify real vulnerabilities and weaknesses. As with virus scanners and intrusion detection systems, the signature repositories of vulnerability scanners need to be updated frequently.

For testers who wish to write their own exploits, the open source Metasploit Project http://www.metasploit.com publishes blackhat information and tools for use by penetration testers, intrusion detection system signature developers, and researchers. The disclaimer on the Metasploit web site is careful to state:

> *This site was created to fill the gaps in the information publicly available on various exploitation techniques and to create a useful resource for exploit developers. The tools and information on this site are provided for legal security research and testing purposes only.*

For Further Reading

Konstantin Rozinov, "Efficient Static Analysis of Executables for Detecting Malicious Behaviors" (thesis, Polytechnic University, May 9, 2005.)
Available from: http://rozinov.sfs.poly.edu/papers/efficient_static_analysis_of_executables_for_detecting_malicious_behaviors.pdf

Penetration Testing, (Washington, DC: US CERT).
Available from: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/penetration.html

5.5.2.3 Compile Time Fault Detection and Program Verification

Compilers are routinely expected to detect, flag, and even eliminate certain type-errors in source code before compiling it. Such compiler checks, as a follow-up to code review, can be useful for detecting simple faults overlooked by the reviewer. Compilers cannot, however, be relied on to perform more sophisticated detection of byzantine faults that are often indicative of complex vulnerabilities, although other tools do exist to help developers detect such problems (*e.g.*, Compuware DevPartner).

Some compilers include extensions that perform full formal verification of complex security properties based on formal specifications generated prior to compilation. This type of formal verification can detect errors and dangerous constructs in both the program itself and its libraries. Other compile time program verification tools rely on the developer having annotated the source code with type qualifiers that then enable the compiler to formally verify the program as being free of recognizable faults. Such type qualifiers may be language-independent and enable the detection of unsafe system calls (which must then be examined by the developer). Other type qualifiers are language-specific and help detect vulnerabilities such as use of buffer overflow prone C and C++ library functions such as *printf*.

Still other compilers perform taint analysis, in which specific input data types are flagged as tainted, causing them to be validated before they are accepted by the compiled program.

5.5.2.4 Reverse Engineering: Disassembly and Decompilation

Reverse engineering of binary executables is performed as a precursor to white box testing of software that is only available in binary form. Two techniques are used in reverse engineering of binary code: disassembly and decompilation. In disassembly, an attempt is made to transform the binary code back into assembler code form. This enables a tester who is conversant in the specific assembly language generated by the disassembler to locate the signs of security-relevant coding errors and vulnerabilities that are detectable at the assembly code level.

By contrast with disassembly, decompilation attempts to generate standard source code from the binary executable. This source code can then be subjected to the same white box testing techniques used on other source code, with the limitation that decompiled source code is rarely as structured,

navigable, and comprehensible as original source code. Note that the optimizers within most compilers can make analysis of disassembled code difficult because part of the optimization process involves rearranging the code to make it execute more efficiently. This can result in vulnerabilities arising at the binary level that did not exist at the source code level.

Because they are labor intensive, both techniques are likely to be practical only for trusted or high-assurance software that is considered very high risk, *e.g.,* because of suspicious pedigree. Because of intellectual property protection concerns, many commercial software products use obfuscation techniques to deter reverse-engineering; such techniques can increase the level of effort required for disassembly and decompilation testing, making such techniques impractical. In many cases, commercial software distribution licenses also explicitly prohibit reverse-engineering.

The blackhat community is a rich source of information, tools, and techniques for reverse engineering which, increasingly, whitehat organizations are adopting to make their software and systems more robust against them. Indeed, as they mature, many erstwhile blackhats are getting jobs as ethical hackers, penetration testers, and security consultants. There is even a Blackhat organization [275] devoted to the awareness and training of security practitioners in the mentality, techniques, and tools employed by their adversaries.

For Further Reading

Müller, Hausi A.; Storey, Margaret-Anne; Jahnke, Jens H. [University of Victoria (Canada)]; Smith, Dennis B. (CMU SEI); Tilley, Scott R. (University of California at Riverside); Wong, Kenny [University of Alberta (Canada)], *Reverse Engineering: A Roadmap.*
Available from: http://www.cs.ucl.ac.uk/staff/A.Finkelstein/fose/finalmuller.pdf

Mike Perry and Nasko Oskov, *Introduction to Reverse Engineering Software.*
Available from: http://www.acm.uiuc.edu/sigmil/RevEng

M. G. J.van den Brand and P. Klint (University of Amsterdam, Netherlands), *Reverse Engineering and System Renovation: An Annotated Bibliography.*
Available from: http://www.cs.vu.nl/~x/reeng/REanno.html

Jussi Koskinen (University of Jyväskylä), *Bibliography of Reverse Engineering Techniques.*
Available from: http://www.cs.jyu.fi/~koskinen/bibre.htm

The Reverse Engineering Community.
Available from: http://www.reverse-engineering.net

French Reverse Engineering Team.
Available from: http://www.binary-reverser.org

Reverser's Playground. Crackmes.de.
Available from: http://www.crackmes.de

CodeBreakers Journal.
Available from: http://www.codebreakers-journal.com

5.5.2.5 Forensic Security Analysis

Forensic security analysis of software, supported by static and dynamic analysis tools, is comparable to other computer forensic analyses. After deployed software is successfully compromised, a forensic analysis can help reveal the vulnerabilities that were exploited by the attacker. Forensic analysis of software comprises three different analyses: intra-component, inter-component, and extra-component. Intra-component forensic analysis is used when the exploited vulnerability is suspected to lie within the component itself.

Inter-component analysis is used when the suspected location of the vulnerability lies in the interface between two components. The analysis tools examine the communication and messaging and programmatic interface mechanisms and protocols used by the components, and reveal any incompatibilities between the different components' implementations those interface mechanisms and protocols.

Extra-component analysis is used when the vulnerability is suspected to lie either in the execution environment or in the dynamics of the whole system's behavior not traceable to a single component or interface. The analysis includes reviewing audit and event logs to find indications of security-relevant whole-system behaviors that indicate vulnerabilities caused by configuration problems or system and environment interactions that were targeted by the attacker.

5.5.3 Software Security Analysis and Testing Tools

Most tools that support software security analysis and testing implement either white box or black box techniques, and many are limited to a single technique, such as static analysis or fuzzing. However, vendors have recently started to produce tool sets or suites whereby control of the tools is integrated *via* a central console. Some tool suites also include attack prevention and protection tools such as intrusion detectors and application firewalls. Examples include Fortify Software's tool suite, Ounce Labs' Ounce Solution, Compuware's DevPartner SecurityChecker, Klocwork's 7, and Coverity's Prevent. Microsoft also integrates software security analysis, test, and implementation tools into its larger Visual Studio 2005 integrated development environment, including a code review tool, a fuzzer, a secure C library, and other software security tools used by the company's own developers.

As Kris Britton of the NSA Center for Assured Software (CAS) has observed, [276] the level of integration of such tools has not extended nearly as far as supporting "meta-analysis," *i.e.,* the ability of different tools to interpret, rank, and increase the confidence in the results of other tools. Meta-analysis cannot be achieved without the ability to fuse, correlate, and normalize the outputs of the constituent tools in a toolset/tool suite. These capabilities are beyond the realm of what is offered by the tools in toolsets and suites from a single vendor, let alone by tools obtained from multiple vendors and open sources. (This is, in fact, one of the main research objectives of the NSA CAS.)

To this end, the OMG Software Assurance Special Interest Group (SwA SIG) is developing the Software Assurance Ecosystem, "a formal framework for analysis and exchange of information related to software security and trustworthiness" [277] This ecosystem will leverage related OMG specifications such as the Knowledge Discovery Metamodel (KDM), the Semantics of Business Vocabulary and Rules (SBVR), and the upcoming Software Assurance Meta-model. By improving interoperability among software assurance tools, organizations will be able to better automate their evaluation processes and incorporate results from a variety of tools—from static code analysis tools to formal method verifiers to application vulnerability scanners—into a single coherent assurance case.

NIST SAMATE has been tasked with developing standards against which software and software assurance tools can be measured. In its first phase, SAMATE is focusing on source code analysis tools. The project has produced a reference dataset against which users can test source code analysis tools to see how effective they are at detecting various types of coding errors. In addition, SAMATE is working on a draft functional specification for source code analysis tools. Through the SAMATE project, tool vendors will be able to use a common taxonomy to describe their capabilities. [278]

To aid in this, SAMATE has produced a classification and taxonomy of software assurance tools, many of which are testing tools. The SAMATE tool classification is essentially a refinement and extension of the tool categorization in the market surveys of application security testing tools produced by the DISA Application Security Project in 2002, 2003, and 2004 (see Section 6.1.7). The classes of software testing tools identified by SAMATE are listed in Table 5-12 below. These tools are not strictly limited to security testing tools; they also encompass general software testing tools which can (and in many cases have been) applied to software vulnerability detection or security property verification. Based on a broad survey of the security testing tool market, additional tool classes have been identified that are not included in the SAMATE classification; these additional classes are listed in Table 5-13.

In addition to software testing tools, the SAMATE classification includes tools for testing for security vulnerabilities, or verifying the security properties or secure configuration of components of the software's execution environment (*e.g.,* network operating systems, web servers, *etc.*), and for testing system-level security functions performed between software components (*e.g.,* verification of correctness of WS-Security implementations) and security of communications between software components (*e.g.,* secure exchange of SOAP messages between web services). These tool classes are listed in Table 5-13, as are additional classes of tools in this category.

In both tables, tool classes that are strictly security-focused are indicated with an "X". All others are broader-spectrum tools that either include some security-specific test capabilities, or general testing tools that can be used for security testing.

Table 5-12. Classes of Software Security Test Tools

SAMATE Classes	
Web application vulnerability scanners (and assessment proxies)	X
Dynamic (binary) analysis tools	
Compiler error checking and safety enforcement	
Source code security (static) analyzers	X
Byte code scanners	
Binary code scanners	
Code review assistants	
Additional Classes	
Compiler-based program verification	
Property-based testers	
Property-based testers	
Source code fault injectors	
Binary fault injectors	
Fuzzers	
Penetration testers	X
Buffer overrun detectors	
Race detectors	
Input validation checkers	
Tools for detection of malicious code in source code	X
Pedigree analysis tools	
Code security checklists	X

Table 5-13. Classes of Execution Environment and System-Level Test Tools

SAMATE Classes	
Network (vulnerability) scanners	X
Web services network scanners	
Database (vulnerability) scanners	X
Intrusion detection tools	X
Antispyware (detection) tools	X
Additional Classes	
Operating system vulnerability scanners	X
Web server vulnerability scanners	X
Patch verification tools	
Virus scanners	X

Both NIST SAMATE and NSA CAS are involved in the evaluation of software security tools. These efforts are described in Sections 6.1.2 (CAS) and 6.1.10 (SAMATE).

For Further Reading

Source Code Analysis Tools. Washington (DC): US CERT.
Available from: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/tools/code.html

Black Box Security Testing Tools. Washington (DC): US CERT.
Available from: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/tools/black-box.html

Fong, Elizabeth, editor. NIST SP 500-264, *Proceedings of Defining the State of the Art in Software Security Tools Workshop*. c2005.
Available from: http://samate.nist.gov/docs/NIST_Special_Publication_500-264.pdf

5.5.3.1 Software Security Checklists

Software engineers have found checklists useful in conducting reviews ever since, at least, Michael Fagan [279] invented formal inspections. Gary McGraw, in Software Security, recommends the checklists in Jay Ramachandran's *Designing Security Architecture Solutions* (John Wiley and Sons, 2002) and that published by J.D. Meier *et al.* [280] Several other software security and application security checklists are also available in the public domain; these are listed in Table 5-14 below. In addition, David Gilliam *et al.* [281] have defined guidelines for developing one's own software security checklists.

Table 5-14. Software and Application Security Checklists

Checklist	URL
DISA: *Application Security Checklist* Version 2, Release 1.9 (November 24, 2006)	http://iase.disa.mil/stigs/checklist/app-security-checklist-v2r19-24Nov06.doc
NASA *Software Security Checklist*	Contact David Gilliam, david.p.gilliam@jpl.nasa.gov
OWASP: *Web Application Penetration Checklist* v1.1 (Version 2 is due for release December 31, 2006. The OWASP *Design Review Checklist* is no longer available; it is not clear whether a revision is underway)	http://prdownloads.sourceforge.net/owasp/OWASPWebAppPenTestList1.1.pdf?download (in English) http://www.owasp.org/index.php/Category:OWASP_Testing_Project (in Spanish and Italian)
Charles H. Le Grand, CHL Associates: *Software Security Assurance: A Framework for Software Vulnerability Management and Audit*	http://www.ouncelabs.com/audit/
Visa U.S.A.: CISP Payment Application Best Practices checklist	http://usa.visa.com/download/business/accepting_visa/ops_risk_management/cisp_payment_application_best_practices.doc
Djenana Campara: *Secure Software: A Manager's Checklist* (June 20, 2005)	http://www.klocwork.com/company/downloads/SecureSoftwareManagerChecklist.pdf
Australian Computer Emergency Response Team (AusCERT): *Secure Unix Programming Checklist* (July 2002 version)	http://www.auscert.org.au/render.html?it=1975
Don O'Neill Consulting: Standard of Excellence product checklists (include a security checklist)	http://members.aol.com/ONeillDon2/special_checklist_frames.html
Microsoft Corp./b-sec Consulting Pty Ltd. Business Application Security Assurance Program (BASAP) *Application Security Assurance Framework*, Version 2.2 (June 22, 2005)	http://www.b-sec.com.au/basap/Application%20Security per cent20Assurance%20Framework%20v2.2.pdf
BASAP *Secure Development Process* Framework, Version 2.1 (June 21, 2005)	http://www.b-sec.com.au/basap/Secure%20Development%20Process%20Framework%20v2.1.pdf
Apple Computer: *Secure Coding Guide Security* Development Checklists	http://developer.apple.com/documentation/Security/Conceptual/SecureCodingGuide/Articles/DevSecSoftware.html#//apple_ref/doc/uid/TP40002495-DontLinkElementID_29

5.5.4 System C&A and Software Security Assurance

The trend of malicious attacks has shifted focus from network systems to software vulnerabilities. It is also true that no software can be impervious to attack, and there is no "quick fix" or "canned solution" that clearly defines a practice, process, or methodology for implementing security practices that provide for software assurance. The ultimate goal is to produce secure software that contains properties of repeated regularity and reliability, and this can be accomplished by establishing and maintaining a stringent and accepted set of processes that recognize, minimize, and mitigate vulnerabilities.

There is an obvious need to establish a set of criteria for the implementation, documentation, certification, and accreditation of software accomplished through a formal evaluation approach similar to the C&A process for network systems [*e.g.,* DoD Information Assurance Certification and Accreditation Process (DIACAP), DCID 6/3, National Information Assurance Certification and Accreditation Process (NIACAP), FISMA, NIST]. While software assurance may be addressed as a component of the current mandated C&A processes, it is not nearly inclusive enough to ensure software assurance or software security.

Several models exist today, but a general set of criteria would—

- Help establish a set of defined steps and processes
- Facilitate management oversight for secure programming practices
- Help in the recognition of design patterns for vulnerabilities
- Provide for security verification that security mechanisms have been implemented
- Provide a testing process so the removal of vulnerabilities can be demonstrated
- Support the certification, formal review, and acceptance of software by a designated manager such as a Designated Approving Authority (DAA).

The CC evaluation and C&A disciplines do not provide an adequate basis for assuring software security for many reasons. Most notably, there is a chasm between the disciplines of software assurance and the CC and C&A bodies of knowledge. The tools used for system evaluation in the separate disciplines of CC and C&A consider software assurance as a minor goal in the overall certification of the whole system. (C&A typically assesses security only at the whole-system level.) C&A focuses mainly on infrastructure and architecture models of access control and risk mitigation; these may or may not benefit software security assurance.

One reason for this information gap is that there is very little language in the CC or in standard C&A documentation that specifically addresses software assurance concerns. The separate views can be contributed to differing levels of importance concerning the correctness of information, security controls, and policy enforcement. Software assurance specific language was added to

the draft of CC Version 3. However, before the new version could be approved by ISO/IEC, the consultation period expired, [282] and the future of Version 3 remains undetermined.

In the current version of the CC (Version 2), systems evaluated at EAL4 and below [283] do not require rigorous security engineering practices. The vast majority of COTS software, if evaluated at all, is evaluated at or below EAL4. In addition to these specific reasons for the gap between the studies of software assurance and the CC, not all software is eligible for CC evaluation, and in that case, the software would not need to be evaluated by the CC resulting in a lack of evaluation of security controls altogether.

The C&A discipline, from a security standpoint, deals with many objects such as systems, networks, and application life cycles. In short, the C&A process audits and ensures policies, procedures, controls, and contingency planning. While some information security reports can be obtained about systems from various forms of testing (penetration tests and code reviews), this level of testing is not indicative of software security policies and procedures that alone will provide adequate software assurance.

The main objective of system ST&E is to determine whether the system as a whole satisfies its information system security requirements, *i.e.,* those for the functionalities and constraints that ensure the preservation of the confidentiality, integrity, and availability of data, and the accountability of users. Software security properties, both at the levels of the whole system and of individual components, are seldom considered because C&A is driven by the need of systems to conform with governing information security policy. To date, DoD has neither added software security language to DoDD 8500.1, nor mandated a separate, comparable policy governing security for software.

Even if the certifier wishes to verify the security properties of the components of COTS-based systems, the C&A documents required for DIACAP [which, because its activities begin earlier in the system's life cycle than did their counterparts in DoD Information Technology Security Certification and Accreditation Process (DITSCAP), is thought to more strongly influence systems engineers to begin security engineering at the outset of the life cycle, rather than attempting to "tack security on" at the integration phase], DCID 6/3, NIACAP, and FISMA do not include C&A criteria specific to COTS-based systems. Nor do COTS components generally expose the security information that will enable the certifier exact assessment of each component's conformance even to the stated C&A criteria, let alone the assessment of the security conflicts between components, and the impact of those conflicts on the overall security of the system.

The C&A process does not look deep enough, or extensively at enough of the individual software components to comfortably address software assurance. In many cases, conducting or providing for code review of COTS products is not feasible or likely. Further, such tests, no matter the extent, depth, or thoroughness of the testing, are often at the discretion of the DAA. The results of

these tests as part of a C&A process are often times looked upon as a tertiary or an objective step towards the overall accreditation of the system or network and are not used or even authorized.

The inadequacy of CC evaluation artifacts and C&A artifacts as the basis for establishing software security assurance is addressed further in Section 5.1.4 on assurance cases.

5.6 Secure Software Distribution and Configuration

The principles (if not the practices) for trusted distribution of software and systems defined in NCSC-TG-008 *Guide to Understanding Trusted Distribution in Trusted Systems* (the "Dark Lavender Book") are still broadly applicable to software distributions today. The objective of secure distribution is to minimize the opportunities for malicious or nefarious actors to gain access to and tamper with software during its transmittal (*via* physical media shipment or network download) from its supplier to its consumer.

Secure distribution mechanisms that have become standard as intellectual property rights protections for commercial software are increasingly being used to protect integrity for purposes of security. Such mechanisms include tamperproof or tamper-resistant packaging, read-only media, secure and verifiable distribution channels [*e.g.,* Secure Sockets Layer (SSL)-encrypted links for downloads, registered mail deliveries], and digital integrity mechanisms (hashes or, increasingly, digital watermarks, and/or code signatures).

In addition to the software itself, the software's installation and configuration procedures, routines, tools, and interfaces are also, increasingly, being protected through authentication of installer, cryptographically protected communication channels, separate distribution paths, *etc.*

DHS has been in discussion with NIST about the need for a standard defining the characteristics of a minimum acceptable "secure" default configuration for commercial software. Increasingly, major commercial vendors are shipping their software with such secure default configurations, a practice originated by manufacturers of security systems such as firewalls and trusted operating systems.

DoD and other government departments and agencies produce secure configuration guidelines and checklists for widely used commercial software products. A number of these can be found at—

- NSA Security Configuration Guides.
  Available from: http://www.nsa.gov/snac
- DISA Security Technical Implementation Guides (STIG) and Checklists.
  Available from: http://www.nist.gov/cgi-bin/exit_nist.cgi?url=http://iase.disa.mil/stigs/index.html
- NIST Security Configuration Checklists for IT Products.
  Available from: http://csrc.nist.gov/checklists

In addition to these, the SysAdmin, Audit, Networking and Security (SANS) Institute and the Center for Internet Security has established the Security Consensus Operational Readiness Evaluation (SCORE) [284] program, which is enlisting security professionals from a number of organizations to develop minimum acceptable secure configuration checklists for a number of popular types of systems (*e.g.,* web applications, UNIX and UNIX-derived operating systems) as well as specific implementations of those systems.

References

78 H. Mouratidis and P. Giorgini," Integrating Security and Software Engineering: an Introduction," chap. I in *Integrating Security and Software Engineering*, H. Mouratidis and P. Giorgini, eds. (Hershey, PA: Idea Group Publishing, 2007).

79 Charles B. Haley, Robin C. Laney, and Bashar Nuseibeh, "Deriving Security Requirements From Crosscutting Threat Descriptions," in *Proceedings of the Third International Conference on Aspect-Oriented Software Development*, March 22–26, 2004: 112–121.
Available from: http://mcs.open.ac.uk/cbh46/papers/AOSD04.pdf

80 The term "acquisition" in the context of this specific discussion, focuses on how software comes into existence within a given organization or system. For this reason, it does not include acquisition of contractor development services. However, contractors may be responsible for the integration, assembly, custom development, and/or reengineering of software.

81 L. David Balk and Ann Kedia, "PPT: a COTS Integration Case Study," in *Proceedings of the 22nd International Conference on Software Engineering*, July 2000.

82 Nicholas G. Carr, "Does Not Compute," *The New York Times* (January 22, 2000).
Available from: http://www.nytimes.com/2005/01/22/opinion/22carr.html

83 Jeremy Epstein, "SOA Security: The Only Thing We Have to Fear Is Fear Itself," *SOA Web Services Journal* (December 3, 2005).
Available from: http://webservices.sys-con.com/read/155630.htm

84 Mark G. Graff and Kenneth R. Van Wyk, *Secure Coding, Principles and Practices* (Sebastopol, CA: O'Reilly and Associates, 2003).

85 Frank Tiboni, "Air Force Handles Network Security," *Federal Computer Week* (June 13, 2005).
Available from: http://www.fcw.com/article89159-06-13-05-Print

86 This observation was made by Joe Jarzombek, Director of Software Assurance, DHS CS&C NCSD, at the kick-off of the DHS Software Assurance Program's Acquisition Working Group meeting of October 2005. Compare the two versions of *Federal Acquisition Regulations* (FARS), Part 1, subchapter A,.
Available from: http://www.acquisition.gov/far/05-06/html/FARtoHTML.htm with that available from: http://www.acquisition.gov/far/05-05r1/html/FARtoHTML.htm

87 Mary Linda Polydys (CMU SEI) and Stan Wisseman (Booz Allen Hamilton), SwA Acquisition Working Group, *Software Assurance (SwA) in Acquisition: Mitigating Risks to the Enterprise—Recommendations of the Multi-agency, Public/Private Sector*, draft vers. 0.9 (Washington, DC: DHS CS&C NCSD, February 9, 2007).

88 "OWASP Secure Software Contract Annex" [web page] (Columbia, MD: OWASP).
Available from: http://www.owasp.org/index.php/OWASP_Secure_Software_Contract_Annex

89 David A. Wheeler, "Open Standards and Security" (presentation, July 12, 2006).
Available from: http://www.dwheeler.com/essays/open-standards-security.pdf

90 Jerome H. Saltzer and Michael D. Schroeder, "The Protection of Information in Computer Systems," in *Proceedings of the Symposium on Operating System Principals*, October 1973.
Available from: http://www.cs.virginia.edu/~evans/cs551/saltzer/

91 Peter Neumann (SRI International) and Richard Feiertag (Cougaar Software, Inc.), "PSOS Revisited," in *Proceedings of the 19th Annual Computer Security Applications Conference*, Las Vegas, NV, December 8–12, 2003: 208–216.
Available from: http://www.csl.sri.com/neumann/psos03.pdf

92 Laurianne McLaughlin, "Winning the Game of Risk," *IEEE Security and Privacy* 3, no. 6, (November–December 2005): 9–12.

93 Peter Neumann (SRI International), *Principled Assuredly Trustworthy Composable Architectures, Final Report*, report no. CDRL A001 (December 28, 2004).
Available from: http://www.csl.sri.com/users/neumann/chats4.html

94 "SEI Predictable Assembly From Certifiable Components (PACC)" [web page] (Pittsburgh, PA: CMU SEI).
Available from: http://www.sei.cmu.edu/pacc/

95 "Center for High Assurance Computer Systems (CHACS)," web page (Washington, DC: Naval Research Laboratory).
Available from: http://chacs.nrl.navy.mil

96 Mitchell Komaroff (OSD/OCIO), "DoD Software Assurance Concept of Operations Overview" (slides presented at the OMG Software Assurance Special Interest Group [SwA SIG] meeting, December 9, 2006, Washington, DC [Revised 2007 March]).

97 "Build Security In" portal page.

98 "SSE-CMM: Systems Security Engineering–Capability Maturity Model" [web site] (Herndon, VA: International Systems Security Engineering Association [ISSEA]).
Available from: http://www.sse-cmm.org/index.html

99 E.J. Chikofsky and J.H. Cross, "Reverse Engineering and Design Recovery: a Taxonomy," *IEEE Software 7*, no. 1 (January 7, 1990): 13–17.

100 Requirements analysts, system designers and software architects are included within the broad category of "developer."

101 Robin Milner (University of Edinburgh), "A Theory of Type Polymorphism in Programming," *The Journal of Computer and System Sciences* (April 19, 1978).
Available from: http://www.diku.dk/undervisning/2006-2007/2006-2007_b2_246/milner78theory.pdf

102 Barry W. Boehm, *Software Engineering Economics* (Upper Saddle River, NJ: Prentice-Hall, 1981).

103 John Viega and Gary McGraw, *Building Secure Software: How to Avoid Security Problems the Right Way* (Boston, MA: Addison-Wesley, 2001).

104 Marco M. Morana (Foundstone Professional Services), "Building Security into the Software Life Cycle: a Business Case" (paper presented at BlackHat USA, Las Vegas, NV, August 2–3, 2006)..

105 Charles H. LeGrand (CHL Global Associates), *Managing Software Risk: an Executive Call to Action* (Waltham, MA: Ounce Labs, September 21, 2005).

106 Microsoft Corporation, "Understanding the Security Risk Management Discipline," revised May 31, 2006, chap. 3 in *Securing Windows 2000 Server* (Redmond, WA: Microsoft Corporation, November 17, 2004.
Available from: http://www.microsoft.com/technet/security/prodtech/windows2000/secwin2k/swin2k03.mspx

107 Gary E. McGraw (Cigital Inc.), *Risk Management Framework (RMF)* (Washington, DC: US CERT, September 21, 2005).
Available from: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/risk/250.html

108 Morana, *Building Security into the Software Life Cycle, op. cit.*

109 David P. Gilliam (NASA Jet Propulsion Laboratory), "Security Risks: Management and Mitigation in the Software Life Cycle," in *Proceedings of the Thirteenth IEEE International Workshop on Enabling Technologies*, June 14-16, 2004: 211–216.

110 "Defect Detection and Prevention (DDP)" [web site] (Pasadena, CA: NASA JPL at California Institute of Technology).
Available from: http://ddptool.jpl.nasa.gov/

111 T. Scott Ankrum and Alfred H. Kromholz (The MITRE Corporation), "Structured Assurance Cases: Three Common Standards" (slides presented at the Association for Software Quality [ASQ] Section 509 Software Special Interest Group meeting, McLean, VA, January 23, 2006).
Available from: http://www.asq509.org/ht/action/GetDocumentAction/id/2132

112 In the nuclear power industry, as long ago as 1965, the UK's Nuclear Installations Act was adopted, which required the submission of a safety case as part of a nuclear facility's application for a license to operate. See Peter Wilkinson (Australian Department of Industry, Tourism and Resources), "Safety Cases, Success or Failure?" (seminar paper, May 2, 2002).
Available from: http://www.ohs.anu.edu.au/publications/pdf/seminar_paper_2.pdf.

See also, Dr. Robin Pitblado and Dr. Edward Smith (DNV London), "Safety Cases for Aviation. Lessons from Other Industries," in *Proceedings of the International Symposium on Precision Approach and Automatic Landing*, July 18–20, 2000.
Available from: http://www.dnv.com/binaries/SafetyCasesAviation_tcm4-85501.pdf and

Lord Cullen, "The Development of Safety Legislation" (lecture at the Royal Academy of Engineering and Royal Society of Edinburgh, 2006).
Available from: http://www.royalsoced.org.uk/events/reports/rae_1996.pdf

113 The term may have first been applied to software in the context of the Strategic Defense Initiative. See US Congress Office of Technology Assessment, *SDI, Technology, Survivability, and Software*, report no. OTA-ISC-353 (Washington, DC: US Government Printing Office, May 1988).
Available from: http://govinfo.library.unt.edu/ota/Ota_3/DATA/1988/8837.PDF

114 "SafSec" [web page] (Bath, Somerset, UK: Praxis High Integrity Systems Ltd.).
Available from: http://www.praxis-his.com/safsec/index.asp

115 Sherry Hampton, New York, letter to Paul Croll, King George, VA, June 8, 2006 (approval by IEEE Standards Board of new Project P15026).
Available from: http://standards.ieee.org/board/nes/projects/15026.pdf

116 Robin E. Bloomfield, *et al., Assurance Cases for Security: Report of the Workshop on Assurance Cases for Security*, vers. 01c (January 17, 2006).
Available from: http://www.csr.city.ac.uk/AssuranceCases/Assurance_Case_WG_Report_180106_v10.pdf

117 "The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks" [web page] (New York, NY: IEEE *and* Graz, Austria: International Federation for Information Processing [IFIP]).
Available from: http://www.dsn.org/call/index.htm

118 T. Scott Ankrum and Charles Howell (The MITRE Corporation), "Looking for a Good Argument: Assurance Case Frameworks" (presentation to the meeting of the Software Special Interest Group of the American Society for Quality, Washington, DC, and Maryland Metro Section 509, May 2003).
Available from: http://www.asq509.org/ht/action/GetDocumentAction/id/476

119 Since Metricon 1.0, a second "mini-Metricon" was held in February 2007 at the University of San Francisco. See "Metricon 1.0" web page. securitymetrics.org [Last updated September 20, 2006, by Andrew Jaquith].
Available from: http://www.securitymetrics.org/content/Wiki.jsp?page=Metricon1.0

120 The same can be said about the safety of safes. Although the physical space and the testing methods of safes are much more mature than they are for software, safe security ratings are actually fairly crude and heavily reliant on the disclaimer that the ratings apply only "under given circumstances." New attacks or tailored attacks against different safes (of which, compared with different software packages, there are extremely few) are sometimes discovered or developed as the attack tools or techniques advance, *e.g.*, drilling a particular spot on the safe will release the locking mechanism; unless that spot is known, having a strong drill bit will not in and of itself make the attack possible.

121 "SSE-CMM Security Metrics" [web page] (Herndon, VA: International Systems Security Engineering Association [ISSEA]).
Available from: http://www.sse-cmm.org/metric/metric.asp

122 Ernst and Young, *Using Attack Surface Area and Relative Attack Surface Quotient to Identify Attackability* (New York, NY: Ernst and Young Security and Technology Solutions, May 6, 2003).
Available from: http://www.microsoft.com/windowsserver2003/docs/AdvSec.pdf

123 Crispin Cowan, "Relative Vulnerability: An Empirical Assurance Metric", Presented at the 44th International Federation for Information Processing Working Group 10.4 Workshop on Measuring Assurance in Cyberspace (Monterey, CA, 25-29 June 2003)

124 Brian Chess and Tsipenyuk Katrina, "A Metric for Evaluating Static Analysis Tools", Presented at Metricon 1.0 (Vancouver, BC, Canada, 1 August 2006).

125 Pratsuya Manadhata and Jeannette M. Wing (CMU), *An Attack Surface Metric* (Pittsburgh, PA: CMU, July 2005).
Available from: http://www.cs.cmu.edu/~wing/publications/CMU-CS-05-155.pdf

126 O.H. Alhazmi, Y. K. Malaiya, and I. Ray (Colorado State University), "Security Vulnerabilities in Software Systems: a Quantitative Perspective," in *Proceedings of the IFIP WG 11.3 Working Conference on Data and Applications Security*, Storrs, CT, August 2005.
Available from: http://www.cs.colostate.edu/~malaiya/635/IFIP-10.pdf

127 Pravir Chandra, "Code Metrics", Presented at Metricon 1.0 (Vancouver, BC, Canada, 1 August 2006).

128 Russell R. Barton, William J. Hery, and Peng Liu (Pennsylvania State University), "An S-vector for Web Application Security Management," working paper (Pennsylvania State University, University Park, PA, January 2004).
Available from: http://www.smeal.psu.edu/cdt/ebrcpubs/res_papers/2004_01.pdf

129 John Murdoch (University of York), "Security Measurement White Paper," vers. 3.0. (Washington, DC: Practical Software and Systems Measurement [PSM] Safety and Security Technical Working Group, January 13, 2006).
Available from: http://www.psmsc.com/Downloads/TechnologyPapers/SecurityWhitePaper_v3.0.pdf

130 Riccardo Scandariato, Bart De Win, and Wouter Joosen (Catholic University of Leuven), "Towards a Measuring Framework for Security Properties of Software," in *Proceedings of the Second ACM Workshop on Quality of Protection*, October 27–30, 2006.

131 Victor R. Basili and Gianluigi Caldiera (University of Maryland), and H. Dieter Rombach (University of Kaiserslautern), *The Goal Question Metric Approach* (1994).
Available from: http://wwwagse.informatik.uni-kl.de/pubs/repository/basili94b/encyclo.gqm.pdf

132 Thomas Heyman and Huygens Christophe, Catholic University of Leuven; "Software Security Patterns and Risk", Presented at Metricon 1.0 (Vancouver, BC, Canada, 1 August 2006).

133 McDermott, *Attack-Potential-Based Survivability Modeling for High-Consequence Systems, op. cit.*

134 A metric first proposed in F. Stevens, "Validation of an Intrusion-Tolerant Information System Using Probabilistic Modeling" (MS thesis, University of Illinois, Urbana-Champaign, IL, 2004).
Available from: http://www.crhc.uiuc.edu/PERFORM/Papers/USAN_papers/04STE01.pdf

135 National Computer Security Center (NCSC), *A Guide to Understanding Configuration Management in Trusted Systems*, report no. NCSC-TG-006-88 (Fort Meade, MD: National Security Agency, March 28, 1988).
Available from: http://csrc.nist.gov/secpubs/rainbow/tg006.txt *or*
http://www.fas.org/irp/nsa/rainbow/tg006.htm *or*
http://www.iwar.org.uk/comsec/resources/standards/rainbow/NCSC-TG-006.html

136 Premkumar T. Devanbu and Stuart Stubblebine, "Software Engineering for Security: a Roadmap," in *Proceedings of the Conference on the Future of Software Engineering*, Limerick, Ireland, June 4–11, 2000: 227–239.
Available from: http://www.stubblebine.com/00icse.pdf or http://www.cs.ucdavis.edu/~devanbu/files/dec14.pdf

137 J.D. Meier, *et al*. (Microsoft Corporation), *Improving Web Application Security: Threats and Countermeasures Roadmap* (Redmond, WA: Microsoft Corporation, June 2003).
Available from: http://msdn2.microsoft.com/en-us/library/aa302420.aspx#c04618429_009

138 Alexis Leon, "Configuration Item Selection," chap. 7, sec. 7.4 in *Software Configuration Management Handbook*, 2nd ed. (Boston, MA: Artech House Publishers, 2004).

139 "MKS Integrity" [web page] (Waterloo, ON, Canada): MKS Inc.).
Available from: http://www.mks.com/products/index.jsp

140 "Oracle Developer Suite 10g Software Configuration Manager (SCM)" [web page] (Redwood Shores, CA: Oracle Corporation).
Available from: http://www.oracle.com/technology/products/repository/index.html

141 Sparta, Inc., "Information System Security Operation," from *Secure Protected Development Repository* [data sheet] (Columbia, MD: Sparta, Inc.
Available from: http://www.isso.sparta.com/documents/spdr.pdf

142 Schlomi Fish (Better SCM Initiative), "Version Control System Comparison" [web page].
Available from: http://better-scm.berlios.de/comparison/

143 Information Security Forum, *The Standard of Good Practice for Information Security*, vers. 4.1 (London, UK: Information Security Forum, January 2005).
Available from: http://www.isfsecuritystandard.com/

144 Katya Sadovsky, Carmen Roode, and Marina Arseniev (University of California at Irvine), "Best Practices on Incorporating Quality Assurance into Your Software Development Life Cycle" (paper presented at EDUCAUSE 2006, Dallas, TX, October 9–12, 2006).
Available from: http://www.educause.edu/content.asp?page_id=666&ID=EDU06277&bhcp=1

145 Winston W. Royce, "Managing the Development of Large Software Systems, Concepts and Techniques," in *Proceedings of IEEE Wescon*, Los Angeles, CA, 1970 August 1–9, 1970.
Available from: http://www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf

146 "Defense System Software Development: MIL-STD-2167A" [web page].
Available from: http://www2.umassd.edu/SWPI/DOD/MIL-STD-2167A/DOD2167A.html

147 Mei C. Yatoco (University of Missouri-St. Louis), *Joint Application Design/Development* (fall 1999).
Available from: http://www.umsl.edu/~sauter/analysis/JAD.html

148 *Software Development and Documentation*, MIL-STD-498 (Washington, DC: US Department of Defense, November 8, 1994).
Available from: http://www.pogner.demon.co.uk/mil_498/ *and*

Jane Radatz, Myrna Olson, and Stuart Campbell (Logicon). *CrossTalk: The Journal of Defense Software Engineering*, MIL-STD-498 (February 1995).
Available from: http://www.stsc.hill.af.mil/crosstalk/frames.asp?uri=1995/02/MILSTD.asp

149 Evolutionary development has itself evolved into agile development.

150 "Evolutionary Delivery" [web page] (Bilthoven, The Netherlands: N.R. Malotaux Consultancy).
Available from: http://www.malotaux.nl/nrm/Evo/

151 Walter Maner (Bowling Green State University), *Rapid Application Development* (Bowling Green, OH: Bowling Green State University Computer Science Dept., March 15, 1997).
Available from: http://csweb.cs.bgsu.edu/maner/domains/RAD.htm

152 Erik Arisholm and Dag I.K. Sjøberg (University of Oslo), and Jon Skandsen and Knut Sagli (Genera AS), "Improving an Evolutionary Development Process—a Case Study" (paper presented at European Software Process Improvement (EuropSPI99), Pori, Finland, October 25–27, 1999).
Available from: http://www.iscn.at/select_newspaper/process-models/genera.html

153 Barry W. Boehm, *et al.*, "A Software Development Environment for Improving Productivity," *IEEE Computer* 17, no. 6 (June 1984): 30–44.

154 Sun Microsystems, "Workspace Hierarchy Strategies for Software Development and Release," chap. in *Sun WorkShop TeamWare 2.0 Solutions Guide*, and "Concurrent Development," sec. in *Guide* (Mountain View, CA: Sun Microsystems, Inc., 1996).
Available from: http://w3.mit.edu/sunsoft_v5.1/www/teamware/solutions_guide/hierarchy.doc.html#214

155 Scott W. Ambler, (Toronto, ON, Canada: Ambysoft, Inc., December 15, 2006).
Available from: http://www.ambysoft.com/unifiedprocess/

156 IBM/Rational, "Rational Unified Process Best Practices for Software Development Teams," *Rational Software White Paper,* no. TP026B Rev 11/012001 (November 2001).
Available from: http://www-128.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251_bestpractices_TP026B.pdf

157 Scott W. Ambler, *The Agile Unified Process* (Toronto, ON, Canada: Ambysoft, Inc., June 12, 2006).
Available from: http://www.ambysoft.com/unifiedprocess/agileUP.html

158 Scott W. Ambler, "Enterprise Unified Process (EUP)" [home page] (Toronto, ON, Canada: Ambysoft, Inc., March 1, 2006).
Available from: http://www.enterpriseunifiedprocess.com/

159 Konstantin Beznosov and Philippe Kruchten, "Towards Agile Security Assurance," in *Proceedings of the 11th ACM Workshop on New Security Paradigms*, Nova Scotia, Canada, September 2004.
Available from: http://konstantin.beznosov.net/professional/papers/Towards_Agile_Security_Assurance.html

160 During a security push, the entire product team focuses on updating the product's threat models, performing code reviews and security testing, and revising documentation. The objective of the security push is to confirm the validity of the product's security architecture documentation through a focused, intensive effort, uncovering any deviations of the product from that architecture, and identify and remediate any residual security vulnerabilities. A security push compresses activities that would normally be distributed across multiple SDLC phases into a single relatively short time period.

161 "Oracle Software Security Assurance" [web page] (Redwood Shores, CA: Oracle Corporation).
Available from: http://www.oracle.com/security/software-security-assurance.html

162 "CLASP" [web page], *op cit.*

163 James W. Over (CMU SEI), "TSP for Secure Systems Development" (presentation at CMU SEI, Pittsburgh, PA).
Available from: http://www.sei.cmu.edu/tsp/tsp-secure-presentation/

164 Ivan Flechais (Oxford University), and Cecilia Mascolo and M. Angela Sasse (University College London), "Integrating Security and Usability into the Requirements and Design Process," in *Proceedings of the Second International Conference on Global E-Security*, London, UK, April 2006.
Available from: http://www.softeng.ox.ac.uk/personal/Ivan.Flechais/downloads/icges.pdf

165 Sodiya, Adesina Simon; Onashoga, Sadia Adebukola; Ajayi, Olutayo Bamidele. Towards building secure software systems. In: Proceedings of Issues in Informing Science and Information Technology; 2006 June 25-28; Salford, Greater Manchester, England. Vol. 3. Available from: http://informingscience.org/proceedings/InSITE2006/IISITSodi143.pdf

166 Mohammad Zulkernine (Queen's University), and Sheikh Iqbal Ahamed (Marquette University), "Software Security Engineering: Toward Unifying Software Engineering and Security Engineering," chap. XIV in *Enterprise Information Systems Assurance and System Security: Managerial and Technical Issues*, Merrill Warkentin and Rayford B. Vaughn, eds. (Hershey, PA: Idea Group Publishing, 2006).

167 Royce, *Managing the Development of Large Software Systems, op cit.*

168 Dan Wu, Ivana Naeymi-Rad, and Ed Colbert (University of Southern California), "Extending MBASE to Support the Development of Secure Systems," in *Proceedings of the Software Process Workshop*, Beijing, China, May 25–27, 2005.
Available from: http://www.cnsqa.com/cnsqa/jsp/html/spw/download/Copy%20of%20MBASE_Sec_Ext_danwu_abstract%5B1%5D.v1.revisedv2.1.pdf

169 Dan Wu, Ivana Naeymi-Rad, and Ed Colbert, "Extending Mbase to Support the Development of Secure Systems," in *Proceedings of the Software Process Workshop*, Beijing, China, May 25–27, 2006.
Available from: http://www.cnsqa.com/cnsqa/jsp/html/spw/download/Copy%20of%20MBASE_Sec_Ext_danwu_abstract%5B1%5D.v1.revisedv2.1.pdf

170 Secure Software Engineering portal.
Available from: http://www.secure-software-engineering.com/

171 Microsoft Corporation, *Security Engineering Explained* (Redmond, WA: Microsoft Corporation Patterns and Practices Developer Center, October 2005).
Available from: http://msdn2.microsoft.com/en-us/library/ms998382.aspx

172 Nancy R. Mead (CMU SEI), *Requirements Engineering for Survivable Systems*, tech. note no. CMU/SEI-2003-TN-013, Figure 1, "Coarse-Grain Requirements Engineering Process" (Pittsburgh, PA: CMU SEI, September 2003): 2.
Available from: http://www.cert.org/archive/pdf/03tn013.pdf

173 Jim Johnson (The Standish Group), "Return on Requirements" (presentation at Choas University, Half Moon Bay, CA, February 29–March 3, 2004). Johnson asserts that half of all project failures are caused by faulty or inadequate requirements.

174 Darwin Ammala, "A New Application of CONOPS in Security Requirements Engineering," *CrossTalk: The Journal of Defense Software Engineering* (August 2000).
Available from: http://www.stsc.hill.af.mil/Crosstalk/2000/08/ammala.html

175 Philip E. Coyle (OSD Operational Test and Evaluation), "Simulation-Based Acquisition for Information Technology" (presentation at the Academia, Industry, Government Crosstalk Conference, Washington, DC, May 18, 1999).
Available from: http://www.dote.osd.mil/presentations/Coyle051899/sld001.htm

176 A. Rashid, A.M.D. Moreira, and J. Araújo, "Modularisation and Composition of Aspectual Requirements," in *Proceedings of the Second International Conference on Aspect-Oriented Software Development*, Boston, MA, March 17–21, 2003: 11–20; *and* A. Rashid, A.M.D. Moreira, P. Sawyer, and J. Araújo, "Early Aspects: a Model for Aspect-Oriented Requirement Engineering," in *Proceedings of the IEEE Joint International Conference on Requirements Engineering*, Essen, Germany, September 9–13, 2002: 199–202.

177 Axel van Lamsweerde, Simon Brohez, Renaud De Landtsheer, and David Janssens, "From System Goals to Intruder Anti-Goals: Attack Generation and Resolution for Security Requirements Engineering," in *Proceedings of the Requirements for High Assurance Workshop*, Monterey Bay, CA, September 8, 2003, 49–56. [*Also of interest in the same Proceedings*: Sascha Konrad, *et al.*, "Using Security Patterns to Model and Analyze Security Requirements": 13–22.]
Available from: http://www.sei.cmu.edu/community/rhas-workshop/rhas03-proceedings.pdf *or* http://publica.fhg.de/documents/N-20881.html *and*

Axel van Lamsweerde and E. Handling Letier, "Obstacles in Goal-Oriented Requirements Engineering," *IEEE Transactions on Software Engineering* 26, no. 10 (October 2000): 978–1005.

178 Ian Alexander, Modelling the Interplay of Con icting Goals With Use and Misuse Cases, in *Proceedings of 8th International Workshop on Requirements Engineering Foundation for Software Quality*, Essen, Germany, September 9–10, 2002: 145–152.

179 G. Sindre and A.L. Opdahl, "Eliciting Security Requirements by Misuse Cases," in *Proceedings of the 37th International Conference on Technology of Object-Oriented Languages and Systems*, Sydney, Australia, November 20-23, 2000: 120–131; *and ibid.*, "Templates for Misuse Case Description," in *Proceedings of the Seventh International Workshop on Requirements Engineering Foundation for Software Quality*, Interlaken, Switzerland, June 4–5, 2001.

180 John J. McDermott (NRL CHACS), "Abuse–Case-Based Assurance Arguments," in *Proceedings of the 17th Annual Computer Security Applications Conference*, New Orleans, LA, December 10–14, 2001: 366–374.

181 H. In and Barry W. Boehm, "Using Win-Win Quality Requirements Management Tools: a Case Study," *Annals of Software Engineering* 11 no. 1 (November 2001): 141–174.

182 C.L. Heitmeyer, "Applying 'Practical' Formal Methods to the Specification and Analysis of Security Properties," in *Proceedings of the International Workshop on Information Assurance in Computer Networks, Methods, Models, and Architectures for Network Computer Security*, St. Petersburg, Russia, May 21–23, 2001: 84–89.

183 Charles B. Haley, Robin C. Laney, and Bashar Nuseibeh (The Open University), "Deriving Security Requirements From Crosscutting Threat Descriptions," in *Proceedings of the Third International Conference on Aspect-Oriented Software Development*, Lancaster, UK, March 22–26, 2004: 112–121.
Available from: http://mcs.open.ac.uk/cbh46/papers/AOSD04.pdf

184 John Wilander and Jens Gustavsson (Linköpings University), "Security Requirements: a Field Study of Current Practice," in *Proceedings of the Third Symposium on Requirements Engineering for Information Security*, Paris, France, August 29, 2005.
Available from: http://www.ida.liu.se/~johwi/research_publications/paper_sreis2005_wilander_gustavsson.pdf

185 Charles B. Haley, Jonathan D. Moffett, Robin Laney, and Bashar Nuseibeh (The Open University), "A Framework for Security Requirements Engineering," in *Proceedings of the Second Software Engineering for Secure Systems Workshop*, Shanghai, China, May 20–21, 2006: 5–42.
Available from: http://mcs.open.ac.uk/cbh46/papers/Haley-SESS06-p35.pdf

186 Defense Information Systems Agency (DISA), *Application Security Checklist*, vers. 2, Release 1.9 (Chambersburg, PA: DISA Field Security Operation, November 24, 2006).
Available from: http://iase.disa.mil/stigs/checklist/app-security-checklist-v2r19-24Nov06.doc

187 Monika Vetterling, Guido Wimmel, and Alexander Wißpeintner, "Secure Systems Development Based on the Common Criteria," in *Proceedings of the 10th International Symposium on the Foundations of Software Engineering* (FSE-10), 2000.
Available from: http://www4.in.tum.de/~wimmel/papers/VWW02_FSE.pdf.

[*Also of interest*: Indongesit Mkpong-Ruffin and David A. Umphress, "High-Leverage Techniques for Software Security," *CrossTalk: The Journal of Defense Software Engineering* 20, no. 3 (March 2007): 18–21. Available from: http://www.stsc.hill.af.mil/CrossTalk/2007/03/0703RuffinUmphress.html]

188 Paco Hope and Gary McGraw (Cigital, Inc.), and Annie I. Antón (North Carolina State University), "Misuse and Abuse Cases: Getting Past the Positive," *IEEE Security and Privacy* (May-June 2004): 32–34.
Available from: http://www.cigital.com/papers/download/bsi2-misuse.pdf

189 Nancy R. Mead (CMU SEI), *Requirements Elicitation Introduction* (Washington, DC: US CERT, September 22, 2006).
Available from: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/requirements/533.html; *and*

ibid., Requirements Elicitation Case Studies Using IBIS, JAD, and ARM (Washington, DC: US CERT, September 22, 2006).
Available from: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/requirements/532.html

190 Bruce Schneier, "Attack Trees," *Dr. Dobbs Journal* (December 1999).
Available from: http://www.schneier.com/paper-attacktrees-ddj-ft.html

191 "Common Vulnerability Scoring System Special Interest Group" [web page] (Mountain View, CA: Forum of Incident Response and Security Teams).
Available from: http://www.first.org/cvss/

192 Davide Balzarotti (Milan Polytechnic), Mattia Monda (University of Milan), and Sabrina Sicari (University of Catania), "Assessing the Risk of Using Vulnerable Components," chap. in *Quality of Protection: Security Measurements and Metrics*, Dieter Gollmann, Fabio Massacci, and Artsiom Yautsiukhin eds. (New York, NY: Springer, 2006).
Available from: http://homes.dico.unimi.it/~monga/lib/qop.pdf

193 Saurabh Bagchi (Purdue University), "Achieving High Survivability in Distributed Systems Through Automated Intrusion Response" (presentation at the meeting of IFIP Working Group 10.4, San Francisco, CA, July 2, 2004).
Available from: http://cobweb.ecn.purdue.edu/~dcsl/Presentations/2006/ifip_irs_070206.pdf

194 "Automated Security Self-Evaluation Tool" [web page] (Gaithersburg, MD: NIST Computer Security Division Computer Security Resource Center).
Available from: http://csrc.nist.gov/asset/

195 "CRAMM" [web page] (Walton-on-Thames, Surrey, UK: Siemens Insight Consulting).
Available from: http://www.cramm.com/

196 Donald L. Buckshaw (Cyber Defense Agency), *et al.*, "Mission Oriented Risk and Design Analysis of Critical Information Systems," *Military Operations Research* 10, no. 2 (November 2005).
Available from: http://www.mors.org/awards/mor/2006.pdf

197 "OCTAVE" [web page] (Pittsburgh, PA: CMU SEI CERT).
Available from: http://www.cert.org/octave/

198 Donald Firesmith (CMU SEI), "Modern Requirements Specification," *Journal of Object Technology* 2, no. 2 (March–April 2003): 53–64.
Available from: http://www.jot.fm/issues/issue_2003_03/column6

199 Paco Hope and Gary McGraw (Cigital, Inc.), and Annie I. Antón (North Carolina State University), "Misuse and Abuse Cases: Getting Past the Positive," *IEEE Security and Privacy* (May-June 2004): 32–34.
Available from: http://www.cigital.com/papers/download/bsi2-misuse.pdf

200 Meledath Damodaran, "Secure Software Development Using Use Cases and Misuse Cases," *Issues in Information Systems* VII, no. 1 (2006): 150–154.
Available from: http://www.iacis.org/iis/2006_iis/PDFs/Damodaran.pdf

201 Lamsweerde, *et al., From System Goals to Intruder Anti-Goals, op cit.*

202 Nancy R. Mead (CMU SEI), SQUARE: *Requirements Engineering for Improved Systems Security* (Pittsburgh, PA: CMU SEI CERT, April 21, 2006).
Available from: http://www.cert.org/sse/square.html

203 Constance L. Heitmeyer, *Software Cost Reduction* (Washington, DC: NRL CHACS, 2002).
Available from: http://chacs.nrl.navy.mil/publications/CHACS/2002/2002heitmeyer-encse.pdf

204 Object Management Group (OMG), "Unified Modeling Language: UML Resource Page," (Needham, MA: OMG [last updated January 2, 2007].
Available from: http://www.uml.org/

205 "Security and Safety in Software Engineering" [web page] (Munich, Germany: Technical University of Munich).
Available from: http://www4.in.tum.de/~juerjens/secse/

206 "Software Application Security Services (SASS) Tools: SecureUML" [web page] (Mission Viejo, CA: McAfee Foundstone Division, August 3, 2005).
Available from: http://www.foundstone.com/index.htm?subnav=resources/navigation.htm&subcontent=/resources/proddesc/secureuml.htm

207 For examples of their work, see Geri Georg, Siv Hilde Houmb, and Indrakshi Ray, "Aspect-Oriented Risk Driven Development of Secure Applications" (paper submitted to the 20th Annual IFIP 11.3 Working Conference on Data and Applications Security, Sophia Antipolis, France, July 31–August 2, 2006). Available from: http://www.idi.ntnu.no/grupper/su/publ/siv/DBSEC_georg.pdf; *and*

Geri Georg and Siv Hilde Houmb, "The Aspect-Oriented Risk-Driven Development (AORDD) Framework," in *Proceedings of the International Conference on Software Development*, Reykjavik, Iceland, June 2005: 81–91.
Available from: http://www.idi.ntnu.no/grupper/su/publ/siv/swde-2005-houmb.pdf; *and*

Geri Georg, Indrakshi Ray, and Robert France, "Using Aspects to Design a Secure System" in *Proceedings of the Eighth IEEE International Conference on Engineering of Complex Computer Systems*, Greenbelt, MD, December 2–4, 2002.
Available from: http://www.cs.colostate.edu/~georg/aspectsPub/ICECCS02.pdf

208 Martin Croxford and Roderick Chapman (Praxis High Integrity Systems), "Correctness by Construction: a Manifesto for High-Integrity Software," *CrossTalk: The Journal of Defense Software Engineering* 18, no. 12 (December 2005).
Available from: http://www.stsc.hill.af.mil/CrossTalk/2005/12/0512CroxfordChapman.html' *and*

Martin Croxford and Roderick Chapman (Praxis High Integrity Systems), "The Challenge of Low Defect, Secure Software: Too Difficult and Too Expensive?" *DoD Software Tech News* 8, no. 2 (July 2005).
Available from: http://www.softwaretechnews.com/stn8-2/praxis.html

209 Bertrand Meyer, "On Formalism in Specifications," *IEEE Software 2*, no. 1 (January 1985): 6–26.

210 Sascha Konrad, *et al., Using Security Patterns to Model and Analyze Security Requirements.*

211 "Tropos Requirements-Driven Development for Agent Software" [web page] (Trento, Italy: Università degli Studi di Trento [Last updated March 4, 2007].
Available from: http://www.troposproject.org/

212 Paolo Giorgini, *et al.* (University of Trento), *Requirements Engineering Meets Trust Management: Model, Methodology, and Reasoning*, tech. report no. DIT-04-016 (Trento, Italy: University of Trento, 2004). Available from: http://eprints.biblio.unitn.it/archive/00000534/ *and*

Haralambos Mouratidis and Gordon Manson (University of Sheffield), and Paolo Giorgini (University of Trento), "An Ontology for Modelling Ssecurity: the Tropos Approach," in *Proceedings of the Seventh International Conference on Knowledge-Based Intelligent Information and Engineering Systems*, Oxford, UK, September 4, 2003.
Available from: http://dit.unitn.it/~pgiorgio/papers/omasd03.pdf

213 Joshua J. Pauli and Dianxiang Xu (Dakota State University), "Misuse Case-Based Design and Analysis of Secure Software Architecture," in *Proceedings of the International Conference on Information Technology, Coding and Computing* 2, April 4-6, 2005: 398–403.
Available from: http://www.cs.ndsu.nodak.edu/~dxu/publications/pauli-xu-ITCC05.pdf

214 Anthony Hall and Roderick Chapman (Praxis High Integrity Systems), "Correctness by Construction: Developing a Commercial Secure System," *IEEE Software* (January–February 2002): 18–25.

215 Saltzer and Schroeder, *The Protection of Information in Computer Systems.*

216 Samuel T. Redwine, Jr., ed., *Secure Software: a Guide to the Common Body of Knowledge to Produce, Acquire, and Sustain Secure Software*, draft vers. 1.1 (Washington, DC: US CERT, August 31, 2006). A subsequent version, draft vers. 1.1, dated April 2, 2007,
Available from: https://buildsecurityin.us-cert.gov/daisy/bsi/resources/dhs/95.html?branch=1&language=1

217 The practice of openly exposing one's design and algorithms to scrutiny is widely accepted in the cryptographic community, with the exception of NSA and possibly some of NSA's counterparts in other countries.

218 Morrie Gasser, *Building a Secure Computer System* (New York, NY: Van Nostrand Reinhold, 1988).

219 "xADL 2.0" [home page] (Irvine, CA: University of California Institute for Software Research, 2000–2005). Available from: http://www.isr.uci.edu/projects/xarchuci/

220 Michael E. Shin (Texas Tech University), "Modeling of Evolution to Secure Application System: From Requirements Model to Software Architecture," in *Proceedings of the International Conference on Software Engineering Research and Practice*, Las Vegas, NV, June 26–29, 2006.
Available from: http://ww1.ucmss.com/books/LFS/CSREA2006/SER5235.pdf

221 Sean Barnum and Amit Sethi (Cigital, Inc.), *Attack Pattern Glossary* (Washington, DC: US CERT, November 11, 2006).
Available from: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/attack/590.html

222 Markus Schumacher, *et al., Security Patterns: Integrating Security and Systems Engineering* (New York, NY: John Wiley & Sons, 2005).
For more information, see http://www.securitypatterns.org/

223 The respository is no longer maintained online, but its contents have been published privately by two of the the researchers involved, in the following paper: Darrell M. Kienzle, *et al., Security Patterns Repository Version 1.0*.
Available from: http://www.modsecurity.org/archive/securitypatterns/dmdj_repository.pdf *or* http://www.scrypt.net/~celer/securitypatterns/repository.pdf.

224 Darrell M. Kienzle and Matthew C. Elder, *Security Patterns for Web Application Development, Final Technical Report* (November 4, 2003).
Available from: http://www.modsecurity.org/archive/securitypatterns/dmdj_final_report.pdf *or* http://www.scrypt.net/~celer/securitypatterns/final per cent20report.pdf

225 Bob Blakley (IBM) and Craig Heath (Symbian), *Technical Guide to Security Design Patterns*, cat. no. G031 (San Francisco, CA: The Open Group, April 2004).
Available from: http://www.opengroup.org/products/publications/catalog/g031.htm

226 Christopher Steel, Ramesh Nagappan, and Ray Lai, *Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management* (Indianapolis, IN: Prentice Hall Professional Technical Reference, 2006).
For more information see: http://www.coresecuritypatterns.com/

227 Spyros T. Halkidis, Alexander Chatzigeorgiou, and George Stephanides (University of Macedonia), "A Practical Evaluation of Security Patterns," in *Proceedings of the Sixth International Conference on Artificial Intelligence and Digital Communications*, Thessaloniki, Greece, August 18–20, 2006.
Available from: http://www.inf.ucv.ro/~aidc/proceedings/2006/5 per cent20shalkidis.pdf

228 "First International Workshop on Secure Systems Methodologies Using Patterns" [web page] (Regensburg, Germany: University of Regensburg).
Available from: http://www-ifs.uni-regensburg.de/spattern07/

229 Peter Amey (Praxis High Integrity Systems), *Correctness by Construction* (Washington, DC: US CERT, December 5, 2006).
Available from: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/sdlc/613.html?branch=1&language=1

230 Hall and Chapman, *Correctness by Construction: Developing a Commercial Secure System, op cit.*

231 W. Wang, *et al.*, "e-Process Design and Assurance Using Model Checking," *IEEE Computer* 33, no. 10 (October 2000): 48–53.

232 Peter Ryan, *et al., Modelling and Analysis of Security Protocols*, 2nd ed. (Addison-Wesley Professional, 2001).

233 Edsger W. Dijkstra (Burroughs Corporation), "Guarded Commands, Nondeterminacy and Formal Derivation of Programs," *Communications of the ACM* 18, no. 8 (August 1975): 453–457.

234 Harlan D. Mills, *Software Productivity* (New York, NY: Dorset House Publishing, 1988).

235 Tony Hoare (Microsoft Research), *Communication Sequential Processes* (Upper Saddle River, NJ: Prentice-Hall International, 1985).
Available from: http://www.usingcsp.com/

236 For those unfamiliar with the concept of SOA, helpful background information can be found at: the "Web Services and Service-Oriented Architectures" web site. (Burnsville, MN: Barry & Associates, Inc.).
Available from: http://www.service-architecture.com/index.html; *and in*

Organization for the Advancement of Structured Information Standards (OASIS), Reference Model for Service Oriented Architecture, vers. 1.0 [Approved] (August 2, 2006).
Available from: http://www.oasis-open.org/committees/download.php/19679/soa-rm-cs.pdf; *and in*

"World Wide Web Consortium (W3C): Web Services Architecture," working group note (February 11, 2004).
Available from: http://www.w3.org/TR/ws-arch/

237 José Luíz Fiadeiro." Designing for Software's Social Complexity," *IEEE Computer* 40, no. 1 (January 1, 2007): 34–39.

238 Bertrand Meyer, *Design by Contract*, tech. report no. TR-EI-12/CO (Santa Barbara, CA: Eiffel Software Inc. [formerly Interactive Software Engineering Inc.], 1986).

239 "Eiffel Software" [web site] (Goleta (CA): Eiffel Software Inc.).
Available from: http://www.eiffel.com/ ; *and*

Bertrand Meyer, *Basic Eiffel Language Mechanisms* (August 2006).
Available from: http://se.ethz.ch/~meyer/publications/online/eiffel/basic.html; *and*

Ecma International (formerly the European Computer Manufacturers Association), *Eiffel: Analysis, Design and Programming Language*, standard ECMA-367, 2nd ed. (June 2006). (Note that the Eiffel standard was also approved by ISO/IEC as ISO/IEC 25436.).
Available from: http://www.ecma-international.org/publications/standards/Ecma-367.htm

240 Stephen H. Edwards, M. Sitaraman, B.W. Weide, and E. Hollingsworth (Virginia Tech), "Contract-Checking Wrappers for C++ Classes," *IEEE Transactions on Software Engineering 30*, no. 11 (November 2004): 794–810.

241 Yves Le Traon, Benoit Baudry, and Jean-Marc Jezequel, "Design by Contract to Improve Software Vigilance," *IEEE Transactions on Software Engineering* 32, no. 8 (August 2006): 571–586.

242 Jean-Marc Jezequel and Bertrand Meyer, "Design by Contract: the Lessons of Ariane," *IEEE Computer* (January 1997): 129–130.

243 Sometimes called Fagan inspections, after Michael Fagan who is credited with inventing formal software inspections.

244 BSD = Berkeley Software Distribution.

245 Viega and McGraw, *Building Secure Software, op cit.*

246 Mike Sues, Wendy-Anne Daniel, and Marcel Gingras (Cinnabar Networks, Inc.), "Secure Programming and Development Practices," slide presentation (Ottawa, ON, Canada: Cinnabar Networks, Inc., 2001).
Available from: http://www.cinnabar.ca/library/SecureProgramingTutorial.ppt

247 M.E. Kabay (Norwich University), "Programming for Security," part 1 of *Network World Security Newsletter* (June 4, 2001).
Available from: http://www.networkworld.com/newsletters/sec/2001/00853827.html.

Programming for security, Part 2. Network World Security Newsletter. 2001 June 6.
Available from: http://www.networkworld.com/newsletters/sec/2001/00853837.html.

Programming for security, Part 3. Network World Security Newsletter. 2001 June 11.
Available from: http://www.networkworld.com/newsletters/sec/2001/00871502.html.

Programming for security, Part 4. Network World Security Newsletter. 2001 June 13.
Available from: http://www.networkworld.com/newsletters/sec/2001/00871525.html.

Revised version of all four parts. 2004.
Available from: http://www2.norwich.edu/mkabay/overviews/programming.pdf.

See also: "Mailbag: Programming for Security," *Network World Security Newsletter* (September 1, 2001). Available from: http://www.networkworld.com/newsletters/sec/2001/00991582.html

248 Robert J. Seacord (CMU SEI), "Secure Coding Standard" [web page] (Pittsburgh, PA: CMU SEI CERT). Available from: http://www.securecoding.cert.org/

249 "Information about Standard ML" [web page] (Pittsburgh, PA: CMU School of Computer Science). Available from: http://www.cs.cmu.edu/afs/cs.cmu.edu/project/fox/mosaic/sml.html. Note that ML = metalanguage.

250 "MISRA C" [web site] (Nuneaton, Warwickshire, UK: Motor Industry Software Reliability Association [MISRA]).
Available from: http://www.misra-c2.com/

251 T.M. Austin, S.E. Breach, and G.S. Sohi, "Efficient Detection of All Pointer and Array Access Errors," in *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, Orlando, FL, June 20–24, 1994.

252 CCured "adds a minimal number of run-time checks (in C) to C programs "to prevent all memory safety violations. The resulting program is memory safe, meaning that it will stop rather than overrun a buffer or scribble over memory that it shouldn't touch." For more information, see: "CCured Documentatio" [web page] (Berkeley, CA: University of California).
Available from: http://manju.cs.berkeley.edu/ccured; *and*

George C.. Necula, *et al.* (University of California at Berkeley), *CCured: Type-Safe Retrofitting of Legacy Software, ACM Transactions on Programming Languages and Systems* 27, no.3 (May 2005): 477–526.
Available from: http://www.cs.berkeley.edu/~necula/Papers/ccured_toplas.pdf

253 Originally developed by Michael Hicks, University of Maryland. For more information see: "CYCLONE Project" website. Available from: http://cyclone.thelanguage.org/

254 "Vault: a Programming Language for Reliable Systems Project" [web page] (Redmond, WA: Microsoft Research).
Available from: http://research.microsoft.com/vault/

255 Sumant Kowshik, Dinakar Dhurjati, and Vikram Adve (University of Illinois at Urbana-Champaign), "Ensuring Code Safety Without Runtime Checks for Real-Time Control Systems," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, Grenoble, France, October 8–11, 2002: 288-297.

256 Fail-Safe C "disallows any unsafe memory operation in full ANSI C standard (including casts and unions)." For more information, see: "Fail-Safe C Project" web page (Tokyo, Japan: National Institute of Advanced Industrial Science and Technology Research Center for Information Security).
Available from: http://www.rcis.aist.go.jp/project/FailSafeC-en.html *and*

"Fail-Safe C version 1.0" [web page] (Japan: Tatsurou Sekiguchi, March 29, 2005).
Available from: http://homepage.mac.com/t.sekiguchi/fsc/index.html

257 Les Hatton, *Safer C: Developing Software in High Integrity and Safety-Critical Systems* (Maidenhead, Berkshire, UK: McGraw-Hill Book Company Europe, 1995).

258 Chandrasekhar Boyapati (Massachusetts Institute of Technology), "SafeJava: a Unified Type System for Safe Preprogramming" (PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, February 2004). Available from: http://www.pmg.lcs.mit.edu/~chandra/publications/phd.pdf

259 SPARKAda was designed "for high integrity applications in which many errors are impossible and which has decades of proven results." Available from: "SparkADA" [web page] (Bath, Somerset, UK: Praxis High Integrity Systems Ltd).
Available from: http://www.praxis-his.com/sparkada/

260 *Hermes* publications web page (Hawthorne, NY: IBM Watson Research Center).
Available from: http://www.research.ibm.com/people/d/dfb/hermes-publications.html *and*

Willard Korfhage (Polytechnic University), "Hermes Language Experiences," *Software: Practice and Experience* 25, no. 4 (April 1995): 389-402.
Available from: http://www.cs.ubc.ca/local/reading/proceedings/spe91-95/spe/vol25/issue4/spe950wk.pdf

261 "ERights.org" [home page] (California: ERights.org).
Available from: http://www.erights.org/ *and*

Stiegler, Marc. The E language in a walnut. Draft. 2000.
Available from: http://www.skyhunter.com/marcs/ewalnut.html

262 Fred Spiessens and Peter Van Roy (Catholic University of Leuven), "The Oz-E Project: Design Guidelines for a Secure Multiparadigm Programming Language," in *Proceedings of the Second International Conference on Multiparadigm Programming in Mozart/OZ*, Charleroi, Belgium, October 7–8, 2004, Peter Van Roy (Catholic University of Leuven), ed. (Berlin, Germany: Springer-Verlag, 2005): 3389.
Available from: http://www.info.ucl.ac.be/~pvr/oze.pdf *or*
http://www.info.ucl.ac.be/people/PVR/oze.pdf

263 Chris Hawblitzel (Dartmouth University), "Clay Research Activities and Findings" [web page] (Hanover, NH: Dartmouth University Computer Science Department).
Available from: http://www.cs.dartmouth.edu/research/node101.html

264 Matt Messier and John Viega, "Safe C String Library v1.0.3" [web page] (January 30, 2005).
Available from: http://www.zork.org/safestr/

265 "Libsafe Research" [web page] (Basking Ridge, NJ: Avaya Labs).
Available from: http://www.research.avayalabs.com/gcm/usa/en-us/initiatives/all/nsr.htm&Filter=ProjectTitle:Libsafe&Wrapper=LabsProjectDetails&View=LabsProjectDetails

266 For example, the Template-Based Classes for Microsoft's Visual C++. For more information, see: Microsoft Corporation, "Template-based Classes," in *Visual Studio 2005 Version 8.0* (Redmond, WA: MSDN Library).
Available from: http://msdn2.microsoft.com/en-us/library/f728cbk3(VS.80).aspx *and*

"MAXcode, Template-based Programming" (The Code Project, March 25, 2002).
Available from: http://www.codeproject.com/cpp/maxcode.asp

267 Les Hatton, "Safer Language Subsets: an Overview and a Case History—MISRA C.," *Information and Software Technology* 46 (2004): 465–472.
Available from: http://www.leshatton.org/IST_1103.html

268 "SSCC: The Plum Hall Safe-Secure C/C++ Project" [web page] (Kamuela, HI: Plum Hall Inc.).
Available from: http://www.plumhall.com/sscc.html

269 "SCC: The Safe C Compiler" [web page] (Madison, WI: University of Wisconsin Department of Computer Science [Last updated September 26, 1995]).
Available from: http://www.cs.wisc.edu/~austin/scc.html

270 "Memory Safe C Compiler" [web page] (Stony Brook, NY: State University of New York Secure Systems Laboratory; 1999–2002).
Available from: http://www.seclab.cs.sunysb.edu/mscc/

271 Dowd, *et al., The Art of Software Security Assessment, Identifying and Preventing Software Vulnerabilities. Also see:* Chris Wysopal, *et al., The Art of Software Security Testing: Identifying Software Security Flaws*, 1st ed. (Boston, MA: Addison-Wesley Professional, 2006). *Also see:* Andres Andreu, *Professional Pen Testing for Web Applications* (Indianapolis, IN: Wrox/Wiley Publishing Inc., 2006). *Also see:* Tom Gallagher, *et al., Hunting Security Bugs* (Redmond, WA: Microsoft Press, 200).

272 George Fink and Matt Bishop (University of California at Davis), "Property-based Testing: a New Approach to Testing for Assurance," *ACM SIGSOFT Software Engineering Notes* 22, no. 4 (July 1997): 74–80. Available from: http://nob.cs.ucdavis.edu/~bishop/papers/1997-sen/

273 T. Ball, "The Concept of Dynamic Analysis," in *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Toulouse, France, September 1999: 216–234.

274 Note that many Java source code analysis tools actually analyze the bytecode generated from the Java source code rather than the source code itself. Scanners for platform-specific bytecode (*e.g.*, x86 machine language) are less capable than their JVM counterparts because the low-level platform-specific code incorporates few high-level language constructs. For this reason, the binary analysis tool must reconstruct this logic from the machine code, which takes large amounts of processing power and potentially produces incorrect results. Nevertheless, binary analyzers are continuing to improve in their capabilities and can be beneficial in analyzing the security of closed source COTS products.

275 "Black Hat Digital Self-Defense" [web site] (Seattle, WA: Black Hat).
Available from: http://www.blackhat.com/

276 Kris Britton (NSA CAS), "NSA Center for Assured Software" (presentation to the Director of Central Intelligence's Information Security and Privacy Advisory Board, Gaithersburg, MD: NIST Computer Security Division Computer Security Resource Center, 2006 March 21, 2006).
Available from: http://csrc.nist.gov/ispab/2006-03/March-2006.html

277 Djenana Campara, "Software Assurance Ecosystem Infrastructure" (presentation at the OMG Software Assurance Special Interest Group meeting, Washington, DC, December 9, 2006).
Available from: http://swa.omg.org/docs/swa_washington_2006/ OMG_SwA_AB_SIG_Focus_Direction_n_Next_Steps.pdf

278 "SAMATE" [portal page] *op cit.*

279 Michael E. Fagan, "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems Journal* 15, no. 3 (1976): 258–287 *and* Michael E. Fagan, "Advances in Software Inspections," *IEEE Transactions on Software Engineering* SE-12, no. 7 (July 1986): 744–751.

280 J.D. Meier, *et al.* (Microsoft Corporation), "Improving Web Application Security: Threats and Countermeasures," *NET Framework Security* (Redmond, WA: MSDN Developer Center, June 2003.
Available from: http://msdn2.microsoft.com/en-us/library/ms994921.aspx

281 David P. Gilliam, Thomas L. Wolfe, Josef S. Sherif, and Matt Bishop, "Software Security Checklist for the Software Life Cycle," in *Proceedings of the Twelfth International Workshop on Enabling Technologies, Infrastructure for Collaborative Enterprises*, Linz, Austria, June 9–11, 2003: 243.

282 "Unofficial Common Criteria and the Common Evaluation Methodology" [web page], Common Criteria Project.
Available from: http://www.commoncriteriaportal.org/public/expert/index.php?menu=3

283 NIST and UK Government Communications Head Quarters Communications-Electronics Security Group (CESG), *Common Criteria for Information Technology Security Evaluation User Guide* (Fleet, Hampshire, UK: Syntegra UK, February 21, 2002): 9.
Available from: http://www.commoncriteriaportal.org/public/files/ccusersguide.pdf

284 "Security Consensus Operational Readiness Evaluation (SCORE)" [web page] (Bethesda, MD: SANS Institute).
Available from: http://www.sans.org/score/

6

Software Assurance Initiatives, Activities, and Organizations

In the past 5 years, the Department of Defense (DoD) has become increasingly active in pursuit of software security assurance and application security objectives. To this end, it has established a number of programs to provide guidance, security assessments, and other forms of support in these areas. The most significant of these initiatives are described in Section 6.1.

DoD is not alone in its efforts. The Department of Homeland Security (DHS) and the National Institute of Standards and Technology (NIST) are both very involved in these areas of study, and their activities are also described in Section 6.2.

The private sector has also become very active, not just in terms of commercial offerings of tools and services, but in establishing consortia to collectively address different software security and application security challenges. The most noteworthy of these industry initiatives are described in Section 6.1.9.

Section 6.3 describes several international standards under development to address various software security assurance concerns.

Section 6.4 describes legislation with software security elements or implications at the Federal and state levels in the United States.

Note that the United States appears to be alone in the world in establishing software security assurance initiatives at the national government level.

6.1 US Government Initiatives

The following software assurance, software security, and application security initiatives are being sponsored within the US Government.

6.1.1 DoD Software Assurance Initiative

The roots of the DoD Software Assurance Initiative are the recommendations in the Defense Science Board (DSB) Task Force on Globalization and Security's final report of December 1999. These recommendations included a call for the Secretary of Defense to set up a software assurance program at the Assistant Secretary of Defense (ASD) level, and for DoD to enhance its security and counter-intelligence activities to deal with the potential threats introduced by DoD's reliance on commercial software of foreign manufacture.

These recommendations were addressed in July 2003, when the ASD/NII established a Software Assurance Initiative to examine challenges associated with evaluating the assurance risks of commercially acquired software in advance of deployment in government environments. As a follow-on to this initiative, ASD/Networks and Information Integration (NII) formed a Software Assurance Tiger Team in December 2004—in partnership with the Office of the Under Secretary of Defense/Aquisition Technology and Logistics (OUSD/AT&L)—with the goal of developing a holistic strategy to reduce the Federal Government's susceptibility to these risks.

According to the common definition of software assurance proposed by the Software Assurance Tiger Team:

> *Software assurance (SwA) relates to the level of confidence that software functions as intended and is free of vulnerabilities, either intentionally or unintentionally designed or inserted as part of the software.*

Through a rigorous outreach initiative to both government and industry stakeholders, the Tiger Team further proposed a stratagem of guiding principles as the foundation for reducing software assurance risks:

- Understand the problem(s) from a systems perspective
- Tailor responses to the scope of the identified risk
- Ensure responses are sensitive to potential negative impacts, *e.g.*:
  - Degradation of ability to use commercial software
  - Decreased responsiveness/increased time to deploy technology
  - Loss of industry incentive to do business with DoD
  - Minimize burden on acquisition programs
- Exploit and extend relationships with—
  - National, international, and industry partners
  - DoD initiatives, *e.g.,* trusted integrated circuits and information assurance (IA).

The DoD Software Assurance Tiger Team developed a concept of operations for addressing the issue, focusing on the following areas:

- Prioritization (of systems and components)
- Engineering-in-depth

- Supplier assurance
- Science and technology.

The Tiger Team's approach is to address software security issues at the system level. This approach reflects the Tiger Team's belief that some (many?) threats to software security cannot be addressed cost-effectively at the software level.

The Tiger Team collaborates with the National Defense Industrial Association (NDIA) to lead the industry effort for integrating secure system engineering practices into software product development, focusing on the impact of software assurance on information technology implementations. NDIA hosts Software Assurance Summits and coordinates the authoring efforts of the *Guidebook* (see Section 6.1.1.2). Industry standards that NDIA promotes include International Standards Organization (ISO)/ International Electrotechnical Commission (IEC) 15278, American National Standards Institute (ANSI) 632, IEEE 1220, Electronics Industry Association (EIA) 731, and CMMI.

With OUSD/AT&L and ASD/NII oversight and brokering, the Software Assurance Tiger Team has extended its reach into several forums in an attempt to coordinate a suite of community-driven and community-adopted software assurance methodologies, as well as designate leadership roles for advancing the objectives of the initiative. The overarching goal of the outreach initiative is to "partner with industry to create a competitive market that is building demonstrably vulnerability-free software." This goal requires coordination among industry, academia, and national and international partners to address shared elements of the problems related to assessing software assurance risk. With a shared understanding of the problems, the Tiger Team aims to focus science and technology on research and development of technologies that effectively improve upon existing development tools, strengthen standards for modularizing software, and enhance the ability to discover software vulnerabilities.

Beyond OUSD/AT&L, ASD/NII, and NDIA, major executive contributors to the Software Assurance Tiger Team currently include—

- **Aerospace Industries Association (AIA):**
  - *Role*: Best practices in aviation pathfinder sector; build upon concepts in ARINC 653
  - *Responsibilities*: Help integrate software assurance processes into mainstream integration activities
- **Government Electronics and Information Technology Association (GEIA):**
  - *Role*: Product manufacturing and development standards for core systems; *e.g.*, ISO 9126, ISO 12119
  - *Responsibilities*: Share lessons learned and collaborate with other stakeholders to develop new processes

- **Object Management Group (OMG):**
  - *Role*: Set software modeling and interface standards for software modularity and partitioning; *e.g.,* leverage ISO 150408 for EAL 6/7 target
  - *Responsibilities*: Leverage ongoing standards activities to advance Tiger Team goals.

6.1.1.1 Tiger Team Activities

The activity groupings within the Tiger Team were established to ensure fulfillment of the DoD *Software Assurance CONOPS*. These activities include—

- **Prioritization**: Processes for stakeholder prioritization (through deliberation) of the criticality of systems are being established. In the short term, prioritization will focus on new high-risk acquisitions, such as major DoD acquisitions, systems connected to classified networks, classified programs, and systems identified by DoD leadership. Prioritization is to occur early in the requirements/acquisition phase of the SDLC. The prioritization activity has identified a notional component criticality definition for four levels of assurance:
  - **High+**: Technology where compromise could result in catastrophically degraded mission capability or mission failure of the system (*e.g.,* cryptographic algorithms or cross-domain information solutions)
  - **High:** Technology where compromise could result in seriously degraded mission capability of the system (*e.g.,* system monitoring capability or need-to-know control within a security domain)
  - **Medium+**: Technology where compromise could result in partial or identifiable degradation of mission capability of the system (*e.g.,* IT components of a major business application)
  - **Medium:** Technology where compromise could result in inconvenience (*e.g.,* office automation tools).
- **Engineering in Depth (EiD)**: EiD applies systems engineering approaches to minimize the number and criticality of components that require greater assurance and to manage the residual risks inherent in the use of less assured products. EiD includes implementing risk-mitigating design techniques, such as graceful degradation, isolation, multipathing, and replaceable modules.
- **Supplier Assurance**: Supplier assurance uses all-source information to characterize suppliers according to the level of threat they present to the DoD. Issues of concern include foreign control of suppliers by Countries of Concern and outsourcing of technology/product development. The threat data collected is based on intelligence data and supplier-provided information. The supplier assurance activity has defined the supplier assurance requirements for the four levels of

component criticality defined by the prioritization activity (all four levels require a security-related engineering process and some level of source threat assessment performed):
    - **High+Assurance**: US-owned corporation or authorized US Government (USG) contractor with only cleared US citizens involved
    - **High Assurance**: US-owned corporation or authorized USG contractor with only US citizens involved
    - **Medium+Assurance**: US-owned corporation or authorized USG contractor with software design and engineering control functions primarily under control of US citizens or citizens of countries with historically close ties to the US whereas software development may be performed in a foreign country by foreign nationals
- **Science and Technology (S&T)**: S&T aims to achieve transformational solutions to software assurance problems and to provide state-of-the-art technical resources to the EiD process. S&T also works with industry to develop standards to satisfy EiD needs, and coordinates research and development (R&D) efforts for vulnerability prevention, detection, and mitigation tools and technologies.
- **Industry Outreach**: Industry outreach extends the DoD community to industry by engaging NDIA (systems assurance committee), OMG (software assurance committee), AIA, and GEIA. Additional activities of this bin include:
    - Developing a *Systems Assurance Handbook* (NDIA leadership)
    - Developing industry end-to-end reference models, products, standards and requirements, and product-level assurance properties.

While there are some key private sector participants in this bin, there is a noticeable lack of broader industry commitment or participation.

6.1.1.2 Products

As noted above, the NDIA Systems Assurance Committee, co-chaired by Kristen Baldwin (OUSD/AT&L), Mitchell Komaroff (ASD/NII), and Paul Croll (Computer Sciences Corporation), is producing *Systems Assurance: Delivering Mission Success in the Face of Developing Threats—A Guidebook*. [285] This *Guidebook* describes the differences between traditional secure systems engineering that addresses concerns about malicious developers, administrators, and users, and the need to reduce uncertainty and provide an explicit basis for justifiable stakeholder confidence in software, as well as decision making about software acquisitions and engineering approaches. Intended for multiple audiences, the Guidebook bases its consolidated approach on a variety of current secure system engineering practices, policies, and recommendations.

6.1.2 NSA Center for Assured Software

The National Security Agency (NSA) Center for Assured Software (CAS) was established in November 2005 as a focal point for software assurance issues in the NSA and DoD. The CAS collaborates closely with the DoD Software Assurance Tiger Teams and the DHS-sponsored Software Assurance Working Groups (WG), most notably the Tools and Technology WG. The CAS is also coordinating its tool evaluation efforts with the NIST Software Assurance Metrics and Tool Evaluation (SAMATE) program.

With its overarching mission of minimizing the number of exploitable software vulnerabilities in critical DoD information systems, the CAS is spearheading collaboration efforts across government, academia, and industry to research, coordinate, and prioritize software assurance activities throughout the software assurance community.

In addition to a broader collaboration role, the CAS is attempting to establish methodologies and leverage tools for comprehensively establishing and evaluating software trustworthiness throughout the software life cycle, including COTS and government-developed software utilized in critical DoD systems and networks. As a result of its evaluation activities, the CAS hopes to offer recommendations to the DoD, the Intelligence Community, other Federal agencies, and industry standards bodies (*e.g.,* ISO/IEC, OMG), with respect to changes in policy and standards that may permanently improve or enhance the level of assurance, and the ability to measure the level of assurance inherent to commercial or government-developed software.

In defining a consistently repeatable full life cycle process for evaluating software, the CAS is also identifying the testable software properties that will provide measurable levels of justifiable confidence (*i.e.,* assurance) that—

- The software will securely, appropriately, and predictably perform its intended functions.
- The software will not perform any unauthorized functions.
- The software does not contain exploitable implementation flaws, regardless of whether those flaws were intentionally or accidentally included.

6.1.2.1 Activities

The CAS is refining a software assurance evaluation methodology based on use of existing software code review and testing tools. CAS has also undertaken a series of tool evaluations to determine the suitability and effectiveness of the existing tools for supporting their evaluation methodology. The methodology consists of five evaluation phases, each supported by tools:

1. **Acceptance**: Determine whether tools and techniques exist for evaluating the software, and identify and fill capability gaps between evaluation capability needed and tools/techniques available.

2. **Extraction/Inspection:** Apply available tools and techniques that extract relevant data and metadata from the software (*e.g.,* complexity metrics, module dependencies, reverse-engineered source code). CAS will also work to foster integration of tools, and promote further research in this area.
3. **Analysis**: Apply available tools and techniques that query the extracted metadata for properties or indicators of assurance, such as presence of buffer overflow vulnerabilities or race conditions, improper memory management/object reuse, unexpected functionality, *etc.* In this area, CAS will also work to improve the quality of analysis tools, with particular focus on reducing the number of false positives in the tool results.
4. **Meta-Analysis**: Integrate outputs from analytical tools to rank the relevance of the identified indicators. To perform analytical tests that are beyond the capability of any one tool, and because some tools may increase the confidence in the results from another tool, CAS will work to "weave together" tools into a scalable meta-analysis methodology. This may involve using one tool to focus the analysis of a following tool or filter the results of a preceding tool, using independent indicators to help rank the results and integrate output from multiple analytical tools and techniques into a single meta-output that will enable the analyst to discern higher order assurance indicators.
5. **Reporting**: Transform analytical results into a variety of comprehensible reports that focus on different aspects of the analytical results. CAS will also work to define a set of customer-focused report formats.

The CAS strategy for finalizing and implementing its full life cycle software assurance evaluation methodology is to participate in public software assurance standards activities to influence the technological direction of the commercial and open source tools and techniques available to support the methodology. In addition, the CAS will define internal software assurance standards for NSA, and ensure compliance to those standards; among these is the forthcoming Guidance for Addressing Malicious Code Risk described in Section 6.1.2.1.1. Finally, through outreach to government, industry, and academia, CAS seeks to both influence and benefit from the software assurance practices, technologies, and research in those communities.

6.1.2.1.1 CAMP

The Code Assessment Methodology Project (CAMP) addresses the Government's need for greater software assurance and protection from unintentional and intentionally inserted malicious code by laying the foundation for the development of a software evaluation methodology to analyze untrusted code. The project will address the risk of malicious code (*i.e.,* code that introduces deliberate subversive behavior in software) and maliciously introduced software vulnerabilities.

6.1.3 DoD Anti-Tamper/Software Protection Initiative

Within the Air Force Rome Laboratories (AFRL) Sensors Directorate, the DoD Anti-Tamper and Software Protection Initiative (AT/SPI) [286] was established to thwart US adversaries attempts to reverse engineer and compromise software components and applications within national security systems. The AT/SPI is determining requirements for and guiding development of protection techniques for software.

A main focus of the AT/SPI is on technology that prevents the unauthorized distribution, tampering, or denial of service of critical national security software. The objectives of the SPI include—

- Insertion of protection measures into existing DoD software components, applications, and systems
- Measurement of the effectiveness of current protection measures
- Research into new software protection technology
- Education of the software development community on the SPI software protection philosophy
- Raising awareness of the threat to high-end software and the need for its protection
- Collaboration with the commercial sector on software protection methods
- Research into current software protection policies and development of new policies.

To date, AT/SPI efforts have yielded several important technological advances, including the development of a Secure Development Environment (SDE) to ensure total life cycle protection of software, and the development of tools to simulate attacks and accurately measure the level of protection afforded within a given threat environment.

The SPI has also established the Software Protection Center (SPC), a validated set of tools that support development of code in a secure environment and the application of software protections to that code prior to distribution and deployment. The toolbox contains a wide array of approved technologies to automate the process of software protection; these technologies can be implemented alone or in parallel.

For Further Reading

Jeff Hughes and Martin R. Stytz (AFRL/SN T-SPI Technology Office), *Advancing Software Security: the Software Protection Initiative.*
Available from: http://www.preemptive.com/documentation/SPI_software_Protection_Initative.pdf

Hardware-assisted Software Anti-Tamper.
Available from: http://www.dodsbir.net/sitis/archives_display_topic.asp?Bookmark=29477

Deobfuscating tools for the validation and verification of tamper-proofed software.
Available from: http://www.dodsbir.net/sitis/archives_display_topic.asp?Bookmark=28950

6.1.4 DSB Task Force on Mission Impact of Foreign In uence on DoD Software

The DSB Task Force on Mission Impact of Foreign Influence on DoD Software was established in October 2005. [287] Inspired by the DSB Microelectronic Task Force's publication of a report on the national security implications of migration of semiconductor design and manufacturing to foreign countries, the USD/AT&L, ASD/NII, and Commander of US Strategic Command (USSTRATCOM) announced their co-sponsorship of the task force's report on the impact and influence of foreign suppliers on the trustworthiness and assurability of the software components of DoD systems.

The task force report, which will be published in spring 2007, will characterize the causes for and level of DoD dependence on foreign-sourced software. It will assess the risks presented by that dependence. Whereas the DSB report on the implications of migration of semiconductor design and manufacturing focused on two threats (the risk that the Unites States could be denied access to the supply of chips and the risk that the chips could be maliciously modified), this report will focus primarily on the risk of malicious modifications in DoD software. The report acknowledges the threat that suppliers could deny access to the maintenance of legacy code, but the task force suggests that greater risk is posed by malicious code (though the attack vectors to be considered are not limited to those unique to foreign suppliers).

The report will also—

- Provide recommendations for managing the economics and risks involved in acquiring commercial-off-the-shelf (COTS) and foreign software, specifying the types of applications for which this is an acceptable risk as well those which necessitate cleared US citizens;
- Prioritize DoD software components according to their need for high levels of trustworthiness
- Identify requirements for employing EiD, which specifies assurance requirements for various components in a software-intensive system based on the criticality of a component
- Identify requirements for intelligence gathering to better characterize and monitor the threat from foreign suppliers
- Identify requirements for supplier assurance so that suppliers may be assessed for their trustworthiness
- Identify policies or technological research that can be undertaken to improve the ability to determine and sustain the trustworthiness and assurability of software, and when necessary to improve it
- Provide recommendations for improving the quality and assurance of COTS and open source software (OSS), such as improving the National Information Assurance Partnership (NIAP) evaluation process.

The task force comprises a chairman plus four members, all from private industry, as well as an executive secretary (currently Robert Lentz of ASD/NII). There are also four government advisors to the task force—two from DHS Cyber Security and Communications (CS&C) National Cyber Security Division (NCSD), one from NSA, and one from the Defense Information Systems Agency (DISA), as well as a representative of the DSB Secretariat in an oversight role.

In its first 6 months, the task force members were primarily engaged in data gathering, in part through reports of relevant initiatives in industry and government, including the Intel Trusted Computing Initiative, the NSA prototype of a trusted platform architecture for software (consistent with the DoD SPI), and the DHS Software Assurance Program.

6.1.5 GIG Lite

The most widely stated concern associated with use of software of unknown pedigree (SOUP) is the potential presence of malicious code (or malware). The difficulty stems from lack of effective automated malware detection technologies as well as the unwillingness of government project managers to increase the costs of software acquisitions to accommodate the detailed, extensive code reviews and security tests needed to manually locate malicious logic. Indeed, as briefed by Chris Gunderson, a researcher at the Naval Postgraduate School, at the December 2006 meeting of the DHS Software Assurance Program's Tools and Technology WG, DoD is seeking to model its security evaluation cycles after software industry testing cycles. Gunderson is researching the feasibility of doing this in a research initiative called GIG Lite. Funded by the Joint Interoperability Test Command, GIG Lite seeks to shorten the time involved with certifying and accrediting services and software applications for use in the Global Information Grid (GIG).

The objective of GIG Lite is to speed up the acquisition and certification and accreditation (C&A) schedules, enabling DoD to adopt new products far more quickly, in timeframes comparable to those achieved in the private sector. GIG Lite suggests establishing a small community of vendor-run and independent test labs to create a major test range for the rapid study, prototyping, demonstration, and evaluation of software components and services. Evaluations will focus on proving the value proposition that the target component/service brings and its trustworthiness (from both information assurance and software assurance perspectives). GIG Lite seeks to create an approved products list of components and services that have assigned trustworthiness ratings. The program also plans to develop the means by which potential users of evaluated components/services can map their own requirements against the attributes of products in the list so they can discover those products that come closest to satisfying those requirements.

Recognizing the potential for an increased level of risk that DoD will have to assume to accomplish the shorter testing and C&A cycle, GIG Lite

recommends employing a secure environment that uses IA (including computer network defense) and software assurance measures to detect, isolate, and minimize the impact of any security violations, including those that may result from execution of badly behaved software.

6.1.6 NRL CHACS

The Center for High Assurance Computing Systems (CHACS) [288] within the NRL Information Technology Division conducts research activities in the focus areas of "security, safety, availability and timely delivery of computational results." Major CHACS activities focus on formal methods to support the accuracy and preservation of critical system properties.

Of CHACS' six major research sections, three focus on aspects of software security assurance:

- **Formal Methods**: Researching formal methods for modeling, analysis, and verification of critical properties, *e.g.,* verifying compilers, formal verifications of pseudocode, formally based software tools
- **Software Engineering**: Researching software intrusion tolerance and survivability, *e.g.,* "Attack-potential-based survivability modeling for high-consequence systems" and "Merging Paradigms of Survivability and Security: Stochastic Faults and Designed Faults"
- **Computer Security**: Developing high-assurance building blocks, *e.g.,* "situation-aware middleware," such as survivable intelligent software agents.

6.1.7 DISA Application Security Project

The DISA Application Security Project was established in 2002 within the Applications Division of the DISA Center for Information Assurance Engineering (CIAE), to insert application security best practices into DISA's (and its contractors') web and database application development processes. The project had multiple objectives:

- Produce guidance for developers to help them produce more secure web and database applications
- Survey automated web application and database vulnerability assessment tools to identify "best of breed," and collect these into a toolkit, to be hosted on a "portable assessment unit"
- Use the vulnerability assessment toolkit to perform real-world vulnerability assessments on web and database applications developed by other DISA programs
- Provide application security subject matter expert consulting support as directed.

After its first year, the project's scope expanded to address the security of web services. In years 2 and 3 of the project, the developer guidance was

updated and expanded, and additional guidance documents were produced. As of October 31, 2004, the project had produced the following documents:

- *Application Security Requirements Engineering Methodology* (2004)
- *Reference Set of Application Security Requirements* (2002, 2003, 2004)
- A series of Application Security Developer's Guides (2002, 2003, 2004) comprising—
  - *Developer's Guide to Securing the Software Development Lifecycle*
  - *Developer's Guide to Secure Software Design and Implementation*
  - *Developer's Guide to the Secure Use of Software Components*
  - *Developer's Guide to Availability*
  - *Developer's Guide to Software Security Testing*
  - *Developer's Guide to Secure Web Applications*
  - *Developer's Guide to Secure Web Services*
  - *Developer's Guide to Secure Database Applications*
  - *Developer's Guide to Secure Use of C and C++*
  - *Developer's Guide to Secure Use of Java-Based Technologies*
  - *Developer's Guide to Public Key Enabling*
- *Java 2 Execution Environment (J2EE) Container Security Checklist* (2004)
- *Application Vulnerability Assessment Tool Market Survey* (2002, 2003, 2004)
- *Application Vulnerability Assessment Methodology* (2002, 2003, 2004).

All versions of these documents remain in draft status. Early versions of the *Reference Set* and the *Developer's Guide* were posted on the Information Assurance Support Environment (IASE) website, but the 2004 revisions were never posted to replace them. Of all the documents produced by the project, only the *Reference Set* Version 2.0 was still available on the IASE website as of January 2006.

In addition to producing these documents, the project acquired, integrated, and used the application vulnerability assessment toolkit to perform two security assessments of DoD web applications, the Defense Logistics Agency's Enterprise Mission Assurance Support System (eMASS), and the DISA Global Command Support System (GCSS).

A reorganization in DISA resulted in the transfer of the Application Security Project to the Field Security Operation (FSO). Since that time, the Application Security Project has ceased to exist as a separate effort. Instead, FSO extracted and adapted much of the content of the developer's guidance to produce four documents:

- *Application Services Security Technical Implementation Guide* (latest version: Version 1, Release 1, 17 January 2006; adapted from original *J2EE Container Security Checklist*)
- *Application Services Checklist* (latest version: Version 1, Release 1.1, 31 July 2006; adapted from original *J2EE Container Security Checklist*)
- *Application Security Checklist* (latest version: Version 2, Release 1.9, 24 November 2006; adapted from original *Reference Set of Application Security Requirements* and Application Security Developer's Guides)

- *Application Security and Development* STIG (latest version: Version 1, Release 1, 20 April 2007; adapted from original *Reference Set of Application Security Requirements* and Application Security Developer's Guides);

As of November 2006, FSO still implemented DISA's application-related STIGs and Checklists, which are posted on the DISA IASE website. In addition, a significant portion of the content of the Application Security Project's documents formed the basis for DHS's *Security in the Software Life Cycle,* whereas NIST used the categorization of vulnerability assessment tools in the DISA market surveys as one of the bases for its SAMATE tools classification and taxonomy.

6.1.8 Other DoD Initiatives

The initiatives discussed in the following section were established recently, or are in the process of being established.

6.1.8.1 DoDIIS Software Assurance Initiative

Managed under the auspices of the Applications Solutions Branch (ESI-3A) of the Defense Intelligence Agency (DIA), the newly established DoD Intelligence Information System (DoDIIS) Software Assurance Initiative intends to promote secure coding by all DoDIIS developers. The initiative will also finalize the Agile Development Environment (ADE) being defined for use by all DoDIIS developers. The ADE will provide a single standard set of tools and methods, including the open source GForge code repository. The initiative also intends to integrate secure development practices into all DoDIIS development projects' SDLCs. The initiative's guidance and products will be used by DIA and all other defense intelligence components (*e.g.,* National Geospatial Intelligence Agency, National Reconnaissance Office), and by the military services' intelligence branches (*e.g.,* Air Force Intelligence Agency).

6.1.8.2 US Army CECOM Software Vulnerability Assessments and Malicious Code Analyses

The Security Assessment and Risk Analysis Laboratory of the US Army Communications-Electronics Command's (CECOM) Software Engineering Center has established a software vulnerability assessment and malicious code analysis capability in support of the US Army Command and Control Protect Program. The CECOM assessment team has evaluated and compared three different methodologies for software vulnerability and malicious code analysis, and employs each selectively:

- Primarily manual analysis of source code using static analysis tools
- Semi-automated analysis using tools

- Tools and methods (many still in the research stage) for fully automated vulnerability and malicious code detection and analysis in both source code and executable binaries.

For Further Reading

Samuel Nitzberg, *et al.* (InnovaSafe, Inc.), *Trusting Software: Malicious Code Analyses* (February 18, 2004). Available from: http://www.innovasafe.com/doc/Nitzberg.doc

6.1.8.3 Air Force Application Security Pilot and Software Security Workshop

From January to November 2006, the 554th Electronic Systems Wing (554 ELSW) at Gunter Air Force Base ran an application security pilot in which they evaluated a range of vulnerability assessment, source code analysis, and application virtualization and defense tools for their potential utility in securing US Air Force application systems. The categories of tools and solutions evaluated during the pilot included:

- **Source Code Analysis**: Tools from two vendors were used to assess eight Java applications, revealing over 30,000 instances of common vulnerabilities and nonsecure coding practices. After the pilot ended, the 554 ELSW purchased tool licenses and used them in assessing a major application system. The same source code analysis tools, with customized rule sets, are expected to be used for code audits, in which they will scan the entire application code base prior to build. Code audits were not performed during the pilot.
- **Runtime Analysis**: The pilot included a demonstration of a runtime analysis tool tracing the propagation of vulnerabilities throughout the base, with reporting of runtime code coverage metrics.
- **Penetration Testing**: A tool with scripted "hacks" was demonstrated against an application executing in a controlled environment. The 554 ELSW is considering implementing penetration testing as part of their standard application regression testing.
- **Application Virtualization**: This pilot kicked off at the end of the Application Security Pilot period (*i.e.,* October 26, 2006), using a personal computer virtual machine (VM) product that enabled running of both Microsoft and Java applications, to secure desktop systems in instances in which legacy applications were incompatible with the security policy requirements of the Air Force's Standard Desktop Configuration. When the pilot ended, additional pilots were being considered for the 554 ELSW at Wright-Patterson Air Force Base and one or more sites in US Air Force-Europe.
- **Application Defense**: The pilot included implementation of a Honey Pot, as well as intrusion detection, monitoring, and prevention techniques and solutions.

- **Centralized Project Management**: All other tools and solutions used in the pilot were centrally managed through a web-based management "dashboard" that implemented standardized vulnerability reporting across all mission areas and program management offices.

On April 11–12, 2007, the Air Force Materiel Command 754th Electronic Systems Group at Gunter AFB co-sponsored an Armed Forces Communications Electronics Association Software Security Workshop, at Auburn University in Montgomery, Alabama. The papers presented focused on software and application security threat, vulnerability, risk, and remediation issues during the various phases of the SDLC used by the 554 ELSW-WP as an organizing principle for their Applications Security Pilot. Those phases include define, design, code, test, and monitor.

6.1.9 DHS Software Assurance Program

The DHS Software Assurance Program provides a framework and comprehensive strategy for addressing people, process, technology, and acquisition throughout the SDLC. The program seeks to transition from patch management as the standard approach for dealing with security weaknesses and vulnerabilities to the broad ability to routinely develop and deploy software products that start out trustworthy, high in quality, and secure, and remain so throughout their lifetimes.

Through the hosting and co-sponsoring of various meetings and forums, the DHS Software Assurance Program leverages public-private WG collaborations to generate a broad range of guidance documents and other products described in Sections 6.1.9.1 and 6.1.9.2.

6.1.9.1 Software Assurance Working Groups

The DHS Software Assurance Program currently sponsors seven working groups (WG), which meet every other month (and more frequently, if a particular concern needs to be addressed).

1. **Business Case (Marketing & Outreach)**: This WG focuses on advancing the awareness, understanding, and demand for assured software. This WG is establishing a consistent message that can be presented in different formats and at different levels of understanding to express the need for information assurance. In September 2006, this WG collaborated with the CIO Executive Council to develop the survey that focused on "Software Assurance" (*via* the terms "Reliability" and "Stability"), the findings of which are being assessed by the WG to form the basis for the emphases of its future activities.
2. **Technology, Tools, and Product Evaluation**: This WG focuses on the technology aspect of software assurance. This WG is looking at product evaluations and the tools necessary to accomplish this objective.

Currently, this WG tracks and influences the activities of the DHS-sponsored NIST SAMATE program, the MITRE Common Weakness Enumeration (CWE), and efforts of the NSA CAS.

3. **Acquisition**: This WG focuses on requirements for ensuring the acquisition of secure software. This WG is looking at enhancing software supply chain management through improved risk mitigation and contracting for secure software. The first product of this WG is a draft guidance document to assist acquisition managers in drafting software assurance-relevant language in procurement documents (*e.g.,* requests for proposal, statements of work) for software products and services, establishing security evaluation criteria for solicitation responses, and using those criteria in the assessment of those solicitation responses. The *Acquisition Management Guide for Software Assurance* is being jointly developed by contributors from academia, industry, and government, and will address concerns of all parties involved in acquisition. The *Guide* will include—
   - Templates for acquisition language and evaluation based on successful models
   - Common or sample statement of work or procurement language that includes provisions on liability
   - Due diligence questionnaires designed to support risk mitigation efforts by eliciting information about the software supply chain.
4. **Processes and Practices**: This WG focuses on improving software development processes to increase their likelihood of producing secure software. The WG is specifically identifying and, in some cases, producing guidance, standards, practice examples, configuration guidance, and conformance checks that help promote secure SDLC activities. The main activities of this WG have been the production of the *Security in the Software Life Cycle* document, and the tracking and influencing of ISO/IEC, Institute of Electrical and Electronics Engineers (IEEE), and OMG software assurance-relevant standards initiatives.
5. **Workforce, Education, and Training**: This WG produced the draft Common Body of Knowledge (CBK) as a basis from which academic instructors can develop secure software engineering curricula for universities, community colleges, and other academic institutions. Draft Version 1.1 of the CBK was released in August 2006, publicly reviewed, and discussed at a specially convened workshop for software engineering and information assurance academics in fall 2006. The draft CBK is expected to be published shortly after a final revision that will address public and workshop comments. The WG has also drafted a counterpart "essential body of knowledge" (EBK) that will be offered as the basis for development of workforce training programs and classes.
6. **Measurement**: This WG includes representatives from government, industry (including members and executives of the International

Systems Security Engineering Association), and academia. The WG is chartered to consider ways in which the degree of assurance provided by software can be assessed using quantitative and qualitative methods and techniques. The WG has just released for review within the DHS Software Assurance Program *Practical Guidance for Software Assurance and Information Security Measurement* (Draft Version 2.0). This document defines an approach for quantifying and assessing the degree to which software assurance techniques have been integrated into SDLC processes, and for evaluating the effectiveness of such integration in terms of the level of increased trustworthiness of the software produced by those processes. The document also seeks to facilitate compatibility of outputs from existing network-, system-, and software-level testing, assessment, and monitoring tools, and metrics from measurement approaches such as CMMi; Practical Software Measurement (PSM); NIST SP 800-55, *Security Metrics Guide for Information Technology Systems*; and the draft of ISO/IEC 27004, *Information Security Management Measurement*. In addition to its work on the *Guide*, the WG also contributed to the OMG request for proposal to develop a software metrics metamodel.

7. **Malware**: This, the most recently formed of the WGs, has to date focused on enumerating the attributes of malicious software ("malware"), so that the different types of malware can be characterized and the attribute-base characterizations can be combined with the emerging legal definitions for the different types of malware. Using the glossary published by the *Anti-Spyware Coalition's Working Report*, [289] the Malware WG's efforts are intended to complement those of the Common Malware Enumeration (CME) initiative described in Section 3.2.3.2.

The activities of the DHS Software Assurance WGs are briefed at the twice-yearly Software Assurance Forums co-sponsored by DHS and DoD.

6.1.9.2 Other Products and Activities

The DHS Software Assurance Program also sponsors a number of activities and artifacts not directly linked to specific WGs.

- **US-CERT BuildSecurityIn Portal**: Located on the World Wide Web at https://buildsecurityin.us-cert.gov/, the BuildSecurityIn portal is a compendium of theoretical background and practical guidance information developed and assembled by multiple contributors throughout the software development, software assurance, and software security communities, and directed towards an audience of software developers, architects, and security practitioners.
- **Software Assurance Landscape**: The Software Assurance Landscape has been envisioned to provide a single place for interested parties to find

descriptions of past, current, and planned activities, organizations, practices, and technologies that characterize the current "landscape" of the software assurance community. In addition to identifying a wide range of information resources, the Landscape will identify gaps in the current landscape that are hindering the universal adoption of software assurance ethos, processes, and practices, and will suggest solutions to close those gaps. An annotated outline of the Landscape was released for public discussion on October 16, 2006. The anticipated size, scope, and volatility of the Landscape's content have led DHS to consider publishing it in the form of an online knowledge base, rather than as a document. A second draft was briefed at the Software Assurance Forum in early March 2007, after which development of the Landscape's content was begun. To minimize duplication of effort and maximize consistency, the Landscape project team is coordinating its efforts with the authors of the DHS *Security in the Software Life Cycle*. It is also anticipated that the Landscape developers will use this SOAR as a key source.

6.1.10 NIST SAMATE

The DHS Software Assurance Program and NIST are jointly funding and overseeing the SAMATE project, [290] which they established to accomplish two primary objectives:

- Develop metrics to gauge the effectiveness of existing software assurance tools
- Assess current software assurance methodologies and tools to identify deficiencies that may introduce software vulnerabilities or contribute to software failures.

Specifically, the SAMATE project is intended to address concerns related to assessing "confidence" in software products—*i.e.,* quantifying through well-established metrics the level of assurance in software products with respect to security, correctness of operation, and robustness. This goal extended to assessing "confidence" in the effectiveness of existing software assurance tools—namely the accuracy of reporting features and the incidence of false positives and negatives. Due to the variability across tool suites, however, a standard testing methodology was needed to establish a structured and repeatable baseline for evaluating software effectiveness.

The project has established a publicly accessible, web-based Software Reference Dataset (SRD) of more than 1,700 examples for evaluating tools. These examples include contributions from academia, government, and security researchers, as well as examples written for specific tests. The SRD contains code with known weaknesses and, to assist in measuring false positive rates in testing tools, code without weaknesses. Most of the examples are short pieces of code written in C, with some examples in Java and C++, and some

larger code examples extracted from public software packages. The SRD also includes sample designs in Unified Modeling Language (UML), requirement specifications, and executable code.

In an attempt to thoroughly scope the problem of measuring software assurance confidence and tool effectiveness and achieve vendor community buy-in, the SAMATE project has defined an approach roadmap that relies on several community feedback loops, including workshops focused on helping SA tool developers, researchers, and users prioritize particular software assurance tool functions and define metrics for measuring the effectiveness of these functions.

Products and activities to be produced by SAMATE include—

- Taxonomy of classes of software assurance tool functions
- Workshops for software assurance tool developers and researchers and users to prioritize particular software assurance tool functions
- Specifications of software assurance tool functions
- Detailed testing methodologies
- Workshops to define and study metrics for the effectiveness of software assurance functions
- A set of reference applications with known vulnerabilities
- Papers in support of SAMATE metrics, including a methodology for defining functional specifications, test suites, and software assurance tool evaluation metrics.

Informally announced at the DHS Software Assurance WG meetings in late January 2007, a Software Assurance Labs Consortium is also being planned to fall under the umbrella of SAMATE activities. This consortium will represent both private and government test labs involved in the evaluation or assurance of software product security, with possible future objective of establishing standard software assurance "ratings" for software-intensive systems and commercial software products.

6.1.11 NASA RSSR

The National Aeronautics and Space Administration (NASA) Reducing Software Security Risk (RSSR) program [291] sponsored by the NASA Software IV&V Facility is working to define a formal analytical approach for integrating security into existing and emerging practices for developing high-quality software and systems. The RSSR seeks to address several problems typically associated with security in the SDLC. From November 2000 to December 2005, engineers from NASA Jet Propulsion Laboratory (JPL) (at the California Institute of Technology) and from University of California at Davis collaborated under the RSSR through an Integrated Approach project [292] to develop the Software Security Assessment Instrument (SSAI) (referred to in Section 5.1.3.2), which seeks to address the following problems:

- Absence of security in software and system engineering
- High cost of formally specifying security properties
- Cycle of "penetrate and patch"
- Predominantly piecemeal approach to security assurance.

To address these problems, the SSAI includes the following tools, procedures, and instruments:

- Software security checklist
- Vulnerability matrix
- Modeling instrument
- Property-based testing tool
- Training.

These tools and instruments can be used individually or in tandem to support the following functionality:

- **Model-Checking**: The SSAI's model checking involves building a state-based model of the system, identifying properties to be verified, and checking the model for violations of specified properties. Its flexible modeling framework features adaptability to early life cycle events. The SSAI includes a model-based verification and a flexible modeling framework that provides newly discovered vulnerability scenarios to *VMatrix*, a matrix of known vulnerability probabilities. Model-based verification also provides life cycle verification results to the property-based tester (PBT).
- **Property-Based Testing**: Using an instrumenter, test execution monitor, and program verifier, PBT employs a code-slicing technique to iteratively test software for violations of security properties. PBT examines data from program executions to assess as many control paths within the Java, C, or C++ source code as possible. The verifier is used subsequently to ensure that security property violations have not been reintroduced into source code during later coding.

With the cooperation of PatchLink Corporation, the NASA Independent Verification and Validation (IV&V) Center successfully used the SSAI to verify the security properties of PatchLink's Java-based UNIX agent. PatchLink is using the findings of the NASA assessment to improve the security of the product.

6.1.11.1 Recent Research Results

Since the release of the SSAI in 2005, RSSR has focused on—

- Achieving a higher level of automation in the SSAI.
- Release of an updated version of the PBT, and definition of future enhancements to the tool.

- Integration of a mechanism to automatically transform natural language security requirements into formal specifications of those requirements that can then be model checked using the Spin model checker. The Spin model checker results would then be verified by the PBT to ensure they are not violated in the actual code. (Completion of this effort depends on release by NASA of further funding.).
- Improved specification and validation of formal models, simulations, and measurements.
- Improved techniques for defect detection and prediction.
- Normalization of various modeling artifacts of the to enable improved analysis.
- Enhanced scope and capabilities of the software IV&V tools used by NASA.

Though not all of these projects specifically focus on software security, NASA anticipates that the resulting improvements in software quality and dependability will also benefit security.

6.2 Private Sector Initiatives

The following initiatives have been undertaken in the private sector, typically as consortia with membership primarily from the commercial sector (*e.g.,* software tool vendors, major software suppliers), but also with significant participation by organizations and individuals in the academic and government sectors.

6.2.1 OWASP

Open Web Application Security Project (OWASP) [293] defines itself as "an open source community" of software and application security practitioners dedicated to helping organizations in the private and public sectors develop, purchase, and maintain trustworthy application software. OWASP produces tools and documents, and sponsors forums and chapters. Its products are distributed under approved open source license to any interested party.

OWASP promotes the notion of web application security as a composite of people, processes, and technology. The extensive information and software it produces are developed collaboratively by OWASP members and outside participants. OWASP warrants that the information in its publications is independent of any individual commercial or proprietary interest.

OWASP projects are organized as collections of related tasks with a single defined roadmap and team leader; the team leader is responsible for defining the vision, roadmap, and tasking for the project. OWASP projects have produced artifacts ranging from guidance documents, to tools, teaching environments, checklists, and other materials.

6.2.1.1 Tools

To date, OWASP has released two significant tools:

- **WebGoat**: This deliberately insecure, interactive web application is designed to be used in a tutorial context. As it runs, WebGoat prompts users to demonstrate their knowledge of security by exploiting vulnerabilities in the WebGoat application.
- **WebScarab**: This vulnerability assessment framework is used to analyze web applications and web services. Written in Java and thus portable to many platforms, WebScarab's various modes of operation are implemented by a number of plugins.

6.2.1.2 Documents and Knowledge Bases

OWASP has also published extensive tutorial and guidance material, including—

- **AppSec FAQ**: This FAQ answers common developer questions about web application security. Its content is not specific to a particular platform or language; instead, it addresses the most common threats to all web applications regardless of language or platform.
- ***Guide to Building Secure Web Applications***: This document is a comprehensive manual on designing, developing, and deploying secure web applications. Now in its second version, the OWASP *Guide* has served as a key source of guidance for many architects, developers, consultants, and auditors. According to OWASP, the *Guide* has been downloaded more than 2 million times since its publication in 2002 and is referenced by several leading government, financial, and corporate security and coding standards.
- **Legal knowledge base:** This project has established a knowledge base of materials on the legal aspects of secure software, including contracting, liability, and compliance.
- **Top Ten Web Application Security Vulnerabilities**: This is the first of OWASP's major projects, and the one that brought the organization into international prominence. The Top Ten identifies and describes a broad consensus of opinion on the most critical security vulnerabilities and weaknesses in web applications. The 2002 Top Ten has been widely cited as a minimum standard for web application security, providing the basis upon which a number of other application and software security vulnerability taxonomies have been defined. All of the major vendors of application vulnerability scanners advertise their products' ability to detect the vulnerabilities listed in the OWASP Top Ten. OWASP published a new version of the Top Ten in May 2007.

6.2.2 OMG SwA SIG

The OMG SwA Special Interest Group (SIG) [294] works with the OMG Platform and Domain Task Forces and several external software industry organizations to coordinate and establish a common framework for analysis and exchange of information to promote software trustworthiness. The larger framework can be broken down into the following components:

- A framework of software properties that can be used to present any/all classes of software so software suppliers and acquirers can represent their claims and arguments
- Verification of products, ensuring satisfactory characteristics for system integrators who will use those products to build larger assured systems
- Enablement of industry, improving visibility into the current status of software assurance, and developing automated tools that support the common framework.

The SwA SIG is able to leverage related OMG specifications, such as Knowledge-Driven Modernization and Software Process Engineering Metamodel, and particularly the various quality and maturity models for security with OMG specifications. The SwA SIG is working to identify scenarios in which the OMG Software Assurance Framework can be applied

6.2.2.1 Products

A noteworthy SwA SIG project is the development of the Software Assurance Ecosystem, which is emerging as a framework and infrastructure for the exchange of information related to software assurance claims, arguments, and evidence. Initially, the Software Assurance Ecosystem infrastructure integrates tools and outputs from three different realms of software engineering: formal methods, reverse engineering, and static analysis. The purpose of the Ecosystem is manifold:

- To provide an effective vehicle for communications of assurance information between developers and stakeholders on the one hand, and certifiers and evaluators on the other
- To provide a repository for gathering assurance claims and arguments
- To improve the objectivity and accuracy of evidence collection
- To enable the rapid evaluation of evidence and building of evidence correlation models
- To automate validation of claims against evidence, based on arguments
- To enable more accurate and highly automated risk assessments.

6.2.3 WASC

Web Application Security (WASC) [295] was founded in January 2004 by a group of web application security tools vendors (Application Security, KaVaDo, Sanctum, SPI Dynamics, and WhiteHat Security) with the stated objective of establishing, refining, and promoting Internet security standards.

The consortium's members, which include not only corporations but individual experts and industry practitioners as well as noncommercial organizational representatives, collaborate on research, discussion, and publication of information about web application security issues and countermeasures to specific threats. Their intended audience includes individuals as well as enterprises.

WASC also acts as an advocate for Internet users in general, and for web application security practitioners in particular. Though WASC welcomes corporate members, it claims to be vendor neutral. WASC also differentiate itself from OWASP, which appears to share many of the same objectives. According to WASC, its main objectives—to provide a public resource for industry guidance, freely exchangeable literature (no open source license is required), and documented standards, and to promote web application security standards and best practices—differ from OWASP's, which WASC considers more "goal-oriented" in its multiplicity of open-source web security software development projects and documentation initiatives. (Interestingly, in November 2005, WASC published its *Web Security Threat Classification*, apparently an answer to the OWASP Top Ten.)

Some industry insiders, however, have suggested that WASC was started by former OWASP members who were unhappy with the level of influence a few corporate members had in that organization. WASC detractors in OWASP, on the other hand, have criticized WASC for being little more than a marketing platform for its corporate founders. They also deplore WASC's overall tone as promoting fear, uncertainty, and doubt (FUD).

Ultimately, as one blogger put it, "How is WASC going to play with OWASP? Time will tell, but in my opinion the more Web application security awareness the better." Indeed, several individuals and organizations see value in both consortia, and belong to both.

6.2.4 AppSIC

The Application Security Industry Consortium (AppSIC) [296] is a nonprofit organization founded in 2005 as a "community of security and industry thought leaders" consisting of experts, technologists, industry analysts, and consumers in the application security sector. The organization's goal is to establish and define international cross-industry security metrics and guidelines to help organizations measure security return on investment (ROI) and apply metrics to buying security products.

AppSIC aims to serve as a bridge between the academic, industrial, vendor, and business user communities on application security. It seeks to produce business and technically relevant results. AppSIC distinguishes itself from other consortiums by synthesizing the views of a diverse range of companies and experts. Founding members include executive-level representatives of Security Innovation, Microsoft, Red Hat, Oracle, IDC, Gartner,

Internationale Nederlanden Groep (ING), Systems Applications and Products (SAP), Compuware, Secure Software, the Florida Institute of Technology, and Yoran Associates. Other significant organizations have joined, including Ounce Labs and Credit Suisse. AppSIC membership is open to all interested parties; the consortium charges no membership fee.

The broader goals of AppSIC include—

- Developing metrics for effectiveness of secure software development processes
- Generating application security assessment criteria
- Developing guidelines to help software development organizations address application security issues in their life cycle processes
- Developing business metrics for measuring ROI for application security spending.

AppSIC's first deliverable, in July 2006, was a position paper entitled *What Security Means to My Business* that attempts to capture a business case for software security and to lay the foundation for metrics for measuring business risk that stems from insecure software, with a view towards substantively mitigating that risk.

6.2.5 SSF

Launched in February 2005, the Secure Software Forum (SSF) [297] is a collaborative initiative between major commercial software vendors to provide a starting place for cross-industry discussions and education on how best to implement Application Security Assurance Programs (ASAP). The forum at inception was co-sponsored by Microsoft Corp., Fortify Software, Information Systems Security Association, and Mercury Interactive Corp. Its sponsorship has since expanded to include SPI Dynamics, Visa, and Wintellect.

The forum is designed to facilitate the sharing of industry best practices and key issues that must be solved to ensure more secure software development. Their key efforts to date have been—

- Sponsorship of SSF events for executive-level attendees involved in security operations, software development, and quality assurance
- A 2005 survey on the state of software security awareness and practices among their events' attendees (Results of this survey are discussed in Section 7.2.3.2)
- Publication of a white paper reporting the activities and successes of SSF's members to date
- Drafting by SPI Dynamics of a guidance document for use by software firms that are seeking to implement ASAPs.

6.2.5.1 "State of the Industry" White Paper

In 2006, SSF contracted Reavis Consulting Group to draft *Developing Secure Software: The State of the Industry as Determined by the Secure Software Forum*. This white paper describes the findings of SSF-sponsored industry collaboration events (roundtables, workshop, and webcast events) conducted throughout 2005 with information security and application development executives from Global 2000 organizations.

The paper surveys industry problems in software development and suggests aspects of available solutions. Specific topics addressed include—

- Threats caused by insecure software
- Common software development methodologies
- Progress to date of Microsoft's Trustworthy Computing Initiative, including the firm's use of its SDL
- Efforts to improve software security through adoption of SPI Dynamics' concept of the ASAP, with a proposed Maturity Model to help drive ASAP adoption.

6.2.5.2 ASAPs

The SSF is promoting the creation of ASAPs by organizations that produce application software. According to the SSF, these ASAPs should embrace the following set of broad principles to improve the security of software:

- There must be executive level commitment to secure software.
- Security must be a consideration from the very beginning of the software development life cycle.
- Secure software development must encompass people, processes, and technology.
- Metrics must be adopted to measure security improvements and enforce accountability.
- Education is a key enabler of security improvements.

Beyond stating these principles, the ASAP guidance provided in the *Developing Secure Software* white paper does not suggest a specific process improvement model or SDLC methodology, nor does it even identify the required features such a model or methodology would need to help accomplish ASAP objectives.

The SSF is now discussing a proposal to further refine the ASAP Maturity Model proposed by SPI Dynamics to help drive adoption of ASAPs. This model is described in *Developing Secure Software*.

6.2.6 Task Force on Security Across the Software Development Life Cycle

In 2003, DHS co-sponsored the first National Cyber Security Summit, an assembly of public and private sector leaders convened to discuss how the DHS CS&C NCSD should move forward in implementing the President's National

Strategy to Secure Cyberspace, released in February 2003. The Summit was co-hosted by four leading industry associations: the US Chamber of Commerce, the Business Software Alliance, the Information Technology Association of America, and TechNet. Collectively, these industry organizations formed themselves into the National Cyber Security Partnership (NCSP).

Coincidentally with the summit, at the behest of the DHS, the NCSP established a Task Force on Security Across the Software Development Life Cycle composed of four subgroups:

1. **Education**: Focused on present and future developers, this subgroup recommended that—
   - Security become a core component of software development programs at the university level with sufficient resources to build the academic capacity to improve secure software development
   - Industry, government, and education certificate programs be established for IT professionals
   - Security be improved by enhancing academic curricula.

   The Education subgroup produced a set of recommendations.
2. **Process**: Looked into developing and sharing best practices to improve the quality of software as well as the production processes so systems are more resilient to attack. The subgroup produced a report entitled *Processes to Produce Secure Software,* [298] which documented software development practices, tools, and strategies that software producers could use to produce (more) secure software.
3. **Incentives**: Focused on identifying incentives that—
   - Motivate development of more secure software during every phase of software development
   - Promote effective interaction between security researchers and software vendors
   - Demotivate cyber criminals from demonstrating malicious behavior.

   The Incentives subgroup produced an Incentives Framework outlining recommendations to aid policymakers, developers, companies, and others in developing effective strategies and incentives for producing, acquiring, and using software in ways that increases its security.
4. **Patching**: Focused on defining steps that can be taken to enhance the patching process to reduce complexity, increase its effectiveness, improve reliability, and ultimately, minimize costs and risk. The subgroup identified and categorized specific recommendations for technology providers, critical infrastructure providers, and independent software vendors.

In 2004, the Security Across the Software Development Life Cycle Task Force published a report entitled *Improving Security across the Software Development Life Cycle.* [299] This report summarized the activities of the summit, and the recommendations of the task force's four subgroups.

The DHS Software Assurance Program's WGs can be seen as successors to the task force's four subgroups.

6.2.7 New Private Sector Initiatives

As with the new DoD initiatives documented in Section 6.1.8, the following new private sector initiatives have only recently been announced or established, and thus there is little to report as yet on their activities.

6.2.7.1 Software Assurance Consortium

Announced by Concurrent Technologies Corporation at the DHS Software Assurance WG plenary on January 24, 2007, and chartered on March 8, 2007, the objective of the Software Assurance Consortium will be to engage a significant sector that has not been involved in any of the other software assurance activities or consortia now in existence, *i.e.,* the software consumer. The "consumer" in this context is represented by CIOs in the public and private sectors.

The governing constraint on the charter of the Software Assurance Consortium is that all activities will be consumer driven and consumer focused. The consortium's steering committee and officers will all come from consumer organizations, and legally enforceable criteria will be defined to govern the participation of consumer organizations that are also involved in the production of software (*e.g.,* systems integrators, organizations that develop software for their own use).

The Software Assurance Consortium will not duplicate the efforts of other existing consortia, such as the Secure Software Forum or OWASP. Instead, it will provide a framework in which existing and emerging consortia representing other communities (*e.g.,* software producers, tools vendors, test labs, standards groups) can come together to pursue discussions and activities of joint interest as long as those discussions/activities have the explicit objective of benefiting the software consumer.

Some of the activities planned or under consideration for the consortium include—

- Gather and coordinate consumer software assurance needs, requirements, concerns, and priorities
- Define requirements for risk assessment and testing of software; do so using language that includes standard representations of software vulnerabilities (*e.g.,* Common Vulnerabilities and Exposure (CVE), CWE)
- Identify and provide information about end-user tools that can solve specific user/consumer software security problems (*e.g.,* anti-malware, anti-spyware)

- Establish a scheme for rating software products' security, quality, assurance
- Identify software best practices, guidance, *etc.*, of benefit from a consumer perspective
- Fund research that will benefit consumers and fill perceived R&D gaps.

The consortium will have several desired outcomes:

- Consumers will become more explicit, specific, comprehensive, and consistent in expressing requirements for software that is dependable and secure.
- Consumers will have a better basis (in terms of knowledge, motivation, and tools) for acquiring software that is secure, and for securely using/managing their current and future software.
- Output from the consortium will inform the choices, activities, and strategic and tactical directions of other software communities (vendors, integrators, test labs, acquisition organizations, policy authors, standards bodies, academics).

In March 2007, it was announced that Dan Wolf, formerly the Director of NSA Information Assurance Directorate and initiator of the NSA CAS, had agreed to take on the role of Executive Director of the Software Assurance Consortium.

6.3 Standards Activities

A number of mainly process-oriented standards activities are focused on achieving security assurance in the software development life cycle. OMG standards are addressed in Section 6.2.2.

6.3.1 IEEE Revision of ISO/IEC 15026: 2006

This standard is discussed in Section 5.1.4.2.2.

6.3.2 IEEE Standard. 1074-2006

IEEE Standard 1074-2006 is a revision of the IEEE Std. 1074-1997, *Developing Software Project Life Cycle Processes*, intended to add support for prioritization and integration of appropriate levels of security controls into software and systems. The new standard adds a small number of security-focused activities to the SDLC process defined in the 1997 version of the standard.

IEEE Std. 1074-1997 formed the basis for developing ISO/IEC 12207.1, *Standard for Information Technology–Software Life Cycle Processes;* and 12207.2, *Software Life Cycle Processes–Life Cycle Data*. The main objective of both the IEEE and ISO/IEC standards is to define a quality-driven SDLC process. Neither standard contains specific security guidance, although ISO/IEC 12207 does suggest the need for security activities and references the very few security standards in existence when 1074-1997 and 12207 were first adopted that pertained to the software or system life cycle.

Unlike ISO/IEC 12207, IEEE 1074-2006 includes documentation of security risks and solutions throughout the SDLC; to do so, it leverages Common Criteria assurance principles and assets, defines a security profile for evaluation of software integrity as well as documentation needed to ensure secure operations, and generally covers security areas not addressed in ISO/IEC 12207 (*e.g.,* the security risks inherent in system and software change control).

As the source for new security activities or artifacts to be added to IEEE 1074-1997 Project Activities, the IEEE team that undertook revision of the standard started by analyzing ISO/IEC 17799:2000, *Code of Practice for Information Security Management* and ISO/IEC 15408 *Common Criteria for Information Technology Security Evaluation*. The resulting new 1074 Project Life Cycle Process Framework elevates the visibility and priority of security to that of other compelling business needs.

The IEEE team also ensured that IEEE Std. 1074-2006 aligns with several quality assurance and improvement standards:

- QuEST Forum's TL 9000, the telecommunication industry's extension to ISO 9000 (IEEE 1074-2006 requires the definition of a user's software life cycle consistent with this standard)
- ISO 9001, *Quality Management Systems—Requirements*, Section 7.1, "Planning of Product Realization"
- ISO/IEC 9003 (superseded by ISO 9001:2000)
- CMMI Organizational Process Definition, which requires the establishment of standard processes, life cycle model descriptions, tailoring criteria and guidelines, and establishment of a measurement repository and process asset library)
- ISO/IEC 15288, *Systems Engineering Life Cycle*
- ISO/IEC 12207, *Software Life Cycle*.

Unlike these earlier quality-driven standards, however, the new IEEE Std. 1074-2006 includes guidance for prioritizing security and supporting security measurement for both software projects and software products. The revised standard provides a systematic approach to defining specific security requirements and producing quality security artifacts for each discreet life cycle activity, as well as ongoing audit, improvement, and maintenance of product and process security. The standard supports acceptance testing and validation of security, and requires that products attain security accreditation by an independent security/integrity auditor. The guidance in IEEE Std. 1074-2006 was intentionally structured to be easily adapted for tools-based conformance measurement. The standard also defines enhanced security training activities.

6.3.3 ISO/IEC Project 22.24772

The Other Working Group on Vulnerabilities within the ISO/IEC Joint Technical Committee on Information Technology (JTC1) SubCommittee on Programming Languages (SC22) has been assigned responsibility for

project 22.24772. [300] The mandate of this Project, which is currently being organized, is to produce a technical report (TR) [301] entitled *Guidance to Avoiding Vulnerabilities in Programming Languages Through Language Selection and Use*. This TR, scheduled for publication in January 2009, will provide guidance for programmers on how to avoid the vulnerabilities that exist in the programming languages selected for use on their software projects. While it is not the explicit purpose of the TR, the guidance is expected to help programmers select the most appropriate languages for their projects and choose tools that may help them evaluate and avoid known language vulnerabilities. The vulnerabilities to be identified in the TR will be derived from various non-ISO efforts underway to identify and categorize vulnerabilities and other forms of weaknesses, including the MITRE CWE.

For routine vulnerabilities, the TR will suggest alternative coding patterns that are equally effective but which avoid the vulnerability or otherwise improve the predictability of the compiled program's execution. When such measures are not possible, the TR may suggest static analysis techniques for detecting vulnerabilities and guidance for coding in a manner that will improve the effectiveness of this analysis. When static analysis is not feasible, the TR may suggest the use of other testing or verification techniques. Whenever possible, the report will help users understand the costs and benefits risk avoidance, and the nature of residual risks.

In addition to publishing the TR explaining the different kinds of vulnerabilities and how they can be avoided in different programming languages, Project 22.24772 is considering liaison with the ISO/IEC standards committees responsible for individual programming language standards to determine what issues might be examined in those languages. The project is also examining several existing coding guidelines as potential sources for its TR, including (but not limited to) the CMU CERT's Secure Coding Standards for C and C++; the US Nuclear Regulatory Commission's *Review Guidelines for Software Languages for Use in Nuclear Power Plant Safety Systems: Final Report* (Nuclear Regulation NUREG/CR-6463, Rev. 1, 1997); and ISO/IEC TR 15942:2000, *Guide for the Use of the Ada Programming Language in High Integrity Systems.*

6.3.4 ISO/IEC TR 24731

ISO/IEC JTC1/SC22 WG 14 is focused on defining safety and security standards for programming languages. One of the first products of its efforts is TR 24731, *Information Technology–Programming Languages, Their Environments and System Software Interfaces–Extensions to the C Library–Part I: Bounds-checking Interfaces.* [302] TR 24731 defines a set of standard extensions to the C programming language (standardized in ISO/IEC 9899:1999) that will add memory bounds checking capability, thus reducing the risk of buffer overflows in programs written in C.

6.3.5 IEEE Standard P2600 Section 9.9.2

Section 9.2.2 of the draft IEEE Std. P2600, *Hardcopy System and Device Security,* is [303] entitled "Methodologies and Processes for the Development of Secure HCDs (hardcopy devices)." Section 9.2.2 provides informative (rather than normative) guidance on principles, risk management considerations, and life cycle processes, methodologies, and practices that, if undertaken by developers and managers, are intended to produce secure software for use in hardcopy devices and systems.

It is interesting to note that the P2600 WG, peopled solely by representatives from hardcopy device/system vendors, included in the P2600 standard guidance that reflects current secure software engineering best practices gleaned from several of the most frequently cited software security books and resources. The inclusion of software security guidance in P2600 is an encouraging indicator that software security awareness efforts and publications are having a positive influence on the broader community of software practitioners.

6.4 Legislation Relevant to Software Security Assurance

Developers' liability for insecure products is not addressed in current legislation either at the Federal or state levels. However, prohibitions on certain activities that affect the security of software (*e.g.,* tampering, denial of service, malicious code) are included in a number of laws pertaining to computer security or Internet security. The relevant language in these laws is identified in Table 6-1.

Table 6-1. Legislation with Software Security Relevance

Prohibition	Federal Code	State Code
Against tampering with and denial of service to software programs	▶ 18 United States Code (U.S.C.) Part I, Chapter 47, Section (§) 1030(5)(A)(i) ▶ Note that language in the earlier Subsection (g) expressly absolves vendors of all responsibility for producing vulnerable software, to whit: "No action may be brought under this subsection for the negligent design or manufacture of computer hardware, computer software, or firmware."	- Alaska Statute § 11.46.740 - Arizona Revised Statute § 13-2316 - Arkansas Code § 5-41-202 - California Penal Code § 502 - Colorado Revised Statute § 18-5.5-102 - Illinois Criminal Code Chapter 720 -Illinois Compiled Statutes (ILCS), 5/16D-3 - Michigan Compiled Laws § 752.795 - Minnesota Statute § 609.88 - Nebraska Revised Statute § 28-1345 - Nevada Revised Statute § 205.4765 - New Jersey Statute Annotated § 2C:20-25 - New Mexico Statute § 30-45-4 - North Carolina General Statute § 14-455 - Ohio Revised Code Annotated § 2909.07 - 18 Pennsylvania Consolidated Statute Annotated § 7611, 7612, 7615 - South Carolina Code Annotated § 16-16-20 - Tennessee Code § 39-14-602 - Texas Penal Code § 33.02 - West Virginia Code § 61-3C-7

Table 6-1. Legislation with Software Security Relevance - *continued*

Prohibition	Federal Code	State Code
Against intellectual property violations that involve tampering, denial of service, or unauthorized copying, disclosure, or distribution of software	17 U.S.C. § 1201	- Florida Statute § 815.04 - Mississippi Statute § 97-45-9 - 18 Pennsylvania Consolidated Statute Annotated § 7614
Against distribution and/or installation of malware (including spyware)	None known	- Arkansas Code § 5-41-202 - Colorado Revised Statute § 18-5.5-102 - Georgia Code Annotated § 16-9-153 - Florida Statute § 815.04 - Maine Revised Statute Title 17-A, § 433 - Michigan Compiled Laws § 752.795 - Minnesota Statute § 609.88 - Nebraska Revised Statute § 28-1345 - Nevada Revised Statute § 205.4765 - New Hampshire Revised Statute § 638:17 - North Carolina General Statute § 14-455 - North Dakota Cent. Code § 12.1-06.1-08 - Ohio Revised Code Annotated § 2909.07 - 18 Pennsylvania Consolidated Statute Annotated § 7616 - South Carolina Code Annotated § 16-16-20 - Tennessee Code § 39-14-602 - West Virginia Code § 61-3C-7

Not all states have established legislation, and it is very common for legislation that restricts technology use to lag behind the technology's development. In this case, the technology is a computer or the Internet. Violations of legislation and policies are considered as high as Class III felonies, depending on the severity of the violation.

Software purchasers, including government purchasers, are increasingly holding the software industry accountable for its software, especially when software's vulnerability leads to or contributes to security breaches. In his doctoral dissertation, Jari Råman argues: "Without appropriate regulatory intervention, the level of security of software will not improve to meet the needs of the networked society as a whole." [304] He further asserts that the incentives necessary to entice software companies are not being provided, and that required activities such as vulnerability disclosure should be implemented consistently.

In her article, *Who is Liable for Insecure Systems,* [305] Nancy Mead of the CMU SEI provides an overview of key published opinions in the United States about legal liability for software security problems. Mead notes that

the issue is being considered by two different communities—those also considering liability for poor software quality, and those involved with legal liability for computer security, cyber security, and Internet security breaches. Mead suggests that though a standard for a reasonable level of due diligence by software developers has yet to be established, it is likely that civil liability suits will start to appear on court dockets, with software development firms as the defendants. Mead's own recommendations focus on preemptively motivating and empowering software developers to produce higher quality, more secure software before the threat of liability lawsuits becomes commonplace.

In Europe, the TrustSoft Institute at the University of Oldenburg (in Germany) routinely educates developers about legal liabilities for developing faulty software. . This curriculum includes information on warranty, liability, scope, and personal consequences for the individual programmer. Though the law on which the TrustSoft Institute curriculum is based is specific to Germany, it provides a model for other universities and colleges to follow.

For Further Reading

Nancy Mead (CMU SEI), "Who is Liable for Insecure Systems?" *IEEE Computer.* (July 24): 27-34.

Bruce Schneier, "Sue Companies, Not Coders." Wired. (October 20, 2005),.
Available from: http://www.schneier.com/essay-092.html

References

285 "Systems Assurance Committee" [web page] (Arlington, VA: NDIA).
Available from: http://www.ndia.org/Template.cfm?Section=Systems_Engineering&Template=/ContentManagement/ContentDisplay.cfm&ContentID=17472&FusePreview=True

286 "AFRL/SNT (Anti-Tamper Program & Software Protection Initiative) fact sheet" [web page] (Dayton, OH: Wright-Patterson Air Force Base).
Available from: http://www.wpafb.af.mil/library/factsheets/factsheet.asp?id=6308

287 Kenneth J. Krieg (OUSD/AT&L), memorandum to Chairman, Defense Science Board, Subject: "Terms of Reference—Defense Science Board Task Force on Mission Impact of Foreign In uence on DoD Software," Washington, DC, October 5, 2005.
Available from: http://www.acq.osd.mil/dsb/tors/TOR-2005-10-05-MIFIDS.pdf

288 "NRL CHACS" [portal page] (Washington, DC: NRL CHACS).
Available from: http://chacs.nrl.navy.mil/

289 *Anti-Spyware Coalition. Definitions and Supporting Documents*, working report (June 29, 2006).
Available from: http://www.antispywarecoalition.org/documents/index.htm (Scroll down to "Definition and Supporting Documents".)

290 "SAMATE" [portal page] *op cit.*

291 "Reducing Software Security Risk Through an Integrated Approach Research Program Results" [web page] (Fairmont, WV: NASA IV&V Facility).
Available from: http://sarpresults.ivv.nasa.gov/ViewResearch/60.jsp

292 "Reducing Software Security Risk Through an Integrated Approach Project" [web page] (Davis, CA: University of California at Davis Computer Security Laboratory).
Available from: http://seclab.cs.ucdavis.edu/projects/testing/toc.html

293 "OWASP" [portal page] (Columbia, MD: OWASP).
Available from: http://www.owasp.org/

294 "OMG Software Assurance Special Interest Group (SwA SIG)"[web page] (Needham, MA: Object Management Group).
Available from: http://swa.omg.org/

295 "Web Application Security Consortium" [portal page] (Web Application Security Consortium [WASC]).
Available from: http://www.webappsec.org/

296 "Application Security Industry Consortium" [portal page] (Wilmington, MA: Application Security Industry Consortium [AppSIC]).
Available from: http://www.appsic.org/

297 "Secure Software Forum" [portal page] (Atlanta, GA: SPI Dynamics Inc.).
Available from: http://www.securesoftwareforum.com

298 Samuel T. Redwine, Jr., and Noopur Davis, eds., "Processes to Produce Secure Software: Towards More Secure Software," in *Report of the Software Process Subgroup of the Task Force on Security across the Software Development Lifecycle to the National Security Summit*, March 2004: 1.
Available from: http://www.cigital.com/papers/download/secure_software_process.pdf

299 DHS Security Across the Software Development Lifecycle Task Force, *Improving Security Across the Software Development Lifecycle, Task Force Report* (Washington, DC: DHS CS&C NCSD, April 1, 2004).
Available from: http://www.itaa.org/software/docs/SDLCPaper.pdf

300 "ISO/IEC Project 22.24772" [web page] (New York, NY: ISO/IEC JTC 1/SC 22 Secretariat).
Available from: http://aitc.aitcnet.org/isai/ *and*
proposal for a new work item: *Guidance to Avoiding Vulnerabilities in Programming Languages Through Language Selection and Use* (New York, NY: ISO/IEC JTC 1/SC 22 Secretariat, June 23, 2005).
Available from: http://www.open-std.org/jtc1/sc22/open/n3913.htm

301 By contrast with ISO/IEC standards, which define requirements, ISO/IEC TRs provide non-normative guidance.

302 ISO/IEC JTC 1/SC 22, "Information Technology—Programming Languages,Their Environments and System Software Interfaces—Extensions to the C library," part I of *Bounds-Checking Interfaces*, doc. ref. no. ISO/IEC TR 24731 (New York, NY: ISO/IEC JTC 1/SC 22 Secretariat, October 25, 2005).
Available from: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1146.pdf *and*

ISO/IEC JTC 1/SC 22, "Rationale for TR 24731 Extensions to the C library," part I of *Bounds-Checking Interfaces*, doc. ref. no. ISO/IEC TR 24731 (New York, NY: ISO/IEC JTC 1/SC 22 Secretariat, October 26, 2005).
Available from: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1147.pdf

303 IEEE, "IEEE P2600/D25c Draft Standard for Information Technology: Hardcopy System and Device Security," in *Methodologies and Processes for the Development of Secure HCDs*, doc. no. IEEE P2600/D24c. 9.2.2 (New York, NY: IEEE, December 2006).
Available from: http://grouper.ieee.org/groups/2600/drafts/FullSpec/IEEE_P2600_v24c.pdf

304 Jari Råman, "Regulating Secure Software Development: Analysing the Potential Regulatory Solutions for the Lack of Security in Software" (dissertation, University of Lapland, Rovaniemi Finland, May 26, 2006).
Available from: http://www.ulapland.fi/includes/file_download.asp?deptid=22713&fileid=9480&file=20061023140353.pdf&pdf=1 *and*

Jari Råman, "Contracting Over the Quality Aspect of Security in Software Product Markets," in *Proceedings of the Second ACM Workshop on Quality of Protection*, Alexandria, VA, 2006.

305 Nancy Mead (CMU SEI), "Who Is Liable for Insecure Systems?," *IEEE Computer* 37, no. 7 (July 2004):27–34.

306 Daniel Winteler (University of Oldenburg), *Liability of Programmers for Defects in Software* (Oldenburg, Germany: University of Oldenburg TrustSoft Institute, June 7, 2005).
Available from: http://trustsoft.uni-oldenburg.de/Members/daniel/Microsoft per cent20PowerPoint-Liability-TRUSTSOFT.pdf

7

Resources

7.1 Software Security Assurance Resources

The surge of interest and activity in software security and application security has brought with it a surge of online and print information about these topics. The following sections highlight those that are most often cited by practitioners in these fields.

7.1.1 Online Resources

None of the online resources listed here require registration or payment of fees prior to use. While a number of other resources available do require registration (*e.g.,* the Ounce Labs Library), we have chosen not to include them here to avoid the appearance of promoting the sponsoring organizations' commercial activities.

7.1.1.1 Portals, Websites, and Document Archives

Some of the most extensive and noteworthy online resources are maintained by the organizations whose initiatives are described in Section 7. These include—

- US Computer Emergency Response Team (US-CERT) BuildSecurityIn portal.
  Available from: https://buildsecurityin.us-cert.gov
- National Institute of Standards and Technology (NIST) Software Assurance Metrics and Tools Evaluation (SAMATE) portal.
  Available from: http://samate.nist.gov
  NOTE: SAMATE's email discussion list can be subscribed to from this portal.
- Open Web Application Security Project (OWASP) portal.
  Available from: http://www.owasp.org

In addition to these portals, individual software vendors (including software security tools vendors) and software security experts mantain a number of online resources. Among the most extensive are—

- Microsoft Security Developer Center.
  Available from: http://msdn2.microsoft.com/en-us/security/default.aspx
- SearchAppSecurity.com.
  Available from: http://searchappsecurity.techtarget.com

- Secure Programming.com.
  Available from: http://www.secureprogramming.com
- SPI Dynamics Software Security Training Tools.
  Available from: http://www.spidynamics.com/spilabs/education/index.html
- Fortify Security Resources.
  Available from: http://www.fortifysoftware.com/security-resources
- Secure Software Inc. Resources.
  Available from: http://www.securesoftware.com/resources
- Cigital Inc. Resources.
  Available from: http://www.cigital.com/resources
- PHP Security Consortium.
  Available from: http://phpsec.org
- SysAdmin, Audit, Networking, and Security (SANS) Reading Room.
  Available from: http://www.sans.org/reading_room—see the categories on "Application/Database Sec," "Best Practices," "Auditing & Assessment," "Malicious Code," "Scripting Tips," "Securing Code," and "Threats/Vulnerabilities."

7.1.1.2 Weblogs

The following blogs maintain updated resources and ideas on security topics. These are independent blogs. None of them are operated by software security tool vendors.

- Dana Epps (SilverStr's Sanctuary).
  Available from: http://silverstr.ufies.org/blog
- Michael Howard.
  Available from: http://blogs.msdn.com/michael_howard
- Gunnar Peterson (1 Raindrop).
  Available from: http://1raindrop.typepad.com
- Rocky Heckman (RockyH—Security First).
  Available from: http://www.rockyh.net
- "TrustedConsultant" (Writing Secure Software).
  Available from: http://securesoftware.blogspot.com
- David A. Wheeler.
  Available from: http://www.dwheeler.com/blog

7.1.1.3 Electronic Mailing Lists

The following electronic mailing lists provide open forum discussion on security topics:

- Secure Coding List (SC-L), moderated by Ken Van Wyk, co-author of *Secure Coding: Principles and Practices*.
  Available from: http://www.securecoding.org/list
- Web Application Security (webappsec), operated by OWASP.
  Available from: http://lists.owasp.org/mailman/listinfo/webappsec

- Web Security Mailing List (websecurity), operated by WASC.
  Available from: http://www.webappsec.org/lists/websecurity
- The archives of several now-defunct SecurityFocus mailing lists. These include the SECPROG (secure programming), VulnDev (undeveloped vulnerabilities), and Web Application Security mailing lists. SecurityFocus maintains the BugTraq database of software flaw and error reports.
  Archives are available from: http://www.securityfocus.com/archive.

In addition to these, a number of the organizations whose initiatives and projects are discussed in Section 7 run their own e-mail discussion lists for participants and other interested parties. Those mailing lists are usually publicized on the projects' web pages/portals.

7.1.2 Books

The number of books published annually on software security has increased steadily since the early 2000s, with some coming out in their second editions. The following are books (printed, not electronic) published on software security topics, listed by year of publication in reverse chronological order.

2008 (Scheduled for Publication)

- Alan Krassowski, and Pascal Meunier, *Secure Software Engineering: Designing, Writing, and Maintaining More Secure Code* (Addison-Wesley, 2008).

2007 (Some of These Are Only Scheduled for Publication)

- Brian Chess and Jacob West, Security Matters: *Improving Software Security Using Static Source Code Analysis* (Addison-Wesley Professional, 2007).
- MichaelCross, *Developer's Guide to Web Application Security* (Syngress Publishing, 2007).
- Eduardo Fernandez-Buglioni, Ehud Gudes, and Martin S. Olivier, *Security in Software Systems* (Addison Wesley, 2007).
- Donald G. Firesmith, *Security and Safety Requirements for Software-Intensive Systems* (Auerbach Publishers, 2007).
- Michael Howard and David LeBlanc, *Designing Secure Software* (McGraw-Hill, February 2007).
- Michael Howard, *Writing Secure Code for Windows Vista* (Microsoft Press, 2007).
- Haralambos Mouratidis and Paolo Giorgini, eds., *Integrating Security and Software Engineering: Advances and Future Visions* (Idea Group Publishing, 2007).
- Herbert Thompson, *Protecting the Business: Software Security Compliance* (John Wiley & Sons, 2007).

- Maura van der Linden, *Testing Code Security* (Auerbach Publishers, 2007).
- Chris Wysopal, *et al., The Art of Software Security Testing* (Addison Wesley/Symantec Press, 2007).

2006

- Mike Andrews and James A. Whittaker, *How to Break Web Software: Functional and Security Testing of Web Applications and Web Services* (Addison-Wesley Professional, 2006).
- Dominick Baier, *Developing More Secure ASP.NET 2.0 Applications* (Microsoft Press, 2006).
- Neil Daswani and Anita Kesavan, eds., *What Every Programmer Needs to Know About Security* (Springer-Verlag, 2006).
- Mark Dowd, John McDonald, and Justin Schuh, *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities* (Addison-Wesley Professional, 2006).
- Michael Howard and Steve Lipner, *The Security Development Lifecycle* (Microsoft Press, 2006).
- Gary McGraw, *Software Security: Building Security In* (Addison-Wesley, 2006).

2005

- Clifford J. Berg, *High Assurance Design: Architecting Secure and Reliable Enterprise Applications* (Addison-Wesley, 2005).
- Eldad Eilam, *Reversing: Secrets of Reverse Engineering* (John Wiley & Sons, 2005).
- James C. Foster, *et al., Buffer Overflow Attacks: Detect, Exploit, Prevent* (Syngress Publishing, 2005)
- Michael Howard, David LeBlanc, and John Viega, *19 Deadly Sins of Software Security* (McGraw-Hill Osborne Media, 2005).
- Robert Seacord, *Secure Coding in C and C++* (Addison-Wesley Professional, 2005).
- Herbert H. Thompson and Scott G. Chase, *The Software Vulnerability Guide* (Charles River Media, 2005).

2004

- Mark Burnett, *Hacking the Code, ASP.NET Web Application Security* (Syngress Publishing, 2004).
- Greg Hoglund and Gary McGraw, *Exploiting Software: How to Break Code* (Addison-Wesley, 2004).
- Sverre H. Huseby, *Innocent Code: A Security Wake-Up Call for Web Programmers* (John Wiley & Sons, 2004).

- Jan Jürjens, *Secure Systems Development With UML* (Springer, 2004).
- Jack Koziol, *et al., The Shellcoder's Handbook: Discovering and Exploiting Security Holes* (John Wiley & Sons, 2004).
- Vladimir Vasilievitch Lipaev, *Functional Security of Software Systems* (Synteg, 2004).
- Frank Swiderski and Window Snyder, *Threat Modeling* (Microsoft Press, 2004).
- Paul Watters, Michael Howard, and Steven Dewhurst, *Writing Secure Applications Using C++* (Osborne/McGraw-Hill, 2004).

2003

- John Barnes, *High Integrity Software: The SPARK Approach to Safety and Security* (Addison Wesley, 2003).
- Matt Bishop, Chapter 29, "Program Security," *Computer Security: Art and Science* (Addison-Wesley Professional, 2003).
- Irfan A. Chaudhry, *et al., Web Application Security Assessment* (Microsoft Press, 2003).
- Simson Garfinkel, Gene Spafford, and Alan Schwartz, Chapter 16, "Secure Programming Techniques," and Chapter 23, "Protecting Against Programmed Threats," *Practical Unix & Internet Security,* 3rd Ed. (O'Reilly & Associates, 2003).
- Mark G. Graff and Kenneth R. Van Wyk, *Secure Coding: Principles and Practices* (O'Reilly Media, 2003).
  Available from: http://www.securecoding.org/
- Microsoft Corporation, *Improving Web Application Security: Threats and Countermeasures* (Microsoft Press, 2003).
- John Viega and Matt Messier, *Secure Programming Cookbook for C and C++* (O'Reilly Media, 2003).
- James A. Whittaker and Herbert H. Thompson, *How to Break Software Security* (Addison Wesley, 2003).

2002

- Pavol Cerven, *Crackproof Your Software* (No Starch Press, 2002).
- Michael Howard and David LeBlanc, *Writing Secure Code*, 2nd Ed., (Microsoft Press, 2002).
- Art Taylor, Brian Buege, and Randy Layman, *Hacking Exposed: J2EE & Java—Developing Secure Web Applications with Java Technology* (McGraw-Hill/Osborne Media, 2002).

2001

- Ross J. Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems* (John Wiley & Sons, 2001).

- Gary McGraw and John Viega, *Building Secure Software: How to Avoid Security Problems the Right Way* (Addison-Wesley Professional, 2001).
- Ryan Russell, *Hack Proofing Your Web Applications: The Only Way to Stop a Hacker Is to Think Like One* (Syngress Media, 2001).

2000

- Gary McGraw and Edward W. Felten, *Securing Java: Getting Down to Business with Mobile Code, 2nd Ed.* (John Wiley & Sons, 1999).
- Michael Howard, *Designing Secure Web-Based Applications for Microsoft Windows 2000* (Microsoft Press, 2000)

1999 and before

- Simson Garfinkel and Gene Spafford, Chapter 11, "Protecting Against Programmed Threats" and Chapter 23, "Writing Secure SUID and Network Programs," *Practical Unix & Internet Security* 2nd Ed., (O'Reilly & Associates, 1996).
- Steven M. Bellovin, "Security and Software Engineering," *Practical Reusable Unix Software,* B. Krishnamurthy, ed. (John Wiley & Sons, 1995).
- Morrie Gasser, *Building a Secure Computer System* (Van Nostrand Reinhold, 1988).

7.1.3 Magazines and Journals With Significant Software Security Content

The following publications are either devoted to software security, have columns or sections devoted to software security, or frequently publish articles on software security topics:

- *Secure Software Engineering Journal*, peer-reviewed European journal devoted to security in the software development life cycle, established in 2007 by the developer of S2e (the last research methodology discussed in Section 5.1.8.2.6).
  Available from: http://www.secure-software-engineering.com
- *CrossTalk: The Journal of Defense Software Engineering*, publishes semi-annual issues devoted to software assurance (and sponsored by the DHS Software Assurance Program).
  Available from: http://www.stsc.hill.af.mil/crosstalk
- *IEEE Security and Privacy*, includes a monthly "BuildSecurityIn" column that focuses predominantly on software security issues.
  Available from: http://www.computer.org/portal/site/security
- IEEE *Transactions on Dependable and Secure Computing*; scholarly journal that includes at least one paper each month on research in software security tools and techniques.
  Available from: http://www.computer.org/portal/site/transactions/menuitem.a66ec5ba52117764cfe79d108bcd45f3/index.jsp?&pName=tdsc_home/&

- *Dr. Dobbs Journal*, includes a "Security" department that covers both software and information security issues.
  Available from: http://www.ddj.com

7.1.4 Conferences, Workshops, *etc.*

The following is a listing of conferences, workshops, and fora devoted to secure software themes.

- DoD/DHS Software Assurance Forum
  Available from: https://buildsecurityin.us-cert.gov/daisy/bsi/events.html
- OWASP Application Security Conference
  Available from: http://www.owasp.org/index.php/Category:OWASP_AppSec_Conference
- Software Security Summit
  Available from: http://www.s-3con.com
- International Workshop on Software Engineering for Secure Systems
  Available from: http://homes.dico.unimi.it/~monga/sess07.html
- International Workshop on Secure Software Engineering
  Available from: http://www.ares-conference.eu/conf/index.php?option=com_content&task=view&id=26&Itemid=33
- IEEE International Workshop on Security in Software Engineering
  Available from: http://conferences.computer.org/compsac/2007/workshops/IWSSE.html
- Secure Systems Methodologies Using Patterns
  Available from: http://www-ifs.uni-regensburg.de/spattern07
- German Society of Informatics Special Interest Group on Security Intrusion Detection and Response Conference on Detection of Intrusions and Malware and Vulnerability Assessment
  Available from: http://www.gi-ev.de/fachbereiche/sicherheit/fg/sidar/dimva

In March 2006, the first IEEE International Symposium on Secure Software Engineering (available from: http://www.jmu.edu/iiia/issse/) was held in Arlington, Virginia; and the Workshop on Secure Software Engineering Education and Training was held a month later in Honolulu, Hawaii. In March 2007, the first OMG Software Assurance Workshop (available from: http://www.omg.org/news/meetings/SWA2007/index.htm) was held in Fairfax, Virginia. It is not clear whether any of these events will be repeated.

Several conferences and workshops that focus on system , information, or network/cyber security, or software dependability topics include significant software security content in their programs. These include—

- Internet Society Network and Distributed System Security Symposium
  Available from: http://www.isoc.org/isoc/conferences/ndss
- Annual Computer Security Applications Conference
  Available from: http://www.acsac.org

- USENIX Security Symposium
  Available from: http://www.usenix.org/events/sec07
- Black Hat Briefings & Training
  Available from: http://www.blackhat.com/html/bh-link/briefings.html
- Workshops on Assurance Cases for Security (see Section 5.1.4.3)

7.2 Secure Software Education, Training, and Awareness

In *The Economic Impacts of Inadequate Infrastructure for Software Testing*, [307] a May 2002 report prepared for NIST, the author estimated that the annual cost of software defects in the United States was $59.5 billion. Each defect that remains undetected until after a software product has shipped can cost the supplier tens of thousands of dollars to address and patch; the cost to users of the product is often orders of magnitude higher.

Further exacerbating the problem is the fact, noted by Roger Pressman in his book *Software Engineering: A Practitioner's Approach*, that the cost of fixing a fault that originated in the software's requirements definition phase is multiplied by a factor of 10 with each subsequent life cycle phase. Another unsettling statistic: According to research done by Microsoft, 64 percent of in-house business software developers have admitted that they lack confidence in their own ability to write secure applications.

All the secure software processes, practices, technologies, and tools in the world will be of little use to the developer who has no idea how to use them or even why they are necessary. Recognizing this, software assurance practitioners in academia, government, and industry have begun redefining the components of software engineering education and software developer and programmer training and certification. Some of the fruits of their efforts are described in this section.

For Further Reading

US CERT, *Training and Awareness,* (Washington (DC): US CERT).
Available from: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/training.html

7.2.1 Academic Education in Secure Software Engineering

Most traditional university-level software engineering courses have not included specific content on software security. The majority of these security courses offered as part of computer science or software engineering programs have focused on system security engineering and/or information assurance. However, consistent with the overall increase of interest and activity in secure software engineering and software security assurance, a growing number of colleges and universities have begun to include content in current courses, and even dedicated courses, on security topics directly pertaining to secure software.

Courses on secure software development, secure programming, *etc.*, typically begin by introducing common attacks against software-intensive information systems and the vulnerabilities targeted by those attacks, then progress to

modeling, design, coding, and testing practices that software developers can adopt to reduce the likelihood that exploitable vulnerabilities will appear in the software they produce. The following is a representative sampling of such courses:

- Arizona State University: Software Security
- Ben-Gurion University (Beer-Sheva, Israel): Security of Software Systems
- Carnegie Mellon University (CMU) and University of Ontario (Canada): Secure Software Systems
- George Mason University: Secure Software Design and Programming
- George Washington University: Security and Programming Languages
- Catholic University of Leuven (Belgium): Development of Secure Software
- New Mexico Tech: Secure Software Construction
- North Dakota State University: Engineering Secure Software
- Northeastern University: Engineering Secure Software Systems
- Northern Kentucky University, Rochester Institute of Technology, and University of Denver: Secure Software Engineering
- Polytechnic University: Application Security
- Purdue University: Secure Programming
- Queen's University (Kingston, ON, Canada): Software Reliability and Security
- Santa Clara University: Secure Coding in C and C++
- University of California at Berkeley, Walden University (online): Secure Software Development
- University of California at Santa Cruz: Software Security Testing
- University of Canterbury (New Zealand): Secure Software
- University of Nice Sophia-Antipolis (Nice, France): Formal Methods and Secure Software
- University of Oxford (UK): Design for Security
- University of South Carolina: Building Secure Software.

As noted earlier, other schools offer lectures on secure coding and other software security relevant topics within their larger software engineering or computer security course offerings. At least two universities—the University of Texas at San Antonio and University of Dublin (Ireland)—have established reading groups [308] focusing on software security.

As part of its Trustworthy Computing initiative, Microsoft Research has established its Trustworthy Computing Curriculum program [309] for promoting university development of software security curricula. Interested institutions submit proposals to Microsoft, and those that are selected are provided seed funding for course development.

Another recent trend is post-graduate degree programs with specialties or concentrations in secure software engineering (or security engineering for software-intensive systems). Some of these are standard degree programs, while others are specifically designed for the continuing education of working professionals. The following are typical examples:

- James Madison University: Master of Science in Computer Science with a Concentration in Secure Software Engineering
- Northern Kentucky University: Graduate Certificate in Secure Software Engineering
- Stanford University: Online Computer Security Certificate in Designing Secure Software From the Ground Up
- University of Colorado at Colorado Springs: Graduate Certificate in Secure Software Systems
- Walden University (online): Master of Science in Software Engineering with a Specialization in Secure Computing
- University of Central England at Birmingham: Master of Science in Software Development and Security
- Chalmers University (Gothenburg, Sweden): Master of Science in Secure and Dependable Computer Systems.

In another interesting trend (to date, exclusively in non-US schools), entire academic departments—and in one case a whole graduate school—are being devoted to teaching and research in software dependability, including security, *e.g.*—

- University of Oldenburg (Germany) TrustSoft Graduate School of Trustworthy Software Systems
- Fraunhofer Institute for Experimental Software Engineering (IESE) (Kaiserslautern, Germany): Department of Security and Safety
- Bond University (Queensland, Australia): Centre for Software Assurance.

An impressive amount of research in a wide range of software security topics is also underway at colleges and universities worldwide. This research is discussed in Section 7.3.

As noted in Section 6.1.9.1, to support academics in adding software security to their curricula, the DHS Software Assurance Program's Education and Workforce Working Group (WG) drafted the Software Assurance CBK. No doubt, DHS had no idea how controversial this draft CBK would be among academics and industry software assurance practitioners. While the draft CBK has been praised by a number of software assurance practitioners and educators, and even in draft form has been adopted as the basis for modifying existing curricula at some schools, a significant number of detractors in academia and industry have voiced their concerns that the draft CBK is not only questionable in terms of its utility but also potentially damaging if used for its intended purpose. In response to these critics, DHS's Education and Workforce WG has undertaken consultations with academia and is reviewing a major revision to the draft CBK that is intended to address many of their concerns. As this new edition has yet to be publicly released, it remains to be seen whether it will be less controversial than the last.

For Further Reading

Rose Shumba (Indiana University of Pennsylvania), *et al.*, "Teaching the Secure Development Lifecycle: Challenges and Experiences: 2006," *Proceedings of the 10th Colloquium for Information Systems Security Education,* June 5-8, 2006, 116-123.
Available from: http://www.cisse.info/colloquia/cisse10/proceedings10/pdfs/papers/S04P02.pdf

Eduardo B. Fernandez and Maria M. Larrondo-Petrie (Florida Atlantic University), "A Set of Courses for Teaching Secure Software Development: 2006," *Proceedings of the 19th Conference on Software Engineering Education and Training Workshops*; April 19-21, 2006, 23.
Available from: http://doi.ieeecomputersociety.org/10.1109/CSEETW.2006.4

James Walden and Charles E. Frank (Northern Kentucky University), "Secure Software Engineering Teaching Modules: 2006," *Proceedings of the 3rd Annual Conference on Information Security Curriculum Development,* September 2006, 19-23.
Available from: http://portal.acm.org/citation.cfm?id=1231052&coll=Portal&dl=ACM&CFID=18481565

Zhaoji Chen (Arizona State University) and Stephen S. Yau, "Software Security: Integrating Secure Software Engineering in Graduate Computer Science Curriculum: 2006," *Proceedings of the 10th Colloquium for Information Systems Security Education,* June 5-8, 2006, 124-130.
Available from: http://www.cisse.info/colloquia/cisse10/proceedings10/pdfs/papers/S04P03.pdf

William Arthur Conklin (University of Houston) and Glenn Deitrich (University of Texas at San Antonio), "Secure Software Engineering: A New Paradigm," *Proceedings of the 40th Annual Hawaii International Conference on System Sciences,* 272.
Available from: http://doi.ieeecomputersociety.org/10.1109/HICSS.2007.477

Andy Ju An Wang (Southern Polytechnic State University), "Security Testing in Software Engineering Courses: 2004," *Proceedings of the 34th ASEE/IEEE Frontiers in Education Conference,* October 20-23, 2004.
Available from: http://fie.engrng.pitt.edu/fie2004/papers/1221.pdf

Kurt Stirewalt, [*Mississippi State University (MSU)], Integrating Threat Modeling into the Software Design Course at MSU.*
Available from: http://www.cse.msu.edu/~enbody/ThreatModeling.htm

Linda A. Walters (Norfolk State University), *Integration of Software Security Model in Existing Introductory Technology Courses (TR NSUCS-2004-004).*
Available from: http://www.cs.nsu.edu/research/techdocs/TR004_Linda_Walters.pdf

Brian Roberts, Doug Cress, and John Simmons, [University of Maryland Baltimore Campus (UMBC)], *A Strategy to Include Defensive Programming Tactics in the Undergraduate Computer Science Curriculum at UMBC,* CMSC 791 Information Assurance Project, c2003.
Available from: http://www.cs.umbc.edu/~cress1/cmsc791.html

7.2.2 Professional Training

The professional technical training community is also offering individuals and organizations an increasing number of courses on application security, secure programming, and related topics.

A number of professional training classes and seminars are being offered in software security by commercial training firms, software development or security services/consulting firms, software tools vendors, and academic institutions. For example (these lists are representative)—

- **Consulting Firms**: Security Compass, Aspect Security, Security Innovation, Systems and Software Consortium, KrVW (Kenneth R. Van Wyk) Associates, Secure Software, EnGarde Systems.
- **Tools Vendors**: Foundstone, Symantec, LogiGear, Microsoft, Paladion Networks, Next Generation Security Software, Siegeworks, Netcraft

- **Training Firms**: SANS Institute, Security University, Software Quality Engineering
- **Academic Institutions**: Catholic University of Leuven (Belgium).

Typical of the types of courses being offered are those in the Security Training series offered by Software Quality Engineering:

- Software Security Testing and Quality Assurance (2 days)
- Risk-Based Security Testing (1 day)
- Software Security Fundamentals (2 days)
- Creating Secure Code (2 days)
- How to Break Software Security (2 days)
- How to Break Web Software Security (2 days)
- Creating Secure Code for C/C++ (1 day)
- Creating Secure Code for ASP.NET (1 day)
- Creating Secure Code for Java (1 day)
- Architecting Secure Solutions (2 days)

As noted in Section 6.1.9.1, consistent with its mission to increase awareness and knowledge, the DHS Software Assurance Program's Education and Workforce WG is developing an Essential Body of Knowledge (EBK) as a counterpart to its draft Common Body of Knowledge (CBK). The intent is to provide professional training organizations and departments with a basis for developing classes, seminars, and workshops on secure software engineering.

7.2.2.2 Professional Certifications

A handful of security certification and technology training organizations have established professional certifications that validate competency in secure software development or knowledge of information security issues as they pertain to software (or software-intensive system) development.

The first of these professional certifications was announced by the International Council of Electronic Commerce Consultants (EC-Council): the EC-Council Certified Secure Programmer and Certified Secure Application Developer certifications. [310] More recently, Security University has begun offering a Software Security Engineer Certification [311] to people who attend Security University's regime of a half-dozen courses on secure software topics.

In mid-2006 the SANS Institute announced its national Secure Programming Skills Assessment (SPSA), [312] an examination to be rolled out nationwide in 2007 (a multi-institution test of the SPSA was undertaken in 2006, to refine it in preparation for worldwide release). There will, in fact, be four versions of the SPSA examination, which is intended to help employers in government and industry gauge how well their programmers have mastered knowledge about common software programming flaws that manifest as vulnerabilities, and how to avoid or correct them. These are—

- Secure programming skills using C and C++
- Secure programming skills using Java and JSP
- Secure programming skills using Perl and PHP
- Secure programming skills using .NET [313] and ASP.

In March 2007, SANS established its new Software Security Institute and announced that the Institute would award one of three levels of Certified Application Security Professional (CASP) certification to anyone who passes the SANS-administered SPSA exam. The three CASP levels represent three secure programming skill levels: 1 = minimally skilled, 2= advanced, and 3 = expert.

The SPSA examinations will be offered in three ways: (1) three annual exams administered by SANS at designated testing sites; (2) enterprise-licensed exams for SANS's Secure Programming Enterprise Partners (private and public organizations that contract with SANS) administered by SANS at the Partner facility; (3) self-assessments available at any time online. Self-assessments are not formally administered; they are intended for study/practice only and will not earn the test-taker a CASP.

Prior to establishing the SPSA and the Software Security Institute, SANS, in coordination with the Global Information Assurance Consortium (GIAC), had already established three certification specialties in domains relevant to software security:

- Level 5 Web Application Security
- Level 6 Reverse Engineering Malware
- Level 6 Security Malware.

Microsoft Corporation, which has been very active in its promotion of developer education as part of its own Trustworthy Computing Initiative Security Development Lifecycle initiative, now offers two optional exams for developers attempting to achieve Microsoft Certified Application Developer (MCAD) or Microsoft Certified Software Developer (MCSD) certifications: Implementing Security for Applications with Microsoft Visual Basic .NET, and Implementing Security for Applications with Microsoft Visual C# .NET. Unfortunately, at this time neither of these software security-relevant exams is mandatory, nor may either exam be counted as one of the four core exams required to attain an MCAD or MCSD certification.

The Canadian Engineering Qualifications Board of the Canadian Council of Professional Engineers Examination Syllabus for Software Engineering includes an elective exam on security/safety.

The Certification Labs of the International Institute for Training, Assessment, and Certification (http:/www.IITAC.org) offer a Certified Secure Software Engineering Professional certification as well as Certified Reverse Code Engineering Professional and Certified Exploit and Shell Code Development Professional. These certifications, which can be attained entirely

via web-based training, are structured in compliance with ISO/IEC 17024, General Requirements for Bodies Operating Certification of Persons.

7.2.3 Efforts to Raise Awareness

In recent years, several initiatives have been undertaken to raise awareness about software security. These initiatives include a number of surveys of organizations that are large consumers of software and those that are major producers. The purpose of such surveys is to raise awareness at the executive level of the software security issues that affect their businesses, to gauge current levels of awareness of and activity to address those issues, and to establish a basis for identifying and prioritizing additional needed software security practices. Of the three most recent surveys, one was administered in the user community [represented by the Chief Information Officer (CIO) Executive Council], the second in the vendor community (represented by the Secure Software Forum), and the third by independent academic researchers (at the University of Glasgow). These surveys are described in Sections 7.2.3.1, 7.2.3.2, and 7.2.3.3, respectively.

In addition to surveying, other strategies have been undertaken for building software security awareness in the public and private sectors. Those of the DoD, DHS, and Software Assurance Forum are described in Section 7. Two additional awareness campaigns of note have been undertaken by very influential organizations in the financial services sector—the first as a joint effort by two trade associations, BITS and the Financial Services Roundtable, and the second by Visa International. These activities are described in Sections 7.2.3.4 and 7.2.3.5 respectively.

The Gartner Group, which has long been a bellwether for "hot" technology concerns, has since the mid-2000s focused a significant amount of attention on application security and secure software and has published four oft-quoted, frequently cited reports on these topics in the past 3 years:

- John Pescatore, *Management Update. Keys to Achieving Secure Software Systems,* September 22, 2004.
- Amrit T. Williams and Neil MacDonald, *Organizations Should Implement Web Application Security Scanning,* September 21, 2005.
- Amrit T. Williams, *Implement Source Code Security Scanning Tools to Improve Application Security,* April 4, 2006
- Rich Mogull, *et al.*, *Hype Cycle for Data and Application Security 2007,* December 21, 2006

Other respected research groups (*e.g.,* the Burton Group, which has established Application and Content Security as a research focus area that includes a specialization in Malware) are also focusing on application/software security.

The DHS software assurance program formed the Business Case WG at the March 2006 DHS Software Assurance Forum. The Business Case WG aims to advance the awareness and understanding of and the demand for assured

software within the IT community. By providing a forum for academia, industry, and government to work together, DHS has established a dialog that can be joined by other government agencies and industry partners. One of the first activities the Business Case WG undertook was to reach out to the CIO Executive Council to gather information about the opinions of industry as a whole on the current state of the practice of software development. See Section 6.1.9.1 for more information on this WG.

The DoD Software Assurance Program's Outreach tiger team seeks to improve the state of commercial-off-the-shelf (COTS) products through supplier assurance. This tiger team is discussed in Section 6.1.1. By reaching out to industry and academia, DoD aims to improve the state of software security assurance through constructive dialog rather than through regulation. The tiger team is developing a Systems Assurance Whitepaper outlining DoD's concerns with respect to software security and discussing solutions DoD is currently researching. By sharing this information with industry and academia, the tiger team will be able to test the waters before recommending any changes to existing DoD policy. In addition, the Industry Outreach tiger team produced several documents describing DoD's vision for improving the state of software security assurance: a *Software Assurance CONOPS* (see Section 5.1.1.2.1) and (2) *Systems Assurance* guidebook (see Section 6.1.1.2).

Several private sector outreach activities are described below.

7.2.3.1 CIO Executive Council Poll

Established in the United States in April 2004, the CIO Executive Council aims to promote collaboration and exchange of views on technology issues among chief information officers around the world. It serves as a nexus point for CIOs to act as resources for one another and mutually advance the CIO position in industries worldwide, and comprises hundreds of CIOs worldwide. All council members must serve as the senior-most IT executive in their organization and have purchase authority for their organization's information technology products and services.

In September 2006, the CIO Executive Council held a poll [314] to gauge its members' opinions on the state of security in software development. The respondents, 84 CIOs around the country, were nearly universal (95 percent) in their agreement that reliability and not functionality or "features," as is widely claimed by software vendors, is the software attribute most important to their organizations. [315] The respondents also shared a lack of confidence in the security of their current software, including its ability to function free of flaws, vulnerabilities, and malicious code. The poll indicated that 86 percent of CIOs rate the software used by their firms as "vulnerable" or "extremely vulnerable."

The CIOs also indicated that they would welcome improved practices within the software vendor community, including certifying that the software they distribute meets a designated security target and providing evidence that the software has been scanned for flaws and security vulnerabilities using

qualified tools. Just under half of the CIOs also said they would like vendors to provide a list of the flaws and security vulnerabilities that they know to be inherent to their software.

Finally, the CIOs were in agreement about the impact of software vulnerabilities on their organizations' productivity, mainly due to the need to redeploy staff to deal with incidents caused by software faults, exploits, and malicious code. A majority of respondents also reported an increase in associated IT costs and a reduction in the productivity yielded by their IT systems as a result of such incidents.

7.2.3.2 SSF Survey

During its 2005 conference, the Secure Software Forum (SSF) (see Section 6.2.5) surveyed its executive members to determine the existing level of industry effort to improve the security of commercial software products. [316] The survey reported overall shortcomings across the industry in the security of its software development processes. While over a third of respondents reported having implemented developer training programs in secure coding practices, only 30 percent reported having established a security assurance program in their own development process, while even fewer, only 25 percent, had adopted a software security testing process using sophisticated tools. This despite the fact that 82 percent of respondents' firms develop software for use outside of their own organizations, thus acknowledging that their software security failures affected not only themselves, but their customers.

7.2.3.3 University of Glasgow Survey

The computer science department of the University of Glasgow (UK) is active in research into techniques and tools for security engineering in software-intensive systems. From July–August 2005, Glasgow researchers administered their Web Engineering Security Application Survey [317] to 16 employees in several technical roles in the security department of a Fortune 500 financial services firm that was chosen as being representative of a major corporation that counted heavily on dependability in its software.

When asked a range of questions about their firm's application life cycle development process, the respondents conveyed a largely negative view of the role security plays in that process. Through the survey, the Glasgow researchers confirmed the existence of a conflict between the firm's application developers and those responsible for implementing best security practices. Respondents reported that application development projects seldom reached their predetermined goals; developers' perception that the need for security presented an obstacle to goal-fulfilment was perceived as the main reason. Half of the survey respondents complained that developers were not held accountable or penalized when they did not abide by established secure

development practices. The same number of respondents believed security should play a larger role in the organization's development environment.

7.2.3.4 BITS/Financial Services Roundtable Software Security and Patch Management Initiative

BITS established its Software Security and Patch Management Initiative for its member financial institutions. The Initiative has three primary goals:

- Encourage software vendors that sell to critical infrastructure industries to undertake a higher "duty of care"
- Promote the compliance of software products with security requirements before those products are released
- Make the patch-management process more secure and efficient, and less costly to software customer organizations.

A fourth goal of the initiative is to foster dialogue between BITS members and the software industry to produce solutions to software security and patch management problems that are effective, equitable, and achievable in the near term.

BITS is acting jointly with the Financial Services Roundtable to encourage the financial services companies that make up their collective membership to—

- Develop best practices for managing software patches
- Communicate to software vendors clear industry business requirements for secure software products
- Facilitate CEO-to-CEO dialogue between the software industry and the financial services industry and also critical infrastructure sectors
- Analyze costs associated with software security and patch management; communicate to the Federal Government the importance of investing to protect critical infrastructure industries
- Explore potential legislative and regulatory remedies.

As part of the initiative, BITS has established the BITS Product Certification Program (BPCP) that tests software applications and supporting infrastructure products used by financial institutions against a set of baseline security criteria established by the financial services industry and aligned with the Common Criteria. The BPCP was established in hopes of influencing the vendor community to include security considerations in its development processes, and to improve the security of software products used in the financial services industry. The outcome of a successful BPCP test is a BITS Tested Mark that certifies that the tested product passed BITS BPCP testing.

Together, BITS and the Financial Services Roundtable have also issued what they call their Software Security and Patch Management Toolkit. [318] This "toolkit" is a set of documents that provide recommended security requirements, a template for a cover e-mail message, and procurement

language for ensuring that security and patch management requirements are incorporated in the procurement documents issued by financial institutions to their potential software suppliers. The toolkit also includes a set of talking points and background information designed to help executives of financial institutions communicate with their software suppliers and the media about software security and patch management.

Finally, BITS and the Financial Services Roundtable are encouraging the software industry to notify companies about vulnerabilities as early as possible.

7.2.3.5 Visa USA Payment Application Best Practices

As part of its Cardholder Information Security Program (CISP), the credit card company Visa USA developed a set of Payment Application Best Practices (PABP) [319] to assist the software vendors of payment applications in developing those applications to support their users' (*i.e.,* merchants and service providers, including those who operate online) compliance with the Payment Card Industry Data Security Standard.

Visa promotes its best practices through a combination of software vendor education and a certification scheme whereby a software vendor contracts with a Qualified Data Security Company (QDSC) designated by Visa to audit the application for conformance with the PABP.

Applications that pass the QDSC audit are added to Visa's widely publicized list of CISP-Validated Payment Applications. To maintain their applications on the list, the vendors must also undergo an Annual On-Site Security Reassessment by a QDSC. Visa also mails out letters and publishes other awareness information for their member merchants and service providers strongly encouraging them to use only PABC-validated, Payment Card Industry Data Security Standard-compliant applications.

7.3 Research

The following sections describe observed trends in academic research in a number of software security-relevant areas, both regarding the location of research organizations and projects, and the most active topic areas being pursued by them.

7.3.1 Where Software Security Research is Being Done

The proliferation of research centers, labs, and groups, in both US and non-US institutions, devoted to software security research reflects the broadening of focus by researchers formerly concerned primarily with software safety, reliability, or quality. For example, research centers, laboratories, and groups at the following universities (listed in alphabetical order) have expanded their activities to include significant research in software security areas:

- Bond University (Queensland, Australia): Centre for Software Assurance
- Carnegie Mellon University: Software Engineering Institute
- City University (London, UK): Centre for Software Reliability

- Fraunhofer Institute for Experimental Software Engineering (IESE) (Kaiserslautern, Germany): Department of Security and Safety
- Queen's University (Kingston, Ontario, Canada): Queen's Reliable Software Technology group
- Radboud University Nijmegen (The Netherlands): Laboratory for Quality Software
- Swinburne University of Technology (Melbourne, Australia) Reliable Software Systems group
- Technical University of Munich (Germany): WG on Security and Safety in Software Engineering
- University of California at Santa Barbara: Reliable Software group
- University of Idaho: Center for Secure and Dependable Systems
- University of Mannheim (Germany): Laboratory for Dependable Distributed Systems
- University of Newcastle upon Tyne (UK): Centre for Software Reliability
- University of Mannheim (Germany): Dependable Distributed Systems group
- University of Stuttgart (Germany): Software Reliability and Security group
- University of Virginia: Dependability Research group.

Software security research has also evolved from research in the areas of IA, system security, and computer security. The following universities (again listed in alphabetical order) have research centers, groups, and labs with significant software security research projects underway:

- Auburn State University Samuel Ginn College of Engineering: Information Assurance Laboratory
- Catholic University of Leuven (Belgium) Department Elektrotechniek-ESAT: Computer Security and Industrial Cryptography group
- Catholic University of Leuven (Belgium) DistriNet Research Group: Security WG
- Fraunhofer Institute for Experimental Software Engineering (Germany): Department. of Security and Safety
- Naval Postgraduate School: Center for Information Systems Security Studies and Research
- Pennsylvania State University: Systems and Internet Infrastructure Security Laboratory
- Purdue University: Center for Education and Research in Information Assurance and Security
- Radboud University Nijmegen Laboratory for Quality Software: Security of Systems group
- State University of New York (SUNY) at Stony Brook: Secure Systems Laboratory
- Stevens Institute of Technology: Secure Systems Laboratory

- Technical University of Denmark at Lyngby: Safe and Secure IT Systems group
- Technical University of Munich Competence Center in IT: Security, Software and Systems Engineering group
- University of Auckland (New Zealand): Secure Systems group
- University of California at Davis: Computer Security Laboratory
- University of Cambridge (UK) Computer Laboratory: Security group
- University of Idaho: Center for Secure and Dependable Systems
- University of Illinois at Urbana-Champaign: Information Trust Institute
- University of Maryland at College Park: Institute for Advanced Computer Studies Center for Human Enhanced Secure Systems
- University of Texas at Dallas: Cybersecurity Research Center and Security Analysis and Information Assurance Laboratory
- University of Tokyo (Japan) Yonezawa Group: Secure Computing Project.

A large number of universities support dedicated secure software and secure programming research groups. This is not surprising considering many of the vulnerabilities discovered on an almost daily basis result from programming errors. Examples of universities with such research groups are—

- CMU: CyLab Software Assurance Interest Group
- Fraunhofer Institute for Experimental Software Engineering (Germany): Department of Security and Safety
- German Research Center for Artificial Intelligence: Secure Software group
- Northeastern University: Software and Architecture Security group within the Institute for Information Assurance
- Princeton University: Secure Internet Programming group
- Purdue University CERIAS: Software Vulnerabilities Testing, Secure Patch Distribution, and Secure Software Systems (S3) groups
- Tokyo Institute of Technology Programming Systems Group (Japan): Subgroup on Implementation Schemes for Secure Software
- University of Bremen (Germany): Security and Safety in Software Engineering WG
- University of Oulu (Finland): Secure Programming Group
- University of Stuttgart (Germany) Institute for Formal Methods in Computer Science: Secure and Reliable Software Systems Group.

While most academic research in software assurance and software security is being pursued in North America and Europe, several research projects are underway on other continents in countries as far flung as Australia and New Zealand, Japan, Iran, Tunisia, and Nigeria.

In addition to academic institutions, commercial firms and government agencies often sponsor dedicated research organizations, a number of which are actively researching aspects of application and software security. The Naval Research

Laboratory's (NRL's) Center for High Assurance Computer Systems in particular has been engaged in a number of projects with software security assurance dimensions. In the private sector, research establishments exist both within large corporations such as Microsoft Research and IBM Research's Secure Software and Services Group and in small firms such as Cigital and Next Generation Security Software.

7.3.2 Active Areas of Research

In the United States where the software security community is populated primarily by entities originally involved in information assurance or software quality, the predomination of research activities tends to reflect this. For example, US institutions dominate in academic research devoted to vulnerability and malicious code classification and detection, reverse engineering, and security risk assessment and threat modeling for software. The United States also has a near-monopoly on research into extending software security concerns to the development of high-assurance information systems and the application of high-assurance system development approaches to the development of secure software or execution environments. Typical projects of this type are underway at the Naval Postgraduate School (High Assurance Security Program) and University of Idaho [Multiple Independent Levels of Security (MILS)].

Outside the United States and particularly in Europe, research trends reflect the emergence of security as an extension of software safety: the majority of European software security research is being performed by institutions long active in software safety research. Many of these research projects focus on extending software safety concepts, methodologies, and technologies to also address the security property of software. Government agencies in Europe have also long relied on safety cases as the basis for approving safety-critical systems for operation. The same research institutions that have been involved in safety case research are now adapting or extending those software safety cases to serve as software security assurance cases.

Research into application of formal methods to software security is being conducted primarily in the United States, Europe, and Australasia. Not only have Formal methods have not only long been used in the development of safety-critical software but also in the evaluation of cryptographic algorithms, cryptosystems, security protocols, and trusted kernels. The requirement for formal models and proofs is the most significant differentiating prerequisite for TOEs aspiring to the highest Common Criteria (CC) Evaluation Assurance Level (EAL) (EAL 7), as it was previously for systems aspiring to evaluation at Trusted Computer System Evaluation Criteria (TCSEC) Class A1. (From an academic standpoint, formal methods provide an excellent teaching tool as they represent an unambiguous application of pure mathematics to the problem of system and software engineering.)

Trusted hosts [*e.g.,* Trusted Processor Modules (TPM), MILS] and constrained execution environments (*e.g.,* virtual machines, sandboxes) are subjects of

significant research in the United States and Japan, while research into software tamperproofing and obfuscation is being pursued in North America, Europe, and Australasia, often initially in the context of intellectual property protection and only afterward in the context of its applicability to software security assurance.

The most technology transfer from academic research into commercial products appears in the areas of software security testing (especially static analysis) techniques and tools. Many leading commercial and open source tools originated in academic research projects. Upon graduation, the researchers either formed their own companies or joined existing tool vendors or software security/application security firms to continue evolving their academic prototypes and open source distributions into commercial products. Over time, the same commercial firms have continued to collaborate with and fund academia to pursue additional research that ultimately yields enhancements for those commercial products as well as improves the state-of-the-art of software security testing in general.

Other software security research areas of significant activity include—

- Language-based security, including secure compilers and compilation techniques, type and memory safety, secure languages and variants on standard languages, and proof- and model-carrying code (which applies formal methods to language-based security). By far the most active area of research, with significant projects in more than 20 institutions across the globe.
- Component-based development of secure software systems.
- Reverse engineering and reengineering for security.
- Vulnerability and malware detection, analysis, and prevention.
- Security of mobile code and mobile software agents; when the research areas specific to software security concerns are augmented by research into secure agent computing frameworks and secure inter-agent communications, this becomes one of the most active areas of software and application security research.
- Secure SDLC processes and methodologies, which can be phase specific *(e.g.,* requirements analysis and specification, architectural modeling) or whole-life cycle (for examples of the latter, see Section 5.1.8).
- Software security metrics for measuring both security of software itself and impact of different life cycle processes on the ability to achieve secure software.

Security of embedded software is an area that is apparently of limited interest (notable projects can be found at Princeton University and Radboud University Nijmegen in the Netherlands). Similarly, very few institutions are conducting significant research in the application of artificial intelligence techniques (*e.g.,* artificial immunology) or in software diversity, though the latter has been pursued by the University of Virginia [under funding by the Defense Advanced Research Projects Agency (DARPA) Information Processing Technology

Office's (IPTO)] Self-Regenerative Systems program, and by University of California at Davis and the State University of New York at Stony Brook.

Appendix H provides an extensive listing of significant (unclassified) academic research projects throughout the world devoted to software security assurance (as of February 2007).

Few attempts appear to have been made to document the trends in the area of software security research. An exception is the 2006 National Research Council (NRC) of Canada's report undertaken by George Yee, a researcher in the NRC's Institute for Information Technology. Yee's report [320] attempted to characterize secure software research over the nearly 25 year period from 1981–2005. Using key-phrase searches, Mr. Yee retrieved papers from the ACM and IEEE document archives that contained either the phrase "secure software" or the phrase "application security." He assigned the research projects described in those papers to categories of topics, including "Vulnerability Identification," "Threat Identification," "Coding Methods," "Testing," "Integrity Verification," *etc.*, then counted the number of projects in each category. Unfortunately, Yee's methodology is so seriously flawed—by searching only on those two key phrases, his document retrievals omitted the many, many relevant research papers that contained neither phrase. In so doing, Yee rendered the NRC report findings highly suspect in terms of completeness, accuracy, or meaningfulness of the research trends it purports to characterize.

References

307 G. Tassey (Research Triangle Institute), *The Economic Impacts of Inadequate Infrastructure for Software Testing*, planning rep. no. 02-3 (Gaithersburg, MD: NIST Program Office Strategic Planning and Economic Analysis Group, May 2002).
Available from: http://www.nist.gov/director/prog-ofc/report02-3.pdf

308 At University of Texas, a Secure Software Reading Group; at University of Dublin, a Formal Methods and Security Reading Group.

309 "Trustworthy Computing Curriculum 2005 Request for Proposals (RFP) Awards" [web page] (Redmond, WA: Microsoft Research).
Available from: http://research.microsoft.com/ur/us/fundingopps/RFPs/TWC_Curriculum_2005_RFP_Awards.aspx *and*

"Trustworthy Computing Curriculum 2004 RFP Awards" [web page] (Redmond, WA: Microsoft Research).
Available from: http://research.microsoft.com/ur/us/fundingopps/TWC_CurriculumRFPAwards.aspx

310 "EC-Council Certified Secure Programmer (CSP) and Certified Secure Application Developer (CSAD)" [web page] (Albuquerque, NM: International Council of Electronic Commerce Consultants [EC-Council]).
Available from: http://www.eccouncil.org/ecsp/index.htm

311 "Secure University Software Security Expert Bootcamp" [web page] (Stamford, CT: Security University).
Available from: http://www.securityuniversity.net/classes_SI_SoftwareSecurity_Bootcamp.php

312 "SANS Software Security Institute" [portal page] (Bethesda, MD: SANS Institute).
Available from: http://www.sans-ssi.org/

313 It is not clear whether or not the .NET exam will include C#.

314 Conducted between September 12 and 24, 2006, among CIO Council members and deputy members.

315 Karen Fogerty, "New CIO Executive Council poll reveals CIOs Lack Confidence in Software,". press release (Framingham, MA: CIO Executive Council, October 11, 2006).
Available from: https://www.cioexecutivecouncil.com/nb/?nid=9422bb8:434a3b6:8c77c:dd04c:80cdf83990

316 Stacy Simpson (Merrit Group), and Ashley Vandiver (SPI Dynamics Inc.), "Findings Reveal Organizations Are Taking Steps to Implement Security Assurance Programs but Most Programs Are Still Immature," press release (Atlanta, GA: SPI Dynamics Inc. for Secure Software Forum, February 15, 2006).
Available from: http://www.securesoftwareforum.com/SSF2006/press.html

317 William Bradley Glisson and Ray Welland (University of Glasgow), *Web Engineering Security (WES) Application Survey*, tech. report No. TR-2006-226 (Glasgow, Scotland: University of Glasgow Dept. of Computer Science, August 24, 2006).
Available from: http://www.dcs.gla.ac.uk/publications/paperdetails.cfm?id=8287

318 BITS/Financial Services Roundtable, *BITS/FSR Software Security Toolkit* (Washington, DC: BITS, February 2004).
Available from: http://www.bitsinfo.org/downloads/Publications%20Page/bitssummittoolkit.pdf

319 "Cardholder Information Security Program Payment Applications" [web page] (San Francisco, CA: VISA USA.).
Available from: http://usa.visa.com/business/accepting_visa/ops_risk_management/cisp_payment_applications.html.

Note that the PABP are presented in an application security checklist that includes descriptions of best practice requirements and provides test procedures for verifying that PABP practices have been satisfied in applications. The objective of the PABP is to ensure that applications comply with the PCI Data Security Standard; the majority of the 13 PABP best practices and supporting sub-practices focus on data protection, although three best practices directly address software security issues: Best Practice #5, Develop Secure Applications; Best Practice #7, Test Applications to Address Vulnerabilities; and Best Practice #10, Facilitate Secure Remote Software Updates.

320 George Yee (National Research Council Canada), *Recent Research in Secure Software*, report no. NRC/ERB-1134 (Fredericton, NB, Canada: NRC Institute for Information Technology, March 2006).
Available from: http://iit-iti.nrc-cnrc.gc.ca/publications/nrc-48478_e.html

8
Observations

The state-of-the-art of software security assurance has been constantly improving since research in the field began in the 1970s. Nevertheless, there are a number of particularly intractable challenges identified as far back as 1999 by the Government Accountability Office (in its report, *DoD Information Security: Serious Weaknesses Continue to Place Defense Operations at Risk*) and the INFOSEC Research Council (in its first *National Scale INFOSEC Research Hard Problems List).*

The general observations below highlight significant achievements, trends, anomalies, and deficiencies associated with the activities, standards, methodologies, techniques, and technologies described in Sections 2–8 of this SOAR.

8.1 What "Secure Software" Means

As recently as 2006, Gary McGraw felt the need to remind his readers that, "We must first agree that software security is not security software." The problem arising from the confusion of these terms [321] is that it can lead to two incorrect conclusions:

- The only software that needs to be secure is software that performs security functions.
- Implementing security functions will make software secure.

An unscientific survey of recent articles, books, papers, presentations, blogs, and other discussions on "software security" and "secure software" indicate that the distinction between *security* software and *secure* software is, at last, becoming widely understood…at least in theory. Providing developers with examples of what is meant by secure software, *e.g.,* software that does not contain buffer overflows, race conditions, and other exploitable faults, is one way in which the confusion about these distinct concepts has been clarified.

When it comes to practical advice, however, this distinction still seems muddy. A significant amount of literature and teaching about "secure software

engineering" still includes equal parts advice on how to avoid vulnerabilities and guidance on how to implement security functions in software applications. The discussion below on security design patterns includes a specific example of a technique for implementing security functions in software that is frequently, and incorrectly, discussed in the context of producing software that is in and of itself secure. [322]

8.2 Outsourcing and Offshore Development Risks

As indicated by the increasing number of articles, studies, and reports on outsourcing and offshoring risks, the Government particularly, but also increasingly, industry is still primarily concerned with the potential security threat posed by foreign software developers and suppliers. In some cases, that threat is expressed solely in terms of the possibility of a foreign-influenced developer implanting malicious code in software-for-export. Far less attention has been paid to the problem of the insider threat, *i.e.,* bad actors within US-based software development and integration firms. When the insider threat is raised, the focus is primarily on the problem of foreign workers in US firms. Little attention is being paid to the possibility of US developers being suborned by hostile foreign governments, criminal organizations, *etc.*

The main response to the increasing alarm regarding the risks posed by foreign-sourced software has been to rethink how acquisition is done and to provide guidance (although not yet policy or legislation) to aid the acquirer in attaining some level of confidence in the trustworthiness of the software supplier. In the technological realm, solutions such as those from Palamida and Blackduck Software have been developed to discover information about pedigree or provenance of software (mainly in source code form) for purposes of license enforcement and intellectual property protection (mainly for open source software [OSS]) or reengineering of legacy code. These solutions are being expanded and recast as security analysis techniques for software, either in deployment or in pre-acquisition evaluation. However, because many commercial software licenses prohibit reverse engineering for any purpose, it is not clear that this approach can be adopted without violating license agreements for analysis of anything but noncommercial binary code, which severely limits its usefulness.

8.3 Malicious Code in the SDLC

Hand in hand with concern about offshore-developed software is concern regarding the potential for malicious code entering the user's environment not only through operational means (*i.e.,* network-based viruses, worms, spyware, *etc.*), but through implanting malicious code (Trojan horses, logic and time bombs, malicious bots, *etc.*) in software prior to its distribution. In this area, at least, the possibility of the US-based developer as an insider threat has been considered. Still unresolved, at least to some extent, is the definition of what code actually is malicious. Definitions that designate any undocumented

code as malicious require Easter eggs, spyware, and adware to be considered malicious in all cases. Other definitions base their designation of malicious code on the code's intent (either stated or apparent). However, intent can be extremely difficult to determine; moreover, even if the code itself is benign, the fact that it is present and undocumented makes it a high value target for attackers seeking hard-to-detect entry points into the system.

An increasing amount of guidance for addressing malicious code (malware) includes discussions of detecting and blocking the inclusion of malicious code during the software's development life cycle, rather than only after its deployment. In academia, a significant amount of research is being devoted to devising technologies for detecting the presence of malicious code (or malicious logic) in source code under development, as well as in binary executables prior to installation.

8.4 Vulnerability Reporting

One of the most active areas of research and practical application focuses on reporting vulnerabilities in software. Over the past 5 years, there has been a move to standardize and centralize the collection of vulnerability reporting data, culminating with the MITRE Corporation's Common Vulnerabilities and Exposures (CVE) and the National Institute of Standards and Technology (NIST) National Vulnerability Database (NVD). The ability to make the vast amount of collected vulnerability data easier for specific audiences to analyze and act upon has led to classification, taxonomy, and metadata efforts such as the Common Weakness Enumeration (CWE) and Vulnerability Exploit Description and Exchange Format (VEDEF).

In an attempt to move the state-of-the-art of vulnerability knowledge forward to allow whitehats compete with black hats and criminals has led to the initiation of schemes whereby vulnerability researchers are paid royalties for exclusive intellectual property rights to their reports. These schemes have raised questions regarding the ethics and efficacy of such financial incentives. Some critics are concerned that such efforts will lead to bidding wars with cyber criminals. In fact, the fees offered per vulnerability were more than seven times their initial 2002 value after only 3 years.

What is not yet clear is whether many more security violations reported to various Computer Emergency Response Teams (CERT), Computer Security Incident Response Team (CSIRT), could be unambiguously attributed to the exploitation of vulnerabilities in software, and whether the various vulnerability databases and classifications described here do, in fact, accurately reflect the real state of vulnerabilities in software. The CWE effort was launched to help address this problem.

8.5 Developer Liability for Vulnerable Software

On both sides of the Atlantic, the need to overcome the impracticality of holding software suppliers, and even individual developers, liable for the poor quality and vulnerability of their software has been widely debated. Academics are investigating possible approaches, both regulatory and contractual, to achieve some level of developer and supplier liability and regulation, comparable to that in other engineering disciplines. However, unless and until software quality and security metrics are defined that are precise, meaningful, and reliable enough to hold up in court, it is doubtful that any real progress can be made toward regulating the software industry in this way.

8.6 Attack Patterns

Research and use of attack patterns is being pursued primarily in government and industry, and to a lesser extent in academia. Initially investigated in the context of information system vulnerability assessment and threat modeling, [323] attack patterns are now being promoted (*e.g.,* by Cigital) as providing a useful basis for security analyses throughout the software life cycle, specifically to—

- Specify explicit requirements for resistance of, tolerance of, and resilience to specific attack patterns
- Avoid design and implementation problems that would make the software vulnerable to specific attack patterns
- Include attack pattern-based criteria and checks in design and code reviews, security tests, and post-deployment vulnerability assessments [324] so that the rate of vulnerability detection and the effectiveness of vulnerability mitigations are increased.

To date, only known attack patterns have been used in the definition of abuse/misuse cases, threat models, attack graphs and trees, *etc.* This limitation of focus to known attack patterns presents problems, however. Attackers are very near to their "holy grail"—the routine production of zero-day exploits—novel, and thus *unknown*, attacks that target and exploit vulnerabilities that have yet to be publicly reported let alone mitigated through patches or other countermeasures. While there is consensus that the improved discovery, avoidance, and mitigation of vulnerabilities to *known* attack patterns will improve the security of software, they will not be effective when the software is confronted with unknown attack patterns. Currently, research into the application of fault tolerance, autonomic computing, artificial immunology, and other techniques to address the challenge of making software less vulnerable to unknown attack patterns is still in its infancy.

8.7 Secure Software Life Cycle Processes

In the vast majority of organizations, the security of the software they used was taken on faith, at least until Microsoft publicly acknowledged vulnerabilities in some of its key strategic products and very publicly undertook to modify not only its products, but the software development life cycle (SDLC) processes by which those products were built, to reduce their overall vulnerability rates. At the same time, a growing number of other secure software process models and methodologies have been published, mainly in academia, but also in the private sector by software security luminaries John Viega and Gary McGraw.

With the exception of Microsoft's Security Development Lifecycle (SDL) and Oracle's Software Security Assurance Process, there is no documented evidence that any of these secure software methodologies has been used in real-world software development beyond a few relatively small pilot projects (either purely academic, or under the auspices of academic–industry partnerships). Microsoft alone has published some crude metrics (comparing numbers of vulnerabilities in pre-SDL and post-SDL versions of strategic software products) as indicative of the effectiveness of its methodology.

The process improvement community has not been negligent in this area, either. Back in 2002, the Department of Defense (DoD) and Federal Aviation Administration (FAA) established a joint project to define safety and security extensions to the Integrated Capability Maturity Model (iCMM) and Capability Maturity Model Integration (CMMI), [325] and the efforts by International Standards Organization (ISO)/International Electrotechnical Commission (IEC) and Institute of Electrical and Electronics Engineers (IEEE) to revise ISO/IEC 15026, *System and Software Engineering—System and Software Assurance*, focus on adding security assurance activities to the system and SDLCs defined in ISO/IEC 12207 and 15288. However, as of this writing, the DoD/FAA "extensions" document is still only a proposal that does not appear to even be under consideration for adoption in DoD, while the ISO/IEC 15026 revision is still under development; even when (if) it is approved by ISO/IEC, as an international standard, its acceptance by DoD is far from guaranteed. The other potential problem with process standards is that they must apply in the majority of cases to have any hope of widespread adoption. For this reason, they tend to reflect the lowest common denominator of agreed best practices. They should be seen, then, as providing high-level frameworks into which more specific, extensive methodologies and practices can be inserted and integrated.

What is interesting is that despite all of the uncertainty about specific security-enhanced methods and process models, the simple philosophy of "fix the process, and you'll fix the software" is being increasingly touted as self-evident. [326] A major shift is underway, both among software suppliers and software users, moving them away from exclusive reliance on application security (defense-in-depth [DiD]) measures and "penetrate-and-patch" activities toward the recognition that the way to deal with vulnerabilities

in software is by changing the way it is specified, designed, implemented, compiled, tested, and distributed to reduce the likelihood that such vulnerabilities will be present in the first place. When, how, and to what extent this shift in mindset will translate into the widespread, disciplined adoption of security-enhanced development practices and processes is still unclear.

8.8 Using Formal Methods for Secure Software Development

Formal methods have long been the subject of academic study (they appear to provide an ideal teaching tool because they bridge the gap between theoretical mathematics and practical computer science), but have been limited in their practical use to assuring the correctness of cryptographic algorithms and security protocols, and of small high-consequence software programs, such as embedded safety-critical software programs and trusted operating system kernels. Only recently have formal methods started to be discussed as a means for assuring the required security properties of software-intensive systems. Also emerging are several "semi-formal" methods (*e.g.,* Praxis High Integrity Systems' Correctness by Construction) that add aspects of formalism to otherwise non-formal structured development processes. Perhaps most significant of all is the increasing amount of work in the formal methods community to automate as many formal activities as possible to make these otherwise extremely arcane, labor-intensive activities practical for use by nonexperts in assuring the safety and security of larger software systems.

8.9 Requirements Engineering for Secure Software

It is widely argued that the reason software is not secure is that adequate requirements for software security are never specified. It is also a widely accepted rule that negative and nonfunctional requirements, because they are not "testable" or "actionable," should not be documented. The problem is, to capture requirements that will result in secure software, the requirements analyst must feel free to go through a mental process that includes stating all of the negative and nonfunctional requirements that are crucial to defining the constraints on software behavior that results in its being secure. The next step, then, is to analyze those negative and nonfunctional requirements in order to map them to positive functional requirements that will enable the software to satisfy them. This is, in fact, true whether the negative and nonfunctional requirements pertain to security, reliability, performance, or any other required software property.

Unfortunately, the evolution of negative/nonfunctional requirements into positive functional requirements is still an unresolved "hard problem" for which, although it is widely acknowledged and actively researched, there has still been no practical solution proposed outside of academia.

8.10 Security Design Patterns

The definition and use of design patterns—a concept that originated in architecture in the late 1970s, and 10 years later was suggested for achieving reuse at a higher level of abstraction than source code, *i.e.,* reusable designs for common software functions—for security functions in information systems and software applications was first proposed in the mid-2000s, and was soon the focus of a number of books [327], websites [328], and other publications. [329] To date, this work has focused exclusively on design patterns for common security functionality (*e.g.,* authentication, authorization, encryption, digital signature, *etc.*). There have yet to emerge any definitions of reusable design patterns for functions that would directly contribute to the security of software, such as input validation, [330] attack pattern detection, fault tolerant exception handling, *etc.* Nor does there appear to be any research underway on this topic. This is noteworthy because much of the guidance on secure software development recommends the use of security design patterns without reconciling or even acknowledging the conflict between what security design patterns actually are *(i.e.,* patterns for security functions) and the types of design patterns that would actually contribute to the attack-resistance, tolerance, and resilience of software.

8.11 Security of Component-Based Software

As with security design patterns, the majority of research into the "composability of assurance" for component-based software has used information system security (*i.e.,* confidentiality, integrity, and availability of information processed by the system) as the basis for its concept of whether a system is "secure." The focus of security for component-based development has been on issues like security of information flows across intercomponent interfaces and what the composite assurance level of a system assembled from components with different Common Criteria (CC) Evaluation Assurance Levels (EAL) will be. The research most directly relevant to secure software development has focused on how components reveal their security properties and assumptions to each other and how component-based systems interact with, and receive security protection from, their execution environments. Also, the significant attention paid to issues of security of commercial-off-the-shelf (COTS), OSS, and other nondevelopmental software products is directly relevant, because such software products are often used as components of larger software-intensive systems.

8.12 Secure Coding

If the numerous books and other guidance resources on secure coding are indicative, "coding" is a term that is not very precisely understood. While most definitions equate "coding" with "programming," which is defined as an implementation-phase activity, most guidance on secure coding and secure

programming not only includes information but does little to distinguish among secure design principles (mapped to the design phase of the SDLC), secure coding practices (mapped to the implementation phase of the SDLC), and security testing techniques including post-compilation (blackbox) techniques (late coding phase for whitebox, testing/integration phase for blackbox).

The problem is that because they attempt to cover so much of the life cycle, these broad-scope secure coding guides tend to provide their recommendations at a level of detail that may be right for an architect or designer, but is too high level and imprecise to be of much help to the actual coder. At best, this case of false advertising means that the programmer who plans to invest in one or more of these "secure coding" guides should first review them thoroughly to determine whether, in fact, the information they contain is at a level of detail and extensive enough (secure coding guides should, ideally, include actual source code examples) to be of practical use.

At a lower level, compilers and programming libraries are working to improve the security of software code. Several "safe" versions of the C library are available—and Microsoft provides one with new versions of its Visual Studio compiler. Similarly, modern C compilers will detect-and-correct or detect-and-flag a number of well-known coding errors that have historically led to vulnerabilities in software. Because developers routinely rely on compilers to detect syntax errors, using a similar interface for security errors improves security without modifying the programmer's workflow.

8.13 Development and Testing Tools for Secure Software

The current focus of many commercial and open source software security testing tools is finding problems that are indicative of vulnerabilities in source code that has already been written. The majority of these tools fall into the category of static analysis tools, although the number of tools for fuzzing and fault injection is increasing, as is the number of verifying compilers that perform security checks.

There are an increasing number of "safe" and "secure" programming languages. Many of these are variants on C and C++ that compensate for the lack of type-safety and memory-safety in those languages. One of the most widely used is Microsoft's C#. Even more widely used is Java, which is frequently cited as an inherently secure language, due to both its built-in type safety and its supporting Java Virtual Machine (JVM) environment. (C# similarly benefits from the code security features of the .NET environment which, because they are even more tightly integrated into the overall execution environment, are seen as having some advantages over the JVM in ease of use.)

8.14 Software Security Testing

Trends in software security testing are the adaptation of several techniques from the software safety, software quality, and blackhat communities for use in software security testing (*e.g.,* fuzzing, fault injection, reverse engineering). Significant effort within both the NIST Software Assurance Metrics and Tool Evaluation (SAMATE) and National Security Agency (NSA) Center for Assured Software (CAS) programs has gone into the classification, taxonomization, and comparison of these and other types of security testing tools (and even some development tools).

As the listing on the NIST SAMATE website attests, the number of software security testing tools is continually increasing. These tend to be test type, language, and platform specific. Even so, a first level of tool integration has been achieved, through use of central management "consoles" that provide a single interface to control the operation of a variety of testing tools (albeit usually from a single vendor). The result is a kind of multifunction tool "framework." As yet, however, there are few if any individual tools that are multifunction. Nor are there either individual tools or frameworks that are multilingual and multiplatform, *i.e.,* able to run a variety of tests that target software written in different languages and hosted on different platforms.

Another level of tool integration that seems far from being achieved is the correlation, fusion, and normalization of test results from different types and brands of software testing tools, as well as across test tools at the software, system, and network levels. Object Management Group's (OMG) Software Assurance Ecosystem represents a first step toward this second level of tool integration, although it is still questionable whether the approach of using a meta-language to normalize test results from different tools will require capturing those results at such a high level of abstraction as to cause the loss of important lower-level results.

Tools and techniques for building security test cases for software are becoming increasingly advanced. Unified Modeling Language (UML)-based misuse and abuse case modeling, threat modeling (whether based on Microsoft's methodology, or any of several others), attack patterns, trees, and graphs are all techniques that have emerged as equally useful for helping specify security requirements and architectures for software, and for defining security test cases for implemented software. It is increasingly true that software security testing is increasing in sophistication and scope beyond simple verification against generic application security checklists (such as those listed in Section 5.5.3.1).

8.15 Security Assurance Cases

As with formal methods, assurance cases are routinely used for the verification of safety-critical systems and high-assurance security systems. The first assurance cases were, in fact, safety cases for physical systems—in this case, nuclear facilities—mandated by law in Europe in the late 1960s. Other European safety laws followed regulating railroads, avionics, *etc.*, and all mandating safety

cases. In the United States, no similar reliance on safety cases emerged to verify compliance with safety regulations. However, the concept of security assurance cases has become inculcated through the CC evaluation process, where they take the form of Security Target documents.

The need for assurance cases is widely acknowledged in the software assurance community, where they are seen as a means for software producers to make claims about the security of their software and to provide evidence supporting those claims. Assurance cases are looked to as providing the basis for security evaluations of software products. Significant effort has already been expended by three important standards bodies (ISO/IEC, IEEE, and OMG) to define standards for (1) the content of assurance cases for software and (2) the process by which those assurance cases can be verified. At the same time, the software safety community has begun to adapt and extend its standards for software safety cases to also address the need to establish and verify software security properties. The SafSec standard developed by Praxis High Integrity Systems for the United Kingdom Ministry of Defence (UK MOD) is an example of such a standard for a "hybrid" safety/security assurance case for software.

8.16 Software Security Metrics

How much security is enough? At present, there appears to be no way to answer that question. Most software security guidance is either excessively pragmatic (do only just enough to get by) or excessively rigorous (do it all, or you'll still have vulnerable software). In short, without means to measure the effectiveness not just of whole processes but of individual practices and techniques, under specific circumstances and in particular environments, software security still appears to be a daunting all-or-nothing proposition.

Currently, existing and proposed software security metrics focus almost exclusively on counting and comparing vulnerabilities in implemented software, measuring the attack surface, or measuring complexity. No one has yet determined whether the comparison of numbers of vulnerabilities in earlier *vs.* later versions of software programs (the Microsoft metric) or the average number of vulnerabilities per *x* lines of code are, in fact, meaningful metrics in terms of indicating whether efforts to produce more secure software have succeeded, or for predicting the likelihood that software that appears to be secure in the development environment will, in fact, prove to be secure "in the wild" (*i.e.,* in deployment).

Both the information security and the software engineering communities appear to be looking at the problem of defining meaningful metrics for measuring the security of software. The metrics they are researching are based on existing metrics for system security measurement, and software quality, reliability, and safety measurement. The Department of Homeland Security's (DHS) Working Group on security metrics and measurement seems to be focused mainly on metrics adapted from the information security community. SAMATE, despite the "Metrics" in its name, has to date focused on the "TE" (tools evaluation) portion of its charter, and so has yet to begin pursuing work in this area.

8.17 Secure Software Distribution

The notion of "trusted distribution" defined by the Trust Computer System Evaluation Criteria (TCSEC) has been revived, although the focus now is on distribution *via* network-based downloads rather than physical media. One of the most noteworthy trends is the use of digital watermarking and digital rights management technologies initially developed for intellectual property protection and license enforcement as integrity and authorization techniques for downloaded software executables. Because these technologies were initially conceived for widespread usage in environments that lack any kind of security infrastructure, they are increasingly replacing or augmenting digital signature and cryptographic hashes as integrity mechanisms on software. The problem of trustworthy download source identification (*i.e.,* making sure the site from which software is downloaded is, in fact, a valid supplier site) is beginning to be addressed through authentication of download channels and through code signatures using certificates issued by trusted certification authorities.

8.18 Software Assurance Initiatives

During the last month in which this SOAR was being written, the Defense Intelligence Agency (DIA) and the Air Force both initiated software assurance initiatives. This is indicative of the rapidly increasing interest in software security assurance. Moreover, an unprecedented level of coordination and communication exists between existing software assurance initiatives across DoD and DHS, ensuring that duplication of effort is minimized and use of limited resources is optimized. What has not yet been achieved, however, is a similarly close coordination (beyond some specific projects, such as the iCMM/CMMI safety and security extensions) with the software assurance programs of other agencies such as the National Aeronautics and Space Administration (NASA), the FAA, and the Department of Energy. Even less communication and coordination appears to exist between US government and foreign allied government programs, which raises the question of the implications of this lack of cooperation for multinational coalitions and partners in the war on terror. For instance, if regulation and legislation and licensing of developers is ever achieved, without international cooperation, it is doubtful that it will have any impact given the global nature of the software industry.

Activities across the whole DoD/DHS software assurance community range from development of secure development toolsets for use across a single department (DoD) to definition of international standards (that are expected to benefit DoD and DHS). By nature of its mission, the focus of the DoD software assurance program is somewhat more parochial than that of DHS: DoD is focusing on its own needs and problems, while DHS, with its charter to secure the nation's critical infrastructures (including its cyber information infrastructure), necessarily takes a much broader view in defining the scope of its Software Assurance Program's activities.

Starting with application security consortia such as Open Web Application Security Project (OWASP) and Application Security Industry Consortium (AppSIC), and now expanding into software security consortia such as the Secure Software Forum and the newly-chartered Software Assurance Consortium, industry has also become increasingly and actively engaged in community-wide software security programs and activities.

8.19 Resources on Software Security

The number of books, websites, portals, blogs, and particularly papers and articles on software security topics has reached a point where it is impossible to effectively survey them all. As of January 2007, there is now even a magazine solely devoted to secure software engineering. This said, the accuracy, currency, and quality of content across this multiplicity of resources varies widely. Much is left online and on bookshelves that has, in fact, been superseded by later publications (often by the same authors); a handful of the earliest resources even carry disclaimers stating that they are probably no longer completely valid, and that the reader should seek more recent information. [331] Perhaps someday a researcher will undertake a semantic web-based portal through which all reviews and commentaries on print and online software security resources can be centrally accessed (and, ideally, appended with Amazon.com or TripAdvisor style ratings) to aid those seeking the best resources in locating them.

8.20 Knowledge for Secure Software Engineering

Particularly in the agile development community, the hotly debated question is how to inject the required security knowledge into software projects. The agile approach is predicated on the establishment of teams of developers who are all equally knowledgeable in the practices, techniques, and technologies needed throughout the agile life cycle to produce software that functions correctly. Agile development teams reject the idea of including specialists on their teams: there is not enough time or resources to accommodate team members who cannot contribute fully to all life cycle activities. The question of how security knowledge is acquired by software development teams is not limited to agile developers. Many development teams (including agile teams that are willing to accommodate specialists) hope that by adding a security expert or two to their ranks, they can "offload" responsibility for all software security concerns to those experts, while they themselves continue to write software as they have always done. Nevertheless, there is growing consensus within the software security assurance community that *all* participants in a software development project (including managers) need at least some knowledge of security.

Unlike systems engineering, where specialist security architects can to a great extent "overlay" security components and interfaces on top of business logic components and interfaces, secure software engineering entails the amplification and adaptation of good software engineering principles, practices,

and techniques rather than the addition of a whole separate and unrelated set of "secure" principles, practices, and techniques. In other words, secure software engineering is really a matter of adapting existing principles, practices, and techniques used in developing high-quality software. The question is how to best impart the knowledge developers need to know, which principles, practices, and techniques to adapt, and how to make those adaptations.

If Microsoft's example is salubrious, a combination of security specialists and experts on software teams and increased developer security knowledge is needed. The developer knowledge will focus on how to improve the practices, *etc.* the developer already uses, while the experts will be relied on to verify the security of the resulting software, and also to consult with the developers throughout the software process to continue reinforcing their new knowledge.

8.21 Software Security Education and Training

There is still widespread disagreement throughout academia about the best way to add software security to existing software engineering curricula, courses, and lectures. The problem most frequently cited is the perception that the software security subject matter can only be added to already full curricula, and this will be at the expense of other subjects. Few academics are yet willing to reassess the content of their existing curricula to determine whether it is all as important as (1) it was when their curricula were originally conceived and as (2) the software security subject matter that they are convinced must displace it. Further, there is disagreement on how best to teach software security: should be a separate course, an addendum to every course, or an augmentation similar to English departments' grammar clinics.

In the realm of professional training and certification, the number of courses on secure programming, security testing, and other practical aspects of software security is growing, as is the number of professional certifications for "secure programmers," "secure application developers," and "secure software engineers." DoD has already stated its intention to begin requiring developers of security-critical DoD systems to hold the certification associated with the SysAdmin, Audit, Networking, and Security (SANS) National Secure Programming Skills Assessment.

8.22 Software Security Research Trends

A caveat on the research trends reported in this SOAR: The majority of data on research trends comes from academia. Far less information is publicly available about research in industry and government in general, and about classified research in particular. Because this SOAR is unclassified, it avoids making any observations on classified research programs or projects. In any case, much of what is termed "government research" is, in fact, accomplished through government funding of academic and private research institutions. The apparent lack of coordination and communication between research teams and

laboratories (*vs.* individual researchers) across academia, and in some cases even within the same institution, [332] appears to be typical of how research in academia is conducted. One can only speculate on possible reasons for this lack of cooperation and communication. Perhaps there is a belief among academics that good results are more likely to emerge if more researchers work independently to solve a given problem than if fewer, albeit larger, coordinated groups of researchers collaborate to do so. (If this belief indeed explains the lack of cross-institution collaborations, it would be interesting to determine whether it arises simply from a "gut feeling" or whether empirical evidence supports it.) Another possibility is that within the context of teaching institutions, the need to monitor, mentor, and grade individual student researchers within research teams would be unacceptably difficult were the team to span multiple institutions. (In that case, it would be interesting to find out how professors who do manage this challenge feel about the additional level of effort, if in fact there is any.) A less savory possibility is that individual researchers' desire for credit and/or concern about retaining intellectual property rights to their work outweighs their desire to pursue the most optimal approach (*i.e.,* collaboration with other institutions) for attaining their research goals. [333]

References

321 Gary McGraw, *Software Security: Building Security In* (Boston, MA: Addison-Wesley; 2006).

322 The following article provides a classic example of confusing security software with software security: Kevin Sloan and Mike Ormerod, "How a Variety of Information Assurance Methods Deliver Software Security in the United Kingdom," *CrossTalk: The Journal of Defense Software Engineering* (2007 March 2007): 13–17. Available from: http://www.stsc.hill.af.mil/CrossTalk/2007/03/0703SloanOrmerod.html

323 Bruce Schneier, in *Attack Trees, Modeling Security Threats*, was one of the first to suggest using attack patterns for this purpose.

324 The following paper describes a typical application of this concept to software vulnerability assessment: Michael Gegick and Laurie Williams, "Matching Attach Patterns to Security Vulnerabilities in Software-Intensive Systems Designs," in *Proceedings of the Workshop on Software Engineering for Secure Systems*, St. Louise, MO, May 15–16. (Published as *ACM SIGSOFT Software Engineering Notes* 30, no. 4 [[New York, NY: ACM Press 2005]).

325 Keep in mind that a process improvement model, even a security-enhanced one, will only be as successful in improving the state of the resulting software as the process it is meant to codify and discipline. It is just as possible to codify a bad process, and to repeat it with quasi-mechanical discipline and rigor, as it is a good process.

326 This is, in fact, true of the growing interest in software security generally. It is widely reported that no real business case has been made to justify investment in software security, and no return-on-investment statistics have yet been captured. And yet the level of discussion, publication, and activity (*i.e.,* buzz") about software security continues to grow along what appears to be an increasingly steep upward trajectory. Is it a matter of the tail wagging the dog? Is all the "buzz" actually creating an illusion of a business case where one does not actually exist?

327 For example: Steel, *et al.,. Core Security Patterns*.

328 For example: "SecurityPatterns.Org" [web portal].
Available from: http://www.securitypatterns.org *and*

"Identity Management Security Patterns" [home page] (Santa Clara, CA: Sun Microsystems, Inc.)
"Java.Net Project" [website].
Available from: https://identitypatterns.dev.java.net

329 For example, the *Open Group Technical Guide to Security Design Patterns* and Joseph Yoder (University of Illinois at Urbana-Champaign), and Jeffrey Barcalow (Reuters Information Technology), "Architectural Patterns for Enabling Application Security," in *Proceedings of the Fifth Pattern Language of Programming Conference*, Monticello, IL, September 3–5 1997.
Available from: http://st-www.cs.uiuc.edu/~hanmer/PLoP-97/Proceedings/yoder.pdf

330 It should be noted, however, that several application frameworks now provide standard input validation support for applications that run within them.

331 One cannot help but wonder, given these caveats, why the authors of these earlier works continue to keep them online. Historical interest? Ego gratification?

332 For example, the CyLab and SEI at CMU are both pursuing software assurance research in parallel but without any coordination or cooperation between them.

333 Either that, or the disincentives in terms of concerns over intellectual property and patent rights outweigh any incentives in terms of benefiting from other researchers' findings and techniques before they are published.

A Acronyms

The following is a list of all abbreviations and acronyms used in this document, with their amplifications.

AA	Application Area
ACID	Atomic, Consistent, Isolated, and Durable
ACM	Association for Computing Machinery
ACSAC	Annual Computer Security Applications Conference
ADE	Agile Development Environment
AEGIS	Appropriate and Effective Guidance in Information Security
AFB	Air Force Base
AFRL	Air Force Rome Laboratories
AIA	Aerospace Industries Association
ANSI	American National Standards Institute
ANUBIS	Analyzing Unknown Binaries
AOM	Aspect Oriented Modeling
AOSD	Aspect Oriented Software Development
API	Application Programming Interface
AppSIC	Application Security Industry Consortium
ARINC	Aeronautical Radio, Incorporated
AS/NZS	Australian/New Zealand Standard

ASAP	Application Security Assurance Program
ASASI	Environment for Addressing Software Application Security Issues
ASD	Assistant Secretary of Defense
ASD	Adaptive Software Development
ASP	Active Server Pages
ASP	Agile Software Process
ASSET	Automated Security Self-Evaluation Tool
AT&L	Acquisition Technology and Logistics
AT/SPI	Anti-Tamper and Software Protection Initiative
AUP	Agile Unified Process
AusCERT	Australian CERT
AVDL	Application Vulnerability Description Language
BAN	Burrows-Abadi-Needham
BASAP	Business Application Security Assurance Program
BPCP	BITS Product Certification Program
BSD	Berkeley Software Distribution
BSI	Build Security In
C&A	Certification and Accreditation
CAML	Categorical Abstract Machine Language
CAMP	Code Assessment Methodology Project
CAPEC	Common Attack Pattern Enumeration and Classification
CAS	Center for Assured Software
CASE	Computer Aided Software Engineering
CASL	Common Algebraic Specification Language
CASP	Certified Application Security Professional
CASSEE	Computer Automated Secure Software Engineering Environment
CBK	Common Body of Knowledge
CC	Common Criteria
C3I	Command, Control, Communications and Intelligence
CCS	Calculus of Communicating Systems
CCTA	Central Communication and Telecommunication Agency
CD&R	Capability Development and Research
CECOM	Communications-Electronics Command
CERIAS	Center for Education and Research in Information Assurance and Security
CERT	Computer Emergency Response Team
CERT/CC	Computer Emergency Response Team Coordination Center
CHACS	Center for High Assurance Computer Systems
CHATS	Composable High-Assurance Trustworthy Systems
CHESS	Center Human Enhanced Secure Systems
CIAE	Center for Information Assurance Engineering
CISP	Cardholder Information Security Program
CISSP	Certified Information Systems Security Professional

CLASP	Comprehensive Lightweight Application Security Process
CM	Configuration Management
CME	Common Malware Enumeration
CMM	Capability Maturity Model
CMMI	Capability Maturity Model Integrated
CMU	Carnegie Mellon University
CNSS	Committee on National Security Systems
COAST	Computer Operations, Audit, and Security Technology
CONOPS	Concept of Operations
CORAS	Consultative Objective Risk Analysis System
CORBA	Common Object Request Broker Architecture
COSIC	COmputer Security and Industrial Cryptography
COTS	Commercial Off-The-Shelf
CRAMM	CCTA Risk Analysis and Management Method
CS&C	Cyber Security and Communications
CSDP	Certified Software Development Professional
CSDS	Center for Secure and Dependable Systems
CSIA	Central Sponsor for Information Assurance
CSIRT	Computer Security Incident Response Team
CSIS	Center for Strategic and International Studies
CSL	Computer Science Lab
CSP	Communicating Sequential Processes
CSSC	Control Systems Security Center
CSSE	Center for Systems and Software Engineering
CTMM	Calculative Threat Modeling Methodology
CTS	Construction, Transition, and Support
CVE	Common Vulnerabilities and Exposures
CVSS	Common Vulnerability Scoring System
CWE	Common Weakness Enumeration
DAA	Designated Approving Authority
DACS	Data and Analysis Center for Software
DARPA	Defense Advanced Research Projects Agency
DbC	Design by Contract
DCID	Director, Central Intelligence Directive
DDP	Defect Detection and Prevention
DDR&E	Director of Defense Research and Engineering
DES	Data Encryption Standard
DFARS	Defense Federal Acquisition Regulation Supplement
DHS	Department of Homeland Security
DIA	Defense Intelligence Agency
DIACAP	DoD Information Assurance Certification and Accreditation Process
DiD	Defense-in-Depth
DISA	Defense Information Systems Agency

DITSCAP	Defense Information Technology Security Certification and Accreditation Process
DOC	Detector of Obfuscated Calls
DoD	Department of Defense
DoDAF	DoD Architecture Framework
DoDD	DoD Directive
DoDI	DoD Instruction
DoDIIS	Department of Defense Intelligence Information System
DoS	Denial of Service
DOVES	Database of Vulnerabilities, Exploits, and Signatures
DREAD	Damage potential, Reproducibility, Exploitability, Affected users, Discoverability
DSB	Defense Science Board
DSDM	Dynamic System Development Method
DTIC	Defense Technical Information Center
EAL	Evaluation Assurance Level
EBK	Essential Body of Knowledge
EC	European Community
EC-Council	International Council of Electronic Commerce Consultants
EIA	Electronic Industries Association
EiD	Engineering in Depth
EIS	Electronics and Information Systems (department)
ELSW	Electronic Systems Wing
eMASS	Enterprise Mission Assurance Support System
EU	European Union
EUP	Enterprise Unified Process
FAA	Federal Aviation Administration
FAQ	Frequently Asked Questions
FAR	Federal Acquisition Regulation
FDD	Feature-Driven Development
FDR	Failure Divergence Refinement
FIPS	Federal Information Processing Standards
FISMA	Federal Information Security Management Act
FMEA	Failure Modes and Effects Analysis
FSO	Field Security Operations
FSR	Final Security Review
FUD	Fear, Uncertainty, and Doubt
FX/MC	Function Extraction for Malicious Code
GAO	Government Accountability Office
GCSS	Global Communication Support System
GEIA	Government Electronics and Information Technology Association
GIAC	Global Information Assurance Consortium
GIG	Global Information Grid
GOTS	Government Off-The-Shelf

GRL	Goal-oriented Requirements Language
GSEC	(GIAC) Security Essentials Certification
HBAL	Heap Bounded Assembly Language
HCD	Hard Copy Device
HIPAA	Healthcare Information Portability and Accountability Act
HTTP	HyperText Transfer Protocol
I&A	Identification and Authentication
IA	Information Assurance
IAC	Information Assurance Center
I2WD	Information and Intelligence Warfare Directorate
IASE	Information Assurance Support Environment
IATAC	Information Assurance Technology Analysis Center
IAVA	Information Assurance Vulnerability Alert
iCMM	Integrated Capability Maturity Model
IDA	Institute for Defense Analyses
IE	Inception and Elaboration
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
IESE	Institute for Experimental Software Engineering
IET	Institute of Engineering and Technology
IETF	Internet Engineering Task Force
IFIP	International Federation for Information Processing
IIS	Internet Information Server
IITAC	International Institute for Training, Assessment, and Certification
ILCS	Illinois Compiled Statutes
IMCS	Information Management Core Services
INFOSEC	Information Security
ING	Internationale Nederlanden Groep
INRIA	Institut National de Recherche en Informatique et en Automatique
IPS	Intrusion Prevention System
IPSec	Internet Protocol Security
IPTO	Information Processing Technology Office
IRC	INFOSEC Research Council
ISO	International Standards Organization
ISTLab	Information Systems Technologies Laboratory
IT	Information Technology
ITI	Information Trust Institute
IV&V	Independent Verification and Validation
IW	Information Warfare
JAD	Joint Application Design
JAD	Joint Application Development
J2EE	Java 2 Enterprise Edition
JPL	Jet Propulsion Lab
JSP	Java Server Pages

JTC	Joint Technical Committee
JVM	Java Virtual Machine
KDM	Knowledge Discovery Metamodel
LaQuSo	Laboratory for Quality Software
LCA	Life Cycle Architecture
LCO	Life Cycle Objectives
LD	Lean Development
MAFTIA	Malicious and Accidental Fault Tolerance for Internet Applications
MBASE	Model-Based Architecting and Software Engineering
MCAD	Microsoft Certified Application Developer
MCC	Model-Carrying Code
MCSD	Microsoft Certified Software Developer
MDA	Model Driven Architecture
MDD	Model Driven Development
MILOS	Méthodes d'Ingenierie de Logicels Securisés
MILS	Multiple Independent Levels of Security
MISRA	Motor Industry Software Reliability Association
MIT	Massachusetts Institute of Technolgoy
MOD	Ministry of Defense
MORDA	Mission Oriented Risk and Design Analysis
MOTS	Modified Off-The-Shelf
MSF	Microsoft Solutions Framework
MYSEA	Monterey Security Architecture
NASA	National Aeronautics and Space Administration
NCES	Net-Centric Enterprise Services
NCSC	National Computer Security Center
NCSD	National Cyber Security Division
NCSP	National Cyber Security Partnership
NDA	Non-Disclosure Agreement
NDIA	National Defense Industrial Association
NFR	Non-Functional Requirement
NGSS	Next Generation Security Software
NIACAP	National Information Assurance Certification and Accreditation Process
NIAP	National Information Assurance Partnership
NII	Networks and Information Integration
NIPRNet	Non-Sensitive Internet Protocol Network
NISCC	National Infrastructure Security Co-ordination Centre
NIST	National Institute of Standards and Technology
NRC	National Research Council
NRL	Naval Research Laboratory
NRM	Network Rating Methodology
NSA	National Security Agency
NSS	National Security System

NSTB	National SCADA Test Bed
NUREG	Nuclear Regulation
NVD	National Vulnerability Database
OASIS	Organization for the Advancement of Structured Information Standards
OCL	Object Constraint Language
OCTAVE	Operationally Critical Threat, Asset, and Vulnerability Evaluation
OMG	Object Management Group
OSD	Office of the Secretary of Defense
OSS	Open Source Software
OUSD	Office of the Under Secretary of Defense
OVAL	Open Vulnerability and Assessment Language
OWASP	Open Web Application Security Project
PA	Process Area
PABP	Payment Application Best Practices
PACC	Predicable Assembly from Certifiable Components
PAG	Program Analysis Group
PAM	Pluggable Authentication Module
PBT	Property-Based Tester
PELAB	Programming Environments Laboratory
PEPA	Performance Evaluation Process Algebra
PITAC	President's Information Technology Advisory Committee
PL	Protection Level
PLOVER	Preliminary List of Vulnerabilities Examples for Researchers
PRL	Program/Proof Refinement Logic
PSM	Practical Security Management
PSM	Practical Software Measurement
PSOS	Provably Secure Operating System
PTA	Practical Threat Analysis
QA	Quality Assurance
QDSC	Qualified Data Security Company
QRST	Queen's Reliable Software Technology (group)
R&D	Research and Development
RAD	Rapid Application Development
RAID	Redundant Array of Independent Disks
RAISE	Rigorous Approach to Industrial Software Engineering
RASQ	Relative Attack Surface Quotient
RBAC	Role-Based Access Control
RESE	Reconfigurable Reliability and Security Engine
RFID	Radio Frequency Identification
RFP	Request for Proposal
RIPP	Rapid Iterative Production Prototyping
RISOS	Research into Secure Operating Systems
RM	Reference Model

RMF	Risk Management Framework
ROI	Return on Investment
RSL	RAISE Specification Language
RSSR	Reducing Software Security Risk
RTCA	Radio Technical Commission for Aeronautics
RUP	Rational Unified Process
RUPSec	Rational Unified Process-Secure
S&S	Safety and Security
S&T	Science and Technoloy
S2e	Secure Software Engineering
S3	Secure Software Systems (group)
SAFECode	Static Analysis For safe Execution of Code
SafeCRT	Safe C/C++ Run-Time
SAL	Standard Annotation Language
SAMATE	Software Assurance Metrics and Tools Evaluation
SAML	Security Assertion Markup Language
SANS	SysAdmin, Audit, Networking, and Security
SAP	Systems Applications and Products
SBVR	Semantics of Business Vocabulary and Rules
SC	Subcommittee
SCADA	Supervisory Control and Data Acquisition
SC-L	Secure Coding List
SCM	Software Configuration Management
SCORE	Security Consensus Operational Readiness Evaluation
SCR	Software Cost Reduction
SDE	Secure Development Environment
SDL	Security Development Lifecycle
SDLC	Software Development Life Cycle
SE	Systems Engineering
SECPROG	Secure Programming
SECURIS	Secure Information Systems
SEI	Software Engineering Institute
SENSE	Software Engineering and Security
SERC	Software Engineering Research Center
SFDEF	Susceptibility and Flaw Definition
SIDAR	Security Intrusion Detection and Response
SiES	Security in Embedded Systems
SIG	Special Interest Group
SLAM	Software specification, Language, Analysis, and Model-checking
SML	Standard ML
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
SOAR	State-of-the-Art Report
SoS	Security of Systems (group)

SOUP	Software of Unknown Pedigree
SOW	Statement of Work
SP	Special Publication
SPattern	Secure Systems Methodologies Using Patterns
SPC	Software Protection Center
SPI	Software Protection Initiative
SPSA	Secure Programming Skills Assessment
SQL	Structured Query Language
SQUARE	Secure Quality Requirements Engineering
SRD	Software Reference Dataset
SRI	Stanford Research Institute
SSAA	System Security Authorization Agreement
SSAI	Software Security Assessment Instrument
SSCP	Systems Security Certified Practitioner
SSDM	Secure Software Development Model
SSE-CMM	Secure Systems Engineering Capability Maturity Model
SSF	Secure Software Forum
SSL	Secure Sockets Layer
SSO	Single Sign-on
ST	Security Target
ST&E	Security Test and Evaluation
STIG	Security Technical Implementation Guide
STRIDE	Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, Elevation of Privilege
SUNY	State University of New York
SwA	Software Assurance
TAMPER	Tamper and Monitoring Protection Engineering Research (Lab)
TCB	Trusted Computing Base
TCP/IP	Transmission Control Protocol/Internet Protocol
TCSEC	Trusted Computer System Evaluation Criteria
TCX	Trusted Computer Exemplar
TDD	Test-Driven Development
TF-CSIRT	Task Force-Computer Security Incident Response Team
T-MAP	Threat Modeling based on Attacking Path
TOE	Target of Evaluation
TPM	Trusted Processor Module
TR	Technical Report
TSP	Team Software Process
TSP-Secure	Team Software Process for Secure Software Development
UI	User Interface
UMBC	University of Maryland Baltimore Campus
UML	Unified Modeling Language
URL	Uniform Resource Locator
USC	University of Southern California

USG	US Government
USSTRATCOM	US Strategic Command
V&V	Verification and Validation
VCP	Vulnerability Contributor Program
VINCS	Virtual Infrastructure for Networked Computers
VDM	Vienna Development Method
VEDEF	Vulnerability Exploit Description and Exchange Format
VM	Virtual Machine
WASC	Web Application Security Consortium
WG	Working Group
WISDOM	Whitewater Interactive System Development with Object Models
WS	Web Services
xADL	eXtensible Architecture Description Language
XCCDF	XML Configuration Checklist Data Format
XML	eXtensible Markup Language
XP	eXtreme Programming

B
Definitions

The following are key terms used in this document and their definitions as those terms are meant to be understood in this document.

Abuse | Malicious misuse, usually with the objective of alteration, disruption, or destruction.

Assurance | Justifiable grounds for confidence that the required properties of the software have been adequately exhibited. In some definitions, assurance also incorporates the activities that enable the software to achieve a state in which its required properties can be verified or assured.

Attack | An attempt to gain unauthorized access to a system's services or to compromise one of the system's required properties (integrity, availability, correctness, predictability, reliability, *etc.*). When a software-intensive system or component is the target, the attack will most likely manifest as an intentional error or fault that exploits a vulnerability or weakness in the targeted software.

Availability | The degree to which the services of a system or component are operational and accessible when needed by their intended users. When availability is considered as a security property, the intended users must be authorized to access the specific services they attempt to access, and to perform the specific actions they attempt to perform. The need for availability generates the requirements that the system or component be able to resist or withstand attempts to delete, disconnect, or otherwise render the system or component inoperable or inaccessible,

regardless of whether those attempts are intentional or accidental. The violation of availability is referred to as *Denial of Service* or *sabotage*.

Blackhat | A person who gains unauthorized access to and/or otherwisecompromises the security of a computer system or network.

Component | A part or element within a larger system. A component may be constructed of hardware or software and may be divisible into smaller components. In the strictest definition, a component must have —

- A contractually specified interface (or interfaces)
- Explicit context dependencies
- The ability to be deployed independently
- The ability to be assembled or composed with other components by someone other than its developer.

In the less restrictive definition used in this SOAR, a component may also be a code module or code unit. A code unit is either one of the following:

- A separately testable element of a software component
- A software component that cannot be further decomposed into constituent components
- A logically separable part of a computer program.

A code module is one of the following:

- A program unit that is discrete and identifiable with respect to compilation, combination with other units, and loading, *i.e.,* a code unit
- A logically separable part of a computer program, *i.e.,* a code unit.

Compromise | A violation of the security policy of a system, or an incident in which any of the security properties of the system are violated.

Correctness | The property that ensures that software performs all of its intended functions as specified. Correctness can be seen as the degree to which any one of the following is true:

- Software is free from faults in its specification, design, and implementation.
- Software, documentation, and other development artifacts satisfy their specified requirements.
- Software, documentation, and other development artifacts meet user needs and expectations, regardless of whether those needs and expectations are specified or not.

In simple terms, software that is correct is (1) free of faults, and (2) consistent with its specification.

Countermeasure | An action, device, procedure, technique, or other measure that reduces the vulnerability or weakness of a component or system.

Critical Software | Software the failure of which could have a negative impact on national security or human safety, or could result in a large financial or social loss. Critical software is also referred to as *high-consequence* software.

Denial of Service (DoS) | The intentional violation of the software's availability resulting from an action or series of actions that has one of the following outcomes:

- The system's intended users cannot gain access to the system.
- One or more of the system's time-critical operations is delayed.
- A critical function in the system fails.

Also referred to as *sabotage*.

Dependability | The ability of a system to perform its intended functionality or deliver its intended service correctly and predictably whenever it is called upon to do so. The following properties of software directly contribute to its dependability:

- Availability
- Integrity
- Reliability
- Survivability
- Trustworthiness
- Security
- Safety.

Execution Environment | The aggregation of hardware, software, and networking entities surrounding the software that directly affects or in uences its execution.

Error | (1) Deviation of one of the software's states from correct to incorrect. (2) Discrepancy between the condition or value actually computed, observed, or measured by the software, and the true, specified, or theoretically correct value or condition. Some sources give a third meaning for error: (3) a human action that leads to a failure. For clarity, this SOAR uses the word *mistake* to convey this third meaning.

Failure | (1) Non-performance by a system or component of an intended function or service. (2) Deviation of the system's performance from its specified, expected parameters (such as its timing constraints).

Fault | The adjudged or hypothesized cause of an error.

Flaw | A mistake of commission, omission, or oversight in the creation of the software's requirements, architecture, or design specification that results in an inadequate and often weak design, or in one or more errors in its implementation. Some software assurance practitioners object to the word " aw" because it is often confused with "error," "fault," and "defect." (Just as "defect" is sometimes similarly confused with " aw.")

Formal | Based on mathematics. (This narrow definition is used in this SOAR to avoid the confusion that arises when "formal" is used both to mean "mathematically based" and as a synonym for "structured" or "disciplined.")

Formal Method | A process by which the system architecture or design is mathematically modeled and specified, and/or the high-level implementation of the system is verified, through use of mathematical proofs, to be consistent with its specified requirements, architecture, design, or security policy.

Independent Verification and Validation (IV&V) | Verification and validation performed by a third party, *i.e.,* an entity that is neither the developer of the system being verified and validated, nor the acquirer or user of that system.

Integrity | The property of a system or component that re ects its logical correctness and reliability, completeness, and consistency. Integrity as a security property generates the requirement for the system or component to be protected against intentional attempts to do one of the following:

- Alter or modify the software in an improper or unauthorized manner. (Note that attempts to destroy the software in an improper or unauthorized manner are considered attacks on the system's availability, *i.e.,* Denial of Service attacks)
- Through improper or unauthorized manipulation to cause the software to either perform its intended function(s) in a manner inconsistent with the system's specifications and the intended users' expectations, or to perform undocumented or unexpected functions.

Least Privilege | The principle whereby each subject (*i.e.,* actor) in the system is granted only the most restrictive set of privileges needed by the subject to perform its authorized tasks, and whereby the subject is allowed to retain those privileges for no longer than it needs them.

Malicious Code | Undocumented software or firmware intended to perform an unauthorized or unanticipated process that will have adverse impact on the dependability of a component or system. Malicious code may be self-contained (as with viruses, worms, malicious bots, and Trojan horses), or it may be embedded in another software component (as with logic bombs, time bombs, and some Trojan horses). Also referred to as *malware*.

Mistake | An error committed by a person as the result of a bad or incorrect decision or judgment by that person. Contrast with "error," which is used in this document to indicate the result of a "mistake" committed by software (*i.e.,* as the result of an incorrect calculation or manipulation).

Misuse | Usage that deviates from what is expected (with expectation usually based on the software's specification).

Quality | The degree to which a component, system, or process meets its specified requirements and/or stated or implied user, customer, or stakeholder needs and expectations.

Predictability | The properties, states, and behaviors of the system or component never deviate from what is expected.

Problem | Used interchangeably with anomaly, although "problem" has a more negative connotation and implies that the anomaly is, or results from, a flaw, defect, fault, error, or failure.

Reliability | The probability of failure-free (or otherwise satisfactory) software operation for a specified or expected period or interval of time, or for a specified or expected number of operations, in a specified or expected environment under specified or expected operating conditions.

Risk | The likelihood that a particular threat will adversely affect a system by exploiting a particular vulnerability.

Robustness | The degree to which a component or system can function correctly in the presence of invalid inputs or stressful environmental conditions, including inputs or conditions that are malicious in origin.

Sabotage | *See* Denial of Service.

Safety | Persistence of dependability in the face of realized hazards (unsponsored, unplanned events, accidents, mishaps) that result in death, injury, illness, damage to the environment, or significant loss or destruction of property.

Sandboxing | A method of isolating application-level components into distinct execution domains, the separation of which is enforced by software. When run in a sandbox, all of the component's code and data accesses are confined to memory segments within that sandbox. In this way, sandboxes provide a greater level of isolation of executing processes than can be achieved when processes run in the same virtual address space. The most frequent use of sandboxing is to isolate the execution of untrusted programs (*e.g.,* mobile code, programs written in potentially unsafe languages such as C) so that each program is unable to directly access the same memory and disk segments used by other programs, including trusted programs. Virtual machines (VM) are sometimes used to implement sandboxing, with each VM providing an isolated execution domain.

Secure State | The condition in which no subject can access another entity in an unauthorized manner for any purpose.

Security | Protection against disclosure, subversion, or sabotage. To be considered secure, software's dependability (including all constituent properties of that dependability) must be preserved in the face of threats. At the system level, security manifests as the ability of the system to protect itself from sponsored faults, regardless of whether those faults are malicious.

Service | A set of one or more functions, tasks, or activities performed to achieve one or more objectives that benefit a user (human or process).

Software Security Assurance | Justifiable grounds for confidence that software's security property, including all of security's constituent properties (*e.g.,* attack-resistance, attack-tolerance, attack-resilience, lack of vulnerabilities, lack of malicious logic, dependability despite the presence of sponsored faults, *etc.*), has been adequately exhibited. Often abbreviated to *software assurance.*

Software-Intensive System | A system in which the majority of components are implemented in/by software, and in which the functional objectives of the system are achieved primarily by its software components.

State | (1) A condition or mode of existence that a system or component may be in, for example the input state of a given channel. (2) The values assumed at a given instant by the variables that define the characteristics of a component or system.

Subversion | The intentional violation of the software's integrity.

Survivability | The ability to continue correct, predictable operation despite the presence of realized hazards and threats.

System | A collection of components organized to accomplish a specific function or set of functions.

Threat | Any entity, circumstance, or event with the potential to harm the software system or component through its unauthorized access, destruction, modification, and/or denial of service.

Trustworthiness | Logical basis for assurance (*i.e.,* justifiable confidence) that the system will perform correctly, which includes predictably behaving in conformance with all of its required critical properties, such as security, reliability, safety, survivability, *etc.,* in the face of wide ranges of threats and accidents, and will contain no exploitable vulnerabilities either of malicious or unintentional origin. Software that contains exploitable faults or malicious logic cannot justifiably be trusted to "perform correctly" or to "predictably satisfy all of its critical requirements" because its compromisable nature and the presence of unspecified malicious logic would make prediction of its correct behavior impossible.

User | Any person or process authorized to access an operational system.

Verification And Validation (V&V) | The process of confirming, by examination and provision of objective evidence, that —

- Each step in the process of building or modifying the software yields the right products (verification). Verification asks and answers the question "Was the software built right?" (*i.e.,* correctness).
- The software being developed or modified will satisfy its particular requirements (functional and nonfunctional) for its specific intended use (validation). Validation asks and answers the question "Was the right software built?" (*i.e.,* suitability).

In practical terms, the differences between verification and validation are unimportant

except to the theorist. Practitioners use the term V&V to refer to all of the activities that are undertaken to ensure that the software will function according to its specification. V&V is intended to be a systematic and technical evaluation of software and associated products of the development and maintenance processes. Independent V&V is a process whereby the products of the software development life cycle are reviewed, verified, and validated by an entity that is neither the developer nor the acquirer of the software, which is technically, managerially, and financially independent of the developer and acquirer, and which has no stake in the success or failure of the software.

Vulnerability | A development fault or weakness in deployed software that can be exploited with malicious intent by a threat with the objective of subverting (violation of integrity) or sabotaging (violation of availability) the software, often as a step toward gaining unauthorized access to the information handled by that software. Vulnerabilities can originate from weaknesses in the software's design, faults in its implementation, or problems in its operation.

Weakness | A aw, defect, or anomaly in software that has the potential of being exploited as a vulnerability when the software is operational. A weakness may originate from a aw in the software's security requirements or design, a defect in its implementation, or an inadequacy in its operational and security procedures and controls. The distinction between "weakness" and "vulnerability" originated with the MITRE Corporation Common Weaknesses and Exposures (CWE) project (http://cve.mitre.org/cwe/about/index.html).

Whitehat | A person who is ethically opposed to the abuse of computer systems. Motivated by that opposition, the whitehat frequently uses the blackhat's techniques and tools in order to confound blackhat compromise attempts and to protect the systems and networks targeted by them.

C

Types of Software Under Threat

The majority of literature, activity, and research in the software security community focuses on information systems software, especially application software, although some consideration is also given to middleware and operating system software, not least thanks to the publicity garnered by Microsoft's Trustworthy Computing Initiative, [334] a process improvement initiative adopted by Microsoft in 2002 to improve the security, privacy, and reliability both of its software products and its business practices.

Less attention is being paid to the security of networking software outside of wireless and mobile networks, or to the security of security-enforcing software, such as firewalls, intrusion detection systems, virtual machine monitors, operating system kernels, *etc.*, even though this software is often depended on to compensate for vulnerabilities and weaknesses in other software. This is not to say that software in these categories is not under threat: only that it appears to receive less explicit attention from software assurance practitioners, initiatives, vendors, *etc.*

Note: The authors recognize that there is a great deal of activity focused on developing and establishing techniques, technologies, tools, *etc.* for improving other dependability properties (reliability, safety, quality) in all software and software-intensive systems, including those that fall into the categories discussed here—enough activity, in fact, to fill several additional State-of-the-Art Reports. It has been clearly demonstrated that concentration on these other properties can coincidentally improve the security of software. However, as the Purpose of the document states, the focus of this SOAR is narrowly on those activities, initiatives, *etc.*, that are primarily or exclusively concerned with security as a property of software.

When information system software is targeted by an attacker, the ultimate objective is most often to bypass or exploit the privileges and mechanisms that enable the targeted software to access information the attacker has no authorization to access.

Some software-intensive systems are considered "high consequence." Failure of the software in such systems (whether accidental or intentionally induced) is expected to have effects with great and even catastrophic consequences. These effects may be immediately apparent, *e.g.,* the crash of a plane resulting from a failure in air traffic control system software, or the effects may only propagate over time, gradually increasing to a critical level. For example, consider the fairly new category of information system, the electronic voting system. It is difficult to quantify the long-term effect a compromise of dependability would have in such a system. However—and possibly because it is so difficult—many experts believe that tampering with electronically captured election results could have a political impact over time that is both critical and long-lasting.

The potential threat of political corruption through electronic voting system compromise is taken seriously enough for a number of standards bodies, government and private organizations, and academic institutions [335] to be working intensely on the means to develop highly reliable, secure software for electronic voting systems and the means to assure the reliability and security of that software with a high degree of confidence. As "high consequence" software, electronic voting software joins safety-critical and mission-critical national security software, both in terms of the criticality of its dependable functioning, and the need for an extremely high level of assurance in that dependability.

Of course, not all software is found in information systems. Software is also used in systems that do not process or control information but rather monitor, measure, and control physical processes, *e.g.,* the Supervisory Control and Data Acquisition (SCADA) systems within large distributed critical infrastructure systems (*e.g.,* chemical, physical transport, municipal water supply, electric power distribution and generation, gas and oil pipeline, nuclear power systems). [336]

Numerous SCADA security initiatives have been undertaken to address the vulnerable nature of SCADA systems. Valuable contributions have been made by all of the stakeholders in improving SCADA security: system owners, vendors, consultants, academic institutions, national laboratories, independent associations and bodies, and government organizations. Two most significant initiatives are: National SCADA Test Bed (NSTB) at Idaho National Laboratory and Sandia National Laboratory, and Control Systems Security Center (CSSC), which is managed by the Idaho National Laboratory.

The NSTB program is funded by Department of Energy, while the CSSC is funded by the Department of Homeland Security. Both programs use the same facilities and testbeds. The NSTB program is focused on reducing vulnerabilities of the electrical sector, while the CSSC program is concerned with all of the critical infrastructures in the United States.

Security of SCADA systems, including concern regarding the exploitability of vulnerabilities in their software components, has become a major focal point of several security organizations. In September 2006, the National Institute of Standards and Technology (NIST) published the initial public draft of Special Publication 800-82, *Guide to Supervisory Control and Data Acquisition (SCADA) and Industrial Control Systems Security.* [337] A year earlier, Idaho National Laboratory proposed a set of *Cyber Assessment Methods for SCADA Security* [338] that includes consideration of software vulnerabilities. Idaho National Laboratory is also a key participant in the New York State Office of Cyber Security and Critical Infrastructure Coordination's SCADA and Control Systems Procurement Project. [339]

In 2006, the latter project formed a working group (WG) with representatives of major utilities and control system software vendors to draft a set of guidelines and specific requirements for acquisition of SCADA systems. [340] *Cyber Security Procurement Language for Control Systems* is intended to elevate security to an explicit element of contractual negotiations between customers and suppliers of critical infrastructure systems software and hardware. The level of concern regarding SCADA software security has been elevated enough to inspire the emergence of a small industry of vendors who specialize in secure SCADA software. Spearheaded by RealFlex Technologies [341] and Hexatec, [342] more established software suppliers and tools vendors such as Symantec [343] are also taking on the SCADA security challenge.

At a smaller scale are embedded systems, in which software and firmware is used to control the physical operations and interactions of devices ranging from military, aerospace, and commercial vehicles to weapons systems to medical devices to communications devices (from satellites to cellular telephones), as well as a host of other physical devices, many of which have safety-critical aspects. As their name implies, "safety-critical" systems are those wherein people's lives and health depend directly on the system's (including its software's) reliability.

Firmware and embedded software are becoming increasingly present in commercial off-the-shelf (COTS) devices that previously had no software components. It is not uncommon for a network controller, such as a wireless Ethernet card, to host full-blown embedded systems that control their communications and user interface functionality. The increased use of firmware in COTS network devices presents a security risk. The firmware is rarely, if ever, patched, and wireless networking devices are particularly susceptible to remote attack. In response to this increased threat, a small segment of the open source software community is militating against the inclusion of binary-only firmware in the Linux kernel because it is empirically destabilizing to the operating system's reliability and security. [344]

In Chapter 7 of *Exploiting Software,* [345] Greg Hoglund and Gary McGraw effectively describe the threat to embedded systems:

> *For no valid technical reasons, people seem to believe that embedded systems are invulnerable to remote software-based attacks. One common misconception runs that because a device does not include an interactive shell out of the box, then accessing or using "shell code" is not possible. This is probably why some people (wrongly) explain that the worst thing that an attacker can do to most embedded systems is merely to crash the device. The problem with this line of reasoning is that injected code is, in fact, capable of executing any set of instructions, including an entire shell program that encompasses and packages up for convenient use standard, supporting [operating system]-level functions. It does not matter that such code does not ship with the device. Clearly, this kind of code can simply be placed into the target during an attack. Just for the record, an attack of this sort may not need to insert a complete interactive TCP/IP shell. Instead, the attack might simply wipe out a configuration file or alter a password.*
>
> *There are any number of complex programs that can be inserted via a remote attack on an embedded system. Shell code is only one of them. Even the most esoteric of equipment can be reverse engineered, debugged, and played with. It does not really matter what processor or addressing scheme is being used, because all an attacker needs to do is to craft operational code for the target hardware. Common embedded hardware is (for the most part) well documented, and such documents are widely available.*

One of the most widely publicized successful attacks on an embedded system was the 2002 hack of the flash memory of the Microsoft XBox game cube in order to access the algorithm used by the game cube's cryptosystem to decrypt and verify its bootloader. [346]

The vast majority of software-intensive embedded systems were conceived as non-networked, standalone systems, while most software-intensive control systems were, if networked at all, connected only to private dial-up links. However, an increasing number of such systems are being connected to and remotely administered and operated *via* Internet links or other public networks. Even embedded controllers in automobiles are being monitored *via* wireless network-based systems such as OnStar. It is widely believed that it is only a matter of time before the same network-based systems now limited to monitoring or information update and reporting functions will be used to reset embedded processors, reconfigure embedded software, and download new software and firmware versions. Embedded software in implanted medical devices is now accessible *via* radio frequency identification (RFID) interfaces, [347] while in telemedicine applications, software-controlled surgical robots are being controlled *via* satellite uplinks between in-theater medical facilities and US-based military hospitals. [348]

Network connectivity means that these software-intensive systems are being exposed to threats that the systems' designers never anticipated, and the impact of a compromise made possible by such new exposure is likely to be catastrophic. In a weapons system, for example, subversion of the software that performs the latitude/longitude calculations for targeting could render the system ineffective or even cause it to target friends rather than foes. Subversion or sabotage of a software-based temperature control in a nuclear power plant could result in a meltdown, while sabotage of avionic software in a fighter jet could result not only in the crash and loss of the jet, but the death of the pilot.

To date, the approach to preventing software's susceptibility to attack has started with the understanding of the various types of sources of threats to that software, how those threats manifest as attacks, and the nature and exposure of vulnerabilities and weaknesses that are typically targeted by those attacks. Methodologies have been developed to identify and model threats, attacks, and vulnerabilities, including Microsoft Threat Modeling, the European Community's CORAS, PTA Technologies' CTMM, the open source Trike methodology, USC's T-MAP, and SEI's OCTAVE, all of which are discussed in Section 5.2.3.1.

For Further Reading

"Threats to Voting Systems." NIST.
Available from: http://vote.nist.gov/threats

Aviel David Rubin, *Brave New Ballot: The Battle to Safeguard Democracy in the Age of Electronic Voting,* (Morgan Road Books, 2006).
Available from: http://www.bravenewballot.org

"Avi Rubin's E-voting Security page".
Available from: http://avirubin.com/vote

Cameron Barr, "Security of Electronic Voting is Condemned," *The Washington Post* (December 1, 2006).
Available from: http://www.washingtonpost.com/wp-dyn/content/article/2006/11/30/AR2006113001637.html

Bruce Schneier, "The Problem with Electronic Voting Machines," *Schneier on Security* (November 10, 2004).
Available from: http://www.schneier.com/blog/archives/2004/11/the_problem_wit.html

Dan S. Wallach, "Hack-a-Vote: Demonstrating Security Issues with Electronic Voting Systems," *IEEE Security & Privacy* 2, no.1 (January/February 2004): 32–37.
Available from: http://www.cs.rice.edu/%7Edwallach/pub/hackavote2004.pdf

Control Systems Cyber Security Awareness, (United States Computer Emergency Readiness Team. *Circa* July 7, 2005).
Available from: http://www.us-cert.gov/reading_room/Control_System_Security.pdf

Sandia National Laboratories Center for SCADA Security.
Available from: http://www.sandia.gov/scada/home.htm

United Kingdom (UK) National Infrastructure Security Coordination Centre (NISCC), "Vulnerabilities in Embedded Software," *The Quarterly*, 4 (April 2004).
Available from: http://www.niscc.gov.uk/niscc/docs/re-20041231-00959.pdf

Srivaths Ravi, Anand Raghunathan, Paul Cocher, and Sunil Hattangady, "Security in Embedded Systems: Design Challenges," *ACM Transactions on Embedded Computing Systems* 3, no.3 (August 2004): 461–491.
Available from: http://portal.acm.org/citation.cfm?id=1015049

Warren Webb, *Hack This: Secure Embedded Systems,* (EDN, July 22, 2004).
Available from: http://www.edn.com/article/CA434871.html

References

334 "Trustworthy Computing" [home page] (Redmond, WA: Microsoft Corporation).
Available from: http://www.microsoft.com/mscorp/twc/default.mspx

335 Examples include: "NIST Improving US Voting Systems Project" [web page] (Gaithersburg, MD: NIST).
Available from: http://vote.nist.gov *and*

"ACCURATE Project" [portal page] (Washington, DC: National Science Foundation CyberTrust Program).
Available from: http://accurate-voting.org *and*

"Voting System Standards" [web page] (San Francisco, CA: Verified Voting Foundation).
Available from: http://www.verifiedvotingfoundation.org/article.php?list=type&type=89 *and*

"Electronic Voting Machine Project" [portal page] (Granite Bay, CA: Open Voting Consortium).
Available from: http://evm2003.sourceforge.net/

336 Marc Maiffret (eEye Digital Security), "Cyber-terrorism: Is the Nation's Critical Infrastructure Adequately Protected?" (testimony to the US House Subcommittee on Government Efficiency, Financial Management, and Intergovernmental Relations Oversight, July 24, 2002).
Available from: http://research.eeye.com/html/Papers/download/Maiffret-Congress-Infrastructure.pdf *and*

Samuel G. Varnado (Sandia National Laboratories), "SCADA and the Terrorist Threat: Protecting the Nation's Critical Control Systems" (statement to the US House Committee on Homeland Security, Subcommittee on Economic Security, Infrastructure Protection, and Cyber Security and the Subcommittee on Emergency Preparedness, Science, and Technology, October 18, 2005).
Available from: http://www.sandia.gov/news/resources/testimony/pdf/051018.pdf *and*

Dana A. Shea. (Congressional Research Service), *Critical Infrastructure: Control Systems and the Terrorist Threat*, report for Congress, order code RL31534 (Washington, DC: Congressional Research Service, January 20, 2004).
Available from: http://www.fas.org/sgp/crs/homesec/RL31534.pdf

337 Keith Stouffer, Joe Falco, and Karen Kent (NIST). *Guide to Supervisory Control and Data Acquisition (SCADA) and Industry Control Systems Security*, initial public draft, special pub. 800-82 (Gaithersburg, MD: NIST Computer Security Division, September 2006)..
Available from: http://csrc.nist.gov/publications/drafts/800-82/Draft-SP800-82.pdf

338 May Robin Permann and Kenneth Rohde (Idaho National Laboratory). Cyber Assessment Methods for SCADA Security" (presentation at the 15th Annual Joint Instrumentation, Systems and Automation Society POWID/EPRI Controls and Instrumentation Conference, Nashville, TN, 2005 June 5–10, 2005).
Available from: http://www.oe.energy.gov/DocumentsandMedia/Cyber_Assessment_Methods_for_SCADA_Security_Mays_ISA_Paper.pdf

339 "SCADA and Control Systems Procurement Project" [web page] (New York: Multi-State Information Sharing and Analysis Center [MS-ISAC)]).
Available from: http://www.msisac.org/scada/

340 Robert Lemos, "SCADA System Makers Pushed Toward Security," *SecurityFocus* (July 26, 2006).
Available from: http://www.securityfocus.com/news/11402

341 "RealFlex Technologies Ltd." [web page]
Available from: http://www.real ex.com/

342 "Hexatec" [web page].
Available from: http://www.hexatec.co.uk/

343 Symantec Canada, "Symantec and AREVA T&D Partner to Deliver SCADA Security Solutions for Electric Power Industry," press release (Vancouver, BC, Canada: Symantec Canada, June 14, 2004).
Available from: http://www.symantec.com/region/can/eng/press/2004/n040614.html

344 For a discussion of this specific issue, see the following articles: Mike Barton (InfoWorld, San Francisco, CA: mike_barton@infoworld.com), "Researchers Hack Wi-Fi Driver to Breach Laptop," in Slashdot [Internet blog] [operated by the Open Source Technology Group, Inc., Fremont, CA]: June 22, 2006, 01:08 a.m.
Available from: http://it.slashdot.org/article.pl?sid=06/06/22/0312240 *and*

Timothy (timothy@monkey.org), "Less Than a Minute to Hijack a MacBook's Wireless," in Slashdot [Internet blog] [operated by the Open Source Technology Group, Inc., Fremont, CA]: August 3, 2003, 08:10 a.m.
Available from: http://it.slashdot.org/article.pl?sid=06/08/03/129234 *and*

Zonk (zonk@slashdot.org), "Wi-Fi Exploits Coming to Metasploit," in Slashdot [Internet blog] [operated by the Open Source Technology Group, Inc., Fremont, CA]: October 26, 2006, 05:23 p.m.
Available from: http://it.slashdot.org/article.pl?sid=06/10/26/2052223 and

Zonk (zonk@slashdot.org), "Code Execution Bug in Broadcom Wi-Fi Driver," in Slashdot [Internet blog] [operated by the Open Source Technology Group, Inc., Fremont, CA]: November 12, 2006, 07:10 a.m.
Available from: http://it.slashdot.org/article.pl?sid=06/11/12/0824250

345 Greg Hoglund and Gary McGraw, *Exploiting Software: How to Break Code* (Boston, MA: Addison-Wesley, 2004).
Available from: http://www.exploitingsoftware.com/

346 Andrew "Bunnie" Huang, "Keeping Secrets in Hardware: the Microsoft XBox Case Study," *Massachussetts Institute of Technology AI Memo*, no. 2002-008 (May 2002).
Available from: http://web.mit.edu/bunnie/www/proj/anatak/AIM-2002-008.pdf

347 T.J. Becker, "Improving Medical Devices: Georgia Tech Research Center Expands Testing Capabilities to Help Reduce Potential Interference," *Georgia Tech Research News* (July 25, 2006).
Available from: http://www.gtresearchnews.gatech.edu/newsrelease/eas-center.htm

348 Robert K. Ackerman, "Telemedicine Reaches Far and Wide," *SIGNAL Magazine* (March 2005).
Available from: http://www.afcea.org/signal/articles/anmviewer.asp?a=693&print=yes *and*

H., Shimizu Murakami, *et al.* (Tohoku University Sendai), "Telemedicine Using Mobile Satellite Communication," *IEEE Transactions on Biomedical Engineering* 41, no. 5 (May 1994): 488–497.

D

DoD/FAA Proposed Safety and Security Extensions to ICMM and CMMI

The Safety and Security Extension Project Team jointly established by the Federal Aviation Administration (FAA) and the Department of Defense (DoD) produced the draft report *Safety and Security Extensions to Integrated Capability Maturity Models* [349] in September 2004. The report defined a Safety and Security (S&S) Application Area (AA) to be used in combination with either the FAA's Integrated Capability Maturity Model (iCMM) or the Software Engineering Institute's (SEI) Capability Maturity Model Integration (CMMI) to achieve process improvements that would improve the safety and security of software produced by software development life cycle (SDLC) processes guided by the iCMM or CMMI.

The new Safety and Security (S&S) Application Area (AA) extension implements practices within the relevant iCMM and CMMI Process Areas (PA) of the iCMM and CMMI. The additional practices were derived from existing US DoD and UK MOD, National Institute of Standards and Technology (NIST), and ISO/IEC security and safety standards, including ISO/IEC 21827, *SSE-CMM,* and ISO/IEC 17799, *Code of Practices for Information Security Management.*

In the case of CMMI, the methodology for mapping the safety/security practices to the appropriate PAs sometimes required the insertion of an iCMM PA into the CMMI when no comparable CMMI PA existed or the existing CMMI PA was inadequate in achieving the desired safety or security practice.

Table D-1 illustrates the mapping of iCMM and CMMI PAs to the new S&S AA practices.

Table D-1. Safety and Security Extensions to iCMM/CMMI

ICMM PA	CMMI PA	S&S AA Pratice
PA 22: Training	Organizational Training	AP01.01: Ensure S&S Competency
PA 19: Work Environment	Work Environment	AP01.01: Ensure S&S Competency AP01.02: Establish Qualified Work Environment AP01.05: Ensure Business Continuity AP01.11: Objectively Evaluate Products
PA 17: Information Management	(add iCMM PA 17 to CMMI)	AP01.03: Control Information AP01.12: Establish S&S Assurance Argument
PA 10: Operation and Support	(add iCMM PA 10 to CMMI)	AP01.04: Monitor Operations and Report Incidents AP01.10: Develop and Deploy Safe and Secure Products and Services
PA 13: Risk Management	Risk Management	AP01.05: Ensure Business Continuity AP01.06: Identify S&S Risks AP01.07: Analyze and Prioritize Risks AP01.08: Determine, Implement, and Monitor Risk Mitigation Plan AP01.14: Establish a S&S Plan
PA 00: Integrated Enterprise Management	Organizational Environment for Integration Organizational Innovation and Deployment (add iCMM PA 00)	AP01.05: Ensure Business Continuity AP01.09: Identify Regulatory Requirements, Laws, and Standards AP01.13: Establish Independent S&S Reporting AP01.14: Establish an S&S Plan AP01.16: Monitor and Control Activities and Products
PA 01: Needs PA 02: Requirements	Requirements Development Requirements Management	AP01.09: Identify Regulatory Requirements, Laws, and Standards AP01.10: Develop and Deploy Safe and Secure Products and Services
PA 03: Design PA 06: Design Implementation	Technical Solution	AP01.10: Develop and Deploy Safe and Secure Products and Services
PA 08: Evaluation	Verification Validation	AP01.11: Objectively Evaluate Products AP01.12: Establish S&S Assurance Argument

Table D-1. Safety and Security Extensions to iCMM/CMMI - *continued*

ICMM PA	CMMI PA	S&S AA Pratice
PA 15: Quality Assurance and Management	Process and Product Quality Assurance	AP01.12: Establish S&S Assurance Argument AP01.13: Establish Independent S&S Reporting AP01.16: Monitor and Control Activities and Products
PA 11: Project Management	Project Planning Project Monitoring and Control Integrated Project Management Quantitative Project Management	AP01.01: Ensure S&S Competency AP01.13: Establish Independent S&S Reporting AP01.14: Establish a S&S Plan AP01.16: Monitor and Control Activities and Products
PA 16: Configuration Management	Configuration Management	AP01.16: Monitor and Control Activities and Products
PA 18: Measurement and Analysis	Measurement and Analysis	AP01.16: Monitor and Control Activities and Products
PA 05: Outsourcing PA 12: Supplier Agreement Management PA 09: Deployment, Transition, and Disposal	Supplier Agreement Management Integrated Supplier Management	AP01.15: Select and Manage Suppliers, Products, and Services AP01.10: Develop and Deploy Safe and Secure Products and Services
PA 21: Process Improvement	Organizational Process Focus	AP01.16: Monitor and Control Activities and Products

The FAA report goes on to provide extensive information on the activities, typical work products associated with each AA, and recommended practices for achieving the objectives of each AA as it is integrated with the iCMM or CMMI process of the organization. Sixteen of the activities in the FAA's Safety and Security Extensions were also integrated into the proposed revision of ISO/IEC 15026 (see Section 5.1.4.2.2).

For Further Reading

Linda Ibrahim, (FAA). "Sixteen Standards-Based Practices for Safety and Security," *CrossTalk: The Journal of Defense Software Engineering* (October, 2005).
Available from: http://www.stsc.hill.af.mil/CrossTalk/2005/10/0510Ibrahim.html

References

349 Linda Ibrahim, *et al.*, *Safety and Security Extensions for Integrated Capability Maturity Models* (Washington, DC: Federal Aviation Administration (FAA), September 2004). Available from: http://www.faa.gov/about/office_org/headquarters_offices/aio/documents/media/SafetyandSecurityExt-FINAL-web.pdf

E
Security Functionality

This does not purport to be a comprehensive list of system security functions. Instead, it identifies and describes security functions that are implemented in the majority of software-intensive information systems.

E.1 Security Protocols and Cryptographic Mechanisms

The design of secure systems typically involves the design, selection, and use of cryptographic mechanisms and security protocols for protecting the confidentiality and integrity of information (*e.g.,* application data, user authentication credentials) stored by the system or transmitted between the system and its users or other entities. Most security protocols include cryptographic elements, and as a result, need to be designed to manage cryptographic keys. Even in the many secure system designs that still use passwords as the user authentication credential, encryption of passwords both at rest and in transit is critical to the integrity of the authentication process. Authentication devices, such as tokens and smart cards (typically used to store user credentials such as digital certificates or biometric templates), are also becoming increasingly typical in secure systems. A common cause of security protocol failure is a changing environment. Environment changes often result in the invalidation of at least some of the system's underlying assumptions about that environment; such changes can make it impossible for a security protocol to cope with new threats.

To verify the correctness of protocols and implementations of cryptographic algorithms, systems engineers and researchers may apply formal methods. An example is BAN (Burrows-Abadi-Needham) logic, [350] which provides a formal method for reasoning about the logic of belief of principals in cryptographic protocols. Formal methods can also be used to find bugs in security protocol designs. Formal methods are discussed in Sections 5.1.2, 5.2.3.2.5, and 5.3.4.1. Systems engineers also typically rely on the National Institute of Standards and Technology (NIST) Federal Information Processing Standard (FIPS) 140-1 and FIPS 140-2 Cryptographic Module Validation Lists, [351] which list all commercial cryptographic modules that have successfully been validated as conforming to either FIPS 140-1 or FIPS 140-2. NIST also publishes validation lists for cryptographic algorithms. [352]

E.2 Authentication, Identity Management, and Trust Management

Trust management incorporates several security functions, including identity management, user authentication, authorization, and access control. Trust management specifically enables the sharing of these functions among different domains (*e.g.,* domains belonging to different business or trading partners). Trust management is effectively a sophisticated version of single sign-on (SSO) that comprises several functional security components:

- **Identity Management**—the means of managing user identity and account information that will be used as the basis for identifying, authenticating, and authorizing the user's access to a system or resource. Information managed by the identity management component of a trust management system includes the user's personal information, encrypted passwords, biometric data, financial information, security clearance, organizational roles, history of access/usage, *etc.*
- **Authentication**—Each user's identity must be authenticated before the user is allowed to access a system or resource. This authentication is based on the presentation by the user of a unique identifier plus one or more credentials, such as a password, a digital certificate, or a biometric, to the authentication component of the trust management system. In a federated trust management system, each organization in a partnership is responsible for authenticating its own users, while trusting the other partners to do the same. In most cases, the different organizations' systems convey evidence of each authentication [*e.g.,* in a Security Assertion Markup Language (SAML) assertion] to the other systems.
- **Authorization**—Authorization to access specific systems, applications, services, or resources is granted according to a security policy, which is represented as a set of rules that can be dynamically interpreted by the authorization component of the trust management system. This enables the trust management system to decide whether an authenticated user is authorized to access a restricted service or

protected resource. The policy for authorization may be based on the matching of required access privileges with the privileges associated with the user's individual identity (as in discretionary access control), the user's role (as in role-based access control), the user's clearance (as in mandatory access control), the perceived level of risk the access poses to the system (as in risk-adaptive access control), or some other attribute or combination of attributes (as in attribute-based access control). See Section E.3 for a discussion of access control.

Systems that rely on external trust management systems (or only on trust management subsystems such as identity management systems or SSO authentication systems) may require certain capabilities and interfaces, such as ability to interface with a public key infrastructure, ability to parse SAML assertions, *etc.* If passwords are used as authentication credentials, the system may need to incorporate its own password management capability. Password management is a difficult design problem for the systems engineer in developing secure systems. Physical control devices, passwords, and biometrics are the typical authentication methods. Devising a password protection scheme typically involves understanding the psychology of users and of the attackers, and judging whether a password scheme is sound by considering the type of attacks it must defend against. Consideration also needs to given of whether attackers will target particular accounts or target any account possible. Technical protection issues, such as whether passwords can be snooped by malicious software or network eavesdropping, also need to be considered.

E.3 Access Control

Systems engineers are responsible for defining and designing access control methods for systems. According to Ross Anderson in *Security Engineering*, access control's "function is to control which principals (persons, processes, machines)…have access to which resources in the system…"

Access controls are required at multiple levels in a system:

1. **Application—**Access control mechanisms may be provided at this level to implement complex security policies involving third-party trust establishment, user authorization, *etc.*
2. **Middleware**—Application frameworks (such as .NET and Java Enterprise Edition), database management systems, web servers, and public key infrastructures are three types of middleware that provide access control mechanisms. [353]
3. **Operating System—** The operating system, from a security perspective, provides mechanisms for—
   - File system control of access to data and resources (may enforce one or more access control models, including discretionary, mandatory, role-based, attribute-based, risk-adaptive)

- Cryptographic and/or steganographic data confidentiality and integrity protection to augment access control.

4. **Hardware—**Computer chip architectures are designed to control access to memory addresses, thus preventing one process from interfering with another process or overwriting the second process' data. Certain trusted operating systems tightly link their access controls with the underlying hardware architecture. [354] Another form of hardware-based access control is the Trusted Platform Module (TPM). A TPM is a hardware component that supports process isolation (similar in intent to the process isolation provided by virtual machines, but with stronger enforcement because of the reliance on hardware). TPM process isolation enables trusted software processes, their resources, and data to be physically isolated from untrusted processes and their resources and data. The result is that execution of the untrusted processes, with any coincidental insecure behaviors, cannot affect the execution or environment of the trusted processes. The most widely used TPMs are those that conform to Trusted Computing Group standards.

References

350 Michael Burrows, Martin Abadi, and Roger Needham (Digital Equipment Corp.), *A Logic of Authentication*, DEC SRC research rpt. No. 39 (February 28, 1989).
Available from: http://ftp.digital.com/pub/DEC/SRC/research-reports/abstracts/src-rr-039.html) *and*
Kyntaja, Timo (Helsinki University of Technology), A Logic of authentication by Burrows, Abadi and Needham. 1995.
Available from: http://www.tml.tkk.fi/Opinnot/Tik-110.501/1995/ban.html

351 NIST, "*Validation Lists for Cryptographic Modules*" [web page] (Gaithersbur, MD: NIST Computer Security Division).
Available from: http://csrc.nist.gov/cryptval/140-1/1401val.htm

352 NIST, Validation Lists for Cryptographic Algorithm Standards (Gaithersburg, MD: NIST Computer Security Division).
Available from: http://csrc.nist.gov/cryptval/140-1/avallists.htm

353 Virtualization is emerging as a capability at other layers as well. Long provided in the IBM VM operating system ("VM" is, in fact, an acronym for "virtual machine"), it is now provided as a standard feature in the latest version of Linux. Unix "chroot jails" and Solaris containers may also be referredn as operating system-level VM mechanisms. Virtualization is also provided at the hardware level as a standard feature of the latest 64-bit computer chips.

354 The BAE/DigitalNet XTS-400's STOP operating system and Aesec's GEMSOS operating system provide discrete software "rings" that are isolated by the kernel's mandatory access control policy enforcement; each operating system ring maps one-for-one (in the case of STOP) or two-for-one (in the case of GEMSOS) into a hardware ring of the underlying Intel chip architecture.

F

Agile Methods: Issues for Secure Software Development

The following are some further considerations with regard to use of agile methods to produce secure software.

F.1 Mismatches Between Agile Methods and Secure Software Practices

With one exception, the agile methods listed in Table F-1 reflect their creators' formal commitment to support the core principles of the Agile Manifesto. [355]

Table F-1. Agile Methods

Method	Acronym	Creator/Affiliation
Agile Software Process	ASP	Mikio Aoyama/Nanzan University and Fujitsu (Japan)
eXtreme Programming	XP	Kent Beck, Ward Cunningham/Tektronix; Ron Jeffries/Object Mentor and XProgramming.com
Crystal Family of Methods	None	Alistair Cockburn/IBM
Adaptive Software Development	ASD	Jim Highsmith, Sam Bayer/Cutter Consortium
Scrum	None	Ken Schwaber/Advanced Development Methods; Jeff Sutherland/PatientKeeper

Table F-1. Agile Methods - *continued*

Method	Acronym	Creator/Affiliation
Feature-Driven Development	FDD	Jeff De Luca/Nebulon
Dynamic System Development Method	DSDM	DSDM Consortium (UK)
Lean Development *	LD	Bob Charette/ITABHI Corp. *
Whitewater Interactive System Development with Object Models	Wisdom	Nuno Jardim Nunes/Universidade da Madeira; João Falcão e Cunha/Universidade dão Porto

* not committed to the Agile Manifesto

The Agile Manifesto's core principles are listed in Table F-2, which originally appeared in DHS' *Security in the Software Life Cycle,* lists the security implications of each core principle. Those principles not listed are completely neutral regarding security. This table is copied from *Security in the Software Lifecycle.*

Table F-2. Core Principles of the Agile Manifesto that Have Security Implications

No.	Principle	Implication for Security
1	The highest priority of agile developers is to satisfy the customer. This is to be achieved through early and continuous delivery of valuable software.	Negative, unless customer is highly security-aware. There is a particular risk that security testing will be inadequate or excluded because of "early delivery" imperatives.
2	Agile developers welcome changing requirements, even late in the development process. Indeed, agile processes are designed to leverage change to the customer's competitive advantage.	Negative, unless customer is careful to assess the security impact of all new/changing requirements and include related requirements for new risk mitigations when necessary.
3	Agile projects produce frequent working software deliveries. Ideally, there will be a new delivery every few weeks or, at most, every few months. Preference is given to the shortest delivery timescale possible.	Negative, unless customer refuses to allow schedule imperatives to take precedence over security.
4	The project will be built around the commitment and participation of motivated individual contributors.	Neutral. Could be Negative when the individual contributors are either unaware of or resistant to security priorities.
5	Customers, managers, and developers must collaborate daily, throughout the development project.	Neutral. Could be Positive when all participants include security stakeholders (*e.g.,* risk managers) and have security as a key objective.

Table F-2. Core Principles of the Agile Manifesto that Have Security Implications - *continued*

No.	Principle	Implication for Security
6	Agile developers must have the development environment and support they need.	Neutral. Could be Positive when that environment is expressly intended to enhance security.
7	Developers will be trusted by both management and customers to get the job done.	Negative, unless developers are strongly committed and prepared to ensure security is incorporated into their process and products.
8	The most efficient and effective method of conveying information to and within a development team is through face-to-face communication.	Negative, as the assurance process for software is predicated on documented evidence that can be independently assessed by experts outside of the software project team.
9	The production of working software is the primary measure of success.	Negative, unless "working software" is defined to mean "software that always functions correctly *and* securely."
12	Agility is enhanced by continuous attention to technical excellence and good design.	Positive, especially when "technical excellence and good design" reflect strong expertise in and commitment to software security.
13	Simplicity, which is defined as the art of maximizing the amount of work not done, is essential to successful projects and good software.	Positive, if simplicity is extended to the design and code of the software. Simplicity of design and code will make them easier to analyze and their security implications and issues easier to recognize.

In addition to those listed above, other significant mismatches have been identified by various writers and panelists. (Several of these authors and panelists produced resources listed under "For Further Reading" at the end of Section 5.1.8.1.) These mismatches include—

- **Project Management Mismatches**
  - Agile development does not allow for specialization of development team members, which precludes inclusion of security experts on the development team.
  - Planning for software security practices and reviews is not easily accommodated in agile project planning processes.
  - Implicit trust in developers results in all members of the agile team having access to all software artifacts, including those not developed by them. This runs counter to the imperatives of separation of roles and separation of duties, and the ability to perform secure configuration management. It also implies a need for security background checks and a comparable level of clearance and need-to-know for all developers, thus potentially increasing the costs associated with staffing.

- Pair Programming, a core practice in eXtreme Programming and other agile methods, which entails one developer continually reviewing a second developer's code as it is being written (a kind of "on the fly" code review), is not possible in organizations in which workstation sharing is not permitted.

- **Requirements Engineering Mismatches**
  - Agile methods lack the ability to specify security and assurance requirements.
  - Acceptance of changing requirements, even late in the life cycle, necessitates constant impact analyses, and runs counter to the need to establish an unchanging security baseline for purposes of Certification and Accreditation (C&A) or Common Criteria (CC) evaluation.
  - Agile requirements modeling and capture methods do not accommodate capture of nonfunctional requirements.
  - Agile methods do not include, or allow time for, threat modeling.
- **Design and Implementation Mismatches**
  - Developers may not understand the security impact of design decisions made "on the fly."
- **Testing Mismatches**
  - "Working" software is the primary measure of test success. Agile testing does not extend to penetration testing or other nonfunctional security tests, and it does not include key activities of software security testing.
  - Agile methods do not include, or allow time for, threat modeling or security-focused code reviews.
  - Test case definition and test execution in the context of test-driven development (TDD) do not accommodate software security (*vs.* functional security) tests.
  - The agile imperative of early, frequent, and continuous software deliveries does not realistically allow for independent, third-party security assessments because these would have to be planned and budgeted iteratively and repeatedly throughout the agile life cycle. Options for security independent verification and validation (IV&V) do not exist in agile methods.
  - Daily collaboration between developers and customers results in the conflicting need for independence of security reviewers and testers.

- **Other Mismatches**
  - Preference for face-to-face communication to the exclusion of written communication clashes with the requirements of C&A and CC evaluations for extensive software documentation. It also makes IV&V impractical—independent testers rely on written documentation to become familiar with the system they are to test, because they must avoid direct contact with the system's developers in order to maintain their objectivity and independence.
  - Agility is a philosophy, not just a methodology. Agile developers may not understand the importance of software security. Attaining this comprehension may be difficult for them, because security compromises core principles of the Agile Manifesto.

F.2 Suggested Approaches to Using Agile Methods for Secure Software Development

The state-of-the-art in using agile methods to develop secure software revolves around extending existing agile methods to accommodate security practices, and adapting security practices so they more easily "fit" into existing agile methods.

For example, TDD (a.k.a. *continuous testing*) is a cornerstone of all agile methods. TDD requires every specified requirement to be verified through a test case before coding of the implementation of that requirement can begin. TDD is automated to the greatest extent possible to make it easier to run the continuous, iterative series of test cases against code as it is developed. In *Towards Agile Security in Web Applications,* [356] Vidar Kongsli noted that the benefits of automatic testing created a high degree of acceptance, but that manual testing was still required for verification of some requirements.

Equivalent security-benefiting practices are not found in all agile methods. To date, the only way that proponents of security-enhancing agile methods appear to have determined that their particular agile development method of choice could be effectively adapted to achieve software security objectives was through trial and error, i.e., they adapted the agile method by adding security training, modeling, analyses, reviews, and testing to various phases of the SDLC as it is defined by the method. A question of importance to agile advocates is whether such a security-enhanced methodology, if it must allow for too many extra activities, can still be considered "agile."

At this point, most of the suggested additions to agile methods are talking points, which have not been proven through practical application in agile development projects. Some of these suggested additions include those identified in Table F-3 (with attribution for each recommendation).

Table F-3. Recommended Security Extensions to Agile Methods

Addition to Agile Method	Source of Recommendation
Increasing security awareness and ownership of security issues by the development team.	Kongsli [357]
Adding a security engineer role to the development team to inform and educate the other team members.	Wäyrynen *et al.* [358]
Developing security-related (and customer-specified) user or misuse stories (XP). Modeled misuse stories could be related to the user story. If the misuse story can be described as an acceptance test, the acceptance test is the formal security requirement that needs to be implemented.	Wäyrynen *et al.,* Kongsli
Integrating the security solution with the customer's environment early in the project.	Beznosov [359]
Performing security reviews through pair programming (XP).	Wäyrynen *et al.*
Implementing of code scanning tools and security testing tools, allowing testing to define what the "good enough" security solution is and help to gain confidence in its quality as well as functionality.	Cornell, [360] Beznosov
Adhering to common coding standards and security guidelines.	Cornell
When closing an iteration, running automated customer acceptance tests but making sure to include negative testing for identified threats.	Cornell

Recent research efforts combine agile approaches with security engineering in hopes of producing a kind of secure agile engineering method. It has been suggested that "agile security engineering" can be achieved by introducing agile software engineering values to the traditional practice of mitigating security risks in software. This approach is suggested by Tappenden *et al.* [361] who assert that by applying security elements to agile development life cycle phases (perhaps in parallel and without all the steps necessarily being included), secure software can be developed in an agile manner.

It has also been suggested that security-enhancement may be easier for feature-driven development (FDD), a methodology that follows the general principles of agile development but which also provides scalability and planning support that agile methods do not. Table F-4 summarizes possible security enhancements to both agile methods and FDD.

Table F-4. Summary of Security Enhancements to Agile Methods and FDD

Life Cycle Phase	Enhancement to Agile Methods	Enhancements to FDD
Requirements Analysis	▶ Identify and list key, security-sensitive assets of the organization. ▶ Make listed items candidate security objects denoted in the appropriate agile method notation (*e.g.,* security-enriched use cases). ▶ Identify and list organizational authorized users. ▶ Make listed persons candidate security subjects and actors in the appropriate agile method notation. ▶ Establish rank order for each organization the security objects from most sensitive (Top Secret) to least sensitive (Unclassified). ▶ Analyze which security actors have access to which security objects by applying use case notation. Check for completeness. ▶ Analyze potential threats using abuse cases. ▶ Estimate cost of recovery from an attack against listed security objects within the abuse case scenario. ▶ Perform risk analysis of abuse cases and cost of recovery data.	▶ Capture requirements with use cases that identify security subjects and security objects ▶ Identify and document abuse cases ▶ Develop an overall model of the system ▶ Construct candidate classes from use cases ▶ Derive security levels of classes from use cases and incorporate into classes ▶ Build a feature list ▶ Specify abuse case scenarios ▶ Specify countermeasures to prevent abuse cases ▶ Relate use cases to abuse cases to features ▶ Classify features into feature sets based on activity
Design	▶ Using selected agile design diagrams, include security subjects (actors) and security objects. ▶ Include the security classification of actors and objects in the modeling notation. ▶ Using risk management, prioritize features.	▶ Plan by feature ▶ Define order of features to be developed and tested ▶ Prioritize security countermeasures (features) ▶ Design by feature ▶ Incorporate security elements and security classification into objects as attributes ▶ Sketch sequence diagram of each security feature

Table F-4. Summary of Security Enhancements to Agile Methods and FDD - *continued*

Life Cycle Phase	Enhancement to Agile Methods	Enhancements to FDD
Implementation	▶ Implement countermeasures and design according to sensitivity. ▶ Prioritize functions to be implemented. ▶ Unit test the highest priority functions first.	▶ Build by feature ▶ Implement in security countermeasure feature priority order, adding the most important security measures first
Testing	▶ Examine use cases and abuse cases to develop test strategy. ▶ Prioritize the test list based on risk analysis. ▶ Test highest priority items first.	▶ Test abuse case scenarios that are most sensitive first ▶ Test based on security countermeasure feature priority list

References

355 Kent Beck, *et al., Manifesto for Agile Software Development* (2001). Available from: http://agilemanifesto.org/

356 Vidar Kongsli, "Towards Agile Security in Web Applications," in *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Portland, OR, October 22-26, 2006.

357 *Ibid.*

358 J. Wäyrynen, M. Bodén, and G. Boström, "Security Engineering and eXtreme Programming: an Impossible Marriage?" in *Extreme Programming and Agile Methods—XP/Agile Universe 2004*, C. Zannier, H. Erdogmus, and L. Lindstrom, eds. (Berlin, Germany: Springer-Verlag, 2004): 117–128.

359 Beznosov Konstantin, "Extreme Security Engineering: on Employing XP Practices to Achieve 'Good Enough Security' Without Defining It" (presentation at the First ACM Workshop on Business Driven Security Engineering, Washington, DC, October 31, 2003). Available from: http://konstantin.beznosov.net/professional/doc/papers/eXtreme_Security_Engineering-BizSec-paper.pdf

360 Michelle Davidson, "Secure Agile Software Development an Oxymoron?," SearchAppSecurity.com (October 25, 2006). Available from: http://searchappsecurity.techtarget.com/originalContent/0,289142,sid92_gci1226109,00.html

361 A. Tappenden, P. Beatty, and J. Miller (University of Alberta), and A. Geras and M. Smith (University of Calgary), "Agile Security Testing of Web-based Systems Via HTTPUnit," in *Proceedings of the AGILE 2005 Conference*, Denver, CO, July 24–29, 2005.

G

Comparison of Security Enhanced SDLC Methodologies

Tables G-1 and G-2 illustrate how the main security enhancements of the most mature of the software development life cycle (SDLC) methodologies described in Sections 5.1.8.2.1 through 5.1.8.2.5 map into the activities of a standard SDLC. In this case, the life cycle model used is that illustrated by the lower right quadrant of Barry Boehm's spiral development model, with some minor modifications for purposes of clarity. The life cycle phases in this table do not include a Disposal (or Retirement or Decommissioning) phase because one or the other of the following is true—

- None is included in the Boehm spiral model.
- None of the five security-enhanced methodologies defines specific activities for such a phase.

Legend for Tables

[1] Corresponds with the project management (*vs.* technical) planning activities in the lower left quadrant of the Spiral Model

[2] Maps to Organizational Life Cycle Processes or Supporting Processes in ISO/IEC 12207 lifecycle model

[3] Includes code review and unit testing, explicitly or implicitly, in most models

[4] Includes integration testing, explicitly or implicitly, in some models

[5] Refers to whole-system testing in some models, integration testing in others

[6] Falls within the Implementation phase of Boehm's Spiral Model

Table G-1. Management and Organizational Activities

	Microsoft SDL	Oracle Secure Software Assurance	CLASP	McGraw's 7 Touchpoints	TSP-Secure
Project Planning [1]	Hold security kickoff Register with security team	Ensure developer security awareness	Institute security awareness program		Define operational procedures for secure software production Establish organizational policies, management oversight, resources, training, project plan procedures and mechanisms, and tracking
Multiphase Activities [2]			Monitor security metrics Manage certification process Manage System Security Authorization Agreement (SSAA)	Conduct risk tracking, monitoring, and analysis	Use security-related predictive measures (may include predictive process metrics, checkpoints) Use operational procedures for secure software production, including project risk management, measurement, and feedback

Table G-2. Development Activities

	Microsoft SDL	Oracle Secure Software Assurance	CLASP	McGraw's 7 Touchpoints	TSP-Secure
Needs and Requirements	Identify assets Develop "asset compromise cases" (comparable to misuse/ abuse cases) Identify security requirements Conduct risk analysis of requirements	Ensure developer security awareness	Specify operational environment Identify global security policy Identify user roles, requirements Detail misuse cases Perform security analysis of requirements	Build abuse cases Develop security requirements specification	Develop security specifications Identify asset Develop use cases Develop abuse cases
Architecture/ Product Design	Apply security design best practices Develop security architecture Review attack surface (Conduct threat modeling)		Document security design assumptions Specify resource-based security properties (Address repeated security issues)	Conduct risk analysis of architecture	Use secure design process Conduct threat modeling

Table G-2. Development Activities - *continued*

	Microsoft SDL	Oracle Secure Software Assurance	CLASP	McGraw's 7 Touchpoints	TSP-Secure
Detailed Design	Use security design best practices Conduct threat modeling		Apply security principles to design Research and assess security solutions Build information labeling schemes Design user interfaces (UI) for security functions Specify database security configuration Conduct security analysis of system design Implement and/or elaborate on resource policies Address repeated security issues	Conduct risk analysis of design	Use secure design process (may include conformance to security design principles, design patterns to avoid common vulnerabilities) Review design security

Table G-2. Development Activities - *continued*

	Microsoft SDL	Oracle Secure Software Assurance	CLASP	McGraw's 7 Touchpoints	TSP-Secure
Coding [3]	Use security development tools Use security best practices for development and unit testing	Use secure coding standards Use standard libraries of security functions	(Implement and/or elaborate on resource policies) (Implement interface contracts) Address repeated security issues Review source-level security Conduct software security fault injection testing	Conduct static analysis and code review Conduct risk-based unit testing	Use secure implementation process (may include secure programming, and use of secure language subsets, coding standards, and quality management practices for secure programming) Review code using static and dynamic analysis tools
Integration [4]	Use security development tools Use security best practices for development and integration testing Create security documentation and tools for product	Develop secure configuration guidelines	(Specify database security configuration) Integrate security analysis into build process Implement and/or elaborate resource policies Implement interface contracts (Conduct security functionality usability testing) (Review source-level security) (Conduct software security fault injection testing) Build operational security guide	Conduct risk-based integration testing	Use secure implementation process (may include removing vulnerabilities from legacy software)

Table G-2. Development Activities - *continued*

	Microsoft SDL	Oracle Secure Software Assurance	CLASP	McGraw's 7 Touchpoints	TSP-Secure
Testing [5]	Use security testing tools Follow security best practices for testing Prepare security response plan Perform "Security Push" [362] Conduct penetration testing	Conduct penetration testing Conduct vulnerability scanning Validate against security checklists Conduct third-party security IV&V	Conduct security functionality usability testing (Review source-level security) (Conduct software security fault injection testing) Identify and implement security tests Verify security attributes of resources	Conduct risk-based security testing Conduct penetration testing Conduct external analysis	Use secure review and inspection process Use secure test process (may include security test plans, white box and black box testing, test defect review and/or vulnerability analysis by defect type, and security verification techniques)
Distribution/ Deployment [6]	Conduct final security review		Use code signing		
Maintenance/ Support [6]	Security servicing Response execution	Vulnerability management Critical patch deliveries Apply security fixes to main code base	Manage security issue disclosure process	Solicit feedback from security operations	(May include removing vulnerabilities from legacy software)

References

362 As described by Michael Howard in his article "A Look Inside the Security Development Lifecycle at Microsoft," *MSDN Magazine* (November 2005), a "Security Push" is "a team-wide focus on threat model updates, code review, testing, and documentation scrub.... [I]t is a concerted effort to confirm the validity of the information in the Security Architecture documentation, to uncover changes that may have occurred during the development process, and to identify and remediate any remaining security vulnerabilities…[T]he push duration is ultimately determined by the amount of code that needs to be reviewed for security."

H

Software Security Research in Academia

The listing in Table H-1 provides some indication of the extent and variety of current academic software security and application security research being pursued worldwide.

The methodology for gathering this information was as follows—

1. Compiled an extensive set of software security assurance keywords, keyphrases, and keyword/keyphrase combinations (*e.g.,* malware; "software engineering;" software + vulnerabilities + "static analysis") that was used to search for academic research papers and articles in the online document archives of the Association for Computing Machinery (ACM), Institute for Electrical and Electronics Engineering (IEEE), and CiteSeer.

2. Quickly scanned the papers and articles identified by the search and reviewed in more depth those papers that seemed most relevant to determine—
   - The institutional affiliations of the authors
   - The name of the research project being documented; if found, at least the subject of the research was noted.
   - Researchers/projects referenced in the bibliographies of those papers and articles were also noted.

3. Visited the websites of the authors' universities identified in step 1 (as well as those identified from the bibliographies). As necessary, navigated the university site to find the appropriate schools (*e.g.,* Engineering), and within those schools the appropriate departments (*e.g.,* Computer Science), research laboratories or pages, and so on until the pages of individual research projects, and in some cases individual researchers were found.

4. Reviewed all software security relevant research projects found, and also followed links from those pages to other research teams, laboratories, and projects in the same schools and at different schools.

5. Refined our set of software security assurance keywords and performed further searches to locate additional research teams/projects—
   - Google searches of .edu sites, using keywords and limiting filetypes to .ps, .pdf, .doc, and .ppt files—the file types most likely to be associated with research papers and presentations.
   - Keyword searches at the websites of a number of research organizations, including Defense Advanced Research Projects Agency (DARPA), Air Force Rome Laboratories (AFRL), Naval Research Laboratory (NRL), the National Academies of Science, the National Research Councils of United States and Canada, INRIA (Institut National de Recherche en Informatique et en Automatique), *etc.*

6. We then quickly scanned our search results to identify those documents and web pages that merited further investigation. In all documents, we noted the data points listed in step 2.

7. We contacted a number of academic researchers and asked for additional "leads," and followed those leads in the same way.

Given the unscientific nature of this methodology, the listing of research groups, laboratories, and projects in Table H-1 does not purport to be comprehensive. As indicated at the beginning of this appendix, the data provided here is merely representative, and is provided only for illustrative purposes.

Table H-1 Software Security Research in Academia

School/Department or Center/Group or Lab	Software Security Group, Lab, or Project	URL
United States of America		
Auburn State University/Samuel Ginn College of Engineering/Center for Innovations in Mobile, Pervasive, and Agile Computing Technologies/ Information Assurance Laboratory	1. Software Vulnerability Assurance: Software Architecture Analysis, Decompiling and Disassembly, Tamper-proofing, Open Source Attack Databases 2. Software Process for Secure Software Development 3. AI for Vulnerability Assessment: Genetic Algorithms for Parameter Analysis 4. findssv: static analysis of executables *[with Western Illinois University]*	http://www.eng.auburn.edu/users/hamilton/security/Information_Assurance_Laboratory_Research_Areas_Dec_2003.html
Ball State University/Software Engineering Research Center	Measuring the Effect of Software Design on Software Security	http://www.serc.net/web/research/index.asp
California State University at East Bay/Department of Math and Computer Science	1. Novel code obfuscation algorithms 2. JHide: Code Obfuscation Toolkit	1. http://www.mcs.csuhayward.edu/~lertaul/SER3012.pdf 2. http://www.mcs.csuhayward.edu/~lertaul/JHide
Carnegie Mellon University/CyLab and Software Engineering Institute (SEI)/SEI Computer Emergency Response Team (CERT) Coordination Center	1. CyLab Software Assurance Interest Group 2. Team Software Process (TSP) Secure 3. Secure Quality Requirements Engineering (SQUARE) 4. Computational Security Attributes	1. http://www.cylab.cmu.edu/default.aspx?id=177 2. http://www.sei.cmu.edu/tsp/tsp-security.html 3. http://www.cert.org/sse/square.html 4. http://www.cert.org/sse/csa.html *and* http://www.sei.cmu.edu/pub/documents/06.reports/pdf/06tr021.pdf
Cornell University/Department of Computer Science	Language-Based Security	http://www.cs.cornell.edu/Info/People/jgm/lang-based-security/index.htm

Table H-1 Software Security Research in Academia - *continued*

School/Department or Center/Group or Lab	Software Security Group, Lab, or Project	URL
United States of America		
Dakota State University/Center of Excellence in Computer Information Systems	Use/Misuse cases for Secure Software Requirements and Architecture	http://www.homepages.dsu.edu/paulij/pubs
DePaul University/School of Computer Science, Telecommunications and Information Systems/ Foundations of Programming Languages Group	Temporal Aspects *[with Alcatel-Lucent Bell Labs]*	http://teasp.org
Drexel University/Software Engineering Research Group	• CoSAK: source code analysis toolkit • Gemini: buffer overflow prevention	http://serg.cs.drexel.edu/projects
Florida Atlantic University/Department of Computer Science and Engineering/Secure Systems Research Group	1. Secure design patterns for software-intensive systems 2. Application security/application defense	1. none available 2. http://tcn.cse.fau.edu/asf/asf.htm
George Mason University/Department of Information and Software Engineering	Hardware/Software Approaches to Software Security	http://www.seas.gwu.edu/~narahari/research.html#code-security
Georgia Institute of Technology/College of Computing/Software Engineering, Programming Languages, Analysis, Reasoning, and Compilers group	Compiler Management for Tamper-Resistance and Performance	none available
Harvard University/Engineering and Applied Sciences/Computer Science	1. Cyclone (programming language) *[with University of Washington]* 2. PittSFIeld (sandboxing) *[with MIT]*	1. http://www.eecs.harvard.edu/~greg/cyclone/old_cyclone.html *and* http://cyclone.thelanguage.org 2. http://pag.csail.mit.edu/~smcc/projects/pittsfield

Table H-1 Software Security Research in Academia - *continued*

School/Department or Center/Group or Lab	Software Security Group, Lab, or Project	URL
United States of America		
Iowa State University/College of Engineering	• Secure software architectural design method • Secure configuration management	none available
Massachusetts Institute of Technology (MIT)/ Program Analysis Group (PAG)	PittSFIeld (sandboxing) *[with Harvard]*	http://pag.csail.mit.edu/~smcc/projects/pittsfield
Naval Post Graduate School/Center for Information Systems Security Studies and Research	1. High Assurance Security Program 2. Trusted Computing Exemplar (TCX) 3. Monterey Security Architecture (MYSEA)	1. http://cisr.nps.edu/projects/hasp.html 2. http://cisr.nps.navy.mil/projects/tcx.html 3. http://cisr.nps.navy.mil/projects/mysea.html
New Mexico Tech/Computer Science Department	1. Malware Analysis and Malicious Code Detection 2. Secure Software Construction	1. http://www.cs.nmt.edu/research.html#malware 2. http://www.cs.nmt.edu/research.html#secure
North Carolina State University/College of Engineering/ Computer Science Department	Test-Driven Development of Secure and Reliable Software Applications	none available
North Dakota State University/Department of Computer Science	Secure Software Engineering: A Threat-Driven Approach	http://cs.ndsu.edu/~dxu/research/security.html
Pennsylvania State University/Department of Computer Science and Engineering/Systems and Internet Infrastructure Security Laboratory and Smeal College of Business/eBusiness Research Center	1. Secure-typed languages 2. Hardware security 3. Security Aspects: Design (and Implementation Tools) for Security *[with Western Illinois University]* 4. An S-vector for Web Application Security Management *[with Polytechnic University]*	1. http://siis.cse.psu.edu/lbs.html 2. http://siis.cse.psu.edu/hw.html 3. http://www.cse.psu.edu/~tjaeger/research/aspects.html 4. http://www.smeal.psu.edu/cdt/ebrcpubs/res_papers/2004_01.pdf

Table H-1 Software Security Research in Academia - *continued*

School/Department or Center/Group or Lab	Software Security Group, Lab, or Project	URL
United States of America		
Polytechnic University	An S-vector for Web Application Security Management *[with Penn State]*	http://www.smeal.psu.edu/cdt/ebrcpubs/res_papers/2004_01.pdf
Princeton University/Department of Computer Science	Secure Internet Programming group	http://www.cs.princeton.edu/sip/
Purdue University/Center for Education and Research in Information Assurance and Security (CERIAS)	1. CERIAS Vulnerability Database 2. Secure Patch Distribution Group 3. Secure Programming Educational Material	1. http://www.cerias.purdue.edu/about/history/coast/projects/vuln_test.html and http://www.cerias.purdue.edu/about/history/coast/projects/vdb.php 2. http://www.cerias.purdue.edu/about/history/coast/projects/patch.php 3. http://projects.cerias.purdue.edu/secprog
Purdue University/Department of Computer Science	Secure Software Systems (S3) group	http://www.cs.purdue.edu/s3/
State University of New York at Stony Brook/Department of Computer Science/Secure Systems Lab	• Language-based security • Applications of formal methods in security • Operating system enhancements and related tools for security • Model-carrying code (MCC) • Secure mobile code execution environment • Model-Based Approach for Securing Software Systems • Automated code diversity with address obfuscation • Isolated program execution/Alcatraz tool • Specification-Based Techniques (formal methods) in Information Assurance	http://seclab.cs.sunysb.edu/seclab/research/projects/projects.htm

Table H-1 Software Security Research in Academia - *continued*

School/Department or Center/Group or Lab	Software Security Group, Lab, or Project	URL
United States of America		
Stevens Institute of Technology/Secure Systems Lab	1. Language-based security (*e.g.,* proof-carrying code) 2. Heap Bounded Assembly Language (HBAL)	1. none available 2. http://www.cs.stevens.edu/~abc/hbal
University of Arizona/Department of Computer Science	1. Sandmark: A Tool for the Study of Software Protection Algorithms 2. Dynamic Program Analysis for Secure and Reliable Computing	1. http://sandmark.cs.arizona.edu 2. http://www.cs.arizona.edu/~gupta/research/Projects/slice.html
University of California at Berkeley/Department of Computer Science	1. Software Security Project 2. Cqual 3. SafeDrive	1. http://www.cs.berkeley.edu/~daw/research/ss 2. http://www.cs.berkeley.edu/Research/Aiken/cqual 3. http://ivy.cs.berkeley.edu/safedrive
University of California at Davis/Department of Computer Science/Computer Security Laboratory	1. Secure Programming Clinic 2. Vulnerability analysis 3. DOVES: Database of Vulnerabilities, Exploits, and Signatures 4. Property-based testing	1. http://twinpeaks.cs.ucdavis.edu/clinic/ 2. http://seclab.cs.ucdavis.edu/projects/Vulnerabilities.html 3. http://seclab.cs.ucdavis.edu/projects/DOVES/ 4. http://seclab.cs.ucdavis.edu/projects/NASA-JPLsum.html
University of California at Irvine/Institute for Software Research	Architecture-Based Security and Trust	http://www.isr.uci.edu/architecture/archsnt.html
University of Delaware/Department of Computer and Information Sciences/Software Engineering Research Program/HiperSpace	1. Mobile Code Validation through Static Program Analysis, Steganography, and Transformation Control 2. Testing Program-Based Security Mechanisms	1. http://www.cis.udel.edu/%7Ehiper/projects/integrity.html 2. http://www.cis.udel.edu/%7Ehiper/projects/sec-mech-test.html

Table H-1 Software Security Research in Academia - *continued*

School/Department or Center/Group or Lab	Software Security Group, Lab, or Project	URL
United States of America		
University of Illinois at Chicago/Department of Electrical and Computer Engineering	1. Architecture Support for Software Protection 2. Program Counter Encoding for System Hardening	1. http://arch.ece.uic.edu/security/validating-control-flow.htm 2. http://arch.ece.uic.edu/security/pc-encoding.htm
University of Illinois at Urbana-Champaign/ Information Trust Institute (ITI)	3. Reconfigurable Reliability and Security Engine (RESE) 4. Formal Reasoning on Security Vulnerabilities Using Pointer Taintedness Semantics 5. SAFEcode: Static Analysis For safe Execution of Code	3. http://www.crhc.uiuc.edu/DEPEND/randsengine.htm 4. http://www.iti.uiuc.edu/seminars/2004-12-03/Achieving_Trusted_Systems_by_Providing_Security_and_Reliability.pdf 5. http://safecode.cs.uiuc.edu
University of Idaho/Center for Secure and Dependable Systems (CSDS)	Multiple Independent Levels of Security (MILS)	http://www.csds.uidaho.edu/mils.shtml
University of Louisiana at Lafayette/Center for Advanced Computer Studies/Software Research Laboratory	1. Vilo: Malware Search and Analysis Capabilities 2. DOC: Detector of Obfuscated Calls	1. http://www.cacs.louisiana.edu/labs/SRL/vilo.html 2. http://www.cacs.louisiana.edu/labs/SRL/doc.html
University of Louisiana at Lafayette/Special Interest Group on Cybersecurity/Systems Security Research Group	1. Formal methods for security 2. Buffer overflow detection, prevention 3. Formal methods, program analysis, and reverse engineering of binaries for malicious code detection	http://www.cacs.louisiana.edu/cybersecurity/research/sysres.html
University of Maryland at College Park/Institute for Advanced Computer Studies Center for Human Enhanced Secure Systems (CHESS)	Human Enhanced Code Analysis and Development	http://chess.umiacs.umd.edu/research_cad.htm

Table H-1 Software Security Research in Academia - *continued*

School/Department or Center/Group or Lab	Software Security Group, Lab, or Project	URL
United States of America		
University of Pennsylvania/Department of Computer and Information Science	Security-Oriented Languages	http://www.cis.upenn.edu/~stevez/sol
University of Southern California/Center for Systems and Software Engineering (CSSE)	Costing Software Security	• http://csse.usc.edu/cse/pub/research/software_security • http://sunset.usc.edu/csse/TECHRPTS/2006/usccse2006-600/usccse2006-600.pdf
University of Texas at Arlington/Department of Computer Science	Malware analysis (Cobra, SPiKE, VAMPiRE)	• Cobra: http://data.uta.edu/%7Eramesh/pubs/IEEE-Cobra.pdf • SPiKE: http://data.uta.edu/%7Eramesh/pubs/ACSC06-SPiKE.pdf • VAMPiRE: http://data.uta.edu/%7Eramesh/pubs/ACSAC05-VAMPiRE.pdf
University of Texas at Dallas/Cybersecurity Research Center and Security Analysis and Information Assurance Laboratory	• Language security: buffer overflow defense • Secure component-based software • Embedded systems security • Software assurance • High-Assurance Vulnerability Detection	none available
University of Tulsa/Software Engineering and Architecture Team	Secure component assembly	http://www.seat.utulsa.edu/chronological.php

Table H-1 Software Security Research in Academia - *continued*

School/Department or Center/Group or Lab	Software Security Group, Lab, or Project	URL
United States of America		
University of Virginia/Department of Computer Science/Dependability Research Group	1. Genesis: A Framework for Achieving Component Diversity 2. PHPrevent: Automatically Hardening Web Applications 3. N-Variant Systems Framework	1. http://dependability.cs.virginia.edu/info/Genesis 2. http://dependability.cs.virginia.edu/info/PHPrevent 3. http://dependability.cs.virginia.edu/info/N-Variant_Systems_Framework
University of Washington/Department of Computer Science and Engineering/Advanced Systems for Programming	1. Denali: Lightweight virtual machines 2. Cyclone (programming language) *[with Harvard]*	1. http //denali.cs.washington.edu 2. http//cyclone.thelanguage.org/ *and* http://www.eecs.harvard.edu/~greg/cyclone/old_cyclone.html
Western Illinois University/Department of Computer Science	1. Security Aspects: Design (and Implementation Tools) for Security *[with Penn State]* 2. findssv: static analysis of executables *[with Auburn State]*	1. http://www.cse.psu.edu/~tjaeger/research/aspects.html 2. n/a
Canada		
Queen's University (Kingston, ON)/School of Computing/Software Technology Laboratory/Queen's Reliable Software Technology Group (QRST)	1. Unifying Software Engineering and Security Engineering 2. Intrusion Detection-Aware Software Systems	none available
University of Regina/Department of Computer Science	IsoMod discretionary capability confinement system for the Java Virtual Machine (JVM)	http://www2.cs.uregina.ca/~pwlfong/Pub/esorics2006.pdf
United Kingdom		
City University (London)/School of Informatics/Centre for Software Reliability	International Working Group on Assurance Cases (including software security assurance cases)	http://www.csr.city.ac.uk/AssuranceCases

Table H-1 Software Security Research in Academia - *continued*

School/Department or Center/Group or Lab	Software Security Group, Lab, or Project	URL
United Kingdom		
University of Cambridge/Computer Laboratory/ Security Group	1. Economics of information and software security 2. TAMPER (Tamper And Monitoring Protection Engineering Research) Lab	1. http://www.cl.cam.ac.uk/~rja14/econsec.html 2. http://www.cl.cam.ac.uk/research/security/tamper
University of Newcastle upon Tyne/Centre for Software Reliability	MAFTIA: Malicious-and Accidental-Fault Tolerance for Internet Applications	http://www.csr.ncl.ac.uk/projects/projectDetails.php?targetId=101
Finland		
Oulu University/Computer Engineering Laboratory	Secure Programming Group	http://www.ee.oulu.fi/research/ouspg
University of Lapland/Institute for Legal Informatics	Regulating Secure Software Development	http://www.ulapland.fi/?newsid=6440&deptid=11589&showmodul=47&languageid=4&news=1 *also:* http://www.ulapland.fi/includes/file_download.asp?deptid=22713&fileid=9480&file=20061023140353.pdf&pdf=1
Sweden		
Lindköpings University (Sweden)/Department of Computer and Information Science (IDA)/ Programming Environments Laboratory (PELAB)	Dependence graphs for modeling and visualizing software and source code security properties	http://www.ida.liu.se/~johwi/research_publications
Denmark		
Technical University of Denmark (Lyngby)/Safe and Secure IT Systems/Language-Based Technology	SiES: Security in Embedded Systems	http://www.imm.dtu.dk/English/Research/Safe_and_Secure_IT_Systems/Projects.aspx
Belgium		
Ghent University/Electronics and Information Systems (EIS) department/PARIS research group	Coordinated Research of Program Obfuscation	http://www.elis.ugent.be/obfus/ and http://trappist.elis.ugent.be/~mmadou/drupal

Table H-1 Software Security Research in Academia - *continued*

School/Department or Center/Group or Lab	Software Security Group, Lab, or Project	URL
	Belgium	
Catholic University of Leuven/Department of Computer Science/DistriNet Research Group/Security Working Group and COSIC (COmputer Security and Industrial Cryptography) Research Group	1. SoBeNeT: Software Security for Network Applications 2. AGILE: Agile Software Development of Embedded Systems 3. Measuring framework for software security properties	1. http://sobenet.cs.kuleuven.be/index.jsp 2. http://www.cs.kuleuven.ac.be/cwis/research/distrinet/public/research/showproject.php?ABBREV=AGILE 3. none available
	The Netherlands	
Radboud University Nijmegen/Institute for Computing and Information Sciences/Laboratory for Quality Software (LaQuSo)/Security of Systems (SoS) Group	PIONIER (program security and correctness)	http://www.nwo.nl/projecten.nsf/pages/1600111396 *[in Dutch]* *also:* http://www.cs.ru.nl/~bart/Pionier
	Germany	
German Research Center for Artificial Intelligence (Transfer Center)/Formal Methods group	1. Secure Software group 2. Verification Support Environment	1. http://www.dfki.de/siso *[in German]* 2. http://www.dfki.de/vse/projects/vse-short.html
Fraunhofer Institute for Experimental Software Engineering (IESE) (Kaiserslautern)/Department of Security and Safety	• Quality Models: Examination of critical hardware and software systems for relevant security and safety characteristics. • Development Coaching: during all process phases in the development of safety- or security-critical systems	none available
Technical University of Dresden/Computer Science Department/Systems Engineering Group	AutoPatch	http://wwwse.inf.tu-dresden.de/AutoPatch/

Table H-1 Software Security Research in Academia - *continued*

School/Department or Center/Group or Lab	Software Security Group, Lab, or Project	URL
Germany		
Technical University of Munich/Chair for Theoretical Computer Science and Foundations of Artificial Intelligence/Formal Methods and Theory Group	Software Security	http://www.model.informatik.tu-muenchen.de/research/projects/detail/index.php?id=projects.detail&arg=18
Technical University of Munich/Faculty for Informatics Chair IV/Competence Center in IT/Security, Software, and Systems Engineering	Working Group on Security and Safety in Software Engineering	http://www4.in.tum.de/~secse/group.html
University of Duisburg Essen/Department of Computer Science and Applied Cognitive Science/Software Engineering Workgroup	• Security problem frames for component-based software development • Formal methods for security analysis • Development of secure software	none available
University of Mannheim/Laboratory for Dependable Distributed Systems	Hardware-Software interactions and their Security issues: Dependability (including security) Metrics	http://pi1.informatik.uni-mannheim.de/index.php?pagecontent=site/Research.menu/Projects.page/Dependability%20Metrics.page&show=true
University of Stuttgart/Institute for Formal Methods in Computer Science	Software Reliability and Security Group	http://www.fmi.uni-stuttgart.de/szs/index.en.shtml
Switzerland		
University of Lugano/Faculty of Informatics	Detection of Security Flaws and Vulnerabilities by Guided Model Checking	http://www.inf.unisi.ch/research/#DSFV

Table H-1 Software Security Research in Academia - *continued*

School/Department or Center/Group or Lab	Software Security Group, Lab, or Project	URL
Austria		
Technical University Vienna/Distributed Systems and Automation Systems Groups/Secure Systems Lab	1. Anubis: Analyzing Unknown Binaries (Software Security through Binary Analysis) 2. Pathfinder: Malicious Code Analysis and Detection 3. Software Security Audit using Reverse Engineering	1. http://analysis.seclab.tuwien.ac.at
Greece		
Athens University of Economics and Business/ Department of Management Science and Technology/ Information Systems Technologies Lab (ISTLab)	SENSE - Software Engineering and Security	http://istlab.dmst.aueb.gr/content/groups/g_sense-details.html
Russia		
Russian Academy of Sciences/Institute for System Programming/DMA group	Investigation of algorithmic methods of program protection	http://ispras.ru/groups/dma/projects.html?1#1
Australia		
Bond University (Queensland)/Centre for Software Assurance	Model-Based Testing for Software Security	none available
Swinburne University of Technology (Melbourne)/ Faculty of Information and Communication Technologies/Centre for Information Technology Research/Reliable Software Systems group and Component Software and Enterprise Systems group	1. Characterization and compositional security analysis for software systems 2. Security Engineering for Component-Based Software	1. http://www.swin.edu.au/ict/research/citr/rss.htm 2. http://www.it.swin.edu.au/centres/cecses/projects.html#security *and* http://www.it.swin.edu.au/personal/jhan/jhanPub.html
New Zealand		
University of Auckland/Department of Computer Science/Secure Systems Group	Software obfuscation, watermarking, tamperproofing	http://www.cs.auckland.ac.nz/research/groups/ssg

Table H-1 Software Security Research in Academia - *continued*

School/Department or Center/Group or Lab	Software Security Group, Lab, or Project	URL
	Japan	
Tokyo Institute of Technology/Programming Systems Group	Implementation Schemes for Secure Software	http://anzen.is.titech.ac.jp/index-e.html
University of Tokyo/Yonezawa Lab for Information Science/Yonezawa Group/Secure Computing Project and Virtual Infrastructure for Networked Computers (VINCS) Project and Typed Computing Project	1. Fail-safe C Compiler: A memory-safe ANSI-C Compiler 2. SpecSB: A Sandboxing System that Executes Speculative Security Checks 3. MatSB: A Sandbox System for Protecting Security Systems 4. G'Caml: evaluation with types	1. http://venus.is.s.u-tokyo.ac.jp/~oiwa/FailSafe-C.html 2. http://www.yl.is.s.u-tokyo.ac.jp/projects/specsb 3. http://www.yl.is.s.u-tokyo.ac.jp/projects/matsb 4. http://web.yl.is.s.u-tokyo.ac.jp/~furuse/gcaml
	Taiwan	
Institute of Information Science Academia Sinica	Web Application Software Security (esp. vulnerability assessment)	http://www.iis.sinica.edu.tw
	Iran	
Amirkabir University of Technology (Tehran)/ Department of Computer Engineering and Information Technology	RUPSec	see Section 5.1.8.2.6 under "RUPSec"
	Tunisia	
University of Manouba	Environment for Addressing Software Application Security Issues (ASASI)	http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?isnumber=4041509&arnumber=4041534&count=77&index=23
	Nigeria	
University of Agriculture (Abeokuta)	Secure Software Development Model (SSDM)	see Section 5.1.8.2.6 under "SSDM"

This State-of-the-Art Report is published by the Information Assurance Technology Analysis Center (IATAC). IATAC is a DoD-sponsored Information Analysis Center, administratively managed by the Defense Technical Information Center (DTIC), and the Director, Defense Research and Engineering (DDR&E).