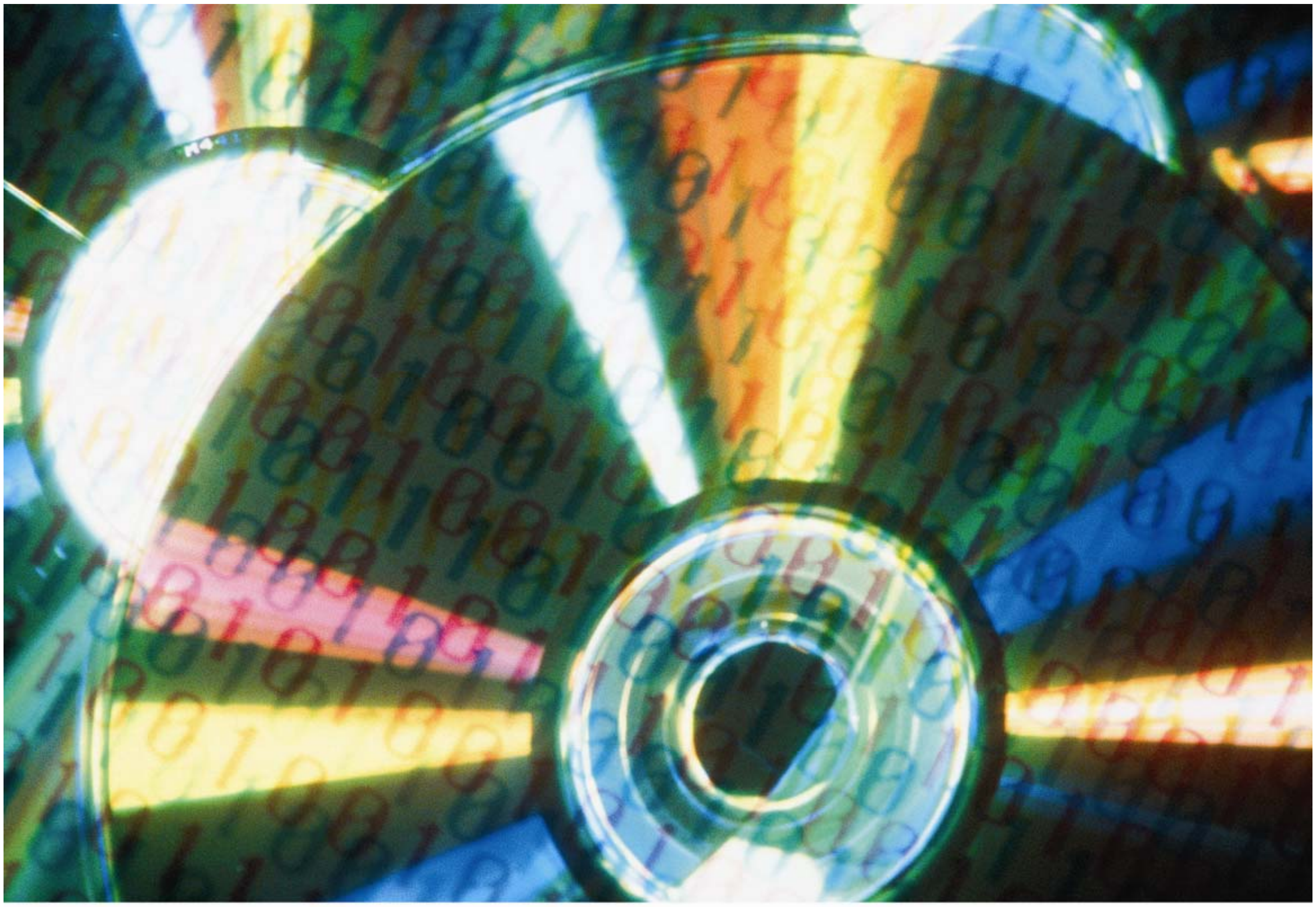


Software Assurance: A Curriculum Guide to the Common Body of Knowledge to Produce, Acquire and Sustain Secure Software

Software Assurance Workforce Education and Training Working Group

October 2007

Homeland Security

Software Assurance: A Curriculum Guide to the Common Body of Knowledge to Produce, Acquire and Sustain Secure Software

Software Assurance Workforce Education and Training Working Group

Draft Version 1.2
October 2007

Version 1.2 29 October 2007

This document is offered for informative use; it is not intended as a policy or standard.

When referring to, quoting, or excerpting from this document please always ensure proper acknowledgement is given.

Citation: Samuel T. Redwine, Jr. (Editor). Software Assurance: A Guide to the Common Body of Knowledge to Produce, Acquire, and Sustain Secure Software Version 1.1.

US Department of Homeland Security, September 2006.

How to Make Contact, Find out More, and Contribute

Through a jointly sponsored working group on Software Assurance Workforce Education and Training, the Department of Homeland Security (DHS) Software Assurance Program is seeking additional input and participation in further developing this Secure Software Assurance Common Body of Knowledge document and related materials. Representatives from government, academia, and the private industry have identified key information needed to educate and train on the developing, sustaining, and acquiring of secure software. This document is offered for informative use; it is not intended as a policy or standard.

If you would like to interact regarding this document, participate in the Workforce Education and Training Working Group, or find out more information about software security and the DHS efforts to improve it; visit the “Build Security In” website at https://buildsecurityin.us-cert.gov.

Foreword

Dependency on information technology makes software assurance a key element of national security and homeland security. Software vulnerabilities jeopardize intellectual property, consumer trust, business operations and services, and a broad spectrum of critical applications and infrastructure, including everything from process control systems to commercial application products. In order to ensure the integrity of key assets, the software that enables and controls them must be reliable and secure. However, informed consumers have growing concerns about the scarcity of practitioners with requisite competencies to build secure software. They have concerns with suppliers' capabilities to build and deliver secure software with requisite levels of integrity and to exercise a minimum level of responsible practice. Because software development offers opportunities to insert malicious code and to unintentionally design and build exploitable software, security-enhanced processes and practices – and the skilled people to perform them – are required to build trust into software.

The Department of Homeland Security (DHS) Software Assurance Program is grounded in the National Strategy to Secure Cyberspace which indicates: "DHS 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."

Software Assurance has become critical because dramatic increases in business and mission risks are now known to be attributable to exploitable software: system interdependence and software dependence has software as the weakest link; software size and complexity obscures intent and precludes exhaustive test; outsourcing and use of un-vetted software supply chain increases risk exposure; attack sophistication eases exploitation; reuse of legacy software interfaced with other applications in new environments introduces other unintended consequences increasing number of vulnerable targets; and the number of threats targeting software are increasing. These all contribute to the increase of risks to software-enabled capabilities and the threat of asymmetric attack. A broad range of stakeholders now need confidence that the software which enables their core business operations can be trusted to perform (even with attempted exploitation).

DHS began the Software Assurance (SwA) Program as a focal point to partner with the private sector, academia, and other government agencies in order to improve software development and acquisition processes. Through public-private partnerships, the Software Assurance Program framework shapes a comprehensive strategy that addresses people, processes, technology, and acquisition throughout the software lifecycle. Collaborative efforts seek to shift the paradigm away from patch management to achieving a broader ability to routinely develop and deploy software products known to be trustworthy. These efforts focus on contributing to the production of higher quality, more secure software that contributes to operations that are more resilient.

In their report to the President, *Cyber Security: A Crisis of Prioritization* (February 2005), in the chapter entitled "Software Is a Major Vulnerability", the President's Information Technology Advisory Committee (PITAC) summed up the problem of non-secure software concisely and accurately:

> *Network connectivity provides "door-to-door" transportation for attackers, but vulnerabilities in the software residing in computers substantially compound the cyber security problem. As the PITAC noted in a 1999 report, the software development methods that have been the norm fail to provide the high-quality, reliable, and secure software that the Information Technology 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. 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.*

The DHS Software Assurance (SwA) program goals promote the security of software across the development life cycle and are scoped to address:

> ***Trustworthiness** – No exploitable vulnerabilities exist, either maliciously or unintentionally inserted;*
>
> ***Predicable Execution** – Justifiable confidence that software, when executed, functions in a manner in which it is intended;*
>
> ***Conformance** – Planned and systematic set of multi-disciplinary activities that ensure software processes and products conform to requirements and applicable standards and procedures.*

Initiatives, such as the DHS "Build Security In" web site (https://buildsecurityin.us-cert.gov) and the developers' guide entitled *Security in the Software Lifecycle: Making Application Development Processes – and Software Produced by Them – More Secure,* will continue to evolve and provide practical guidance and reference material to software developers, architects, and educators on how to improve the quality, reliability, and security of software – and the justification to use it with confidence.

This document, *"Software Assurance: A Curriculum Guide to the Common Body of Knowledge to Produce, Acquire, and Sustain Secure Software,"* (referenced, in short, as SwA CBK) provides a framework intended to identify workforce needs for competencies, leverage sound practices, and guide curriculum development for education and training relevant to software assurance. Because Software Quality Assurance (SQA) and Software Engineering have evolved bodies of knowledge that do not explicitly address security as a quality attribute, and the "National Strategy to Secure Cyberspace" Action/Recommendation 2-14 relating to Software Assurance (SwA) focused on security, integrity and reliability, the initial focus of the SwA education has been to "complete" relevant academic programs. As such, SwA's initial focus has been on persons with knowledge of SQA and Software Engineering but not security.

The SwA CBK introduces the field software-security-related knowledge; points to references for further knowledge, and supplies the background needed to meaningfully select which references might be of interest.

A series of CBK working group sessions have involved participation from academia, industry and federal government to develop this document to address three domains: "acquisition," "development," and post-release "sustainment." Several disciplines contribute to the SwA CBK, such as software engineering, systems engineering, information systems security engineering, safety, security, testing, information assurance, and project management. While SwA is not a separate profession, SwA processes and practices should contribute to enhancing these contributing disciplines. In education and training, Software Assurance could be addressed as: a "knowledge area" extension within each of the contributing disciplines, a stand-alone CBK drawing upon contributing disciplines, or as a set of functional roles drawing upon the CBK, allowing for more in-depth coverage, depending on the specific roles. At a minimum, SwA practices should be integrated within applicable knowledge areas of relevant disciplines.

The SwA working group has reviewed the CBK extensively in the past 18 months, including a separate session held to specifically address the community-accepted Software Security Principles from which the SwA CBK sections were mapped to each principle. Several academic organizations (such as James Madison University, the University of North Carolina, Mississippi State University, the University of Detroit, and the Information

Resouces Management College at the National Defense University) have begun using the draft versions of the SwA CBK to develop their SwA-related courses.

The SwA CBK is a part of the Software Assurance Series, and it is expected to contribute to the growing Software Assurance community of practice. This document is intended solely as a source of information and reference, and is not a proposed standard, directive, or policy from DHS. Because this document will continue to evolve with use and changes in practices, comments on its utility and recommendations for improvement are always welcome. Stakeholders are encouraged to provide lessons learned and sample SwA curriculum.

Authorship and Acknowledgements

The Department of Defense (DoD) and Department of Homeland Security (DHS) Software Assurance efforts encompass software safety, security, reliability, and dependability; yet initially they are concentrating on achieving and assuring security properties and functionality. Their Software Assurance (SwA) Workforce Education and Training Working Group is composed of government, industry, and academic members. This Working Group produced this document as its first step towards achieving adequate US education and training on software assurance. It identifies the additional body of knowledge necessary to develop, sustain, and acquire secure software beyond that normally required to produce and assure software where safety and security are not concerns. The knowledge areas identified and their related references provide a foundation for producing education and training curricula and products as well as being useful to standards developers, evaluators and testers, and others wishing to be knowledgeable in all or part of software-related security.[1]

This document was written and edited by Samuel T. Redwine, Jr. Additional authors included Rusty O. Baldwin, Mary L. Polydys, Daniel P. Shoemaker, Jeffrey A. Ingalsbe and Larry D. Wagoner. Additional contributors included Martin Croxford of Praxis High Integrity Systems Ltd; John McDermid, University of York; Jim Moore, MITRE; Mary Ann Davidson, Oracle; Karen Goertzel, Booz Allen Hamilton; and the Ford Motor Company, whose staff helped in the production of the section on Sustainment, particularly its reference list.

The editor and authors would like to thank their co-authors for mutual help, the Working Group members for the review comments received and the helpful issue discussions that occurred, and external reviewers who contributed their valuable time to improve this guide. In part, James Madison University and its Institute for Infrastructure and Information Assurance supported Mr. Redwine's work.

Reviews of all or parts of drafts of this document were provided by Mark R. Blackburn, Systems and Software Consortium, Inc.;Florian P. Buchholz, James Madison University; J. Katharine Burton, Institute for Defense Analyses ; Mary Ann Davidson, Oracle ; Robert Dupuis, Université du Québec à Montréal; Christopher Fox, James Madison University; Karen Goertzel, Booz Allen Hamilton; Frank Herman, Fraunhofer Center for Experimental Software Engineering, Maryland; George Huber, SRI; David Jackson, QinetiQ; David Ladd, Microsoft; Seok-Won Lee, University of North Carolina Charlotte; Glen Logan, OUSD (AT&L); Vic Maconachy, National Security Agency; Nancy Mead, Software Engineering Institute, Carnegie Mellon; James W. Moore, The MITRE Corporation; Sarah Nash, Institute for Defense Analyses; Ken Nidiffer, Systems and Software Consortium, Inc.; Ioana Rus, Fraunhofer Center for Experimental Software Engineering, Maryland; Joseph M. Saur, Georgia Tech Research Institute, Joint Systems Integration Command; Gary R. Stoneburner, Johns Hopkins University Applied Physics Laboratory; John Viega, MacAfee; Larry Wagoner, National Security Agency; James Walden, Northern Kentucky University; David A. Wheeler, Institute for Defense Analyses; and Stan Wisseman, Booz Allen Hamilton.

Special thanks go to:

Joe Jarzombek, Director of Software Assurance, Department of Homeland Security - National Cyber Security Division, without whose leadership and support this report would not exist.

Mitchell Komaroff, Office of the Chief Information Officer, Information Management and Technology, US Department of Defense for his DOD-related efforts.

Joseph Saur, who championed and managed early efforts to produce the Acquisition portion.

[1] See page *ii* for more information on the Working Group or to contact or join the group.

Jim Moore of The Mitre Corporation was particularly helpful. He personally discussed many aspects of the document, provided detailed reviews and links into the standards community, and made extensive interaction time available with the Process and Practices Working Group.

David Ladd of Microsoft, who attended all the early meetings and made his extensive expertise and advice freely available inside and outside of the meetings. The WG's continuing enlargement of participation owe much to his championing of this need.

Robert Dupuis, who identified many areas where the text presumed more knowledge than exists in the Guide to the Software Engineering Body of Knowledge in which he has been instrumental (any remaining difficulties are solely the responsibility of the authors).

David Wheeler, for pointing out convincingly that the need to be concerned about security is now the "normal" case for software.

Jeff Voas, Shareli Zadeli, Antonio Drommi who provided comments and/or useful participation for the Sustainment portion,

Nancy Mead, Robert Ellison, and others at the Software Engineering Institute, and Gary McGraw and others at Cigital who shared their efforts in this area.

John McDermid of the University of York, who not only contributed quoted text but also generously provided an especially useful discussion one hot afternoon/evening in Washington, as well as pointing the authors to a number of useful references.

Chuck Howell of The MITRE Corporation, who spent an evening providing the benefit of his experience with assurance cases, including discussing the workshop he chaired on the subject.

Mark Vanfleet of NSA for useful discussion and identification of experts

Gary Stoneburner of John Hopkins University, for providing stimulating and useful discussion and comments that were particularly appreciated.

The students in CS480 Secure Software Engineering at James Madison University who helped identify places where the text could be improved.

Lisa Enders, Ginna Baker, Kelly Rote, Christine Jones, and Jeniffer Leahy, James Madison University student assistants who toiled diligently on the many unglamorous tasks that needed to be done

Kimberly Caraway and Cindy Gagliano of Booz Allen Hamilton,(BAH) for administrative assistance and Holly Mckinley and Karen Goertzel, also of BAH, for editing assistance.

Matt Bishop, Cynthia Irvine, Carol Taylor, and Joe Jarzombek, for help organizing a summer 2006 workshop for educators at the Naval Postgraduate School, hosted by Cynthia Irvine. The attendees deserve thanks for their helpful participation and follow-up comments

The Working Group members were Lawrence Baker, Defense Acquisition University; Rusty O. Baldwin, Air Force Institute of Technology; Paul E. Black, Michael Kass, and Michael Koo, National Instituted of Standards and Technology (NIST); Blaine Burnham, University of Nebraska at Omaha; J. Katharine Burton, Sarah Nash, Edward A. Schneider, and David A. Wheeler, Institute for Defense Analyses; Djenana Campara, Klocwork Inc.; Mary Carroll and Linda Polydys, Information Resources Management College at National Defense University; Martin Croxford, Praxis High Integrity Systems Ltd.; Mary Ann Davidson, Oracle; Antonio Drommi, University of Detroit Mercy; Robert Dupuis, Université du Québec à Montréal; Frank Herman, Fraunhofer Center for Experimental Software Engineering, Maryland; Jeffrey A. Ingalsbe, Ford Motor Company; Arthur King, IBM Business Consulting Services supporting OASD(NII) DIAP; l David Ladd, Microsoft Research; Barbara Laswell, Software Engineering Institute, Carnegie Mellon; Seok-Won Lee, University of North

Carolina Charlotte; Tom McGibbon, ITT Industries; James W. Moore, The MITRE Corporation; Samuel T. Redwine, Jr., James Madison University; Joseph M. Saur, Georgia Tech Research Institute, Joint Systems Integration Command; Dan Shoemaker, University of Detroit Mercy; Frank Stomp and Sherali Zeadally, Wayne State University; Jeffrey Voas, Science Applications International Corporation; and Larry Wagoner, National Security Agency.

The (IEEE) International Symposium on Secure Software Engineering and the Software Assurance Forum provide opportunities for additional information in this area. The Workshop on Secure Software Engineering Education and Training is specifically on the topic.

The Working Group's life extends beyond the production of this guide, and its goals remain the same, to help create a workforce capable of developing, sustaining, assuring, and acquiring (more) secure software – but its specific activities may vary. The Working Group welcomes participation in its ongoing activities.[2]

[2] See page ii for directions on how to find out more about, contact, or join the Working Group.

Editor's Preface

Introduction

In June 2005, we took on the task of developing a document intended to provide educators, trainers, and others possessing knowledge of software, but not of security, a companion reference to guide them across the surface of the software security[1] field and point out locations where they can obtain further knowledge by digging into the identified references. We have progressed through a series of publicly reviewed iterations to reach this version. We hope this guide inspires and aids the development of products and services from academia, industry, and government, especially those developing and evolving applicable curriculum, by helping people identify relevant items in the existing body of knowledge.

Along the way, a number of individuals and organizations gave comments both large and small on the many interim drafts providing feedback that I and the other authors greatly appreciated – though we were always eager for more. In the end, this document is the result of those individuals and organizations willing to put in their own resources and step up to do a needed task, and I greatly appreciate them all. In the future, this guide's evolution will benefit from the involvement of even more of the community, including feedback from users.

In selecting the guide's contents and organization, we intended to supply a background and contextual software-security-related or assurance-related knowledge for persons already possessing good software engineering knowledge. While we did not attempt to produce a textbook, and this guide is not one, some may choose to use its text as a high-level introduction to the topic. However, in its primary role, educators, trainers and others will use this text to guide them to the references containing the more detailed software-security-related knowledge relevant to their work. Therefore, its design allows users – after reading the first four introductory sections– to move directly to additional section(s) of interest. Any related knowledge is then cross-referenced. The guide also reads properly front to back with prerequisite knowledge being introduced before it is required.

The primary audiences for this guide are educators and trainers to help them identify both appropriate curricular content and references that detail it. Other audiences include evaluators and testers, acquisition personnel, and standards developers for whom it can help provide a better appreciation of the breadth of subject matter and issues involved. Given the unanticipated accessibility of this guide, studious practitioners could use it to obtain overviews or to guide their learning.

This preface provides a history of the development this guide to the secure software engineering common body of knowledge and guidance on the paths within the document that readers with different interests may take. Those without interest in this history can proceed directly to "How to Read this Document" on page ix or to the Introduction on page 3.

[1] Because of potential confusion among software security, security software, secure software, and the security of software; in addition to using "software security" and "software system security" as inclusive terms, this document sometimes uses the more awkward phrase "software-security-related" to emphasize this inclusiveness. All three inclusive terms certainly include security software and the security of software but also other issues such as the security of additional kinds of computing resources associated with or accessed by software. The phrase "secure software" is used in the same inclusive way but in referring to software or software product, which results in or might affect security.

Some History

In 2003, the US Department of Defense (DoD) launched a Software Assurance Initiative led by Joe Jarzombek,[2] and this was joined in 2004 by the Department of Homeland Security (DHS). In March 2005, Mr. Jarzombek moved to become Director for Software Assurance, National Cyber Security Division within DHS and retains his leadership role in the collaborative interagency Software Assurance efforts including overseeing the development of this document, the SwA CBK.

The DoD and DHS Software Assurance initiatives have submitted internal, interim reports and held jointly sponsored Software Assurance Forums and a number of individual working group (WG) meetings. Among the working groups is one on education and training which includes the individuals from government, industry, and academia that produced this guide. In hopes of soliciting further participation, Joe Jarzombek and others also made a number of appearances at professional events to publicize the efforts and the problems they are addressing.

Driven by an awareness of the rampant worldwide explosion in exploitation of software vulnerabilities, demand is growing for low-defect, secure software systems, in both the defense and commercial sectors. Current, commonplace software specification, design, implementation, and testing practices provide users with software containing numerous defects and security vulnerabilities.[3] Government and industry need processes that effectively and efficiently acquire, develop, and sustain secure software; means to justify confidence in these processes and their products; and practitioners that are motivated, disciplined, and proficient in their execution.

Currently, other Working Groups are concentrating on other aspects of achieving and assuring security properties and functionality[4] including standards, measurement, and tools. The Workforce Education and Training Working Group is addressing the issues related to achieving adequate US software-security-related education and training, including skill shortages within government and industry, and curriculum needs within universities, colleges, and trade schools.[5]

The WG asked the following questions:

1. What are the engineering activities or aspects of activities that are relevant to achieving secure software?
2. What knowledge is needed to perform these activities or aspects?

The early meetings addressing these questions were sometimes uneven and involved brainstorming sessions producing lists of terms and topics – sometimes without shared definitions or understanding. Starting any effort involving a number of people of disparate backgrounds and skills can be hard, and, luckily, the document managed to benefit from these early efforts and progressed well beyond any potential ill effects on its quality.

[2] Then Deputy Director for Software Assurance, Information Assurance Directorate, Office of Assistant Secretary of Defense (Networks and Information Integration)

[3] The February 2005 President's Information Technology Advisory Committee (PITAC) Report to the President, *Cyber Security: A Crisis of Prioritization*, identified the top ten areas in need of increased support, including 'secure software engineering and software assurance'. The findings indicated: Commercial software engineering today lacks the scientific underpinnings and rigorous controls needed to produce high-quality, secure products at acceptable cost. Commonly used software engineering practices permit dangerous errors, such as improper handling of buffer overflows, which enable hundreds of attack programs to compromise millions of computers every year. In the future, the Nation may face even more challenging problems as adversaries – both foreign and domestic – become increasingly sophisticated in their ability to insert malicious code into critical software.

[4] Such software must, of course, also be satisfactory in other aspects as well such as usability and mission support.

[5] The National Software Strategy includes educating and fielding the software workforce to improve software trustworthiness (see report at www.cnsoftware.org/nss2report).

Deciding what knowledge to presume of the readers of this guide took several twists and turns. Initially, the subgroup addressing software development took the Guide to the Software Engineering Body of Knowledge (SWEBOK[6]) [Abran 2004] as a starting point for establishing its presumption of knowledge already identified. The WG knew, however, that the intended readership did not necessarily know this much, and it always tried not to presume more of readers than could be reasonably expected. The WG's and this SwA CBK's goal was to identify the "additional" knowledge needed for developing, sustaining, and acquiring secure software while recognizing that a clear boundary for what is "additional" does not always exist. For some knowledge elements, the difference is mainly in the extremely detailed knowledge or level of rigor required for secure software.

The subgroups on Acquisition and Supply and on Sustainment and Operations had even more difficult problems in finding existing descriptions as bases for their "presumed" bodies of knowledge. As partial answers, the acquisition subgroup used federal acquisition documents and sustainment used the Maintenance section of the SWEBOK Guide and any relevant standards.

Fortunately, the efforts to answer the question "What are the engineering activities or aspects of activities that are relevant to achieving secure software?" benefited from a number of prior efforts and products including:

- National Cyber Security Partnership Taskforce Report on *Processes to Produce Secure Software* [Redwine 2004]
- Safety and Security Extensions for Integrated Capability Maturity Models [Ibrahim et al, 2004]
- IEEE Software and Systems Engineering Standards Committee (S2ESC) collection of IEEE standards
- ISO/IEC JTC1/SC7 WG9 Redefined its terms of reference to software and system assurance (part of Systems Engineering System Life Cycle Processes)
- ISO/IEC 15026 to address management of risk and assurance of safety, security, & dependability within context of system and software life cycles [ISO 15026]
- National Institute of Standards and Technology (NIST) Federal Informaition Security Management Act (FISMA) Implementation Project
- The *Common Criteria* for evaluating the security of software including the new version 3.0 issued in July 2005 [CC 2005]
- The SafSec effort in the UK7 combining concern for safety and security [SafSec Introduction], [SafSec Standard], and [SafSec Guidance]
- A variety of safety-related standards efforts as they address a number of issues shared by safety and security in which the safety community has useful, relevant experience

Members of the WG include editors and authors within several of these efforts. Even with these prior efforts, answering the question about relevant activities occupied much of the initial effort. The WG has also benefited from the work of other Software Assurance Working Groups, such as the one on Software Assurance Processes and Practices, which are also interested in the activities involved in producing and assuring secure software.

For development, the resulting lists needed to be consolidated into some set of categories to form the structure of this document. Among the candidate sets of categories were those from:

- ISO/IEC 15288 – System Life Cycle Processes
- ISO/IEC 12207 – Software Life Cycle Processes

[6] SWEBOK® is an official service mark of the IEEE
[7] By the UK Ministry of Defence and Praxis High Integrity Systems

- IEEE Computer Society – Guide to the Software Engineering Body of Knowledge[8]
- ISO harmonization proposal for ISO/IEC 15288 and ISO/IEC 12207
- IEEE Std 15288-2004 – Adoption of ISO/IEC 15288:2002 System Life Cycle Processes
- IEEE/EIA 12207.0-1966 – Industry Implementation of ISO/IEC 12207:1995

Early discussions revolved around the first two of these. In the end, to facilitate future work and acceptance, the WG picked a set of generic categories easily mapped to the categories used in a number of standards, curricula, and body of knowledge efforts. These are possibly close to the categories used in the SWEBOK Guide – see Table 1.

Answering the second question, moving from activities to required knowledge, is sometimes easy and sometimes difficult. While one is often tempted to state what the knowledge is "The knowledge of activity X," this is usually unsatisfactory except in quite specific cases.

The WG decided to what extent the knowledge included in this body of knowledge document excludes knowledge identified by the second question but already identified by existing standards for software engineering. In the end, the question of what to leave out or make only a passing mention of was answered pragmatically by what was believed to be known (or at least known of) by most of the members of the intended audience for this guide.

In 2005, the WG began efforts to identify the knowledge needed by the previously identified activities. Considerable effort was required before it finally established a satisfactory scope for the knowledge set. Review by experts not intimately involved with the original draft set was crucial to this effort.

Table 1: Comparison with SWEBOK Guide

SWEBOK Guide	This Document
	Nature of Dangers
	Fundamental Concepts and Principles
	Ethics, Law, and Governance[9]
Software Requirements	Secure Software Requirements …
Software Design	Secure Software Design …
Software Construction	Secure Software Construction …
Software Testing	Secure Software Verification, Validation and Evaluation
Software Quality	
Software Engineering Tools and Methods	Secure Software Tools and Methods
Software Engineering Process	Secure Software Processes
Software Engineering Management	Secure Software Project Management
	Acquisition of Secure Software
Software Maintenance	Secure Software Sustainment
Software Configuration Management	

In addition to the more direct sources of knowledge, the software safety community was found to have more experience in some areas than the software security community; these included correctness and assurance.

The WG has benefited from the experiences of the SWEBOK project. For example, the WG has established only one lead author for each section because the SWEBOK experience showed that arrangements with more co-authors worked poorly.

The WG began with the guiding principle that the text should straightforwardly supply the background and contextual knowledge that persons already possessing good software engineering knowledge need to become

[8] Available at http://www.swebok.org
[9] The SWEBOK Guide does have a subsection 11.1.1 on Software Engineering Culture and Ethics as a Software Quality Fundamental

familiar enough with, in addition to the relevant software-security-related or assurance-related knowledge. Next, the group decided that its initial output should have a level of exposition between that used by the SWEBOK Guide (a prose format), and that used by DoD information assurance standards Committee on National Security Systems (CNSS) Training Standards 4011 and 4012[10] (a list format). Members of the WG felt that lists would be too sparse for an audience that lacked prior knowledge of the area, but that lists were adequate for the lowest level of detail and more amenable to production within the short time frame available for its initial product.

In this first version, the Workforce Education and Training Working Group's aim is to be inclusive in enumerating the knowledge needed to acquire, develop, and sustain secure software including assurance (objective grounds for reduced uncertainty or increasing confidence) of its security properties and functionality. To help industry, government, and academia to target their education and training curricula, as well as to aid self-study efforts, the WG may eventually need to provide indications of what knowledge is needed by different roles and other supplementary materials.

The WG aimed to ensure adequate coverage of requisite knowledge areas in contributing disciplines to enable instructors and professionals in several disciplines, such as software engineering (including its many sub-disciplines), systems engineering, project management, etc., to identify and acquire competencies associated with secure software. Because of this wide coverage and applicability as well as historical reasons, the guide's subtitle includes the phrase "Common Body of Knowledge." Indeed, no individual practitioner would probably ever be expected to know all the knowledge identified in this guide, and the guide is designed so that after reading the first four sections readers can go to the sections of their choice.

How to Read this Document

The first major section is an Introduction that explains the scope and purpose and lists the twelve common references that are cited throughout the document. Immediately following the Introduction, Section 2 covers the dangers secure software systems face. Section 3 covers a number of fundamental terms, concepts, and principles. This section, along with Section 4, which is on Ethics, Law, and Governance, is relevant to all the Sections that follow them. Because it provides one set of inputs to requirements, the short section addressing laws, regulations, policies, and ethics related to secure software systems precedes the more technical sections on requirements, design, construction and verification, validation, and evaluation activities. Sections 5-12 follow a nominal lifecycle, plus management and support, with Section 12 addressing the unique considerations after initial deployment.

[10] See http://www.cnss.gov/full-index.html

Some lifecycle aspects are further detailed in a tools and methods section before a process section covers their arrangement, introduction, and improvement. Section 11 addresses differences in managing secure software developments. These sections cover the additional technical, managerial, and supporting activities and interactions related to producing secure software. Section 13 covers the use of existing software, or software externally produced for an organization, including several kinds of sources, commercial and otherwise. The final section gives tips on using this document for several kinds of intended audiences. Thus, depending on one's interests, one can take a number of paths within this guide, as shown in Figure 1.

All readers could also benefit from Section 5, which is on Requirements, and managers could directly benefit from Section 10, which is on Processes.

Because of the iterative nature of many software production efforts, one cannot always draw a clear boundary between "development" including assurance and "sustainment." In addition, acquisition activities require an understanding of software production and sustainment; thus, to maintain coherence, some overlap is inevitable. This overlap is kept to a reasonable level by not having an in-depth treatment of a sub-area in multiple sections, but rather having the other sections reference one another for the details. This may result in some switching back and forth for the readers, but avoids excessive duplication or conflict.

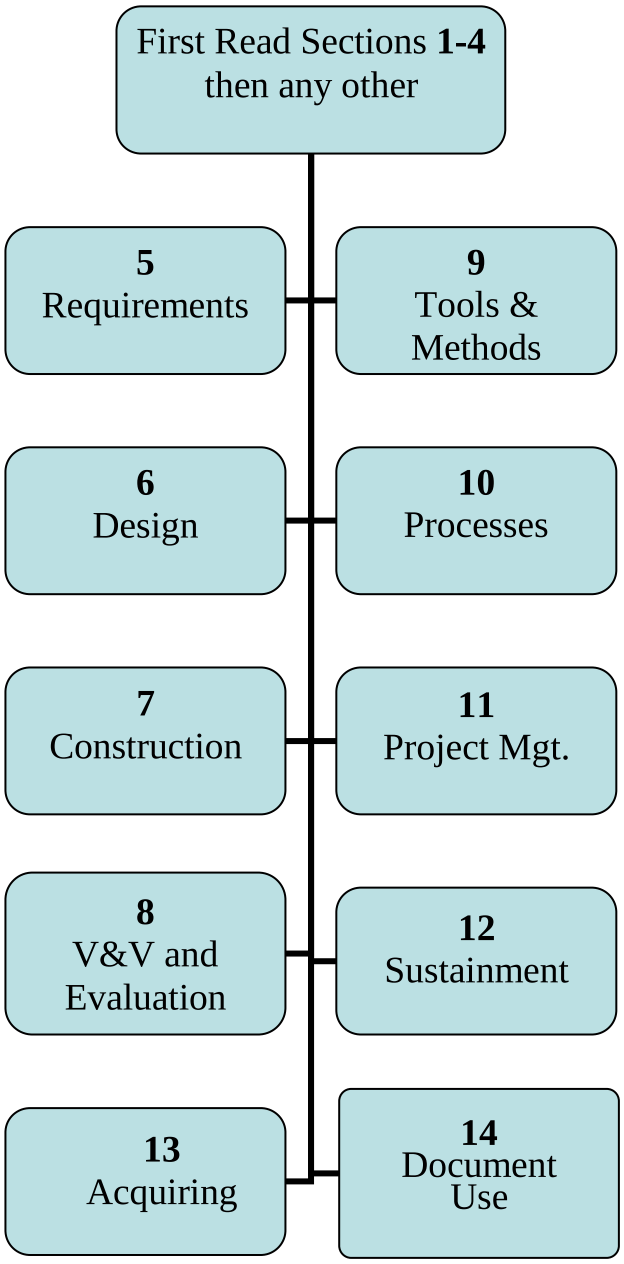


Figure 1: Possible Paths for Readers

A bibliography follows these content sections, listing the items referenced throughout the document and the entries in the lists of further readings (for those who want to pursue areas even further) that appear at the end of each major section. Additionally, the document closes with an Index.

Security Principles of Software Assurance

The sections in this document are organized follow the software development life cycle, providing easy reference for those involved in the development of software. While the primary goal of this document is to aid educators in including software assurance concepts in software development and computer science curricula, it is equally useful for educators wishing to include software assurance concepts in existing information and systems security curricula. As such, this section provides readers an overview of software security principles and where they are addressed.

This set of Software System Security Principles and Guidelines are organized in a fashion that should aid in understanding them individually, as well as how they relate to each other. The organization aims to identify ones that are more basic, abstract, or inclusive than others and provide grounds for arguing completeness, at least at the higher levels. Its underlying purpose is to bring intellectual coherence to an area where items have originated for over thirty-plus years, and authors have tended to provide flat lists, usually organized topically or by importance.[11]

Scope

What various authors have meant by the words “principle” and “guideline” has often been unclear, and this section does not attempt to draw any boundary. Characteristically, each item included in this section is essentially intended to be an “ideal” and usually contains guidance for certain aspects of a secure software system or how activities related to producing secure software should be performed.

Semantically, the principles or guidelines vary from absolute commands to conditional suggestions. However, following the long tradition with computing security started in Saltzer and Schroeder’s seminal 1975 article, the list below is in noun phrases not as commands or suggestions. Generally, you can implicitly place one of the following in front of each noun phrase

- Strive to
- Use
- Find and use

At the higher levels, completeness is a goal, and the items listed aim for intellectual coherence and completeness of coverage. At the middle and lower levels an attempt was made to include a substantial number of items derived from a variety of sources in part to verify the completeness of the higher levels.[12] However, no attempt was made to include every item someone ever labeled a “principle” much less a “guideline”.

Purpose

The ultimate goal of systems or software security is to minimize real-world, security-related, adverse consequences. In practice, this often means to limit, reduce, or manage security-related losses or costs. All secure software issues can fall naturally into three streams and their interaction and can change or vary over time. These streams are labeled:

1. The adverse
2. The system, and
3. The environment

The organization groups around these three streams and within each by their relevant size or nature, plus the streams’ efforts motivated by a number of factors, including a desire for benefits, a dislike of losses, and a yearning for confidence. Occasionally, multiple motivations will exist for the same item. Thus, the principles

[11] These principles are derived from a set of principles or guidelines compiled by Redwine and appearing first topically organized in a 2005 technical report [Redwine 2005].

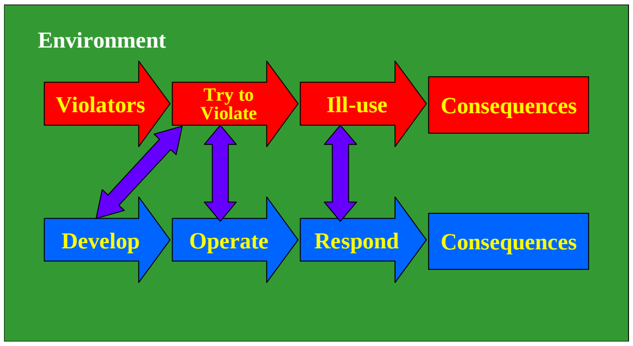
[12] Due to the process used by some of the secondary sources used and the emphasis in this section on organization, some principles or guidelines are individually labeled as to their source of publication (possibly secondary), but most are not. However, an attempt was made to ensure all the sources used be listed in the bibliography. The author welcomes help in better identifying origins.

and guidelines often concern limiting, increasing, reducing, or managing something or the chance or opportunity of something.

Each stream involves sets of entities, potentially overlapping. Usually these can be identified by where their interests or desires lie. For example, stakeholders with interests in adequate/better system security would belong (at least) in the system stream.

Figure 2 gives a simplified conceptual view of a system (blue) and its adverse (red) interaction on field of conflict (green) and suggesting "time," or cause and effect.

Figure 2: The Three Oversimplified



To achieve benefits, avoid losses, and have confidence in these, the entities in each stream need the proper wherewithal and decision-making to use it successfully. Generally, better decisions result from possession of the relevant kinds of information and it having less uncertainty. In addition, decisions are arrived at more quickly and easily (and with less anxiety). Thus they have a need for relevant information with less uncertainty combining to make conclusions with adequately low uncertainty suitable for the decisions they need to make.

They also want the other two streams to facilitate and not harm them. When conflicts are involved, this usually means they also wish to hinder and discourage their opponents and to find, cooperate with, and encourage allies. Thus, the needed wherewithal also depends on other entities and conditions that go beyond one's control and includes tangibles and intangibles, such as morale, mental agility, persistence, and discipline.

Unsurprisingly, the motivations and needs of the entities involved, along with their nature and the nature of their often competitive relationships and their situation, influence the principles and guidelines organized below.

Organization

The principles and guidelines stated below are from the viewpoint of and intended to communicate directly to stakeholders interested in an adequate or better system security. This is often the viewpoint of software system analysts, designers, or their clients.

The principles and guidelines often concern limiting, reducing, or managing the amount or kind of offense, defense, or the environmental attributes or aspects. For brevity, often only one of these words is used, even though more than one might apply. Note that "reduce" only applies if something to reduce already exists, but this term is sometimes used more loosely.

The three streams, each covered in a separate section, are:

- The Adverse
- The System
- The Environment

Each stream is organized into subsections that are concerned with the following:

- Number, size, or amount involved (usually of entities, opportunities, or events)
- Benefits

- Losses
- Uncertainties

Security Principles

This section lists the security principles of software assurance, in their current form, along with references to the sections of this document that address each principle. These security principles represent a first attempt at providing an alternate structure for viewing the common body of knowledge; as such, they may change in future versions of this document. For those interested in how this document maps to these principles, Section 15 provides a matrix illustrating how they relate to one another.

Table 2: The Security Principles of Software Assurance

Principle	Section
1 The Adverse	2.3.1-2.3.2
1.1 Adversaries Intelligent and Malicious	2.3-2.4, 2.6
1.1.1 Potential for significant benefit to attacker will attract correspondingly capable attacker	2.3-2.4, 2.6
1.2 Limit, Reduce, or Manage Benefits to Violators or Attackers	6.3.2
1.2.1 Unequal attacker benefits and defender losses	6.3.2
Attacker's context is different	6.3.2
Think like an attacker	3.4.13, 6.3.2
1.3 Increase Attacker Losses	3.4.12
1.3.1 Increase expense of attacking	3.4.7, 3.4.12
1.3.2 Increase attacker losses and likely penalties	3.4.12
1.3.2.1 Adequate detection and forensics	3.4.12, 6.17
Follow and provide support for legal processes, which lead to successful prosecution	3.4.12
1.4 Increase Attacker Uncertainty	6.15
1.4.1.1.1 Conceal information useful to attacker	6.15
1.4.1.1.2 Exploit deception	6.15
1.5 Limit, Reduce, or Manage Set of Violators	
1.5.1 Users	4.2
1.5.1.1.1 Ensure users know proper use	
1.5.1.1.2 Ensure users know what is abuse	
1.5.1.1.3 Reduce number of malicious insiders	
1.5.2 Limit, reduce, or manage set of attackers	
1.5.2.1 Limit, remove, and discourage aspiration to be attacker	4.2, 4.3
Prevent or discourage each step towards becoming attacker	
Hinder and breakup attacker alliances, association networks, and communications	
Discourage others motivating someone to attempt to violate	
1.6 Limit, Reduce, or Manage Attempted Violations	
1.6.1 Discourage violations	
11.6..1 By Non-malicious Humans	

Principle	Section
1.6.1.1.1 Ease secure operation	
1.6.1.1.2 Acceptable Security	
1.6.1.1.3 Psychological Acceptability	3.4.6, 6.18
1.6.1.1.4 Ergonomic Security	
1.6.1.1.5 Sufficient User Documentation	
1.6.1.1.6 Administrative controllability	
1.6.1.1.7 Manageability	
By Malicious Humans	
1.6.1.2.1 Exploit deception and hiding	
1.6.1.2.2 Appear less attractive than other potential victims	
1.6.2 Limit, reduce, or manage violators’ ease in taking steps towards fruitful violation	
Detect scouting and information collection	
Hinder entities suspected of bad intent	
1.6.2.2.1 Block sources of prior attempts	
1.6.3 Increase losses and likely penalties of attempted attacks	
Detect attempted attacks	
2 The System	
2.1 Limit, Reduce, or Manage Violations	3.3.5
2.1.1 Limit, reduce, or manage origination or continuing existence of opportunities or possible ways for performing violations throughout system’s lifecycle/lifespan	3.3.5, 6.14
2.1.1.1 Accurate Identification	3.3.5, 6.8.1, 6.14
2.1.1.1.1 Positive Identification	3.3.5, 6.14
2.1.1.1.2 Adequate authentication	3.3.5, 6.14
2.1.1.1.3 Valid, tamper-proof identification-related data	3.3.5, 6.14
Separate Identity from Privilege	3.3.5, 3.4.1, 3.4.5, 6.8.1, 6.8.2, 6.14
Positive Authorization	3.3.5, 3.4.3, 3.6.6, 6.8.2, 6.14
Least Exposure	3.3.5, 6.14
2.1.1.4.1 Broadly Eliminate Exposure	3.3.5, 6.14
2.1.1.4.1.1 Isolation from Source of Danger	3.3.5, 6.14
2.1.1.4.1.1.1 Isolation of user groups	3.3.5, 6.14
2.1.1.4.1.1.1.1 Isolate publicly accessible systems from mission-critical resources (e.g., data, processes).	3.3.5, 6.14
2.1.1.4.1.1.2 Domain isolation	3.3.5, 6.14
2.1.1.4.1.2 Continuous Protection of Assets	3.3.5, 6.14
2.1.1.4.1.3 Complete Mediation of Accesses	3.3.5, 3.4.2, 6.14
2.1.1.4.1.4 Separate Policy from Mechanism	3.3.5, 6.14
2.1.1.4.1.5 Least Privilege	3.3.5, 3.4.1, 6.14
2.1.1.4.1.6 Tamper Proof or Resistant	3.3.5, 6.14, 6.16
2.1.1.4.2 Eliminate in Each Situation	3.3.5, 6.14
2.1.1.4.2.1 Secure defaults	3.3.5, 3.4.3, 6.14

Principle	Section
2.1.1.4.2.2 Secure Failure	3.3.5, 3.4.3, 6.14
2.1.1.4.2.3 Secure Shutdown	3.3.5, 6.14
2.1.1.4.2.4 Secure Disposal	3.3.5, 6.14
2.1.1.4.2.5 Secure Diagnosis	3.3.5, 6.14
2.1.1.4.2.6 Secure System Modification	3.3.5, 6.14
2.1.1.4.2.7 Trusted Communication Channels	3.3.5, 6.9, 6.14
2.1.1.4.2.8 Limited Access Paths	3.3.5, 6.14
2.1.1.4.2.9 Minimize Sharing	3.3.5, 6.14
2.1.1.4.2.10 Least Common Mechanism	3.3.5, 3.4.4, 6.14
2.1.1.4.2.11 Limit Trust	3.3.5, 6.14
2.1.1.4.2.11.1 Trust Only Components Known to be Trustworthy	3.3.5, 6.14
2.1.1.4.2.11.2 Hierarchical Trust for Components	3.3.5, 6.14
2.1.1.4.2.11.2.1 Do not invoke less trusted programs from within more trusted ones.	3.3.5, 6.14
2.1.1.4.2.11.2.2 Do not invoke untrusted programs from within trusted ones.	3.3.5, 6.14
2.1.2 Hierarchical Protection	3.3.5, 6.14
2.1.3 Learn, Adapt, and Improve	3.3.5, 6.14
2.1.4 Limit, reduce, or manage undetected violations	3.3.5, 6.14
2.1.4.1.1 Detection of Violations	3.3.5, 6.14
2.1.4.1.1.1.1 Effective Detection	3.3.5, 6.14
2.1.4.1.1.1.2 Self Analysis	3.3.5, 6.14
2.1.4.1.1.1.3 No Need to Detect	3.3.5, 6.14
2.1.4.1.1.1.3.1 Universal Action	3.3.5, 6.14
2.1.4.1.1.1.3.2 Inevitable Action	3.3.5, 6.14
2.1.4.1.2 Recording of Compromises	3.3.5, 6.14
2.1.5 Limit, reduce, or manage lack of accountability	3.3.5, 3.7.4, 6.14
2.1.5.1.1 Accountability and Traceability	3.3.5, 3.7.4, 6.14
2.1.5.1.2 Support Investigation of Violations	3.3.5, 3.7.4, 6.14
2.1.5.1.3 Accurate Clock	3.3.5, 3.7.4, 6.14
2.1.6 Limit, reduce, or manage violations unable to respond to acceptably or learn from	3.3.5, 6.14
2.1.7 Defense in Depth	3.3.5, 3.4.12, 6.14
2.1.7.1.1 Design to defend perfectly, then assume this defense will fail and design to defend after initial security violation(s)	3.3.5, 3.4.12, 6.14
2.1.7.1.2 Diversity in Defenses	3.3.5, 3.4.12, 6.14
2.1.7.1.3 Measures Encounter Countermeasures	3.3.5, 3.4.12, 6.14
2.1.7.1.4 Survivable Security	3.3.5, 3.4.12, 6.14
2.1.7.1.5 Secure Recovery	3.3.5, 3.4.12, 6.14
2.2 Avoid Adverse Effects on System Benefits	
2.2.1 Authorizations Fulfill Needs and Facilitate User	
2.2.2 Encourage and ease use of security aspects	3.4.6

Principle	Section
Acceptable Security	3.4.6
Psychological Acceptability	3.4.6
Sufficient User Documentation	3.4.6
Ergonomic Security	3.4.6
Quickly Mediated Access	3.4.6, 3.4.8
Ease secure operation	3.4.6, 3.4.8
2.2.2.6.1 Administrative controllability	3.4.6
2.2.2.6.2 Manageability	3.4.6
2.2.3 Articulate the desired characteristics and tradeoff among them [jabir 1998]	3.4.14, 3.5.1, 5.2, 5.3.3
2.2.4 Economic Security	3.5.1, 5.3.2
Efficiently Mediated Access	3.4.8
2.2.5 High-Performance Security	
2.2.6 Provide Privacy Benefit	
2.2.7 Provide Reliability	3.3.1
2.3 Limit, Reduce, or Manage Security-related Costs	
2.3.1 Limit, reduce, or manage security-related adverse consequences	
Exclusion of Dangerous Assets	3.3.4
Retain minimal state	
Tolerate Security Violations	3.3.5, 6.3.2, 6.14
2.3.1.3.1 • Limit damage	3.3.5, 6.3.2, 6.14
2.3.1.3.2 • Be resilient in response to events	3.3.5, 6.3.2, 6.14
2.3.1.3.3 • Limit or contain vulnerabilities' impacts	3.3.5, 6.3.2, 6.14
2.3.1.3.4 • Choose safe default actions and values	3.3.5, 6.3.2, 6.14
2.3.1.3.5 • Self-limit program consumption of resources	3.3.5, 6.3.2, 6.14
2.3.1.3.6 • Design for survivability [Ellison 2003]	3.3.5, 6.3.2, 6.14
2.3.1.3.7 • Fail securely	3.3.5, 6.3.2, 6.14
2.3.1.3.7.1 • Ensure system has a well-defined status after failure, either to a secure failure state or via a recovery procedure to a known secure state [Avizienis 2004]	3.3.5, 6.3.2, 6.14
12.3..4 Recover	6.3.2, 6.14
2.3.1.4.1.1 Recover rapidly•	6.3.2, 6.14
2.3.1.4.1.2 Be able to recover from system failure in any state	6.3.2, 6.14
2.3.1.4.1.3 • Be able to recover from failure during recovery (applies recursively)	6.3.2, 6.14
2.3.1.4.1.4 • Make sure it is possible to reconstruct events	3.4.11, 3.7.4, 6.3.2, 6.14
2.3.1.4.1.5 o Record secure audit logs and facilitate periodical review to ensure system resources are functioning, confirm reconstruction is possible, and identify unauthorized users or abuse	3.4.11, 3.7.4, 5.2.13, 6.3.2, 6.14
2.3.1.4.1.6 o Support forensics and incident investigations	3.4.11, 3.7.4, 5.2.13, 6.3.2, 6.14, 6.17
2.3.1.4.1.7 o Help focus response and reconstitution efforts to those areas that are most in need	6.3.2, 6.14
Avoid Single-Point Security Failure	3.4.12

Principle	Section
2.3.1.5.1 • Eliminate "weak links"	
2.3.1.5.2 Avoid Multiple Losses from Single Attack Success	6.3.2
2.3.1.5.3 Separation of Privilege	3.4.5
2.3.1.5.3.1 Separation of duties	3.4.5
2.3.1.5.4 Defense in Depth	3.4.12
2.3.1.5.4.1 • Implement layered security (no single point of vulnerability).	3.4.12
Allocation of Defenses according to Consequences	3.4.7, 3.5.1, 5.3.1-5.3.4
2.3.1.6.1 Inverse Modification Threshold	3.4.7
2.3.1.6.2 Work Factor	3.4.7
2.3.2 Limit, reduce, or manage security-related developmental and operational expenses	
Ease Downstream Security-related Activities	
2.3.2.1.1 Ease Preserving Security while Performing Changes in Product and Assurance Case	
2.3.2.1.2 Ease (cost-effective and timely) Certification and Accreditation	5.2.13
2.4 Limit, Reduce, or Manage Security-related Uncertainties	
Those of stakeholders with interests in adequate/better system security	
2.4.1 Limit, reduce, or manage security-related unknowns	
2.4.2 Limit, reduce, or manage security-related assumptions	
An assumption must have a good reason	
Avoid critical assumptions	
2.4.3 Limit, reduce, or manage unpredictability of system behavior	
Analyzability	3.4.10
2.4.4 Limit, reduce, or manage consequences or risks not addressed in assurance case	
2.4.5 Limit, reduce, or manage consequences or risks related to uncertainty	
Risk Sharing	
2.4.6 Increase Assurance Regarding Product	3.3.6
System Assurability	3.3.6
Reduce Danger from other software or systems	3.3.6, 3.4.5
2.4.6.2.1 Avoid and workaround environment's security endangering weaknesses	3.3.6, 3.4.5
2.4.6.2.2 System does what the specification calls for and nothing else	3.3.6, 3.4.5
Reduce Complexity	3.3.6 3.4.8
2.4.6.3.1 Make Small	3.3.6 3.4.8
2.4.6.3.1.1 Minimized Security Elements	3.3.6 3.4.8
2.4.6.3.2 Simplify	3.3.6 3.4.8
2.4.6.3.2.1 Control complexity with multiple perspectives and multiple levels of abstraction	3.3.6 3.4.8
2.4.6.3.2.1.1 Use information hiding and encapsulation	3.3.6 3.4.8

Principle	Section
2.4.6.3.2.1.2 Clear Abstractions	3.3.6 3.4.8
2.4.6.3.2.1.3 Partially Ordered Dependencies	3.3.6 3.4.8
2.4.6.3.3 Straightforward Composition	3.3.6 3.4.8
2.4.6.3.3.1 Trustworthy Components	3.3.6 3.4.8
2.4.6.3.3.2 Self-reliant Trustworthiness	3.3.6 3.4.8
2.4.6.3.4 To improve design study previous solutions to similar problems [jabir 1998]	3.3.6 3.4.8
2.4.6.3.4.1 Use known security techniques and solutions	3.3.6 3.4.8
2.4.6.3.4.2 Use standards	3.3.6 3.4.8
Change Slowly	3.3.6
2.4.6.4.1 Use a stable architecture	3.3.6, 6.7
2.4.6.4.1.1 To eliminate possibilities for violations – particularly of information flow policies	3.3.6, 6.7
2.4.6.4.1.2 To facilitate achievement of security requirements and evolution	3.3.6, 6.7
2.4.6.4.1.3 Amendable to supporting assurance arguments and evidence	3.3.6, 6.7
Assure Security of Product	3.3.6, 8.2
2.4.6.5.1 Create and Maintain an Assurance Case	3.3.6
2.4.6.5.2 Ensure security preserving composition at all levels of detail	3.3.6
2.4.6.5.3 Secure Distributed Composition	3.3.6
2.4.6.5.4 Ease production of an accompanying assurance case for the security preserving correctness of compositions	3.3.6
2.4.6.5.5 Design to ease traceability, verification, validation, and evaluation	3.3.6
2.4.6.5.6 Analyzability	3.3.6, 3.4.10
2.4.6.5.7 Chain of Trust	3.3.6
2.4.6.6 Use Production Process and Means that Ease and Increase Assurance	3.3.6, 3.4.9, 3.6.5, 3.6.8, 10
2.4.6.6.1 Ease creation and maintenance of an assurance case	3.3.6, 3.4.9, 3.6.5, 3.6.8, 10
2.4.6.6.2 Use Repeatable, Documented Procedures	3.3.6, 3.4.9, 3.6.5, 3.6.8, 10
2.4.6.6.3 Procedural Rigor	3.3.6, 3.4.9, 3.6.5, 3.6.8, 10
2.4.6.6.4 Engineering Rigor	3.3.6, 3.4.9, 3.6.5, 3.6.8, 10
2.4.6.6.5 Open Design	3.3.6, 3.4.9, 3.6.5, 3.6.8, 10
2.4.6.6.5.1 Review for use of design principles (and guidelines	3.3.6, 3.4.9, 3.6.5, 3.6.8, 10
2.4.6.6.6 Chose notations and tools that facilitate achieving security and its assurance	3.3.6, 3.4.9, 3.6.5, 3.6.8, 10
2.4.6.6.7 Have expertise in technologies being used and application domain	3.3.6, 3.4.9, 3.6.5, 3.6.8, 10
2.4.6.6.8 Avoid Known Pitfalls	3.3.6, 3.4.9, 3.6.5, 3.6.8, 10
2.4.6.6.8.1 Avoid common errors and vulnerabilities	3.3.6, 3.4.9, 3.6.5, 3.6.8, 10
2.4.6.6.8.2 Avoid and workaround tools’ security endangering weaknesses	3.3.6, 3.4.9, 3.6.5, 3.6.8, 10
2.4.6.6.8.3 Avoid non-malicious pitfalls	3.3.6, 3.4.9, 3.6.5, 3.6.8, 10
Continuous Risk Management	3.3.6, 3.6.8

Principle	Section
2.4.6.7.1 Consider security or assurance risks together with other risks	3.3.6, 3.6.8
3 The Environment	5.2.4, 5.2.5
3.1 Nature of Environment	5.2.4, 5.2.5, 6.4.1
3.1.1 Security is a system, organizational, and societal problem	5.2.4, 5.2.5, 6.4.1
3.2 Benefits to Environment	5.2.4, 5.2.5
3.2.1 Do not cause security problems for systems in the environment	5.2.4, 6.3.2
3.2.2 Learn, Adapt, and Improve Organizational Policy	5.2.4, 5.2.5
3.3 Limit, Reduce, or Manage Environment-Related Losses	5.2.4, 6.3.2, 6.14
3.3.1 Avoid assumptions about environment	5.2.4, 6.3.2, 6.4.1, 6.14
Make only weak non-critical assumptions about environment	5.2.4, 6.3.2, 6.4.1, 6.14
3.3.2 Trust only services or components in environment known to be trustworthy	5.2.4, 6.3.2, 6.14
3.3.3 More trustworthy components do not depend on less trustworthy services or entities in environment	5.2.4, 6.3.2, 6.14
Do not invoke untrusted services from within system.	5.2.4, 6.3.2, 6.14
3.3.4 Avoid dependence on protection by environment	5.2.4, 6.3.2, 6.14
3.4 Avoid Environment-Related Uncertainties	5.2.4, 5.2.5
3.4.1 Do not rely only on obfuscation or hiding for protection from entities in environment	5.2.4, 5.2.5
3.4.2 Need adequate assurance for dependences	5.2.4, 5.2.5

Table of Contents

Part 1: Introduction

1 Introduction

1.1 Purpose and Scope

For persons with knowledge of software but not security, this document introduces them to the surface of the field software-security-related[1] knowledge and points to references for further knowledge. It supplies the background they need to meaningfully recognize the topic a reference covers and which references might be of interest. Given this, while a guide and not a textbook, this document's text can function as a high-level introduction -- and some may choose to use it that way. However, in its primary role, this high-level description and context combines with the numerous references to serve as a guide for educators and trainers as well as others to the software-security-related knowledge relevant to their work (no job requires it all).

The primary audiences for this guide are educators and trainers who can use this guide to help identify both appropriate curricular content and references that detail it. Other audiences include evaluators and testers, acquisition personnel, program managers, and standards developers for whom it can help provide a better appreciation of the breadth of subject matter and issues involved. In addition, studious practitioners could use it to obtain overviews or to guide their learning. In turn, this document's evolution can benefit from obtaining feedback from users and experts on how it might best be improved or supplemented to become more useful.

This guide concentrates on the additional knowledge associated with producing and obtaining secure software.[2] It mentions only in passing physical, operational, communication, hardware, and personnel security. These are important topics in cyber security but are outside the scope of this guide. Concentrating on software still covers the much of the technical security vulnerabilities being exploited today.

1.2 Motivation

The need for a workforce with better software-security-related skills is clear. The substantial costs of a vulnerability3 to software producers result from a number of activities – initial testing, addressing multiple versions (for other platforms and prior versions still in use), patching, remediation testing, and distribution, as well as negative impact on reputation.4 Thus, producers can suffer serious costs and consequences.

Applying a patch can cost large enterprises tens of millions of dollars. One result of this expense is many computers are not patched and become vulnerable to a known vulnerability. Increasingly, losses of confidential

[1] The terms "software security," "software system security," and "software-security-related" are used as inclusive terms. "Software security," "software system security," and the more awkward but explicitly inclusive phrase "software-security-related" vary only in context and emphasis. They certainly include security software and the security of software but also encompass the security of all kinds of computing resources associated with or accessed by software and topics related to secure software including possible consequences and related uncertainties. The reader needs to avoid confusion among terms such as software security, security software, secure software, and the security of software; in this document usage and context hopefully allow the reader to avoid this confusion.

[2] This document uses the phrase "secure software" in an inclusive way to refer to software or software product, which results in or might affect security. This document uses "security software" for software that performs security functionality.

[3] Ignoring definitional complications, the following definition is useful. "Vulnerability: A flaw or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy." *SANS Glossary of Terms Used in Security and Intrusion Detection,* SANS Institute, 2001

[4] During an earlier period, a study showed that among a set of major vendors announcing a product vulnerability was followed by an average 0.6 percent fall in stock price, or an average $860 million fall in the company's value. This article appears in *new scientist* magazine issue, 25 June 2005, written by Celeste Biever http://www.newscientist.com. Today, however, the market probably has already factored in expectations of these announcements.
This and similar questions have been explored at Workshops on Economics of Information Security since 2002. http://infosecon.net/workshop/index.php.

data result in identity theft and significant fraud losses to firms and customers. Changes in laws and regulations, such as California SB 1386 requiring California citizens whose individual data is compromised to be notified,[5] now result in such data losses becoming public, and established firms have lost business.[6]

In addition to the actual costs for producers and users generated by software-security-related problems, both suffer opportunity costs because valuable resources could be producing added value rather than doing rework and patching.

The problem is not only the result of attempted attacks and insertion of malicious software from both inside and outside organizations, but also other issues as well. An overwhelming majority of security incidents result from defects in software requirements, design, code, or deployment. This combination of attacks and defects means that today's security problems involving computers and software are frequent, widespread, and serious; and since cyber security is an imperative concern for Department of Homeland Security (DHS) and Department of Defense (DoD), initially the choice was made to concentrate their Software Assurance efforts on security and to develop this guide.

Although their joint efforts are recent, the DoD and the DHS Software Assurance initiatives and efforts have encompassed software safety and security and combined a number of disciplines. Currently, the efforts are concentrating on achieving and assuring security properties. The Software Assurance Workforce Education and Training Working Group, composed of government, industry, and academic members, produced this document.

By identifying and providing references for the additional knowledge needed to develop, maintain or sustain,[7] and reuse or acquire either custom or off-the-shelf (OTS) secure software beyond that required to produce and assure software where safety and security are not concerns; the Working Group is taking a first step toward achieving adequate education and training in this area. The knowledge identified spans a number of roles, so no one practitioner would be expected to know it all. Therefore, this guide is designed so that after reading the first four sections readers can go to the sections of their choice.

While the public prominence of software-security-related problems is a recent development, the combination of software and security has long been studied, and, while open questions remain, considerable bodies of relevant research and practice exist.

1.3 Audience

Produced by the Software Assurance Workforce Education and Training Working Group, this compilation is a needed preliminary step toward addressing the issues related to achieving adequate United States (US) education and training on software-security-related knowledge and skills. These issues include the skill shortages within government and industry and curriculum needs within universities, colleges, and trade schools.

The ultimate goal for this document is to improve the software development workforce, as mentioned above; yet, the intended primary audiences go beyond software practitioners. While the most emphasis is on the first two, the intended audiences and the document-related goals for them include:

- Educators – influence and support curricula

[5] See section 4 for more about this and other legal considerations.

[6] ChoicePoint's stock falling 20 percent in the period an incident was disclosed shows another potential impact, even though in this incident, losses resulted from deception, not necessarily faulty software.

[7] While not an attempt to have the software community cease its overly broad use of the term "maintenance," nevertheless, in the interest of better usage, this report generally avoids this overly broad use by using the terms "sustain" or "sustainment" when actions such as adding new functionality are included.

- Trainers – extend/improve contents of training of current workforce
- Acquisition personnel – aid in acquiring (more)[8] secure software
- Evaluators and testers – recognize any relevant content beyond that currently in their evaluations and consider its implications for evaluations of persons, organizations, and products
- Standards developers – encourage and facilitate inclusion of security-related items in standards
- Experts – supply feedback: modifications or validation to document authors
- Practitioners – provide a guide for learning or a high-level introduction to field
- Program managers – help to understand supplying or obtaining secure software and its assurance, including possible approaches, risks, prioritization, and budget

Educators and trainers wishing to develop specific curricula and curricular material will benefit from this comprehensive summation of the topics, facts, principles, and practices within software security including assurance. These represent a body of knowledge (BOK) that provides a benchmark, which will let educators and trainers target and validate detailed learning objectives, develop coherent instructional plans, and structure their teaching and evaluation processes to effectively teach specific content across the wide range of relevant audiences and roles. To be fully effective, these need to be built upon a prerequisite solid foundation in software engineering and quality [Redwine 2004] [Abran 2004].

Evaluators, standards developers, and practitioners including testers will benefit from being able to identify weaknesses or gaps in their personal knowledge that they might address by additional study as well as weaknesses in their organization’s processes or products. They will be able to judge their performance against minimum to advanced levels of practice. Finally, aided by the numerous references, they will be able to tailor specific operational recommendations and processes and to ensure the development, acquisition, deployment, operation, and sustainment or maintenance of (more) secure software within their professional setting.

While the text of this document provides broad coverage, a reader interested in self-education in secure software engineering or acquisition of secure software will, of course, find this guide leads to more depth of knowledge when reading the references.

Thus, this document’s text can provide a high-level introduction, and this text combined with its numerous references can serve to guide, inspire, and support readers in performing their work.

After covering the foundation material in the next three major sections of this document, read the portions of this guide relevant to you, consider what it means for you, and decide if you can use it in its present form or transform it into a form useful to you. You might choose to use this guide or decide to wait until the US government’s software assurance efforts or others expand its contents into a product more useful for your purposes – e.g., one including competencies or mappings into existing curricula. In any case, feedback from you is important to improving future versions and future related products.

Specific initial targets for influence include universities and training organizations willing to be trial users or early adopters, and influencing content of the Institute of Electrical & Electronics Engineers (IEEE) Computer Society’s Guide to the Software Engineering Body of Knowledge [Abran 2004][9].

[8] Following a common practice (used for example by Microsoft) this document uses “more secure software” to mean software better than the prior version or better than would other otherwise be acquired, produced or used, but not necessarily providing a high level of security.. Such improvement reflects the current goals of many involved in software.

[9] Other bodies of knowledge that are potential targets are listed in Section 14 on use of this document.

1.4 Secure Software

After discussing the meaning of security, this section describes the scope of the "additional" knowledge included in this report, outlining what is needed but not available elsewhere. It covers

- Software-security-related properties
- Secure software knowledge
- Boundaries of document scope
- Related subject matter areas

1.4.1 Security Properties

The security-related objectives of software are the preservation of security properties, including confidentiality, integrity, and availability (CIA); and accountability if their preservation fails. Confidentiality, preventing unauthorized disclosure, and integrity, preventing unauthorized alteration, require mechanisms to firmly establish identities – authentication – and to allow only authorized actions – e.g., access control. Preserving availability includes preventing unauthorized destruction and ensuring adequate access or service.

Accountability includes the ability to later reestablish the acts that occurred and their related actors and ensuring relevant actors are unable to deny an act occurred – non-repudiation. Thus, software system security is a question of systems properties. The items being protected by the preservation of these properties may be information, software, executing processes, or other computing resources.

Consequently, security is not just a question of the security mechanisms or functionality; the properties desired must be shown to hold wherever required throughout the system – e.g. the security functionality cannot be bypassed anywhere. In other words, security properties are systems properties.[10]

Security is an omnipresent issue throughout the software lifecycle. [McGraw 2003] Likewise, potential security attacks or difficulties exist throughout the lifecycle of software systems and can result in a variety of security-related requirements for the software and its environment. Deciding upon the required security properties may be intertwined with decisions on the extent of security-oriented development effort and functionality into an overall risk management decision.[11]

Neither in the physical world nor in software can one absolutely guarantee security. Thus, when this guide speaks of "secure software," the actual meaning is "highly secure software realizing – with justifiably high confidence but not guaranteeing absolutely – a substantial set of explicit security properties and functionality including all those required for its intended usage." [Redwine 2004, p. 2] One can also state this in a negative way as "justifiably high confidence that no software-based vulnerabilities exist that the system is not designed to tolerate". Not withstanding these definitions' emphasis on high security, the material in this document covers issues important not just for producing highly secure software but ones important to the many organizations under pressure to produce software merely more secure than the current version.

[10] These are often characterized as emergent properties where an emergent property is one emerging after the system is composed, i.e., a property of the system as a whole.

[11] Much of the risk management concerns in this document address decisions by or effecting software producers. Unfortunately, often current buyers of software do not know what "risk management decisions" the producer made and are thereby hampered in making their decisions.

1.4.2 Secure Software Knowledge

Secure software knowledge falls naturally into three categories: the nature of attacks, how to defend, and the computing system's environment in which the conflict takes place. The guide also concerns itself with secure software acquisition. Thus, the coverage includes

- Attack: entities, objectives, strategies, techniques, and effects
- Defense: roles, objectives, strategies, techniques, and how to develop and sustain software to defend and survive
- Arena: aspects of environments for software across its lifespan with security implications

Because of third-party software's important role in today's software systems, obtaining it has a dedicated section.

- Acquiring secure software: security issues in reuse, selection, and acquisition of software

Defending software includes the entire secure software system lifecycle with approaches and pitfalls from concept through disposal covering all aspects:

- Technical
- Managerial
- Work environment and support

The bulk of needed engineering knowledge remains unchanged throughout original development and sustainment, as well as whether the system is "acquired" or developed for in-house use (e.g., the knowledge of ethics, and legal and regulatory concerns). Sustainment and acquisition both have unique aspects. In addition, the acquisition phase has a large impact on the success of the system, and much of the funds expended on software go toward sustainment. Therefore, in addition to a Preface and this Introduction, this guide contains five main parts:

1. Dangers – attacks and non-malicious acts in conflict that may threaten security – Section 2
2. Fundamental Concepts and Principles; and Ethics, Law, and Governance – needed knowledge across all areas – Sections 3 and 4
3. Development and Sustainment – engineering, management, and support directly involved in producing and sustaining secure software – Sections 5-12
4. Acquiring – reusing, purchasing, or otherwise acquiring secure software – Section 13
5. Use of this Document – tips, thoughts, and suggestions to help the various audiences use this document in their work, including some tips from the first limited usage of drafts – Section 14

Thus, the organization of this guide reflects this division. The Preface discusses the special interest paths a reader can take when reading this guide. Sections 1-4 (this Introduction and parts 1 and 2 above) are essential prerequisites for all readers before going to the later sections that interest them.

1.4.3 Boundaries of Document Scope

Because knowledge for software development is recorded elsewhere in bodies of knowledge, de facto and official standards, curricula standards, and textbooks, this document only covers the "difference" between software produced without concern for security or safety versus with concern for security. It presumes a baseline of currently generally accepted knowledge and identifies a new layer with an upper boundary above this baseline that includes additional the knowledge needed for secure software. In practice, the working group (WG) tried not to presume more of readers than could be reasonably expected. This causes the presumed knowledge about “unsecured” software to vary across the three parts – development[12], sustainment[13], and acquisition[14].

The “additional” secure software knowledge identified includes:

- Knowledge for doing the most rigorous approaches, including using formal methods
- Some knowledge relevant to less rigorous approaches
- An outline of the knowledge often relevant for dealing with legacy software deficient in security
- Knowledge that spans multiple technologies, application domains, or vendors
- Proven knowledge – although the degree of prior use may vary
- Knowledge currently useful or expected to be needed or useful in the near future

While this last bullet calls for well-informed judgments, in a rapidly changing field all such compendiums as this must look ahead to avoid being outdated when published. On the other hand, experts should find little here to surprise them – in part, because much of the knowledge necessary has existed for more than 15 years.

Knowledge descriptions do not cover details of particular products or operational activities since such coverage exists elsewhere. Examples of areas not addressed in detail include

- Specific details of Windows, Unix, Linux, router, and telephone switching operating systems
- Static and dynamic routing tables, network operations, and TCP/IP protocol operations, and routing protocols as they relate to traffic flow on the internet over access provided by common carriers
- The Java-oriented J2EE framework and Microsoft’s .NET
- Rules of evidence or search and seizure, surveillance laws, and investigative methods and procedures

In practice, the WG tried not to presume more of readers than could be reasonably expected and to organize the document recognizing that knowledge from several disciplines is necessary and, as stated above, that it covers knowledge used by a number of roles and no one practitioner would be expected to know all of it.

[12] Any development knowledge presumed can be found in these three sources:

- Knowledge identified in the Guide to the Software Engineering Body of Knowledge (SWEBOK) [Abran 2004]
- The knowledge contained in the SW-CMMI document through Level 5 – while not a body of knowledge, significant knowledge is in the document text itself

Knowledge identified as required in the ACM/IEEE-CS undergraduate curriculum for Software Engineering [ACM 2004]

[13] Any knowledge taken for granted in the Sustainment section can be found by the reader in

- SWEBOK Guide Maintenance chapter
- Relevant sections of ISO/IEC 12207 Information Technology - Software Life Cycle Processes
- ISO/IEC Std. 14764 Information Technology – Software Maintenance

[14] Basic knowledge presumed in acquisition area is knowledge of how to reuse or acquire “unsecured” software and can be found in

- Coverage of reuse throughout SWEBOK Guide [Abran 2004]

IEEE Std 1517-1999, IEEE Standard for Information Technology-Software Life Cycle Processes-Reuse Processes, IEEE, 1999. [IEEE 1517-99]

- Official regulations, standards and guidelines; and in the curricula of the National Defense University or similar master’s level civilian universities.

1.4.4 Related Areas

While the term “software assurance” potentially refers to the assurance of any property or functionality of software, the emphases currently encompass safety and security and integrate practices from multiple disciplines, while recognizing that software must, of course, be satisfactory in other aspects as well, such as usability and mission support.

Related Areas

- Systems engineering
- Security engineering
- Quality
- Computer engineering
- Network security
- Personnel security
- Operational security
- Criminology, legal system, and law and regulatory enforcement
- Intelligence
- Counter-intelligence
- Military strategy and tactics
- Usability engineering
- Executive management

As mentioned previously, the knowledge to develop, sustain, and acquire “unsecured” software is not included in this guide. In addition, a number of areas related to, or included in, the knowledge needed to produce secure software are not included in detail. Some underlying fundamentals are simply presumed, such as

- Discrete mathematics
- Probability theory
- Organizational studies

Three related areas have a number of aspects mentioned, but are not covered in their entirety.

- Systems engineering
- Security engineering
- Quality

In many situations, relevant concerns in secure software development, sustainment, and acquisitions could easily be labeled “systems” concerns. With this in mind, this document sometimes uses the term “software system.”

Other more directly related areas include those in the text box on the right. For these, aspects that are directly relevant are noted, although sometimes briefly or at a high level of abstraction. Finally, domain knowledge of the area applied, be it operating systems or banking, is quite important in practice, but out of the scope of this guide.

While always recognizing the limits of their own expertise, persons producing secure software need to be able to communicate with persons in multiple, related disciplines as well as with stakeholders of many kinds, including users and owners of protected assets and the owners and operators of the software and systems involved.

1.5 Selection of References

Choosing references confronted problems not often encountered within other bodies of knowledge. With a few exceptions such as secure coding, secure software engineering and assurance do not have a set of comprehensive or specialty textbooks that can be used as references. Though the field started in the 1960s, it suffered from the lack of commercial interest during the 1990s and early 2000s. For example, possibly the last introductory text on highly secure software systems was [Gasser 1988].

Thus, today, many of the most relevant references pre-date 1990 or appear in conferences, workshops, technical journals, or even less accessible places. Efforts were made to provide good, recent references, but many older references are still the relevant ones. For this reason, this guide also references several works that might appear, on the surface, to be “old” or “obsolete”, but which in fact remain the best or most useful. Even a topic’s most useful references for educators and trainers are not necessarily instructional materials or practitioner-oriented – but still useful. In addition, because many of the practices established in the discipline of software safety are being adapted or extended and applied to achieve software security objectives, several software safety references are also included in this guide.

While the authors would have preferred to fill readers needs by directing them to only a few references, the fragmented state of the literature results in many sections having a substantial number of references.

1.5.1 Common References

Despite these problems, 13 common references for use across all sections (to partially reduce the number of items users need to acquire) are listed below in subsection 1.7. Six of these are available free, including three of the books and another is free if one has access to the IEEE Computer Society digital library. Uniform resource identifiers (URI) are listed with the references.

While not as extensively referenced in this guide, the items below under Further Reading are also of interest as general references. The first provides context, the second a software application lifecycle view, and the third important, early seminal work.

1.6 Conclusion

Primarily aimed at educators and trainers but useful to many others, this document defines the additional body of knowledge needed to develop, sustain, and acquire (more) secure software beyond that needed for software where safety and security are not concerns. It first addresses the knowledge needed by all, including dangers to software systems, fundamentals, and ethics and legal issues. After covering the areas of knowledge needed to produce and sustain (more) secure software, it finishes with sections on acquiring secure software and using this document followed a unified bibliography and an index.

1.7 Common References Spanning All Sections

[Abran 2004] Abran, Alain, James W. Moore (Executive editors); Pierre Bourque, Robert Dupuis, Leonard Tripp (Editors). *Guide to the Software Engineering Body of Knowledge*. 2004 Edition. Los Alamitos, California: IEEE Computer Society, Feb. 16, 2004. Available at http://www.swebok.org.

[Avizienis 2004] Avizienis, Algirdas, Jean-Claude Laprie, Brian Randell, and Carl Landwehr, “Basic Concepts and Taxonomy of Dependable and Secure Computing,” *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11-33, Jan.-Mar. 2004. Available at http://csdl.computer.org/dl/trans/tq/2004/01/q0011.pdf.

[Berg 2005] Berg, Clifford J. High-Assurance Design: Architecting Secure and Reliable Enterprise Applications, Addison Wesley, 2005.

[Bishop 2003] Bishop, Matt. *Computer Security: Art and Practice*, Addison-Wesley, 2003.

[Gasser 1988] Gasser, M. *Building a Secure Computer System*. Van Nostrand Reinhold, 1988. Available at http://nucia.ist.unomaha.edu/library/gasser.php.

[Howard 2006] Howard, Michael, and Steve Lipner. *The Security Development Lifecycle*. Microsoft Press, 2006.

[Ibrahim et al, 2004] Ibrahim, Linda, et al, *Safety and Security Extensions for Integrated Capability Maturity Models*. Washington D.C.: United States Federal Aviation Administration, Sept. 2004. Available at http://www.faa.gov/ipg/pif/evol/index.cfm.

[McGraw 2006] McGraw, Gary. Software Security: Building Security In. Addison Wesley, 2006.

[Meier 2003] Meier, J.D., Alex Mackman, Srinath Vasireddy, Michael Dunner, Ray Escamilla, and Anandha Murukan, Improving Web *Application Security: Threats* and Countermeasures, Microsoft, 2004. Available at: http://download.microsoft.com/download/d/8/c/d8c02f31-64af-438c-a9f4-e31acb8e3333/Threats_Countermeasures.pdf.

[Redwine 2004] Redwine, Samuel T., Jr., and Noopur Davis (Editors). *Processes for Producing Secure Software: Towards Secure Software*. vols. I and II. Washington, D.C.: National Cyber Security Partnership, 2004. Available at http://www.cigital.com/papers/download/secure_software_process.pdf.

[Sommerville 2006] Sommerville, Ian. *Software Engineering*, 8th ed., Pearson Education, 2006.

[Viega 2005] Viega, J., *The CLASP Application Security Process*, Secure Software, 2005. Available at http://www.securesoftware.com.

[Whittaker and Thompson 2004] Whittaker, J. A. and H. H. Thompson. *How to Break Software Security: Effective Techniques for Security Testing.* Pearson Education, 2004.

1.8 Further Reading

[Anderson 2001] Anderson, Ross J., Security Engineering: A Guide to Building Dependable Distributed Systems. John Wiley and Sons, 2001.

[Goertzel 2006] Goertzel, Karen Mercedes, et al: Security in the Software Lifecycle: Making Application Development Processes—and Software Produced by Them—More Secure, Version 1.0 DRAFT. Washington, DC: Department of Homeland Security, 2006. Available at https://buildsecurityin.us-cert.gov/daisy/bsi/89.html.

[Seminal Papers] Seminal Papers - History of Computer Security Project, University of California Davis Computer Security Laboratory
Available at: http://seclab.cs.ucdavis.edu/projects/history/seminal.html

Part 2: Security Foundations

2 Dangers and Damage

2.1 Introduction

Software and the data it handles face numerous sources and types of damage and risk. This introduction provides the necessary background to understand these threats and risks, which underlie the need for software security. This section is relevant to persons developing, sustaining, and acquiring (more) secure software.

The definition and subtleties of security will be explored at length in subsequent sections. To begin understanding the dangers being addressed, the reader needs to remember that as covered in the Introduction, security is often spoken of as a composite of the three attributes: confidentiality, integrity, and availability. Security often requires the simultaneous existence of (1) confidentiality, (2) availability for authorized actions only, and (3) integrity with absence of 'unauthorized' system alterations [Avizienis 2004, p. 13].

2.2 Dangerous Effects

Probes as a prelude to attacks are continually increasing. Systems in large organizations such as DoD and Microsoft are probed several hundred thousand times per month. Actual attacks are increasing as well. Many attackers exploit already identified vulnerabilities – often as soon as they can compare the old versions to fixed (patched) versions of the software and analyze what changed. They can then attack any non-patched copies of the software. The time between the announcement of a vulnerability and attempted exploits of the vulnerability has diminished from months to a few days, if even that long.[1] Some vulnerabilities are even exploited as "zero-day," meaning that the exploit appears before the vulnerability is formally disclosed. Attacks have moved from primarily targeting widely used software from major vendors to more frequently targeting Web applications.[2] Network traffic to and from Web applications bypasses many traditional network security protections – even though the Web application interfaces directly with an organization's databases or other internal systems.

The amount of malicious software in the wild, spam, phishing, and spyware (all of which are defined in Section 2.4) is increasing, leading to a draining of resources, potential identify theft and loss of sensitive information.[3]

Though the effects of attacks on software security can range from irritating to devastating, no accurate measurements exist to determine the national or worldwide costs of an attack. A Gartner analyst, Avivah Litan, testified the costs of identity theft at a Senate hearing related to the Department of Veterans Affairs loss of 26.5 million veteran identities in May 2006. According to Litan, "a company with at least 10,000 accounts can spend, in the first year, as little as $6 per customer account for just data encryption, or as much as $16 per customer account for data encryption, host-based intrusion prevention, and strong security audits combined." In contrast, Litan said that companies could spend "…at least $90 per customer account when data is compromised or exposed during a breach."[4] The identity theft at the Department of Veterans Affairs was not the result of a software security breach—but many identity thefts are. Regardless, many organizations are starting to realize that the cost of a security breach can far outweigh the costs of security. According to a 2003 CERT/CC report on incident and vulnerability trends, attackers include teenage intruders, industrial spies, foreign governments, criminals, and insiders.[5] Attacks are becoming more sophisticated: targeting specific

[1] Recent estimates say the vulnerability-to-exploit window is now approximately 6 days, but this has been changing rapidly.
[2] Symantec Corp., "Internet Security Threat Report" (September 20, 2004)
[3] For descriptions of cyber crimes see the US Justice Department at http://www.cybercrime.gov/ccdocs.htm and the ACM Risks Forum archives at http://catless.ncl.ac.uk/Risks
[4] Gregg Keizer, "Cleaning Up Data Breach Costs 15x More Than Encryption" (TechWeb, June 9, 2006)
[5] CERT/CC, "CERT/CC Overview: Incident and Vulnerability Trends" (7 May, 2003)

organizations. Security intelligence experts believe that many of these sophisticated and targeted attacks are being performed by organized crime and government espionage.[6] In order to truly understand the repercussions of inadequate software security, some example incidents are provided below.

The earliest known exploitation of a buffer overflow, a common software vulnerability, was in 1988. It was one of the several exploits used by the Morris worm to propagate itself over the Internet. It took advantage of software vulnerabilities in the Unix service, fingerd. Since that time, several Internet worms have exploited buffer overflows to compromise increasingly large numbers of systems. In 2001, the Code Red worm exploited a buffer overflow in Microsoft's Internet Information Service (IIS) 5.0, and in 2003, the SQLSlammer worm compromised machines running Microsoft SQL Server 2000. In 2004, the Sasser worm exploited a buffer overflow in the Local Security Authority Subsystem Service (LSASS), which is part of the Windows operating system that verifies users logging into the computer.

In 2004, a 16-year-old hacker found a few systems on the San Diego Supercomputer Center (SDSC) that had been patched for a software vulnerability but not yet rebooted. He exploited the unpatched software still running on those machines to gain access to the network and install a sniffer to detect users' login sessions and capture login data, such as usernames and passwords.[7]

In May 2006, a large number of spam messages were disseminated from a .de email address. The messages contained a password-stealing Trojan horse[8] called "Trojan-PSW.Win32.Sinowal.u" along with text in German claiming the attachment was an official Microsoft Windows patch. The new Trojan is a member of the Sinowal family of malware first detected in December 2005. The original versions install themselves onto systems using browser exploits while this new variant tricks users into installing it. The malware acts as a man-in-the-middle that captures usernames and passwords when users access certain European bank Web sites.[9]

In May 2006, a zero-day vulnerability in Microsoft Word XP and Microsoft Word 2003 enabled attackers to plant the backdoor provided by the Ginwui Trojan on PCs of users who received emails with malicious Word documents attached. The Trojan enables an attacker to connect to and hijack the compromised PC by installing additional software.[10]

In 2005, security researchers discovered a rootkit distributed by Sony BMG in its compact discs (CDs) that acted as digital rights management (DRM) for the music contained within the CDs. The rootkit installed itself on users' PCs after inserting the CD into the optical drive. The DRM rootkit contained spyware that surreptitiously transmitted details about the user back to Sony BMG. In addition, the rootkit contained software security vulnerabilities that made the PCs vulnerable to malicious code and other attacks. This spurred 15 different lawsuits against Sony BMG to cease selling the audio CDs containing the rootkits.[11]

In 2005, students discovered a weakness in the third-party software used to manage Harvard Business School applications. Students who use the same third-party software at other schools observed that when an application decision is made, applicants visit a series of pages with the final decision appearing at a URL with certain parameters. By using similar parameters on the Harvard Business School Web site, students could view the application decisions before receiving official notice. The applications of students who used this

[6] Paul Stamp, "Increasing Organized Crime Involvement Means More Targeted Attacks" (Forrester, 12 October 2005)
[7] Bill Brenner, "Security Without Firewalls: Sensible or Silly?" (SearchSecurity.com, 5 January 2006)
[8] Trojan House: Program containing hidden code allowing the unauthorized collection, falsification, or destruction of information. [CNSS 4009]
[9] Jeremy Kirk, "Password Stealing Trojan Spreads" IDG News Service (PC World, 30 May 2006)
[10] Jay Wrolstad, "Trojan Targets Microsoft Word Vulnerability" (Top Tech News, 22 May 2006)
[11] Priyanka Pradhan, "New Settlement in Sony BMG Rootkit Case" (CNBC TV 18, 23 May 2006)

technique to view decisions in advance were refused due to an inappropriate ethical mindset for future leaders.[12]

In late 2003, Nicolas Jacobsen accessed T-Mobile's Web site using a vulnerability in BEA WebLogic. While the patch was released in early 2003, T-Mobile had not applied the patch to its own servers. Using this vulnerability, the attacker installed an interface to T-Mobile's customer service database. He had access to numerous T-Mobile account details—including social security numbers.[13]

Even more damage can be caused when the resources of a nation state are directed towards subversion of software or exploitation of software vulnerabilities. Thomas C. Reed, former Secretary of the Air Force and special assistant to President Reagan, detailed one such example in his book *At the Abyss: An Insider's History of the Cold War*. In 1981, it became apparent that the Soviet Union was stealing American technology. Instead of shutting down the Soviet operation, CIA director William Casey and National Security Council staffer Gus Weiss came up with a plan in which the U.S. would intentionally subvert the microchips and software that the Soviets were stealing. According to Reed, "every microchip they stole would run fine for 10 million cycles, and then it would go into some other mode. It wouldn't break down; it would start delivering false signals and go to a different logic." Similarly, the software the Soviets stole to run their natural gas supply systems were programmed to include a "time bomb" that changed processing to a different logic after a set period of time. In 1982, the failure of the gas system software caused the explosion of a major gas pipeline, resulting in "the most monumental non-nuclear explosion and fire ever seen from space." [Reed 2004] The whole U.S. sabotage operation resulted in a huge drain on the Soviet economy. Moreover, the Soviets had based a huge number systems on stolen software and hardware. The realization that some of this stolen technology had been compromised made it virtually impossible for them to determine which equipment was safe and which was untrustworthy.[14]

The pipeline explosion occurred over two decades ago. Since then, the emergence of the internet, coupled with the exponential growth in size, complexity, and ubiquity of software, has made such sabotage operations much easier, and it is feared, much more likely. For example, a recent well-resourced, set of incidents showing a professional-level of skill and sophistication see the *Time* magazine September 5, 2005 cover story on Titan Rain [Thornburgh 2005].

2.3 Attackers

The Russian pipeline explosion demonstrated what a well-resourced attacker could accomplish. The Titan Rain attacks were also by sophisticated attackers. At the other end of the attack spectrum are the novice hackers, also referred to as script kiddies. Understanding attackers' motivations and capabilities helps in adequately defending against them. Understanding an attacker's motivations may allow identification of those portions of a system most likely to be attacked while understanding an attacker's capabilities may allow identification of potential methods of attack and predictions concerning success of attacks. This subsection describes the range and nature of attackers.

2.3.1 Types of Attackers

The spectrum of attackers includes two characteristics:

- Sophistication of technical knowledge – from attackers who develop attacks to 'script kiddies' who must rely on attack scripts provided by someone else

[12] Sverre H. Huseby, "Common Security Problems in the Code of Dynamic Web Applications" (http://www.webappsec.org/projects/articles/062105.shtml, 1 June 2005)
[13] Kevin Poulsen, "Known Hole Aided T-Mobile Breach" (Wired News, 28 February 2005)
[14] Steve Kettman, "Soviets Burned by CIA Hackers?" (Wired News, 26 March 2004)
David Hoffman, "CIA Slipped Bugs to Soviets" (Washington Post, 27 February 2004)

- Ability to cause harm – from those able to determine and execute actions causing significant harm to an organization to those for whom just gaining entry (and perhaps notoriety) is the purpose for the attack.

Script kiddies use what they can glean from hacker web sites to try to attack systems. Their attack operations tend to be very crudely orchestrated and "noisy." Given the low barrier of entry consisting of a computer connected to the Internet, a few exploits gleaned from simply using Google to search for an autorooter (scripts or programs for trying to obtain complete administrative privileges) and a little free time, and the script kiddie is in business.

Their attacks can be conducted from anywhere in the world – even from locations where their activities are not illegal. They realize that their chances of ever being identified, much less being convicted, are extremely low. This is true even if the owners of the system being attacked deem it worthwhile to pursue them and are willing to risk the potentially adverse publicity of acknowledging the success of the attack once arrests are made. The attacks by script kiddies are a drain on resources and provide good cover for the sophisticated adversaries. Script kiddies represent the lower-end of a continuum of attackers with a variety of skill levels, resources, and organization. At the higher-end are technically sophisticated attackers who are able to discover vulnerabilities and exploit them, and have the support and direction of a large organization for which cyber attack is not an end in itself, but rather a means of achieving their desired goals. An example of a high-end attacker would be those executing nation state directed computer network attack.

For the entire range of attacker expertise, it is important to distinguish between the sophistication of the attacker and the sophistication of the attack. Persons with very limited technical ability can now launch very sophisticated attacks thanks to the availability of highly sophisticated, point-and-click attack tools.

Between the script kiddies and the well-resourced adversaries is a continuum of attackers with a variety of skill levels and motivations. These types of attackers and their motivations are explored in the next subsection.

2.3.2 Motivations of Attackers

Attackers have many motivations. Some of the primary reasons that an attacker would attack a system are that the attacker wants something that is on the target system, that the attacker wants to use or control the system, the attacker's plans to perform a denial of service against the system, or destroy information on the system or the system itself. In general, attackers engage in two types of attacks: preserving attacks and destructive attacks.

Preserving attacks must maintain a low profile and the system must continue to appear to work as expected to the users in order not to be discovered since, if the attack were to be discovered, the access would likely terminate. Where does this information come from? What is the source?

Preserving attacks may be contrasted with a destructive attack that is intent on destroying either the integrity of the data accessible from the system or the system itself. These are very dangerous attacks, as the attacker may not ultimately care that the attack is discovered as in the case of a time or logic bomb. Via a time bomb or logic bomb (see Section 2.4.2 for definitions) intentionally implanted in software, an attacker can, with relative ease, target a system on an isolated network (air-gapped) network. Destructive attacks can have long-term impacts through corrupted data or destroyed files or a loss of confidence even after the recovery of the system.

For either type of attack, many motivations exist ranging from ego, intellectual challenge, or desire for acceptance to revenge, theft, psychopathy, espionage, and information warfare. Particular kinds of attackers, however, tend to have certain motivations. Some categories of attacks and their typical motivations are described in Table 1.

Table 3: Attackers and their Motivations

Amateur hackers	recreation, reputation, sense of belonging, malevolence, and learning
Insiders	vengeance (e.g., revenge through sabotage), vindictiveness, to gain sympathy, whistle blowing, vigilantism, stalking or intimidation, embezzlement, “job security” through extortion
Psychopath or sociopath	as a form of destructive or anti-social behavior
Individual criminals, small criminal groups	money including credit card fraud, insider fraud, and identity theft
Social protestors (hactivists)	publicity, hindering and disruption, patriotism, vigilantism, and social or political policy change, or chnage in corporate behavior
Commercial competitors	competitive intelligence for competition or negotiation, industrial espionage, recruitment, subversion, commercial advantage or damage, tacit collusion, and misinformation
Organized crime syndicates	money including fraud, extortion, blackmail; theft, and identity theft; recruitment, corruption, and subversion; intimidation and influence including extortion and blackmail; intelligence on politics, law enforcement, criminal competition, and opportunities and risks for criminal activities; and industrial espionage
Terrorists	intelligence including target identification and information, publicity and propaganda, recruiting, political action, disruption, intimidation, and damage
Nation states	intelligence and counter-intelligence, economic espionage, training, preparation for information warfare, misinformation, sabotage, law enforcement and deterrence, political influence, blocking illegal or subversive content, and general hindering and disruption.

Thus, no shortage exists of attackers and motivations. This list, however, is not exhaustive and attackers vary in their capabilities, resources, intentions, persistence, and degree of risk aversion. They also may be outsiders or someone inside – someone having a significant relationship with the individual or organization that is the target of the attack.

2.4 Methods for Attacks

Attacks can occur during any phase of the software life cycle: from the development, testing, deployment, operation, sustainment, to decommissioning and disposal. The ability to conduct an attack can be made significantly easier if the software being attacked contains vulnerabilities, malicious code, or back doors that were placed intentionally during its development for exploitation during its operation.

Probes are often preludes to attacks. Systems in large organizations such as DoD and Microsoft are probed several hundred thousand times per month. Analysis of software by attackers is also a common prelude to identification of points to attack. Many attackers exploit already identified vulnerabilities, often as soon as they can compare the old versions to fixed (patched) versions of the software and analyze what has changed. They can then attack any non-patched copies of the software. The time between the announcement of a vulnerability and attempted exploits of the vulnerability has diminished from months to a few days.

2.4.1 Malicious Code Attacks

One means by which attackers attempt to achieve their objectives is by inserting malicious software code within a software program, or planting it on an operational system. Malicious code, also referred to as *malicious software* or *malware*, is designed to deny, destroy, modify, or impede the software’s logic, configuration, data, or library routines.

Malicious code can be inserted during software's development, preparation for distribution, deployment, installation, and or update. It can be inserted or planted manually or through automated means. Regardless of when in the software lifecycle the malware is embedded, it effectively becomes part of the software and can present substantial dangers.

Viruses, worms, spyware, and adware are all rampant on the Internet, and some names of malware, such as Code Red and Nimda, have entered the popular vocabulary. Everyone from home computer owners to Fortune 500 information technology (IT) infrastructure system managers is waging a constant battle to protect their systems against these threats.

A software producer clearly needs to be concerned about preventing and handling malicious actions directed against the software he or she develops or sustains, as well as the assets that software protects once it is in operational use. As previously noted, malicious actions can occur at other times as well.

Certain categories of malicious code are more likely to be planted on operational systems, while others are more likely to be inserted into software before it is deployed. The malware categories described below are therefore divided into categories that are likely to "inserted" versus categories that are likely to be "planted".

Increasingly, Malware is combined with deceptive "social engineering" techniques to accomplish complex attacks on unsuspecting users. In some cases, malware is used to enable a deception, as in pharming. In other cases, deception is used to trick the user into downloading and executing malicious code. Another popular deception technique, phishing, is worth noting though it does not require malicious code to succeed. "Social engineering" attacks are described in later.

2.4.1.1 Categories of Malware Likely to Be Inserted During Development or Sustainment

- **Back door or trap doors** – a hidden software mechanism that is used to circumvent the system's security controls, often to enable an attacker to gain unauthorized remote access to the system. One frequently used back door is a malicious program that listens for commands on a particular Transmission Control Protocol (TCP) or User Datagram Protocol (UDP) port.
- **Time bomb** – a resident computer program that triggers an unauthorized or damaging action at a predefined time.
- **Logic bomb** – a resident computer program that triggers an unauthorized or damaging action when a particular event or state in the system's operation is realized, for example when a particular packet is received.

2.4.1.2 Categories of Malware Likely to Be Planted on Operational Systems

- **Virus** – a form of malware that is designed to self-replicate (make copies of itself) and distribute the copies to other files, programs, or computers (and sometimes used for malware that replicates both by being copied and copying itself). A virus may attach itself to and becomes part of another executable program, for example to become a delivery mechanism for malicious code or for denial of service attack.

There are a number of different types of viruses, including (1) boot sector viruses, that infect the master boot record (MBR) of a hard drive or the boot sector of removable media; (2) file infector viruses, that attach themselves to executable programs such as word processing and spreadsheet applications and computer games; (3) macro viruses, that attach themselves to application documents, such as word processing files and spreadsheets, then use the application's macro programming language to execute and propagate; (4) compiled viruses, that have their source code converted by a compiler program into a format that can be directly executed by the operating system; (5) interpreted viruses, composed of source code that can be executed only by a particular application or service; (5) multipartite viruses, which use multiple infection methods, typically to infect both files and boot sectors. More recently, a category of virus called a morphing virus has emerged; as its name suggests, a morphing virus changes as it propagates, making it extremely difficult to eradicate using conventional antivirus software because its signature is constantly changing.

- **Worm** – a self-replicating program that is completely self-contained and self-propagating. a self-replicating computer program similar to a computer virus. Unlike a virus, a worm is self-contained and does not need to be part of another program to propagate itself. Worms frequently exploit the file transmission capabilities found on many computers: self-propagating delivery mechanism for malicious code or for a denial of service (DoS) attack that effectively shuts down service to users. Types of worms include a network service worm, that spreads by taking advantage of a vulnerability in a network service associated with an operating system or application, and a mass mailing worm, that spreads itself by identifying e-mail addresses (often located by searching infected systems) then using either the system's email client or a self-contained mailer built into the worm to send copies of itself to those addresses..
- **Trojan, or Trojan Horse** – a non-replicating program that appears to be benign but actually has a hidden malicious purpose.
- **Zombie** – a program that is installed on one system with the intent of causing it to attack other systems.

Operational software can be modified by the actions of malware. For example, some viruses and worms insert themselves within installed executable software binary files where they trigger local or remote malicious actions or propagation attempts whenever the software is executed.

One noteworthy “delivery” technique for malicious code within Web applications is the cross-site scripting attack:

- **Cross-site scripting** (abbreviated as "XSS", or less often “CSS”) – an attack technique in which an attacker subverts a valid Web site, forcing it to send malicious scripting code to an unsuspecting user’s browser. Because the browser believes the script came from a trusted source, it executes it. The malicious script may be designed to access any cookies, session tokens, or other sensitive information retained by the browser for use when accessing the subverted Web site. XSS differs from pharming in that in XSS, the Website involved is a valid site that has been subverted, whereas in pharming, the Website is invalid but made to appear valid.

2.4.2 Hidden Software Mechanisms

There are categories of “hidden” or “surreptitious” software mechanisms that were originally designed for legitimate purposes but which are increasingly being used by attackers to achieve malicious purposes. When this happens, these mechanisms, for all practical purposes, operate as malware.

Like viruses and worms, these hidden software mechanisms are most likely to be planted on operational systems rather than inserted into software during its development or sustainment. The most common hidden software mechanisms are: [15]

- **Bot** (abbreviation of *robot*) – an automated software program that executes certain commands when it receives a specific input. Bots are often the technology used to implement Trojan horses, logic bombs, back doors, and spyware.
- **Spyware** - any technology that aids in gathering information about a person or organization without their knowledge. Spyware is placed on a computer to secretly gather information about the user andreport it. The various types of spyware include (1) a *web bug*, a tiny graphic on a Web site that is

[15] For more information see the Spyware Coalition website http://www.antispywarecoalition.org/

referenced within the Hypertext Markup Language (HTML) content of a Web page or e-mail to collect information about the user viewing the HTML content; (2) *a tracking cookie*, which is placed on the user's computer to track the user's activity on different Web sites and create a detailed profile of the user's behavior.

- **Adware** Any software program intended for marketing purposes such as including to deliver and display advertising banners or popups to the user's screen or tracking the user's online usage or purchasing activity.

2.4.3 Social Engineering Attacks

The main categories of social engineering attacks are:

- **Spam** – unsolicited bulk e-mail. Recipients who click links in spam messages may put themselves at risk of inadvertently downloading spyware, viruses, and other malware.
- **Phishing** – the creation and use of fraudulent but legitimate looking e-mails and Web sites to obtain Internet users' identity, authentication, or financial information, or to trick the user into doing something he/she normally wouldn't. In many cases, the perpetrators embed the illegitimate Web sites' universal resource locators (URLs) in spam – unsolicited bulk e-mail – in hopes that the curious recipient will click on those links and trigger the download of the malware or initiate the phishing attack.
- **Pharming** – the redirection of legitimate Web traffic (e.g., browser requests) to a illegitimate site for the purpose of obtaining private information. Pharming often uses Trojans, worms, or other virus technologies to attack the Internet browser's address bar so that the valid URL typed by the user is modified to that of the illegitimate website. Pharming may also exploit the Domain Name Server (DNS) by causing it to transform the legitimate host name into the invalid site's IP address; this form of pharming is also known as "DNS cache poisoning".

2.4.4 Physical Attacks

Purely physical attacks can be devastating to an organization. Simply cutting critical cables, stealing computers containing critical information, or stealing or destroying the only copies of critical information could potentially ruin an organization. Physical attacks require very little skill to accomplish and can simply be the result of a poorly planned backhoe operation or other simple mistakes. Although it is impossible to predict and defend against all possible physical attacks, proper planning such as off-site backups and redundancy can mitigate many of the consequences or risks posed by physical attacks.

Theft particularly of laptops containing sensitive information is a serious problem leading some to encrypt such data on laptops, "thumb drives", and other media used to transport data.

2.5 Non-Malicious Dangers to Software

Software in operation may be made vulnerable by a number of unintentional, non-malicious events. However, because these events are unintentional, does not mean they cannot constitute a threat to the software's security.

Some of these events threaten the availability of software. Because one can say security is about maliciousness, one could say that availability is correctly categorized as a security property only when:

- The compromise of availability is intentional, i.e., the result of a denial of service attack; *or*
- The compromise of availability leaves the software (or system) vulnerable to compromise of any of its other security properties (for example, denial of service in a software component relied on to validate

code signatures could leave the software's integrity vulnerable to compromise through insertion of unsigned malicious code).

In practice, the maliciousness or lack thereof has limited impact as the event must be considered in any case, and when a service is unavailable it makes no difference to the users' abilities or in its consequences. It may, however, influence investigation, repair, recovery, and other follow-up activities.

[MoD DefStan 00-56 Part 2/3 2004, page 31] lists a number of unintentional events that may threaten the security of operational software. These include:

- Systematic and random failures;
- Credible failures arising from normal and abnormal use in all operational situations;
- Scenarios, including consequential credible failures (accident sequences);
- Predictable misuse and erroneous operation;
- Faulty interactions between systems, sub-systems, or components;
- Failure in the software operation environment (e.g., operating system failure);
- Procedural, managerial, human factors, and ergonomics-related activities.

In addition, there are physical events that can result in hardware failures (and, by extension, software failures); other physical events may render the hardware unstable, which can cause common mode failures in operational software, or render software in development vulnerable to physical access by unauthorized persons. Such physical events include:

- Natural disasters, e.g., hurricanes, earthquakes;
- Failure in the physical operating environment (hardware failures, power failures);
- Mechanical, electro-magnetic, and thermal energy and emissions;
- Explosions and other kinetic events (kinetic events are those that involve movement of the hardware on which software operates);
- Chemical, nuclear, or biological substance damage to the hardware or network;
- Unsafe storage, maintenance, transportation, distribution, or disposal of physical media containing the software.

On analysis many of these events may not result in a security requirement, but some may. Examples of the effects that must usually be considered are

- Corruption of the software may leave it vulnerable to attacks to which it was not subject before;
- Sudden physical accessibility to the systems or media containing the software may increase the ability of attackers to obtain a copy of the software's source code (in order to study it, or to insert malicious code and produce a subverted version of the software) or binary (with the intention of reverse engineering).

Three additional unintentional event categories may have the same effects on the software as malicious code attacks:

- Unintentional software defects: these can have the same effects as malware – currently the most common source of vulnerabilities
- Intentional extra functionality: can provide additional paths of attack, defects and vulnerabilities, or surprises – particularly unused/unadvertised functionality

- Easter eggs: code placed in software for the amusement of its developers or users

Protecting from or reacting to all the events listed above will seldom fall to a single piece of software, but any could be relevant to a given product. Nevertheless, the list provides insight into the kinds of compromising events that may occur.

2.6 Attacks across Lifecycle

This subsection takes a lifecycle view and goes through the lifecycle surveying concerns, problems, and events. It discusses attackers and methods of attack and when they may occur as well as some of the consequences.

2.6.1 Attacks during Software Production

This subsection addresses conditions and events in the software's lifecycle that can make software vulnerable to malicious code insertions and other compromises.

2.6.1.1 Authorized Access

During the development of software, an inside attacker could intentionally implant malicious code. The malicious code could be an intentional backdoor to allow someone to remotely log in to the system running the software or could be a time or logic bomb. Alternatively, the malicious code could be an intentionally implanted vulnerability[16] that would allow the attacker to later exploit the vulnerability. This method would provide the software company with plausible deniability of intent should the vulnerable code be found.

Inserting the malicious code during the software development process places the malicious code in all copies of the software. There are advantages and disadvantages to placing the malicious code in all copies of the software rather than a targeted few. If the malicious code is in all copies, then wherever the software is running, the malicious code will be present. This averts the danger to the attacker of there existing "pure" and "altered" versions of the software so that a simple comparison of checksums will reveal differences between copies of software that should be the same.

The easiest and most effective time to insert a malicious mechanism into a software product is during its requirements phase. Because the malicious code is conceived at the beginning of the software development process, it can be designed either as an integrated and visible feature or as an unadvertised feature. Stating malware intentions in the requirements phase causes many people become aware of the malicious feature, possibly raising the probability of public disclosure. However, conceivably the requirements could be crafted appropriately to obscure the malicious intent of the software (e.g., backdoor to permit remote observation as a user support feature).

Inserting the vulnerability or malicious functionality at a later stage of software development would potentially expose it to fewer people. In situations where organizations are not taking precautions, inserting the malicious mechanism during the design or implementation phase could be relatively easy and in many organizations would only require the actions of a rogue programmer. The likelihood of the attack being exercised and discovered during testing is not high since normal testing is based on the unmodified specifications.[17] Historically, testing does not look for added functionality, which is one of the reasons Easter eggs (recreational

[16] Such as the buffer overruns or race conditions discussed in Section 7 on Secure Software Construction.
[17] Trying to break software security through testing such as that suggested by [Whittaker and Thompson 2004] has also been rare except among major software vendors and users. This trend has been improving.

insertions)[18] and backdoors have been able to become part of final products relatively easily. Code inspection may reveal the malicious code, but in many shops a part of the code is not likely be examined late in the process unless a problem is discovered during testing. Even then, the same programmer who put the malicious code in the product may be the person asked to fix the problem.

Once coding is complete and testing is in progress, the level of difficulty facing an insider inserting malicious code depends on the environment in which the testing is conducted. If the testers have access to the source code and development environment, inserting malicious code could be easy. Having access only to the compiled binary could make inserting a malicious mechanism much more difficult.

Other avenues of insider attacks can occur when placing the product on a CD or a website, or during sustainment such as when developing or releasing upgrades or patches for the software. The original software or updates could also be modified during the delivery and installation or after it has been installed.

To summarize, an attacker who is part of the software development process might alter the product during any phase of the software development process.

During deployment an attacker might:

- Change the product or update it after approval and before distribution;
- Usurp or alter the means of distribution;
- Change the product at user site before or during installation.

2.6.1.2 Unauthorized Access

For outsiders and insiders, the later in the software development process that the attack is inserted, the less likely it is to be found. Usually, the initial release of a software product receives more analyses and testing than an upgrade or patch. Inserting an attack as part of an upgrade or patch would thus be less likely to be discovered. But, on the other hand, not all systems would perform the upgrade or patch, and the lifespan of the vulnerability or attack would be shorter than if the attack had been in the software initially.

An unauthorized attacker could alter the product electronically by hacking into the distribution site for the software. Performing an attack in this manner would likely be difficult and require substantial skill. Also, as mentioned previously, danger of detection increases when both pure and altered copies exist.

Others who may be unauthorized to change the software, but who have a degree insider access, include secretaries, janitors, and system administrators. Being insiders to the company, they are more likely to be trusted and less closely watched. In the case of system administrators, they might even be the ones responsible for detecting any attacks. These or other trusted insiders might be self-motivated, either an inserted agent of an outside entity or as a subversive, e.g., bribed. They may perform the actions themselves or provide access for an unauthorized outsider.

One danger that was not mentioned was disclosing a vulnerability to attackers or to the public, either by an insider or someone not connected to the development, sustainment, or operation of the secure software system. Knowledge of the vulnerability may be known to the company, but until a patch is available, reasons exist for the details of the vulnerability to be closely held. This time window of opportunity could be valuable to an attacker.

To summarize, attackers during development might:

[18] An Easter egg is hidden functionality within an application program that is activated when an undocumented, often convoluted set of commands and keystrokes is entered. Easter eggs are typically used to display the credits of the application’s development team. Easter eggs are intended to be innocuous; however, because Easter eggs are virtually identical in implementation, if not in intent, to logic bombs, many organizations have adopted policies that forbid the use of software that contains easter eggs. NIST SP 800-28: Guidelines on Active Content and Mobile Code. October 2001.

- Change a product from outside by
  - Initiating electronic intrusion
  - Allowing physical intrusion (combined with electronic)
- Change a product from inside by
  - Inserting an agent
  - Corrupting someone already in place
  - Having an insider who is self-motivated
- Change or disrupt development process by
  - Failing to run or report a test
  - Categorizing a vulnerability's defect report as not a vulnerability

In summary, the means to conduct an attack can take several forms. Also, attacks do not have to compromise the security of a system to be successful at denying the continued operation of a system. An outsider can simply overwhelm critical resources (e.g., communication paths) that a system depends upon. Such an attack can be as effective as a successful intrusion if the objective is to deny the use of the system.

Many paths of attack exist including:

- Intrusion: gaining illegitimate access to a system
- External or Perimeter Effects: acts that occur outside or at the defense perimeter but nevertheless have a damaging effect; the most common one is denial of service from overload of a resource
- Insider: a person with existing authorization uses it to compromise security possibly including illegitimately increasing authorizations
- Subversion: changing (process or) product so as to provide a means to compromise security[19]
- Malware: software placed to aid in compromising security

Attempts to prevent attacks against software during its operation fall into the realm of operational security rather than software security.

2.6.2 Attacks Against Operational Systems

The typical attack on an operational system consists of the following steps:

- Target Identification and Selection: The desirability of a target depends to a great extent on the attacker's motivations, and any evidence of the target's vulnerability that can be discovered through investigation of news reports, incident and vulnerability alerts, etc.
- Reconnaissance: Can include technical means, such as scanning and enumeration of systems and ports, as well as social engineering and "dumpster diving" to discover passwords, and investigation of "open source intelligence" using DNS lookups, Web searches, etc. to discover the characteristics of the system being attacked, and particularly to pinpoint any potentially exploitable vulnerabilities.
- Gaining access: Exploits the attack vectors and vulnerabilities discovered during reconnaissance.

[19] "Subversion" is used to describe subversion of people (e.g. developers), subversion of machines or network nodes, subversion of software, and of other things. [Anderson 2004]

- Maintaining access: Often involves escalation of privilege in order to create accounts and/or assume a trusted role. May also involve planting of rootkits, back doors, or Trojan horses. Depending on the attacker's goal, maintaining access may not be necessary.
- Covering tracks, which may include hiding, damaging, or deleting log and audit files, and other data (temp files, cache) that would indicate the attacker's presence, altering the system's output to the operator/administrator console, and exploitation of covert channels.

Attacks and malicious behavior may be undertaken not only by outsiders [Berg 2005, Chapter 5] but by insiders, including authorized users. The specific techniques used to access systems change on a routine basis. Once a vulnerability has been patched, attackers must use a different technique to gain attack the system. Nevertheless, these attacks can be generalized into certain types of attacks, which are (derived from [NSA 2004]):

Attacks performed by an unauthorized attacker:

- *Eavesdrops* on, or captures, data being transferred across a network.
- Gains unauthorized access to information or resources by *impersonating an authorized user*.
- Compromises the integrity of information by its *unauthorized modification or destruction.*
- Performs unauthorized actions resulting in an *undetected compromise of assets*.
- *Observes an entity during multiple uses* of resources or services and *links these* uses *to deduce* undisclosed information.
- *Observes legitimate use when the user wishes kept private* their use of that resource or service.

Attacks performed by an authorized user:

- *Accesses without permission* from the person who owns, or is responsible for, the *information or resource*.
- *Abuses* or unintentionally performs *authorized actions* resulting in *undetected compromise* of the *assets*
- *Consumes shared resources* and *compromises the ability of other* authorized users to access or use those resources.
- Intentionally or accidentally *observes stored information not authorized to see*.
- Intentionally or accidentally *transmits sensitive information to users not authorized to see it*.
- *Participates* in the transfer of information (either as originator or recipient) and then *subsequently denies having done so*.

Users or operators with powerful privileges can be especially dangerous. Administrators or other privileged users can compromise assets by careless, willfully negligent or even hostile actions. Finally, in certain situations a physical attack may compromise security (e.g. breaking and entering, theft of media, physically tapping cables).

Each of the preceding technical events may result in bad security-related outcomes. The severity of the consequence ranges from the annoying to the severe.

A number of other common attack techniques are described in Appendix C of *Security in the Software Lifecycle*, which can be downloaded from the DHS BuildSecurityIn Web portal.

2.6.3 Attacks after Retirement

Even software in disposal may be subject to attack. By gaining physical access to software when it or the computer or media containing it is disposed of, the attacker may find it easier to recover residual sensitive data stored with or embedded in the software—data that could not be easily recovered when the operational software was protected by environment security controls and mechanisms. Or the attacker may wish to copy disposed software that has been replaced by a later but derivative version of the same program, in order to reverse engineer the older version, and use the knowledge gained to craft more effective attacks against the new version.

To be completely safe, one must carefully make sensitive data of all kinds unreadable before disposal. Given increasing forensic capabilities, this may be non-trivial. Encrypted data may not be adequately protected if weakly encrypted or if sensitive for a sustained future period.

2.7 Information about Known Vulnerabilities and Exploits

Current information on vulnerabilities and the exploits that target them can be found in a number of sources, including books (in which the information may be better organized as an introduction to the subject, but will be less current), articles, vendors' and independent "alert" services, and databases. For examples, see [Whitaker 2004] and [Hoglund 2004], as well as the following:

- OWASP Top Ten - http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project
- SANS Top Twenty - http://www.sans.org/top20/
- NIST National Vulnerability Database - http://nvd.nist.gov/
- US-CERT Vulnerability Notes Database - http://www.kb.cert.org/vuls/
- Open Source Vulnerability Database - http://www.osvdb.org/
- MITRE Corporation dictionary of Common Vulnerabilities and Exposures - http://www.cve.mitre.org/
- MITRE Corporation Common Weaknesses Enumeration - http://www.cve.mitre.org/cwe/
- Internet Security Systems (ISS) X-Force Database - http://xforce.iss.net/xforce/search.php
- SecurityFocus Vulnerabilities - http://www.securityfocus.com/vulnerabilities
- Secunia Vulnerability and Virus Information - http://secunia.com/
- Purdue University Center for Education and Research in Information Assurance and Security (CERIAS) Cooperative Vulnerability Database - https://cirdb.cerias.purdue.edu/coopvdb/public/
- DoD's Joint Task Force-Global Network Operations (JTF-GNO) Information Assurance Vulnerability Alert (IAVA) program - http://www.cert.mil/.

What is publicly known, however, may be less than is known to producers or researchers. Potential attackers may know exploits no one else knows. In addition, even after shipment some software vendors make significant efforts to discover vulnerabilities through internal efforts whose results are often not publicized.

2.8 Conclusion

Computing systems can suffer security violations and "internal" damage – e.g. disclosure, tampering, and crashes. However, security problems involving computing systems can have significant consequences in the real world of people, organizations, physical objects and activities, wealth, and power; and have the potential to cause even greater future disruption and damage.

Software plays a central role in today's computing systems and in their security. However, software is in danger throughout its lifespan, and systems can be in danger both when operating and when not operating, e.g., a laptop containing sensitive information could be stolen. Security, both at the individual software component and whole system levels, is about dealing with these dangers and preserving properties such as confidentiality, integrity, and availability in the face of attacks, mistakes, and mishaps. Protection needs are based on the risk or consequences resulting from the threatening dangers and the value of the items being protected. A rational approach to protection must anticipate dangers and provide defensive, tolerance, and resilience measures based on the potential damage inside and outside the system.

No system can be protected perfectly. Much commonly used software is far from vulnerability free, and attacks based on social deception or physical means can happen at any time as can accidents. Nonetheless, without confidence in the security being adequate, one cannot – as the Soviets could not – rationally rely on software being secure or dangers not becoming realities. The software must not only be adequately secure, but evidence must exist to justify rational confidence that it is secure.

Much of this document is about producing software that can block or tolerate and then recover from attacks while sounding alarms and keeping worthwhile records – and the case for having confidence in it. Before continuing into the details, however, some fundamental concepts and principles need to be covered and the legal and organizational context must be set. These topics are covered in the sections 3 and 4, respectively.

2.9 Further Reading

2.9.1 General

[NIPP 2006] Department of Homeland Security. *National Infrastructure Protection Plan*. Department of Homeland Security, 2006. Available at www.dhs.gov/nipp.

Leeson, Peter T. and Christopher J. Coyne, "The Economics of Computer Hacking", in *Journal of Law, Economics and Policy*, Vol. 1, No. 2, pp. 473-495, 2006. Available at http://www.ccoyne.com/Economics_of_Computer_Hacking.pdf.

Frank Swiderski and Window Snyder. *Threat Modeling*. Microsoft Press, 2004).

2.9.2 Attackers

Alexander, Steven. "Why Teenagers Hack: A Personal Memoir", in *Login*, Vol. 10, No. 1, pp. 14-16, February 2005. Available at http://www.usenix.org/publications/login/2005-02/pdfs/teenagers.pdf.

Besnard, Denis, "Attacks in IT Systems: a Human Factors-Centred Approach". University of Newcastle upon Tyne, 2001. Available at http://homepages.cs.ncl.ac.uk/denis.besnard/home.formal/Publications/Besnard-2001.pdf.

Denning, Dorothy E. *Information Warfare and Security*, pp 46-50. Reading MA: Addison-Wesley, 1999.

Fötinger, Christian S. and Wolfgang Ziegler, "Understanding a Hacker's Mind - A Psychological Insight into the Hijacking of Identities". Krems, Austria: Donau-Universität Krems, 2004. Available at http://www.donau-uni.ac.at/de/studium/fachabteilungen/tim/zentren/zpi/DanubeUniversityHackersStudy.pdf.

Garfinkel, Simson L., "CSCI E-170 Lecture 09: Attacker Motivations, Computer Crime and Secure Coding". Cambridge, MA: Harvard University Center for Research on Computation and Society, 21 November 2005. Available at http://www.simson.net/ref/2005/csci_e-170/slides/L09.pdf.

Jordan, Tim, “Mapping Hacktivism: Mass Virtual Direct Action (MVDA), Individual

Kleen, Laura J. *Malicious Hackers: A Framework for Analysis and Case Study.* Master's Thesis, AFIT/GOR/ENS/01M-09. Wright-Patterson Air Force Base, OH: Air Force Institute of Technology, March 2001. Available at http://www.iwar.org.uk/iwar/resources/usaf/maxwell/students/2001/afit-gor-ens-01m-09.pdf.

Krone, Tony, "Hacking Motives", in *High Tech Crime Brief*. Australian High Tech Crime Centre, June 2005. Available at: http://www.aic.gov.au/publications/htcb/htcb006.pdf.

Lakhani, Karim R. and Robert G Wolf, "Why Hackers Do What They Do: Understanding Motivation and Effort in Free/Open Source Software Projects", in *Perspectives on Free and Open Source Software.* Cambridge, MA: MIT Press, 2005. Available at http://ocw.mit.edu/NR/rdonlyres/Sloan-School-of-Management/15-352Spring-2005/D2C127A9-B712-4ACD-AA82-C57DE2844B8B/0/lakhaniwolf.pdf.

Lin, Yuwei. *Hacking Practices and Software Development.* Doctoral Thesis. York, UK: University of York, September 2004. Available at http://opensource.mit.edu/papers/lin2.pdf.

Rattray, Gregory J., "The Cyberterrorism Threat", Chapter 5 in Smith, James M. and William C. Thomas (editors), The Terrorism Threat and U.S. Government Response: Operational and Organizational Factors. U.S. Air Force Academy, Colorado: March 2001. Available at http://www.usafa.af.mil/df/inss/Ch%205.pdf.

Thomas, Douglas. *Hacker Culture.* Minneapolis, MO: University of Minnesota Press, 2002.

Virtual Direct Action (IVDA) and Cyberwars”, in *Computer Fraud & Security*. Issue 4, 2001.

Wong, Kelvin, "Friend Or Foe: Understanding The Underground Culture and the Hacking Consciousness". Melbourne, Australia: RMIT University, 20 March 2001. Available at http://www.security.iia.net.au/downloads/friend_or_foe.pdf and http://www.security.iia.net.au/downloads/new-lec.pdf.

2.9.3 Methods of Attack

[Hoglund 2004] Hoglund, Greg, and Gary McGraw. *Exploiting Software: How to break code*. Addison-Wesley, 2004.

[Howard 2005] Howard, Michael, David LeBlanc, John Viega, *19 Deadly Sins of Software Security*. McGraw-Hill Osborne Media, 1st edition, 2005.

Bruce Schneier. "Attack Trees: Modeling Security Threats", *Dr. Dobb’s Journal*, December 1999.

Herbert Thompson, Scott Chase. *The Software Vulnerability Guide*. Charles River Media, 2005.

James Whittaker, Herbert Thompson. *How to Break Software Security*. Addison Wesley, 2003.

Sverre H. Huseby. Innocent Code: A Security Wake-up Call for Web Programmers. John Wiley & Sons, 2004).

3 Fundamental Concepts and Principles

3.1 Introduction

Anyone who wishes to develop, sustain, evaluate, or acquire software that is (more) secure needs to know several terms, concepts, and principles that span multiple activities and roles. This section covers those that are prerequisites for understanding the remainder of the guide.[1]

The history of computer security goes back at least into the 1960's, and a substantial amount of important work was done in the 1970's and early 1980's. Luckily, a project at the University of California Davis collected many of these seminal works and placed on the Internet [Seminal Papers].[2] While their contents may be reflected in later publications, these works are still relevant today – for example, [Ware 1970], [Anderson 1972], and [Saltzer and Schroeder 1975].

Over time, many terms have come to be used in within and across the software, information security, software security, and other communities, with each community giving the term a different meaning or nuance. These differences can cause confusion that hinders productive communication as conversations revolve around clarification of someone is using a term in a particular situation, debates over competing dictionaries and ontologies, and arguments about which meaning is the "right" one. This guide aims to present the varying usages of terms while also helping the reader obtain a clear understanding of the underlying concepts, and more subtly, to recognize the differences between what is true versus what is known and how well it is known. Readers should also be able to discern the assumptions implicit in various concepts and statements.

After first covering some basic terms and properties, this section introduces the concept of a software security assurance case including one means of gaining partial assurance, software certification and accreditation. This is followed by a number of basic software system security principles and concepts. Secure software engineering subsection includes the many kinds of security-related stakeholders in the software lifecycle, anintroduction to system security policies, and aspects of system architecture. The section ends by elaborating on security properties.

3.2 Variations in Terms and Meaning

The concepts and properties covered in this section tend to fall into groups or change the related term's effective meaning depending on whether they are about:

- What is needed,
- What is specified,
- What is actually true (e.g. about the software),
- What has been measured, observed, inferred, etc.,
- The uncertainty in these measurements, estimates, conclusions, etc.,
- The degree of confidence one has,
- The related decisions one makes.

[1] Additional concepts and terms that need to be known by all readers appear in Sections 1, 2, and 4.
[2] The gap in attention to software security in the 1990's has reputedly led to many articles being submitted for publication today that repeat this early work because the authors are unaware of it.

Roughly, [NIST Special Publication 800-33 2001] calls the first two of these security goals and security objectives. Others call the security properties specifications "constraints". The third, "What is actually true," about the software or its environment may be referred to but seldom known exactly. Rather one has, "What is measured, observed, inferred, etc." regarding items relevant to the software's security – when properly achieved and organized including the uncertainties about them, these constitute its assurance case.

Each item's uncertainty may have assurance steps taken to reduce it, but ultimately the result is the combined residual uncertainty(ies) that is a basis for an individual's or organization's level of confidence (and any uncertainty in it). These uncertainties are about values that might, for example, lead to the conclusion that the software is adequate or – possibly more commonly today – that it is inadequate.

Thus, one has values, say of technical characteristics or sampling results, and uncertainty about them. One must make decisions such as whether to use a software system in a particular sensitive situation. Human judgment is involved. Is the software system worthy of being trusted to preserve certain security properties in a given situation – a delineated trustworthiness?[3] Are potential consequences and risks acceptable? Knowing these is generally not enough, as the most important consideration is often, "What choices does one have?" Making the decision to have sufficient confidence in a system's adequacy or to use it (with or without that confidence) are of a different nature from the more engineering and objective issues in the assurance case.

Table 4: Alternative Sources of Definitions

Abbreviated Name	Bibliography Reference
CNSSI 4009 National Information Assurance Glossary	[CNSSI 4009]
Common Criteria	[Common Criteria Part 1] [Common Criteria Part 2]
DCID 6-3 Protecting Compartmented Information Appendices	[DCID 6/3 2000]
IATF IS Security Engineering	[NSA 2002]
ISO 9126 Quality Characteristics	[ISO/IEC 9126:1991]
ISO IEC TR 15443 IT Security Assurance	[ISO TR 15443 - 1]
DIMACS Workshop 2002	[McGraw 2003]
FIPS PUB 199 Security Categorization	[FIPS PUB 199]
NIST Special Publication 800-33 Underlying Technical Models for Information Technology Security	[NIST Special Pub 800-33 2001]
NIST Special Publication 800-64	[NIST Special Pub 800-64]
NSTISSI 1000 NIACAP	[NSTISSP No. 11]
OWASP Guide	[OWASP 2005]
Trusted Software Methodology	[SDI 1992]
SafSec Standard	[SafSec Standard]
NRL Handbook	[NRL Handbook 1995]
Safety and Security CMM Extentions	[Ibrahim et al, 2004]

The meaning of a statement about a property also depends on the situational assumptions behind a particular specification or claim. Examples of these assumptions include:

- The bits constituting the software constitute the entirety of what is within an environment where everything it depends on (including assumptions) will work exactly as (completely) specified – say, that the software is the only software on a computer or device whose hardware will work as specified.
- The software in question is not alone within its machine environment with possibly multiple trust domains.
- The environment of the software is or is not predictable and is or is not trustworthy.

[3] "Trustworthiness" is thus dependent on the situation and the entity extending the trust (e.g. entity's degree of risk aversion) and not an invariant characteristic of the software

- The environment of the software potentially allows attempts (or actual) changing of or tampering with the software either readily or with some "resistance" from the environment.

The first of these is an ideal situation that is nevertheless a useful assumption for some analyses and a relevant goal for systems design. The meaning often derives from how much is included in the "system" and the truths (or assumptions) about its environment's interactions with it. The more the software must depend on its environment, the more its environment is undependable or hostile, and the more uncertainties enter into the situation; then the more:

- Difficulties may exist in showing the desired properties are established and maintained;
- Residual uncertainties may exist about whether they really are established and preserved.

Finally, it is important to understand the context in which terms that are used within the security community such as threat[4] or assurance are being used, i.e., whether the speaker (or writer) is referring to an entity, capability, condition, event, consequence (computing or real-world), physical or mental state, action, or process.

Numerous dictionaries and glossaries for security, information assurance, and software engineering exist and varying in their definitions of both basic and advanced terms.[5] A number of these are listed in

. While several have strong proponents and persons who will claim a particular glossary's definition of a term is the only right one, these vary on organization and individual bases.

The variety of usages and subtleties are numerous for many terms. Thus, one needs to carefully keep firmly in mind the underlying concepts while understanding the relevant aspect, assumptions, and definitions behind any statement about security properties or characteristics.

3.3 Basic Concepts

3.3.1 Dependability

Dependability is a qualitative "umbrella" term. Avizienis et al. offer a concise set of definitions for the properties related to dependability, including security [Avizienis 2004, p. 13].

"As developed over the past three decades, dependability is an integrating concept that encompasses the following attributes:

- Reliability: continuity of correct service.
- Safety: absence of catastrophic consequences on the user(s) and the environment.
- Maintainability: ability to undergo modifications and repairs. …
- Integrity: lack of improper modification, alteration, or destruction.
- Availability: continually, reliably accessible and usable in a timely manner[6].

[4] Here is an example of conflicting definitions with entity versus condition or event (emphasis added) – **Threat**: an **adversary** that is motivated to exploit a system vulnerability and capable of doing so. National Research Council (NRC) Computer Science and Telecommunications Board (CSTB): *Cybersecurity Today and Tomorrow: Pay Now or Pay Later*. Washington, DC: National Academies Press, 2002. – ***VERSUS*** **– Threat**: Any **circumstance or event** with the potential to adversely impact an IS through unauthorized access, destruction, disclosure, modification of data, and/or denial of service. [CNSSI 4009]
[5] For some sources on the WWW, see for example, Google's "Definitions on the web" feature and Microsoft's list of resources to Decode Technical Jargon available at http://www.microsoft.com/learning/start/terms.asp.
[6] Availability may include availability to share.

When addressing security, an additional attribute has great prominence,

- Confidentiality: Preservation of authorized restrictions on information access and disclosure, including means for protecting personal privacy and proprietary information."

3.3.2 Security

"Security is a composite of…three…attributes – confidentiality, integrity, and availability. Security [of information] often requires the simultaneous existence of 1) availability for authorized actions only, 2) confidentiality, and 3) integrity with 'improper' meaning 'unauthorized'."[7] [Avizienis 2004, p. 13]

While exhibiting reliability, safety, and maintainability may not directly result in secure software, the last can contribute to keeping security "up to date." All may make it easier to show that software is secure.[8]

Security is not as simple as this sounds. Neither confidentiality nor integrity can be achieved unless identities can be firmly established – authenticated – and only allowable actions are permitted or possible – access control or in some situations encryption. Two related properties are also important [Landwehr 2001]

- Accountability: being able to associate actors with their acts; to include non-repudiation (ensuring actors are unable to effectively deny (repudiate) an action)

Security properties are properties of the whole system. This means that these properties are determinable or observable only[9] in the context of how the multiple components of the system interact among themselves, and how the system responds to stimuli from external entities (e.g., input from users) – thus said to emerge or be "emergent" with system composition.

While necessary in the majority of systems, the mere presence of security functionality does not make a system secure. To provide security for an item, individual pieces of security functionality must be impossible to bypass, corrupt, or cause to fail. Given accurate facts about its environment, these inabilities to corrupt, cause to fail, and bypass can emerge from the inherent properties of the software itself, but dependencies including those on hardware mean these inabilities must ultimately (also) be achieved at the system level.

Since secure software is preserving what might be considered systems level properties and may protect many kinds of computing resources such as data, software, and running processes; and does so in a systems context; the distinction between software- and system-level security concerns is fuzzy and frequently irrelevant. Thus, this document generally avoids trying to label topics with these terms. On the other hand, system-subsystem relationships and differences in levels of abstraction are important within software security.

When attacks occur, the software may also be required to detect those attacks and alert users, continue service, confine damage, rapidly recover, and aid in diagnosis and repair to prevent future problems.

Another central issue concerning security properties is

- Assurance: addressing the question of the amount of uncertainty one should have regarding whether the software system will preserve these properties.

It will be covered in some detail in subsection 3.3.4.

---

[7] Another definition of security is, "All aspects related to defining, achieving, and maintaining confidentiality, integrity, availability, accountability, authenticity, and reliability." [ISO/IEC13335-1].

[8] For further information on this topic, see *Security in the Software Lifecycle*, Section 3.1.3, "Other Desirable Properties of Software and Their Impact on Security".

[9] This does not mean that (1) similar or analogous properties may not exist at lower levels or that (2) one might not design a system so a guarantee can be derived from the behaviors of only a portion of a system and a lack of opportunities (to violate or help violare security) for the remainder.

Security was defined in terms of general kinds of properties. However, a specific software system requires specific specified properties within these categories. These specific properties reflect the greater or lesser concern for each kind of general property related to the system and portions or aspects thereof – for example, confidentiality may only be relevant to a portion of the system's data and integrity concerns relevant to certain operations involving certain data. In addition, the level of assurance (uncertainty) desired regarding each may vary with the level or seriousness of each concern that in turn usually reflects the possible consequences in the real world as well as the digital one.

3.3.3 Software and other Security-related Concerns

In addition to a software system's preservation of required security properties within its digital domain, it can contribute to other systems, organizational, or societal security goals including:

- Establishing the authenticity of users and data;
- Establishing accountability of users;
- Providing usability including transparency to users to gain acceptance and resulting security;
- Providing the abilities to:
  - Deter and mislead attackers,
  - Force attackers to overcome multiple layers of defense,
  - Support investigations to identify and convict attackers.
- Aiding physical security, such as in monitoring and entrance control
- Protecting privacy

Privacy needs are one of the key reasons for security. Privacy is a motivation for confidentiality, anonymity, and not retaining data. Avoiding falsehoods that could damage reputations requires data accuracy and integrity. See the Ethics, Laws, and Governance section for mention of some relevant laws and regulations.

Thus, software can help address security concerns at a number of levels.

3.3.4 Assets

An asset may be information or data,[10] an executing process, or anything else that is of value to stakeholders.[11] Often, the relevant stakeholders are its owner and attacker, but may also be society or the entity about which the data relates. Developers of secure software must identify assets and their protection needs. These asset-protection needs derive from needs to protect or benefit stakeholders.

Assets may be categorized by attributes related to their confidentiality (e.g., Top Secret, Secret, Confidential, or Unclassified); by their degree of integrity (e.g., accurate and up to date versus old and with unknown accuracy); or by criticality of availability or acceptable level of unavailability (e.g., unavailable less than one minute in every 30 days).

[10] Federal Information Processing Standard (FIPS) 199, *Standards for Security Categorization of Federal Information and Information Systems.* Gaithersburg, MD: NIST, February 2004, defines a standard methodology for categorizing information (or data) assets according to their security attributes. FIPS 199 is available at http://csrc.nist.gov/publications/fips/index.html (February 2004).

[11] Assets about whom security-releted properties are to be preserved, do not necessarily include everything that might be at risk or damaged. Examples include the wealth and reputations of persons about whom information is kept and those assets about which the contents of the system could facilitate malicious actions of any kind – e.g. spies, soldiers, shareholder value, facilities, infrastructure, and potential objects of safety hazards, terrorism, or criminality.

3.3.5 Security-Violation-related Concepts

Threatening entities, also referred to as threat sources, threat agents, and attackers, may possess capabilities, resources, and intentions that enable them to pose threats to system-related assets (including software components of systems).[12] To perform their attacks, attackers use their capabilities to take advantage of, or exploit, vulnerabilities – "weaknesses…that could be exploited or triggered by a threat source" [NIST FIPS 200] – in the system.[13] Attackers use specific kinds of attacks or "exploits", often falling within recognizable patterns referred to as "attack patterns", to take advantage of particular vulnerabilities in a system. Systems may employ countermeasures to reduce the ability to exploit vulnerabilities (or weaknesses) and others to reduce the extent and intensity of the damage that would result from a successful or partially successful attack.

3.3.6 Assurance

As shown the aftermath of the Soviet pipeline explosion described in the prior section, we need justifiable confidence prior to depending on a system, especially a software system. The greater the dependence, the greater the need for objective grounds for confidence that this dependence is well placed. One needs the appropriate valid arguments and evidence to establish a rational basis for justified confidence for any claims. The greater the dependence on a system, the greater the need for objective grounds justifying confidence that this dependence is well placed.

Assurance is a term whose usage varies, but they all relate to reducing the level of uncertainty in estimation, prediction, information, inference, or the achievement of a goal. Such a reduction may provide an improved basis for justified confidence.[14] For example, the effort spent in reducing uncertainty about the value assigned to a parameter—thereby increasing assurance in that value—can often be cost-effective in that it improves the basis for decision-making even if the value remains unchanged.

Examples of ways in which the word "assurance" is sometimes used include:

- Actions taken to provide a basis for justified confidence – these actions may constitute how something is done, or the evaluations of something or how it is/was done;
- Arguments and evidence that reduce uncertainty or provide grounds to justify confidence;
- Degree an individual or organization has of justified confidence in something such as the justifiable confidence that a system exhibits all of its required properties and satisfies all of its other requirements.

Assurance may relate to any characteristic of software or systems. This document primarily addresses *security* assurance of software systems.

To be usable in hazardous or hostile environments, a secure software product needs two parts: (1) the "operational product," accompanied by (2) the "assurance case" for that product. The assurance case provides a basis for the confidence end-users need in the product before they feel comfortable using it and it may provide the basis for confidence that the producer needs before releasing the product. These two levels of confidence –

[12] While the literature often does so, this guide seldom uses the term "threat" without a modifier because when used alone it is may have several different meanings

[13] For many purposes such as creating coding standards the meaningfulness and need to separate "vulnerability" causing items from other weaknesses may be low or non-existent. In addition, a question always exists about the current and future contexts that are relevant for "could be exploited or triggered".

[14] According to the Merriam-Webster Dictionary, confidence is "the quality or state of being certain". Trust is related to confidence, but trust is often bestowed without explicit justification for confidence. The entity trusted may or may not be trustworthy. In order to prepare for violations of trust, computer systems enforce accountability. Moreover, an entity's history of behavior will directly affect that entity's reputation. Particularly in e-commerce, mechanisms exist for publicizing each buyer's and seller's accountability, history, and reputation in order to provide evidence that others can then use in making their own determinations of whether confidence in the entity is justified, and thus whether they should trust the entity.

to release the product or to use it – and the evidence required to obtain each, might be different. The user relying only on claims made by the producer's assurance case requires the case for releasing to be at least as strong as that for use.[15] The assurance case can provide reduced uncertainty leading to justified confidence in a software system's adequacy, or if the situation warrants, it might provide grounds for rational lack of confidence. This "assurance case" is discussed below in subsection 3.3.6.1.

To support the assurance case, software assurance commonly needs the execution of a planned and systematic set of activities to provide grounds for confidence in software properties. These activities are designed to ensure that both software processes and products conform to their requirements, standards, and defined procedures [NASA Guidebook, I.A]. "Software processes", in this context, include all of the activities involved in the design, development, and sustainment of software; "software products" (sometimes referred to as "artifacts") include the software itself, the data associated with it, its documentation, and supporting and reporting paperwork produced as part of the software process (e.g., test results, assurance arguments). In the case of secure software, the "requirements" will include requirements for the security properties that should be exhibited by the software. The "standards" may be technical, defining the technologies that can be used in the software, or they may be non-technical, defining aspects of the process that are further delineated by the "procedures" that make satisfaction of the software's requirements possible.

One could think of three mutually supportive and sometimes overlapping sets of activities in the software life cycle – *management, engineering,* and *assurance*. Software management is the set of activities for planning, controlling, and directing the software engineering and assurance activities. In many if not most software projects, engineering includes not just production of software products and their evaluation but evaluation and selection of reusable software components (these may commercial software products, open source programs, legacy components, or software stored in a purpose-built reuse repository). Software assurance ensures the management and engineering efforts result in a total product that satisfies all of its requirements or claims and can be shown to do so.

3.3.6.1 Assurance Case

As mentioned above, a secure software product that is rationally usable in dangerous situations has two aspects, (1) the "operational product" part accompanied by (2) the "assurance" part providing the grounds for the confidence required.[16] This confidence-justifying second part of the "total product" is called its "assurance case." Such a case needs to exist whether written or not – otherwise an essential element needed for use is missing – as highlighted by the aftermath of the Soviet gas pipeline explosion. Thus, the assurance case is central to rational use of software where security is a serious concern.

An Assurance Case[17] is "a reasoned, auditable argument created to support the contention that a defined system will satisfy the…requirements," UK Ministry of Defence Standard 00-42. [Ministry of Defence 2003b, section 4.1] Most assurance cases include:

1. one or more claims about the required properties of the system,
2. a body of evidence supporting those claims,

[15] It has been suggested by comments that: While it is possible to add grounds for confidence with activities such as independent evaluation, the bulk of the wherewithal for assurance might be expected to be satisfied as a by-product from processes that produce high assurance product for how else would the producers and their superiors rationally have and maintain high confidence themselves. (And that the absence of such by-products is grounds supporting a determination of lower confidence.)

[16] While the term "operational product" is used here, concern for understanding, use, and evolution (as well as assurance) result in the need for a number of engineering artifacts such as specifications and designs that although some might not consider "operational" are nevertheless part of the "product" during its operational period.

[17] Sometimes called an "assurance argument;" in this guide the term "assurance argument" or just "argument" is used for the arguments that connect the evidence to the assurance claims/conclusions.

3. arguments that clearly link the evidence to the claims.

In this document, the required properties that are relevant are the system’s security properties. [Williams 1998] provides an extensive introduction to security assurance cases for software systems.[18]

The assurance case’s argument must, of course, be supported by evidence that supports each part of the assurance case. Such evidence comes in many forms including results from tests, mathematical proof checkers, analyses, and reviews as well as process and personnel quality. (See Table 5: Kinds of Evidence and the Verification, Validation, and Evaluation section for further information.) Though not covered, a combined assurance case could include security and safety [SafSec Standard][19] as well as possibly other related requirements.

An assurance case addresses the reasons to expect and confirm successful production of the software system and the possibilities and risks identified as difficulties or obstacles to developing and sustaining a secure software system. The assurance case provides significant stakeholders with an objective basis for their justified degree of confidence and decision making. To convince them successfully, the possibilities and risks they perceive must be addressed – whether developers believe them to be merited or not.

Starting with the initial concept and requirements, the assurance case subsequently includes experienced or postulated possibilities and risks; avoidance and mitigation strategies; and an assurance argument referring to associated and supporting evidence from design and construction activities, verification activities, tests and trials, etc. This may eventually include in-service and field data. Any substantive modifications in the software system or its security requirements will necessitate changes to the assurance case.

The assurance case provides a structure of goals and sub-goals (or claims and sub-claims) with evidence and connecting arguments that show the achievement of the top-level security goal(s) (or claim(s)). The [SafSec Standard] and [SafSec Guidance] documents address this structuring of the assurance case. Crudely, the two principal argument patterns are (1) everything necessary went or is right or close enough and (2) nothing significant went or is too wrong. SafSec primarily uses the latter. Other sources include [Ministry of Defence 2003b] and [Howell 2003].

Any sound approach to producing (or using) secure software must address three points,

1. Specified security properties are valid and meeting them will result in meeting real world intentions and expectations
2. System as designed, built, deployed, and executed will meet its specified security properties
3. Valid, justified knowledge of the degree to which 1 and 2 have or are being achieved and the uncertainty related to this knowledge

The third point is addressed by assurance cases. This point can be important not only for contribution towards gauging feasibility or suitability for release or use but also for corrective action, learning, and improvement. The first two points are the principal potential objectives whose achievement assurance cases address. In practice, depending on its criticality and risk, the first may or may not be the subject of a formalized assurance case. The second is normally complex enough to be difficult to comprehend and ensure its correctness, unless it is systematically recorded and reviewed.

[18] Internet sites with material aiding in learning about assurance cases – albeit with a safety emphasis – include http://adelard.co.uk/iee_pn/ and http://www.esafetycase.com/

[19]The objective of SafSec, developed by Praxis High Integrity Systems is to provide a systems certification and accreditation (C&A) methodology that addresses both safety and security acceptance requirements. SafSec was originally developed for C&A of the United Kingdom Ministry of Defense (UK MOD) Integrated Modular Avionics, and other advanced avionics architectures. SafSec is an example of how approaches from assurance of safety as a required property of software have been successfully applied to the assurance of security properties of software.

In the second, the top goal(s) must include the security properties required and could perhaps include the software system's entire set of security-related requirements for the system. One example might have top goals composed of the combination of required security functionality and properties (e.g. that this functionality cannot be bypassed) as in the Common Criteria v. 3.0 [CC 2005] discussed below.

To address feasibility or suitability for a use, knowledge concerning a fourth point is also needed.

1. Valid knowledge concerning the potential consequences or risks related to point 3 (degree and uncertainty regarding achievement of points 1 and 2)

Risk management and the assurance case factor these in to establish the comprehensive, net or residual potential consequences or risk. These may be balanced against benefits in making decisions.

As a living, top-level control document, an assurance case needs to be managed with its status reported periodically including progress in arguments and linked evidences. The case remains with the software system throughout its life through disposal. An assurance case contains, therefore, a progressively expanding body of evidence built up during development, and responds as required to all changes during development and sustainment.

A large variety of kinds of evidence can be relevant to improving the grounds for justified confidence. Table 5 lists a number of them.

Issues arising from the assurance case arguments for security and the different qualities can highlight tradeoffs of other qualities or functionality against security.

Table 5: Kinds of Evidence

1. The quality and history of people
2. The characteristics, suitability, and history of processes used
3. Data on the quality and fidelity of use of process
4. The quality and suitability of the development environment
5. Production organizational structure characteristics and suitability
6. Security of production
7. The realism of the assumptions made
8. Quality of safety or security policy
9. Agreement of informal and formal statements of security properties (and/or other properties, e.g. safety)
10. Specification quality and consistency with these properties
11. Consistency from representations have confidence in (e.g. specification) through intermediate representations (e.g. design, source code) to the released software
12. Security of deployment
13. Security of installation
14. Security of updating the software
15. Software executed is same as what was installed (and updated)
16. And on across lifespan and possible situations (e.g. stolen laptop)
17. Lack of non-required or unspecified functionality
18. Analysis of structure of system and its interactions
19. Chain of evidence: valid creation and collection of evidence with its integrity maintained until and during its use
20. Proper use of assurance argument and evidence
21. Design including defense in depth and tolerance
22. Characteristics of the code including good practices and use of a safe programming language (subset)
23. Static analysis of code for relevant aspects such as
    a. Vulnerability-prone features
    b. Exceptions
    c. Information flow
    d. Conformance to coding standards
    e. Correctness of proofs
24. Nature and results from
    a. Reviews
    b. Audits
    c. Tests
    d. Measurement (direct or surrogate)
    e. Analyses
    f. Simulations
    g. Proofs
25. Data on the execution history of the software itself – particularly for post-update assurance
26. Ease of recovery and repair
27. Quality of support (particularly response to discovery of defects or vulnerabilities)
28. Evidence from suppliers or other sources
29. Warranty or risk sharing agreement and its creditability

An assurance case will be more complete if it includes consideration for possibilities and risk values that are:

- Known items – ensuring none are overlooked;
- Known kinds of items with unknown existence or characteristics

For completeness, one might also consider the possibility of unknown unknowns – items not known to exist (or be relevant) and nobody knows their value.

What are sometimes called “security assurance activities” for software overlap (possibly entirely) with engineering activities and include those directed towards security evaluations of both the software product and the processes used to develop and sustain it. Security evaluations of the product ensure its security requirements are appropriate and have been satisfied in the product. Security evaluations of the process ensure that none of the development or sustainment activities have compromised the software product’s security, and that the software product (including all of its artifacts), and its development and support tools and supporting control and report data have been consistently controlled and protected. The configuration management and corrective action activities within the software's engineering and management processes should be evaluated to ensure they protect all software artifacts and supporting data from modification or deletion by unauthorized persons at all times, and from modification or deletion by authorized users at inappropriate times. Changes to the software should also be evaluated and should be checked to ensure that no change has allowed a security violation to occur or a vulnerability to be introduced. Finally, physical security protection of the software product (all artifacts), control data, development environment (including tools, systems, and network), and the information the software accesses, manipulates, and creates should all be ensured to be adequate at all times [NASA Guidebook, VIII.C].

The activities related to creating and evaluating the assurance case obviously must also be identified in the project plans. Finally, a description of the proposed assurance case would normally appear in a proposal document during acquisition.[20]

3.3.6.2 Levels of Assurance and Confidence

The degree of confidence that can be justified based on a specific assurance case may vary by individual or organization. The less residual uncertainty the higher the level of assurance and presumably the higher the degree, or level, of justifiable confidence. Conceptually, this conversion of an amount of uncertainty into a level of justified confidence is not straightforward or well understood. The term, "assurance" is often prefixed with the modifier "high", "medium", or "low," and sometimes people use it where (level of) "confidence" or (amount or restrictiveness of) "protection" (mechanisms) should have been used.

Arguably, "high-confidence" is not a synonym for "high-assurance". It is possible to have a high level of unjustifiable confidence. "High confidence", then, indicates a level of actual confidence placed in the system, and not necessarily the lower uncertainty (higher "assurance") that would justify such confidence. To avoid confusion, this guide seldom uses the word "assurance" alone or without explanation.

3.3.6.3 Information Assurance

The term "information assurance" (sometimes referred to as IA) is often used as:

1. A catch-all term for all that is done to assure security of information;
2. The levels of uncertainty or justifiable confidence one has in that security.

The information assurance community (also sometimes labeled "information security" community) defines the term as referring to the collective set of measures that protect and defend information and information systems by ensuring their availability, integrity, confidentiality, and access control, and the authentication, authorization, and accountability of their users, as well as non-repudiation by those users of actions for which they are responsible. Information assurance measures include providing for restoration of information systems and the information they contain by incorporating protection, detection, and reaction capabilities. [CNSSI 4009]

To be considered "high assurance" by many people in this community, an information system would implement mechanisms judged to significantly hinder unauthorized parties from accessing the information, and authorized parties from using the information in unauthorized ways. Some high-assurance systems might be referred to as "trusted systems" although this term might better be used related to the act of placing trust rather than characteristics of the system.

While concern for the engineering, production, and internals of software fall naturally within these concerns; many in the discipline of "information assurance" do not consider them. This can sometimes lead to communication problems across disciplines. Nevertheless, the communities share many concepts and concerns.

3.3.6.4 Security Evaluations, Certifications, and Accreditations

A number of certifications for software system security and accreditations of operational systems – software, hardware, polices and procedures, facilities, physical safeguards (e.g. protective casings), user aids, and people

[20] Also see the Assurance Case section of the *Build Security In* website [DHS BSI].

– exist. Historically, these have been applied primarily to government systems. Professional certifications for individuals exist but usually target network security or “unsecure” software engineering. These activities are discussed again in Section 8, Secure Software Verification, Validation, and Evaluation.

3.3.6.4.1 Common Criteria Certification

This certification process uses first an authorized laboratory to do an assessment followed by a government validation of any recommended certification. [Prieto-Diaz 2003] [Merkow 2005] The criteria are “common” in that they are shared internationally as the result of agreement among a number of countries that formerly had differing criteria.

The Common Criteria standard [CC 2005] contains

- Enumeration of security functionality (e.g. authentication, logging)
- Evaluation Assurance Levels (EAL) 1 (low) through 7 (high) calling for increased levels of documentation, formality, and testing
- Methods to be used by those doing the evaluations and certification

Associated with the Common Criteria are a set of Protection Profiles – standard minimal security requirements, usually lists of required functionality – for a number of kinds of software.

Historically, the Common Criteria and associated Protection Profiles have identified security-oriented functionality to be required of systems [Common Criteria v.3.0, Part 2] – along with an assessment and valuation process. The Common Criteria now also calls for self-protection and non-bypassability of security functionality as well as protection for the boundaries between trust domains – that is between areas with different required security properties or, in practice, different ownership (or different top-level entity responsible for security). [CC 2005, Part 3 pages 96-101]

3.3.6.4.2 FIPS 140 Certification of Cryptographic Software

The US Government National Institute for Standards and Technology (NIST) performs certifications of cryptographic software to standards set by Federal Information Processing Standard (FIPS) 140. This has become a de facto standard for cryptographic software even beyond the US government and considered essential by almost all knowledgeable users of such software.

3.3.6.4.3 BITS Certification

The Financial Services Round Table BITS certification program aims at achieving a set of security functionality suitable for banking and financial services and a certification process simpler than that of the Common Criteria. See http://www.bitsinfo.org/c_certification.html.

3.3.6.4.4 System Accreditation

The US DoD accreditation process applies to operational systems and is governed by DoD Instruction 5200.40, DoD Information Technology Security Certification and Accreditation Process (DITSCAP)[21] supplemented by the DoD8510.1-M Application Manual.

On the civilian side of the US government, the National Institute for Standards and Technology has produced NIST Special Publication 800-37 Guide for the Security Certification and Accreditation of Federal Information Systems. [NIST Special Pub 800-37]

Internationally, ISO/IEC 17799:2005 Information technology. Code of Practice for Information Security Management[22] combined with ISO/IEC 27001 (formerly with UK standard BS7799-2:2002) form a basis for an Information Security Management System (ISMS) certification (sic) of an operational system.

[21] NIACAP covered in Section 5 is also used within DoD

3.3.6.4.5 Professional Certification

The safety community (e.g. commercial aviation) has utilized certification (or licensure) of key personnel as part of its approaches. A number of computer security certifications exist from management-oriented ones to technical ones about specific products – for example, certifications from the International Information Systems Security Certification Consortium $(ISC)^2$ and the SANS Institute. Unfortunately, these so far have concentrated on network and organizational security.

Likewise, several software engineering-related certifications exist but have no significant coverage of security issues. Security, however, is beginning to take a more prominent place in vendor certifications such as the Microsoft Certified Application Developer (MCAD) and Solution Developer (MCSD) certifications where optional examinations now exist specifically focused on security.

3.4 Basic Software System Security Principles

Saltzer and Schroeder published their list of principles in 1975, and they remain valid. [Saltzer and Schroeder 1975] Everyone involved in any way with secure software systems needs to be aware of them. Most of the list below follows the principles proposed by [Saltzer and Schroeder 1975] and liberally quotes edited selections from that text. [Viega and McGraw 2001] has a particularly accessible discussion of many of them. These principles have relevance throughout secure software development and sustainment including requirements; design; construction; and verification, validation, and evaluation.

They cover a number of topics. Several principles help in reducing the number of opportunities for violations. As opportunities or possibilities for violations cannot always be eliminated, steps need to be taken to ensure users properly utilize security and efforts toward security are expended in the best places. To reduce the uncertainties related to the adequacy or correctness of the software system, the portion of the system and mechanisms ensuring security should be as small and simple as practicable and the be thoroughly reviewed and analyzed.

Defenses and protection may not be perfect, and violations will occur. For follow-up, learning, and improvement records of what occurred are needed. In addition, requiring multiple successes by an attacker before substantial damage results can increase time or effort attacker needs to expend and provide some tolerance for vulnerabilities or weaknesses.

3.4.1 Least Privilege

Least privilege is a principle whereby each entity (user, process, or device) is granted the most restrictive set of privileges needed for the performance of that entity's authorized tasks. Application of this principle limits the damage that can result from accident, error, or unauthorized use of a system. Least privilege also reduces the number of potential interactions among privileged processes or programs, so that unintentional, unwanted, or improper uses of privilege are less likely to occur.

3.4.2 Complete Mediation

Every access to every (security-sensitive) object must be checked for proper authorization; and access denied if it violates authorizations. This principle, when systematically applied, is the primary underpinning of the protection system, and it implies the existence and integrity of methods to (1) identify the source of every request, (2) ensure the request is unchanged since its origination, and (3) check the relevant authorizations. It also requires that design proposals to allow access by remembering the result of a prior authority check be examined skeptically.

[22] Expected to become part of the 27000 series.

3.4.3 Fail-Safe Defaults

This principle calls for basing access decisions on permission rather than exclusion. Thus, the default situation is lack of access, and the protection scheme identifies conditions under which access is permitted. To be conservative, a design must be based on arguments stating why objects should be accessible, rather than why they should not.

3.4.4 Least Common Mechanism

Minimize the security mechanisms common to more than one user or depended on by multiple users or levels of sensitivity. Whenever the same executing process services multiple users or handles data from multiple security levels or compartments, this potentially creates an opportunity for illegitimate information flow. Every shared mechanism (as opposed, for example, to non-communicating multiple instances) represents a potential information path between users or across security boundaries and must be designed with great care to ensure against unintentionally compromising security. Virtual machines each with their own copies of the operating system are an example of not sharing the usage of the operating system mechanisms – a single instance is not common across the users of the virtual machines. Thus, one desires the least possible sharing of common mechanisms (instances).

3.4.5 Separation of Privilege

A protection mechanism that requires two keys to unlock it is more robust and flexible than one that allows access to the presenter of a single key. By requiring two keys, no single accident, deception, or breach of trust is sufficient to compromise the protected information.

Redundancy is also used in the traditional "separation of duties" in human financial processes, e.g. the persons who fill out the check and sign it are two different people.

3.4.6 Psychological Acceptability

It is essential that the human interface be designed for ease of use, so that users routinely and automatically apply the protection mechanisms correctly.[23]

3.4.7 Work Factor

The cost of performing to a hiher level of quality or of a countermeasure to eliminate or mitigate a vulnerability should be commensurate with the cost of a loss if a successful attack were to otherwise occur. Generally, the more valuable the asset targeted by an attacker, the more effort and resources that attacker is willing expend, and therefore the most effort and resources the defender should expend to prevent or thwart the attack.

3.4.8 Economy of Mechanism

"Keep the design as simple and small as possible" applies to any aspect of a system, but it deserves emphasis for protection mechanisms, since design and implementation errors that result in unwanted access paths may not be noticed during normal use.

3.4.9 Open Design

Security mechanisms should not depend on the ignorance of potential attackers, but rather on assurance of correctness and/or the possession of specific, more easily protected, keys or passwords. This permits the

[23] Usable security is a significant issue and is addressed in both the Requirements and Design sections.

mechanisms to be examined by a number of reviewers without concern that the review may itself compromise the safeguards.

The practice of openly exposing one’s design to scrutiny is not universally accepted. The notion that the mechanism should not depend on ignorance is generally accepted, but some would argue that its design should remain secret since a secret design may have the additional advantage of significantly raising the price of penetration. The principle still can be applied, however, restricted to within an organization or a “trusted” group.

3.4.10 Analyzability

Systems whose behavior is analyzable from their engineering descriptions such as design specifications and code have a higher chance of performing correctly because relevant aspects of their behavior can be predicted in advance. In any field, analysis techniques are never available for all structures and substances. To ensure analyzability one must restrict structural arrangements and other aspects to those that are analyzable.[24]

3.4.11 Recording of Compromises

If a system’s defense was not fully successful, trails of evidence should exist to aid understanding, recovery, diagnosis and repair, forensics, and accountability. Likewise, records of suspicious behavior and “near misses”, and records of legitimate behaviors can also have value.

3.4.12 Defense in Depth

Defense in depth is a strategy in which human, technological, and operational capabilities are integrated to establish variable protective barriers across multiple layers and dimensions of a system. This principle ensures that an attacker must compromise more than one protection mechanism to successfully exploit a system. Diversity of mechanisms can make the attacker’s problem even harder. The increased cost of an attack may dissuade an attacker from continuing the attack. Note that multiple less expensive but weak mechanisms do not necessarily make a stronger barrier than fewer more expensive and stronger ones.

3.4.13 Treat as Conflict

Because many software-security-related issues and activities concern a conflict pitting the system and those aiding in its defense against those who are attacking or may attack in the future, one needs to bring to the situation all that one can of what is known about conflicts. This includes recognizing when attacks in the computer security arena are part of a wider conflict. Below are some of the key concepts.

3.4.13.1 Intelligent and Malicious Adversary

As enumerated in Section 2, the threats or hazards faced may come from a number of sources including intelligent, skilled adversaries. When possession, damage, or denial of assets is highly valuable to someone else, then considerable skill and resources could be brought to bear. When poor software security makes these actions relatively easy and risk free, even lower value assets may become attractive.

One cannot simply use a probabilistic approach to one’s analyses because, for example, serious, intelligent opponents tend to attack where and when you least expect them – where your estimate of the probability of such an attack is relatively low.[25]

[24] Certainly one would never want a critical structure such as a bridge to be built using a structural arrangement whose behavior could not be analyzed and predicted. Why should this be different for critical software?
[25] In theory, game theory techniques that do not require the probabilities to be known could be applicable, but little progress has been made in practice.

Where judging risks is difficult, one possible approach is to make them at least not unacceptable (tolerable) and "as low as reasonably practicable" (ALARP).[26] In employing the ALARP approach, judgments about what to do are based on the cost-benefit of techniques – not on total budget. However, additional techniques or efforts are not employed once risks have achieved acceptable (not just tolerable) levels. This approach is attractive from an engineering viewpoint, but the amount of benefit cannot always be adequately established.

3.4.13.2 Security is a System, Organizational, and Societal Problem

Security is not merely a software concern, and mistaken decisions can result from confining attention to only software. Attempts to break security are often part of a larger conflict such as business competition, crime and law enforcement, social protest, political rivalry, or conflicts among nation states, e.g. espionage. Specifically in secure software development, the social norms and ethics of members of the development team and organization as well as suppliers deserve serious attention. Insiders constitute one of the most dangerous populations of potential attackers, and communicating successfully with them concerning their responsibilities and obligations is important in avoiding subversion or other damage.

3.4.13.3 Measures Encounter Countermeasures

Despite designers' positive approach to assuring protection, in practice one also engages in a software measure-countermeasure cycle between offense and defense. Currently, reducing the attackers' relative capabilities and increasing systems' resilience dominates many approaches. Anonymity of attackers has led to asymmetric situations where defenders must defend everywhere and always,[27] and attackers can chose the time and place of their attacks. Means for reducing anonymity – thereby making deterrence and removal of offenders more effective – could somewhat calm the current riot of illicit behavior.

3.4.13.4 Learn and Adapt

The attack and defense of software-intensive systems is a normal but not a mundane situation; it is a serious conflict situation with serious adversaries such as criminal organizations, terrorist groups, and nation states, and competitors committing industrial espionage. Serious analyses of past and current experience can improve tactics and strategy.

One should not forget how to be successful in conflicts. While it is difficult to state the principles of conflict in a brief manner, some principles exist such as exploiting the arenas in which the conflict occurs; using initiative, speed, movement, timing, and surprise; using and trading-off quality and quantity including technology and preparation; carefully defining success and pursuing it universally and persistently but flexibly; and hindering adversaries.

3.4.14 Tradeoffs

Tradeoffs exist between security and efficiency, speed, and usability. For some systems, other significant tradeoffs with security may also exist. Design decisions may exacerbate or ease these tradeoffs. For example, innovative user interface design may ease security's impact on usability. [Cranor 2005]

Attempting defense in depth raises the tradeoff between fewer layers of defenses each constructed with more resources or more layers each costing less resources. An attacker may overcome several weak lines of defense with less difficulty than fewer but stronger lines of defense.

[26] ALARP is a significant concept in UK law, and an excellent engineering-oriented discussion of it appears in Annex B of DEF STAND 00-56 Part 2 [Ministry of Defence 2004b]

[27] Defending everything may not be possible or may waste resources. "He who defends everything defends nothing." – Frederick II

Business tradeoffs also exist (or are believed to exist) for software producers between effort and time-to-market, and security. Finally, producers want users and other developers to use features, especially new features, of a product but the likelihood of this is reduced if these are shipped "turned off".

3.5 Safety and Security

In recent years, the software safety community has more examples of successful experience with producing high-confidence software than does the software security community. The safety community's experience provides lessons for software security practitioners, but the engineering safety problem differs from the security one in a critical way – it presumes non-existence of maliciousness. Today, security is a concern for most systems as software has become central to the functioning of organizations and much of it is directly or indirectly exposed to the Internet or to insider attack as well as to subversion during development, deployment, and updating. While safety-oriented systems so exposed now must also face the security problem, this subsection speaks of traditional safety engineering that does not address maliciousness.

3.5.1 Probability *versus* Possibility

The patterns of occurrences of "natural" events relevant to safety are described probabilistically for engineering purposes. The probability of a natural event contrasts with the need for concern for the possibility of an intelligent, malicious action. This distinction is central to the difference between facing safety hazards versus security threats (threatening entities, events, and consequences).[28]

3.5.2 Combining Safety and Security

When both are required, a number of areas are candidates for partially combining safety and security engineering concerns including:

- Goals,
- Solutions,
- Activities,
- Assurance case:
  - Goals/claims,
  - Assurance arguments,
  - Evidence,
- Evaluations.

The SafSec effort provides guidance on one way to do this. [SafSec Standard] [SafSec Guidance]

[28] To some in the safety community, a main distinction is not just probability/accidents vs. possibility/attacks but rather, whether human life (or substantial assets outside the system under scrutiny) is threatened in case of a malfunctioning system (which may malfunction due to accidents, attacks, or whatever). Since spies and soldiers have died because of security compromises, this clearly is a thought emphasizing the asset of most immediate concern. On one hand (safety), the system is visualized as affecting the real world causing hazards to be actualized and, on the other hand (security), as protecting the integrity, confidentiality, or availability of computerized data or computing processes. However, since in both cases the key cost elements are the real-world consequences, this distinction can be overstated.

3.6 Secure Software Engineering

The subsection covers several specific areas of fundamental knowledge about secure software engineering required across development, sustainment, reuse, and acquisition. [Gasser 1988, Chapters 1-4] [Bishop 2003, Chapters 1-2] Generally, engineering disciplines have knowledge of (1) the "right ways" to perform and (2) pitfalls or weaknesses – this is true for both the engineers and their products. While much of recent attention in software security has been on the latter, those doing secure software need to know both..

3.6.1 Stakeholders

Software system security-related development or evolution decisions can have a number of stakeholders who impose constraints or otherwise have a "stake" in the security of the system. A given development will typically have stakeholders from several of the following categories:

- National (possibly multiple nations') and international laws, regulations, treaties, and agreements
- Specific communities (such as government or the banking industry)
- Integrators of the software system into a larger product (e.g. OEMs or enterprise-wide application developers)
- Authorized units within an organization
- Sources of security-related policies (e.g. security, personnel, procurement, and marketing policies)
- Decision makers regarding acquisition and usage for software systems (including RFP writers and issuers as well as makers of final decision to acquire)
- System owners and custodians
- Owners and custodians of items in the system (e.g. data)
- Entities about whom the system contains information (e.g. customers and suppliers)
- System administrators
- Incident investigators, counter-intelligence, and forensic personnel
- End users
- Any non-users whose computational performance, results, or property might be impacted – e.g. entities whose software is executing on the same computer or on computers networked with it
- Standards bodies
- Software certifiers and system accreditors
- Software developers
- The general public
- The attackers

The stakeholders mentioned in the last bullet are important stakeholders who certainly "impose constraints or have interests involved". The existence and characteristics of potential or actual attackers strongly influences decisions. A given development effort may have more or less of these stakeholders involved, but it is the potential attackers and loss sufferers who make security a concern.

3.6.2 System Security Policy

Security policies for a system specify legitimate, secure states (or sometimes histories) of a system – or, alternately, illegitimate, insecure states. Conceptually, the required security properties determine these constraints on the software system states or behavior. For software, however, since the required security properties typically concern assets and actors plus conditions and context, they may require restatement or derivations in terms of the systems contents and the software's role and design.

In addition, organizations may have security policies where some contents do not map to these emergent properties of the system, but instead require, constrain, or prohibit the use of certain mechanisms, techniques, or products. While the system must also conform to these policies, they do not present the same design and verification difficulties as emergent properties such as integrity or non-bypassability.

The security policy for a system is partially derived from the societal (e.g., laws and regulations), organizational, and larger, encompassing system policies as well as the effects of interfacing systems. The remainder of a system's security policy is composed of particular policies needed for a specific system.

System security policies are based in part on organizational security policies that may address a long list of areas, including a policy requiring use of a secure software engineering process. Organizational security policies are covered in Section 4, Ethics, Law, and Governance.[29]

System security policies usually define the ways in which information processed by the system is allowed to (constrained to) be handled and the ways in which the system are allowed to interact with users. System security policies cover such areas as:

- Confidentiality of the information, and integrity and availability of both the system and the assets it contains;
- Identification and authentication of users (which may be humans or software processes);
- Access control (i.e., control of access to information and other system resources) and authorization of users (i.e., granting and revoking of rights/privileges to access system resources and information assets);
- Accountability of users for actions they perform while using the system, and non-repudiation by users of their actions
- Administration of system
- Self-protection of software
- Use of cryptography in the system;
- Forensics for tracing user activities.

Generic Access Control Policies

- **Discretionary** access control (user control or ownership of privileges, e.g. control of access privileges for items user creates)
- **Mandatory** access control (privileges never changeable by users – as opposed, for example. to system administrators)
- **Role-based** access control
- **Workflow** access control
- **Chinese wall** access policy (once one accesses on one side of the wall one cannot access on the other thereby avoiding conflicts of interest)
- **High latency** transactions with provisions/prerequisites and obligations (e.g. credit card transactions)

The security policy for a system often includes elements stated in access control policies. A number of conceptual access control policies exist. [Bishop 2003, p. 381-406] These policies can be used individually or in combination and may have a variety of mechanisms to support them. Some well-known kinds of generic access control policies include those shown in the text box at the right.

From a requirements viewpoint, important areas addressed by the software system's security policy may include:

[29] Areas potentially covered by organizational security policies are listed in subsection 4.4.4.

- Data protection.
- Identification and authentication.
- Communication security.
- Accountability.
- Self-protection of the system’s software and hardware.
- Underlying functionality such as random number generation and time stamping.

A single system may be governed by several security policies.

System security policies may be layered from the more general to the more (application) specific. The mechanisms supporting policy enforcement may also be layered, for example, with a bottom separation layer providing information flow and data isolation facilities that higher layers can use to define and enforce policies—such as application specific policies.

Separate policies may exist across a distributed system requiring their composition or reconciliation.

3.6.3 Specification Properties

Security properties in the software system’s security policy may be stated in terms of assets and entities – often by categories of assets or entities. At the technical level, a software system functionality or property might be formally reflected in a specification as a:

- Safety property: restriction on allowed system states;
- Liveness property: system states that it must reach; required progress or accomplishment;
- Information flow property: restricting flows of information.

Often it is necessary or convenient to state some security specifications in terms of information flow constraints or specifications [McLean 1994] including:

- Access control mechanisms and policies,
- Restrictions on information communicated,
- Noninterference,
- Separation properties,
- Covert channels limitations.

Thus, specifications of required security properties can be stated as properties (conditions or constraints) that must be true of the system, and the system’s functionalities’ behaviors might be specified as comparable properties allowing verification of the compliance of behavior with required security properties.[30]

3.6.4 Security-Related Architectural Concepts

General security-related architecture concepts include:

- Identities should be known for the entities involved and their privileges or authorizations must be known or able to be established

[30] If specified formally, this can allow static analysis of conformity of designs and code potentially adding creditable assurance evidence.

- Software systems should detect legitimate and illegitimate actions, recording both, and report, block, hinder, or make other effective responses to illegitimate actions: e.g., notification (e.g., alarm) and audit log.
- Tolerance may exist for some bad events including security violations: e.g. intrusion tolerance, fault tolerance, robustness (input tolerance), and disaster recovery. Achieving tolerance is discussed several places in this document.

While it provides no guarantees, deception may be useful for aiding concealment, intelligence gathering, and wasting an attacker's resources.

3.6.4.1 Identification and Authentication

Preservation of properties that specify who is allowed to do what requires good identification of the entities involved particularly the actors (e.g. user, process, or device). An initial identification might be done by the entity claiming an identity, by inference, from meta-data, or by recognition. Authentication is the verification of the initial identification of an entity, often as a prerequisite to determining the entity's authorization to access resources in a system. Authentication involves evidence used for the verification – authenticators. An authenticator may be something the entity knows or possesses the context or location of the entity, or an inherent characteristic of the entity. Authentication may be strengthened by requiring additional authenticators of the same or different kinds (e.g. two passwords or a password and a smartcard).

Neither confidentiality nor integrity can be achieved unless the identities of all actors (human and process) that interact with or operate on the object to be secured (software or data) can be firmly established – i.e., authenticated – and only allowable actions conforming to their access privileges or authorizations are possible.

3.6.4.2 Detect and Respond

Detection of an error state, or attempted or actual security policy violation is needed before action, e.g. notification or logging as such. A mechanism identified early in the history of software security is the

- Reference monitor – software module that enforces the authorized access relationships on every access request (reference) and whose implementation must be [Anderson 1972]
  - Tamper proof
  - Always invoked
  - Small enough to be subject to analysis and tests, the completeness of which can be assured

The idea was to have every request for access in the system (Complete Mediation) checked by a relatively small part of the system (Economy of Mechanism). Using only a single module or centralizing authorization information has scalability and manageability problems. This has lead to:

- Distributed access control,
- Layered access control.

The latter may rely on a

- Layered software system structure.
- Distributed access control
- Layered access control

The latter may rely on a

- Layered software system structure

Table 6: Tolerance-related Activities

• Forecasting violations
• Detection of violations
• Notification
• Damage isolation or confinement
• Continuing service although possibly degraded
• Diagnosis of vulnerability
• Repair
• Recovery of the system to a legitimate state
• Recording
• Tactics that adapt to attackers' actions

With or without layered access control mechanisms, such layered structure is often considered critical to effective software system assurance.

Related mechanisms deriving from network security include firewalls, intrusion detection systems, and intrusion prevention systems. [NSA 2002, Chapter 4]

3.6.4.3 Tolerance

Despite best efforts, one's software may not able to perform correctly or be suitable in unforeseen circumstances. For this reason, intrusion tolerance, fault tolerance, and input error tolerance (called by some robustness) are important concepts. [Avizienis 2004] The principles of Failsafe Defaults and Separation of Mechanisms are aimed at providing kinds of tolerance. Table 6 lists general Tolerance. Thus, one has

- Tolerance to (partial) attacker success – including "self healing" approaches

This may include automated system recovery.

3.6.4.4 Adaptive Response

Run-time adaptive defense can include moving processing elsewhere, switching to another copy of data, and changing levels of trust of objects and subjects including changing access rights because of their behavior. Intrusion tolerance can be aided by adaptive defenses. This includes

- Adaptive distributed reconfiguration responses to attacks [MAFTIA]

These adaptive responses approaches can be a possible approach to tolerance.

3.6.4.5 Deception

While deception should never be relied on exclusively, it has useful purposes such as gathering intelligence and misleading attackers to deter or confuse them or to cause wasting of their time and resources.

3.6.4.6 Sensitivity-related Separation

Many approaches attempt to keep data with different levels or compartments of sensitivity separated logically, physically, or by different encryptions. Users or access requestors are also often grouped by shared privileges or "levels of trust". Different rules or policies regarding security may be applied in different "trust domains" separated by explicit boundaries.

Several schemes exist for hierarchical multi-level or "horizontal" multi-lateral security where each level or compartment differs in the sensitivity or degrees of integrity of the assets it contains. Mandatory access control is often used for levels. [Neumann 1995] discusses a number of approaches.

- Multi-Level Security (MLS): both a generic term for multiple levels of sensitivity being involved within the same system and a name sometimes given to a specific scheme where all accesses are checked by a single software trusted computing base;
- Multiple Independent Levels of Security (MILS): bottom separation layer providing information flow and data isolation facilities so higher layers can define and enforce policies themselves [Vanfleet 2005];
- Multiple Single Levels of Security (MSLS): no intermixture,

To avoid the need for sensitivity-related separation some systems have been operated as

- System high – the entire system contents are treated as if they have the sensitivity of the most sensitive

In addition, information crossing the boundary between security domains with different policies (or in practice ownership) raises potential problems for both the originating domain (Should it be allowed to leave? Does it need to be encrypted first?) and the receiving one (Can it be trusted? How should it be handled?).

- Guards may be placed on one or both sides of a trust domain boundary (e.g. filters, guardians, or firewalls)

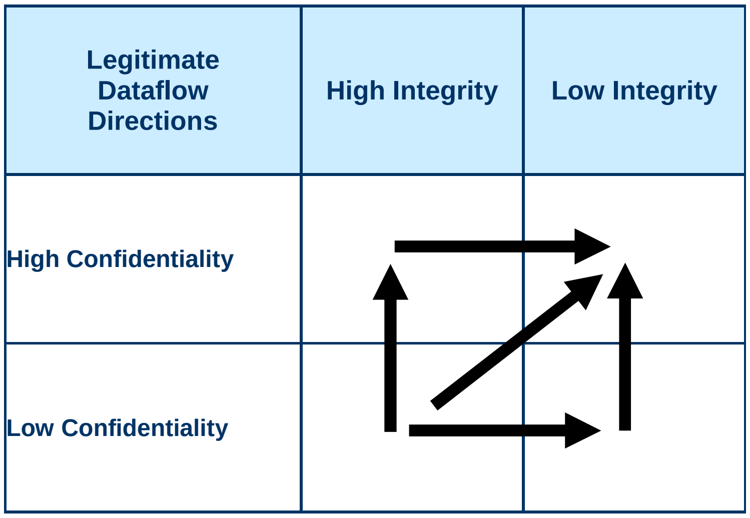
One particular problem is that information already encrypted – say to gain the benefits of end-to-end encryption – cannot be directly inspected at the boundary. Providing cross-domain solutions has become a niche area.

3.6.4.7 Damage Confinement and Isolation

If error states or damage occurs, one mitigation approach is to limit the potential damage by localizing it or reducing its size or effects. This may be done by detecting errors and ensuring correct values rather than the errors propagate to the rest of the system or externally. It can also be done, for example, by making valuable assets unavailable online.

3.6.4.8 Information Flow

Systems that must support information flow between two or more domains (with different security policies – or in practice ownership) must also control information flow among multiple security or sensitivity levels or divisions.



Consistent with the Bell-LaPadula security model [Bell-LaPadula] in multiple-level information systems information is allowed to flow freely from lower (less sensitive) confidentially domains to higher confidentiality ones and from higher integrity ones to lower integrity ones. The diagram at right shows the combined effects of these rules by showing the legitimate directions for information flows. Flowing the other ways, however, normally requires explicit actions to downgrade confidentiality level (e.g. declassification) or upgrade known integrity level (e.g. validation of input).

The following are mechanisms that may be used for information flows or accesses within or across domain boundaries.

- Compartmentalization via:
  - Virtual machines,
  - Separation via encryption,
  - Physical separation,
  - Separation except at point of use,
  - Filters, guardians, and firewalls
  - Access control sub-system (e.g. reference monitor module)

3.6.4.9 Access Control

There are a number of access control concepts, policies, and issues. [Bishop 2003, p. 381-406] [Gasser 1988, Chapter 6] Policies can be used individually or in combination. In addition to the generic access policies listed earlier, concepts and issues include:

- Access control process and mechanisms;
- Access control in distributed systems/databases;
- Disclosure by inference;
- Potential exploitation of access control to create possibilities for disclosure by unusual or covert channels (e.g. variations in the timings of actions).

See above in System Security Policy for generic access control approaches.

3.6.4.10 Cryptography

Basic cryptographic concepts include [Anderson 2001 p. 5] [Bishop 2003 Chapter 9-11]:

- Symmetric key encryption: same key used to encrypt and decrypt;
- Public key encryption: publicly known key used to encrypt and secret key used to decrypt;
- Key management: methods used to originate, distribute, and revoke keys; and identify their possessors;
- Hashing: applying a difficult to reverse function;
- Non-repudiation: impossible to deny performing action, e.g. message encoding contains undeniable evidence identifying who sent it;
- Secret sharing: method of splitting secret among parties[31].

Cryptographic techniques can help preserve confidentiality and integrity as well as ensuring authenticated entities and non-repudiation. These capabilities are also provided through access control and audit logging. Thus, one can sometimes substitute encryption for access control (e.g. during network transfer of data where access control of eavesdropping may be impossible) or serve as a second line of defense.

At least three types of cryptographic expertise are essential. These are:

1. Producing software to encrypt, decrypt, sign, etc.,
2. Cryptographic protocols,
3. Cryptographic issues in system design.

All of these are areas of specialization. Experience shows the first two contain excessive risks if performed by non-specialists.[32] As a result, the domain knowledge for producing cryptographic software was deemed out the scope of this guide.

Experience shows the third may also have dangers if done by non-experts. The security problems of early versions of the IEEE 802.11 wireless protocol are an example. Developers of secure software, therefore, necessarily need significant knowledge of the third. In situations new to the developer, however, expert help can reduce risks as can review by experts.

[31] "Secret sharing" is actually secret splitting in such a way that one must know *t* out of *n* parts to know the secret and, if one knows fewer parts, one knows nothing about the secret.

[32] While presumably produced by specialists and ready for certification, the US Federal government certifiers of cryptographic software report that substantial fractions of submittals have serious problems and 90% have some problem. (CCUF II 2005)

3.6.5 Secure Software Development Activities

Three things aid in reliably producing secure software: [Redwine 2004, p. 3]

1. Outstanding software engineering (including development, testing, and sustainment) performed by skilled people using sound techniques and tools;
2. A sound mastery of the relevant security expertise, practices, and technology;
3. The expert management required to ensure the resources, organization, motivation, and discipline for success.

These implicitly imply an organizational culture that integrates concerns, activities, and rewards for secure software.

Across these aspects, skilled people may be the most significant. Achieving these skills, however, requires considerable knowledge beyond that already required for simply a good software engineering process.

Determining whether developers have the requisite knowledge and expertise in secure software development, however, is still a challenge. With a few exceptions, there are no professional certifications for secure software engineering or secure programming[33]. By contrast with software security practitioners, the safety community (e.g., commercial aviation) has used certification (or licensure) of key personnel for quite a while in order to ensure the qualification of its system developers, managers, and maintainers.

Similarly, in the information security realm, a number of certifications exist for managers, auditors, and engineers. These include both general information security certifications, and security certifications related to specific products. The former include such certifications as those from the International Information Systems Security Certification Consortium (ISC)2 and the SANS Institute, as well as a number of other certifying bodies[34]. So far, these certifications have concentrated on network and organizational security, though (ISC)2 now offers an Information Systems Security Engineering Professional (ISSEP) concentration for holders of its Certified Information Systems Security Professional (CISSP) certification.

Finally, several software engineering-related certifications exist but include no significant coverage of security issues. Security, however, is beginning to take a more prominent place in specific vendors' certifications, such as the Microsoft Certified Application Developer (MCAD) and Solution Developer (MCSD) certifications; optional examinations specifically focused on security now exist for aspirants to these certifications.

This guide presumes knowledge of software engineering activities where security and safety are not concerns. For secure software development, however, a number of activities are new or significantly modified. Table 7 lists activities used by one or more secure software processes.

[33] Noteworthy exceptions include the EC-Council Certified Secure Programmer (ECSP) and Certified Secure Application Developer (CSAD) certifications offered by the International Council of Electronic Commerce Consultants (E-Council).
[34] The Information Systems Security Association lists a number of professional security certifications on their website, at http://www.issa.org/certifications.html.

Table 7: Secure Software Engineering Activities

- Adequately identify and characterize assets, and possible dangers or threats to them
- Analyze dangers or threats, and defenses or mitigations
- Develop security policy(ies) that meets user, organizational, and societal requirements; and formalize them
- Develop a formal top-level specification
- Show top-level specification agrees with security policy
- Develop a security test plan
- Develop an implementation of the system based on the top-level specification providing assurance case with argument and evidence (preferably including proofs) that
  - Design
    - Agrees with top-level specification and security policy
    - Contains only items called for in top-level specification
    - Lacks vulnerabilities to identified threats
  - Code
    - Correspondences to design and security policy
    - Contains only items called for in design
    - Lacks vulnerabilities to identified threats
- Perform penetration testing and test security functionality
- Provide a covert channel analysis
- Perform ongoing monitoring of security threats, needs, and environment
- Provide assurance case including arguments and evidence showing software system meets security requirements
- Perform changes securely maintaining conformance to – possibly revised – security requirements while continuing to provide complete assurance case
- Deploy securely
  - Initially
  - Updates
- Certify software
- Accredit operational system
- Provide for secure
  - Support services
  - Outsider reporting of vulnerabilities

In practice these vary depending whether a heavyweight development process is used [Hall 2002a] [NSA 2002, Chapter 3], a lightweight one as in [Viega 2005], [Lipner 2005a], [Howard 2002], and [Meier 2003] or one especially for the problems of legacy software [Viega 2005, p. 40] and [Meier 2003, p. lxxxi].[35] See the Process section for details.

Generally, the kinds of activities done in heavyweight processes are a modest superset of lightweight processes but with many activities performed with considerably more rigor. The activities used remedially for legacy software are usually similar to a subset of the lightweight ones. [Viega 2005, p. 40] [Meier 2003, p. lxxxi] Note that legacy software may eventually require more serious rework such as major redesign.

Some activities on the list primarily are performed for systems with high security requirements, e.g. covert channel analysis. This list does not reflect the full impact of security on activities. In practice, concern for security affects performance of virtually every activity including, for example, consideration of security in all technical reviews. These activities and effects on other activities must be included in planning and tracking.

Furthermore, production needs to be performed in an appropriately secure work environment and a well-secured development system with the appropriate controls. Two aspects of producing secure software are the security of production and the security provided by the finished product. The former needs to be at least as good as the intentions for the latter. While this guide addresses the latter part in greater length, the first part – subverting the software during production and deployment – as explained in the Dangers section, software producers may not neglect this. Thus, production and deployment need to be:

- Done in a secure environment operated securely, preferably providing counter-intelligence and forensic support;

[35] The first books enumerating steps to produce software appeared in the early 1960's – if not earlier. Software process has been an active area of work in industry, research, and government ever since – within this has been significant work on processes for high-dependability systems. Today, a plethora of books contain general-purpose practices and processes. These range from lightweight processes placing few requirements on developers to heavyweight ones that provide a high level of guidance, discipline, and support. [Boehm 2003] Generally and not surprisingly, success in producing high-dependability systems aimed at safety or security has been greater with software processes closer to the heavyweight end of the spectrum and performed by highly skilled people.

- Possessed of an adequate set of trustworthy tools that have been received, installed, stored, updated, and executed securely;
- Performed by trustworthy people;
- Placed under appropriate controls against inside and outside mistakes, mishaps, and malicious actions.

In addition, adequate assurance must exist that these are true.

Security-related aspects also need to be assessed and improved, for example by:

- Collecting and analyzing security-related measurements;
- Improving security-related:
  - Processes,
  - Personnel-related activities and skills,
  - Artifact quality.

Merely a strong desire for suitable and correct products and services would be sufficient motivation for many of these activities. However, security needs such as privacy provide even more motivations as well as new requirements for production and products. Because of this, some effective organizations establish processes across the organization that integrate security concerns and reuse them.

3.6.6 Security Functionality

Certain kinds of functionality are usually required in secure systems. Among the most common are:

- Identity management of users,
- Access control mechanisms,
- Cryptographic functionality,
- Logging and auditing,
- Control of data flows (e.g. in and out),
- Security administration,
- Malware detection
- Facilities for secure installation and updating,

These other kinds of security functionality identified in [SCC Part 2 2002] and [Common Criteria v.3.0 2005, Part 2]. Common Criteria also calls for self-protection and non-bypassability of security functionality, and protection for boundaries to domains of trust. [Common Criteria v.3.0 Part 3, page 96-101]

Design of security functionalities needs to consider whether the security functions will operate individually or in combination, and whether there are any potential conflicts between the security functions. Security functions may be centralized or decentralized. If COTS and other reusable software or systems are available that implement required security functionality, the characteristics, and appropriate use of those reusable functions need to be carefully considered. See Section 13, Acquisition, for a discussion of this.

3.6.7 Database Security

Databases are parts of many systems with security requirements. Many of the design issues involve establishing proper security access policies. Secure databases are their own subject. [Bertino 2005] provides an introduction to database security concepts as does [Guimaraes 2004] on teaching database security.

3.6.8 Security Risk Management for Software

Producing software that is (more) secure is easier to if risk management activities and checkpoints are integrated throughout the development lifecycle. As discussed earlier software is in danger all its life, and the assurance case provides a framework for organizing and addressing risks.

A risk is an expectation of loss expressed in terms of the probability that a particular bad event or consequence will occur. Although in security-oriented situations the probability may be unknown or unknowable and calculations must be simply from consequences, this is usually, nevertheless, folded under the rubric of “risk management”. Software in which the damage an exploit might cause is quite substantial – i.e., the result would be financial devastation, loss of personal freedom, damage to health, or loss of life – is sometimes referred to as life-critical or mission-critical software, or “high-consequence software”. Such software may include software in large systems, and restitution or recovery could be prohibitively expensive or practically impossible such as in electronic voting or military command and control.

Security-related risks or consequences exist across the entire lifespan of software. Two ways exist to address these. One is to identify all those items or activities that must be performed properly to ensure meeting security requirements. The second is to attempt to identify all the (significant) things that might go wrong. These parallel the knowledge mentioned above of (1) the “right ways” to perform and (2) pitfalls or weaknesses. As labeled in the discussion of assurance cases, these are positive and negative approaches.

The assurance case is the underpinning for software-related security risk management. It addresses claims or goals and risks or consequences related to them attempting to explicitly, rationally, and validly arrive at conclusions regarding them – values and their uncertainties.

3.6.8.1 Assessment of Operational Risks

In discussing risks, risks during the operation of the software have often received the most attention. Attacks on software during operations can lead to compromise either the data the software processes, the software itself, or elements in the software's execution environment. The risk is often viewed as a threat entity will exploit a particular vulnerability with a (harmful) result.

Threat analysis and vulnerability analysis will be discussed at length in the Requirements and Design section respectively. As overview of one approach to risk management is provided next. This approach takes the negative viewpoint and emphasizes identifying vulnerabilities over positively ensuring preservation of security properties.

The objectives of this operational- and vulnerability-oriented software security risk assessment approach are to:

1. Identify all potential threats to the software, and rank them according to likelihood of exploitation and severity and magnitude of impact. The potential of each identified vulnerability to compromise the software itself, or to be exploited to compromise something else, should be captured in the risk assessment;
2. Identify any residual vulnerabilities in the software and identify the changes to the software requirements, design, implementation, or installation configuration that would eliminate or minimize exposure of those vulnerabilities;
3. Estimate the cost of implementing each identified change compared to the amount of potential loss that would result were the vulnerability to be exploited.

Information considered during the security risk assessment process might include:

- The nature of threats[36] to software,
- How those threats manifest as attacks,
- The nature of the vulnerabilities that they are likely to target or exploit,
- The measures necessary to prevent the threats from targeting/exploiting those vulnerabilities,
- The nature of the computing environment in which the conflict takes place,
- The potential consequences both technical (e.g. disclosure) and real-world (e.g. loss of money or life)

The results of the risk assessment guide the risk management process, i.e., the process of identifying, controlling, and eliminating or minimizing (i.e., "mitigating") the uncertain events that may affect the security. Eliminate the vulnerabilities identified in the risk assessment or minimize their exploitability or consequences.

Risk assessments should be performed iteratively throughout the software's lifecycle to maintain an accurate understanding of current risk. Additional security requirements discovered during the design, implementation, testing, and operation phases should be incorporated back into the requirements specification. Security vulnerabilities found during testing should be analyzed to determine whether they originated with the system's requirements, design, implementation, or operational configuration. Similar vulnerabilities should be looked for throughout the software system (and elsewhere), Their root causes should be corrected to remove or mitigate the risk associated with each kind of vulnerability. Risk assessment can also help prioritize and focus resources.

Several software and system security risk assessment methods exist with some having supporting toolsets, Methods include

- Microsoft's threat modeling as described in [Swiderski 2004]
- Microsoft's ACE Threat Analysis and Modeling
- European Union-funded Consultative Objective Risk Analysis System (CORAS) and Research Council of Norway-funded Model-Driven Development and Analysis of Secure Information Systems (SECURIS)
- PTA Technologies' Calculative Threat Modeling Methodology (CTMM),
- Trike
- NASA Reducing Software Risk program's Software Security Assessment Instrument (SSAI)
- CMU SEI's OCTAVE and OCTAVE-Secure (OCTAVE-S) [OCTAVE 2001]
- Siemens' and Insight Consulting's Central Computer and Telecommunications Agency (CCTA) Risk Analysis and Management Method (CRAMM).

While not primarily intended for software, NIST's Draft SP 800-26 Revision 1, "Guide for Information Security Program Assessments and System Reporting Form" (August 2005) and SP 800-30 "Risk Management Guide for Information Technology Systems" (July 2002), supported by the NIST Automated Security Self-Evaluation Tool (ASSET), may also be useful for performing software security risk assessments.

Some risks can be avoided or eliminated, for example by changing the software's design, components, or configuration, or the configuration of its environment. However, unacceptable risks are likely to remain become requirements that must be adequately handled by the software system.

[36] The malicious developer who plants a Trojan horse back door and the hacker who exploits a buffer overflow vulnerability in executing software are two examples of threats to software.

A combination of risk assessment methods can be applied to software throughout the development lifecycle. After an initial risk assessment, deeper targeted assessments can be used to determine which components of the software contribute to the existence of each risk, and which contribute to risk avoidance. Forward analysis can identify the potentially dangerous consequences of a successful attack on the software, while backward analysis can determine whether a hypothesized attack is credible. Applied iteratively through the development lifecycle phases, these methods can help refine the understanding of risk with increasing degrees of detail and granularity.

Specifically, the following aspects of the software should be examined and compared during its risk assessment:

- Mission or business purpose, as captured in its needs statement;
- Objectives, as captured in its requirements specification;
- Structure and interfaces, as depicted in its architecture and design specifications;
- Behaviors, as revealed through analysis, history, or its security testing or evaluation.

After the initial risk analysis is performed, subsequent software lifecycle activities include the objective of minimizing and managing those risks. This includes iteratively re-verifying that the risks have been correctly understood and their required eliminations or mitigations have been adequately addressed. The outcome of these re-verifications will refine the security specifications with specific security properties and mechanisms that must be incorporated into the design, and implemented to mitigate acceptably these security risks.

Even after it has gone into production, periodic risk analyses can ensure that it continues to operate within acceptable risk levels, or to determine whether changes need to be made to the requirement specification, design, or implementation to mitigate or remove risks that may have developed over time or to address new threats.

3.6.9 Domain Knowledge

This guide cannot, of course, enumerate the knowledge needed in all possible fields in which secure software is needed. Nevertheless, adequate application domain knowledge is required of developers, sustainers, and acquirers for proper evaluation of risks and production of a suitable secure system.

3.6.10 Product, Vendor, or Technology Specific Knowledge

Although detailed product or vendor-specific knowledge is not emphasized in this guide, computer and software security are subjects where often “the devil is in the details.” Thus, projects must often have persons with detailed product, vendor, or technology specific knowledge if adequately secure software is to result from their efforts.

3.7 Security Properties Elaborated

Beyond what has been covered so far, this section covers additional conceptual material that should be part of the knowledge of everyone involved or interested in software security. Progressively more in-depth treatments of security properties are available in [Landwehr 2001], [Redwine2005a], and [Bishop 2003]. [Avizienis 2004] contains information on characterization and categorization.

3.7.1 Confidentiality

Computing-related confidentiality topics include access control, encryption, hiding, intellectual property rights protection, traffic analysis, covert channels, inference, and anonymity. The last four are discussed here.

3.7.1.1 Traffic Analysis

The levels, sources, and destinations of communications traffic can sometimes be revealing even if the content is encrypted. For example, traffic increases in organizations tend to foreshadow major events. The main issues in traffic analysis are ease of detection and analyzability. Factors include concealment of origin and destination of communications and the leveling or randomization of traffic volumes and message sizes. [Pfleeger and Pfleeger 2003, p. 410, 453]

3.7.1.2 Covert Channels

Covert channels are "abnormal" means of communication using such means as timing of overt messages, locations in messages not normally used (e.g. unused bits in packet headers), or (un)availability of resources to convey messages. These may be ignored in low or moderate security situations. While covert channels based on resources can potentially be eliminated, the objectives in high-security systems are usually to identify and minimize covert channels of all kinds. Covert communication channels are measured by the bit rate that they can carry. See [Bishop, p. 462-469], [NCSC 1993], and [NRL Handbook 1995, Chapter 8].

3.7.1.3 Data Aggregation Inference

Potential can exist to violate confidentiality or privacy by aggregating data whose individual disclosure would not result in harm. Identity theft is often facilitated by the attacker aggregating data.

3.7.1.4 Inference

Confidential data may be inferable from other data that is available. One example is inferring individual data by comparing data for different groups – an individual's grade in a course can be calculated from the average grade in the course and the average grade of everyone but the individual.

3.7.1.5 Anonymity

Anonymity can involve concealing one's identity, activities, attributes, relationships, and possibly existence. Issues include concealing the identity associated with particular data and who is communicating with whom including determining that the same (but unidentified) entity is involved in two communications – linkage. Desired or required privacy[37] is one motivation for anonymity. [Cannon 2005]

3.7.1.6 Formal Security Models for Confidentiality

A formal security model is a mathematically precise statement of a security policy. Such a model must define a secure state, an initial state, and how the model represents changes in state. The model must be shown to be secure by proving the initial state is secure and all possible subsequent states remain secure. David Bell and Leonard LaPadula of the MITRE Corporation defined the first formal model of confidentiality[38], which stated that if multiple hierarchical levels of confidentiality exist, then one cannot write higher confidentiality data into lower confidential areas and one cannot from a lower confidentiality area read something at a higher level. See [Bishop 2003, Chapter 5] for an extended exposition also including definitions of "basic" and "simple" security.

A more modern (1980's) model is non-interference. The two concepts are that no one at a lower level of confidentiality should see behavior that (1) results in any way from any behavior at a higher level – non-interference [Bishop 2003, p. 448-50] – or alternately (2) from which any information can be derived about behavior at a higher level – probabilistic non-interference [Gray 1990].

[37] Including protection from cyberstalking
[38] David Elliott Bell and Leonard J. LaPadula, "Secure computer systems: mathematical foundations". MITRE Corporation, 1973 - and - "Secure computer systems: unified exposition and MULTICS interpretation". MITRE Corporation, 1976.

3.7.2 Integrity

To maintain system integrity one needs to keep the system in legitimate states or conditions. "Legitimate" must be specified – an integrity security policy could be conditional. For example, it might be allowable for the system to enter otherwise illegitimate states during a transaction, as long as it returns to a legitimate state at the end of the transaction. Early on Biba establish a fundamental integrity property [Biba 1977] and [Clark and Wilson 1987] provides a discussion of commercially relevant integrity.

Two key sub-problems within integrity are:

- Has something changed?
- Were all of the implemented changes authorized?

Checking that data is unchanged can only have meaning in terms of the question, "Since when?" In practice, this usually means that one must query, "Since in whose possession?" (This possession may or may not be at a specified time.)

Kinds of items where proper privileges and authorization can be of concern include:

- Creating,
- Viewing,
- Changing,
- Executing,
- Communicating,
- Sharing
- Encrypting/decrypting
- Deleting/destroying.

In discussing integrity-related change authorizations, changes commonly concern:

- Credentials (evidence of identity and possibly other attributes),
- Privileges,
- Data,
- Software (possibly considered data),
- The point(s) or paths of execution,
- Time (e.g. resetting the system clock).

Sequence and structure can also be the concern of "integrity" properties. For example, transactional integrity ensures that all parts of a transaction succeed, or none do—it is atomic. Relational integrity (in relational databases) enforces that master-detail relationships are correctly maintained (e.g., if you delete a purchase order, you delete related "detail" records such as purchase order lines enumerating items and quantities ordered.). As mentioned, in 1977, K.J. Biba of the MITRE Corporation defined a mandatory integrity policy model that provided a corollary to the Bell-LaPadula mandatory security model.[39]

3.7.3 Availability

Along with reliability, engineering for availability has a long history in computing. Many traditional approaches and means of prediction exist, but all presume lack of maliciousness. (This is no longer so common in the related area of disaster recovery.) As with all security properties, achieving a specified level of availability is a more difficult problem because one must consider maliciousness. Some of the old approaches and almost all the means of calculation no longer work.

[39] K. J. Biba. "Integrity Considerations for Secure Computer Systems" (in MITRE Technical Report TR-3153). The MITRE Corporation, April 1977.

Denial of service attacks from outside – particularly distributed ones originating from many computers simultaneously – can be difficult to successfully overcome. Non-distributed attacks that attempt to take over, exhaust, or destroy resources (e.g. exhaust primary storage) also are a threat. Interestingly, any mechanism designed to deny illegitimate access can tempt attackers to discover a way to use it to deny legitimate access (e.g. locking accounts after a certain number of incorrect passwords tries would allow a malicious person to lock one out of one's account by multiple tries to log as one with random passwords). Speed of repair or recovery can affect availability.

From a security viewpoint, systems need not only to remain available but preserve their other required security properties, e.g. confidentiality, whether available are not.

3.7.4 Accountability

For entities that interact with the system to be held accountable for their actions, those entities must be identified. "Each access to information must be mediated based on who is accessing the information and what classes of information they are authorized to deal with. This identification and authorization information must be securely maintained by the computer system and be associated with every active element that performs some security-relevant action in the system."[40]

Audit information enables actions affecting security to be traced to the responsible party. The system should be able to record the occurrences of security-relevant events in an audit log or other protected event log. The ability to select the audit events to be recorded is necessary to minimize the expense of auditing and to allow efficient analysis.

Audit data must be protected from modification and unauthorized destruction and, in some environments, their confidentiality must be protected. Because they permit detection and after-the-fact forensic investigations of security violations[41], audit logs can become the targets of attacks that attempt to modify or delete records that could indicate an attacker's or malicious insider's actions. In systems that process sensitive data, the audit logs may contain portions of that data, and thus would need to be protected as appropriate for the sensitivity level of that data. In addition, the design of intrusion detection and auditing mechanisms must avoid allowing the exhaustion of log storage space to become a form of attack.

3.7.4.1 Non-Repudiation

Non-repudiation provides proof that any entity that uses a system or acts upon data cannot later deny those actions. Non-repudiation forces users to assume responsibility for their actions so that they cannot disclaim those actions "after the fact" nor deny any event related to themselves—for example, they cannot deny (or repudiate) having been the sender, authorizer, or recipient of a message. Several means of achieving non-repudiation involve cryptographic signatures (more frequently called digital signatures).

ISO/IEC 13888 Information technology – Security techniques – Non-repudiation addresses both symmetric and asymmetric techniques. In symmetric non-repudiation, both the sender and recipient of information are provided with proofs: the sender receives proof that the information was received by the recipient; the recipient receives proof of the identity of the sender. In asymmetric non-repudiation, proof is provided to only one of the parties in a two-party transaction regarding an action of the other party (e.g., sender receives proof of delivery, or recipient receives proof of sender identity, but not both).

[40] Source: DOD 5200.28-STD, Department of Defense Trusted Computer Evaluation Criteria, December 1985.
[41] Other forensic support includes support for identifying suspects and investigating insiders and outsiders. For insiders where the identity of the user may be known, automated recognition of use in an unusual fashion could help support identification of suspects.

3.8 Conclusion

Table 8 provides a high-level list of attributes to limit or in most cases reduce if a software system's description or instance exists. It does not cover all the topics in this section, but does list many of the security-related concerns that decision makers need to address while developing, sustaining, acquiring, and operating software systems.

Table 8: Limits to Aid Software System Security

Limit security-related costs	
	Limit adverse effect on system benefits
	Limit security-related developmental and operational expenses
	Limit security-related (adverse) consequences
Limit violations	
	Limit origination or continuing existence of opportunities or possible ways for performing violations throughout system's lifecycle/lifespan
	Limit undetected violations
	Limit lack of accountability
	Limit violations unable to respond to acceptably or learn from
Limit violators or attackers	
	Limit set of potential violators
	Limit violators' ease of violation
	Limit motivation for violations
Limit security-related uncertainties of stakeholders with interests in adequate/better system security	
	Limit security-related unknowns
	Limit security-related assumptions
	Limit unpredictability of system behavior
	Limit consequences or risks not addressed in assurance case
	Limit consequences or risks related to uncertainty

Source: Sam Redwine

The table includes items related to security-related costs, security violations, violators, and uncertainties. Some entries such as, "Limit origination or continuing existence of opportunities or possible ways for performing violations throughout system's lifecycle/lifespan," cover a broad swath across software security others are narrower but still significant concerns. Since the Table is about a software system, it does not include a limit/reduce entry for the important overall need, "Limit/reduce the existence or usage of systems with inadequate security."

3.9 Further Reading

3.9.1 General

[Abran 2004] Abran, Alain, James W. Moore (Executive editors); Pierre Bourque, Robert Dupuis, Leonard Tripp (Editors). *Guide to the Software Engineering Body of Knowledge*, 2004 Version. IEEE Computer Society, Feb. 16, 2004. Available at http://www.swebok.org

[Anderson 2001] Anderson, Ross J., Security Engineering: A Guide to Building Dependable Distributed Systems. John Wiley and Sons, 2001.

[Bernstein 2005] Bernstein, Lawrence and C. M. Yuhas. *Trustworthy Systems through Quantitative Software Engineering.* Wiley-IEEE Computer Society Press, 2005. About reliability not security.

[Bishop 2006] Bishop, Matt, and Sophie Engle. "The Software Assurance CBK and University Curricula." *Proceedings of the 10th Colloquium for Information Systems Security Education*, 2006.

[NASA Guidebook] National Aeronautics and Space Administration (NASA) *Software Assurance Guidebook* (NASA-GB-A201). Available at http://satc.gsfc.nasa.gov/assure/agb.txt.

[NIST Special Pub 800-27 Rev A 2004] Stoneburner, Gary, Hayden, Clark and Feringa, Alexis. *Engineering Principles for Information Technology Security (A Baseline for Achieving Security)*, Revision A, NIST Special Publication 800-27 Rev A, June 2004.

[NRC 2001] National Research Council (NRC) Computer Science and Telecommunications Board (CSTB). *Cybersecurity Today and Tomorrow: Pay Now or Pay Later.* National Academies Press, 2002. Available at http://darwin.nap.edu/books/0309083125/html.

[Riguidel 2004] Riguidel, Michel, Gwendal Le Grand, Cyril Chiollaz, Sved Naqvi, Mikael Formanek, "D1.2 Assessment of Threats and Vulnerabilities in Networks", Version 1.0. European Union Security Expert Initiative (SEINIT), 31 August 2004. Available at http://www.seinit.org/documents/Deliverables/SEINIT_D1.2_PU.pdf

[SDI 1992] Department of Defense Strategic Defense Initiative Organization. *Trusted Software Development Methodology*, SDI-S-SD-91-000007, vol. 1, 17 June 1992.

3.9.2 System Engineering

[Alexander 2001] Alexander, Ian. *Systems Engineering Isn't Just Software*. 2001. Available at http://easyweb.easynet.co.uk/~iany/consultancy/systems_engineering/se_isnt_just_sw.htm.

[Bahill 1998] Bahill, A.T. and B. Gissing, "Re-evaluating Systems Engineering Concepts Using Systems Thinking". IEEE Transaction on Systems, Man and Cybernetics, Part C: Applications and Reviews, Vol. 28 No. 4 pp. 516-527, November 1998.

[INCOSE] International Council on Systems Engineering (INCOSE). *Guide to Systems Engineering Body of Knowledge (G2SEBoK)*. Available at http://g2sebok.incose.org/.

[Rechtin 2000] Rechtin, E. *Systems Architecting of Organizations: Why Eagles Can't Swim*. Boca Raton, FL: CRC Press, 2000.

3.9.3 Information Security

[CNSSI 4009] Committee on National Security Systems (CNSS) Instruction 4009: *National Information Assurance (IA) Glossary*. Revised May 2003. Available at http://www.cnss.gov/Assets/pdf/cnssi_4009.pdf.

[ISO/IEC 12207] International Standards Organization/International Electrotechical Commission Standard 12207:1995, Software Life Cycle Processes, plus Amendement 1:2002 and Amendment 2:2004. Available at http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=21208

[ISO/IEC 15026] ISO/IEC Standard 15026:1998, System and Software Integrity Levels. Available at http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=26236

[ISO/IEC 15288] ISO/IEC Standard 15288:2002, Systems Engineering - System Life Cycle Processes. Available at http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=27166

[NIST FIPS 200] NIST: Federal Information Processing Standards Publication (FIPS PUB) 200: *Minimum Security Requirements for Federal Information and Information Systems.* March 2006. Available at http://csrc.nist.gov/publications/fips/fips200/FIPS-200-final-march.pdf.

[NIST SP 800-27] NIST Special Publication 800-27: Engineering Principles for Information Technology Security (A Baseline for Achieving Security). Revision A, June 2004. Available at http://csrc.nist.gov/publications/nistpubs/800-27A/SP800-27-RevA.pdf.

[NIST Special Pub 800-33 2001] NIST SP 800-33, *Underlying Technical Models for Information Technology Security*, December 2001

[NRC 1999] Committee on Information Systems Trustworthiness, *Trust in Cyberspace*, Computer Science and Telecommunications Board, National Research Council, 1999.

3.9.4 Security Functionality

[DoDI 8500.2] Department of Defense Instruction 8500.2 (6 February 2003). *Information Assurance (IA) Implementation*. Washington, DC: US Department of Defense, 2003. Available at http://www.dtic.mil/whs/directives/corres/pdf2/i85002p.pdf.

[NIST FIPS 200] NIST: Federal Information Processing Standards Publication (FIPS PUB) 200: *Minimum Security Requirements for Federal Information and Information Systems.* March 2006. Available at http://csrc.nist.gov/publications/fips/fips200/FIPS-200-final-march.pdf.

[NIST Special Pub 800-53] Ross, Ron et al. *Recommended Security Controls for Federal Information Systems*, NIST Special Publication 800-53, Feb. 2005. Available at http://csrc.nist.gov/publications/nistpubs/800-53/SP800-53.pdf. and its Revision 1 Draft available at Available at http://www-08.nist.gov/publications/drafts/800-53-rev1-ipd-clean.pdf.

4 Ethics, Law, and Governance

4.1 Scope

This section addresses the ethics, laws, regulations, and standards peculiar to developing secure software and is intended for all readers. Given the relatively recent interest in the security of software, much of the knowledge in this section is in a formative stage. Often it is derived from work from a larger domain. For example, no code of ethics has been proposed specifically for secure software developers, but there are ethical codes for information security professionals, computing professionals, and software engineers [Bynum and Rogerson 2004].

Legal and regulatory knowledge in the domain of secure software development tends to focus on issues such as privacy, intellectual property, and liability. Although many laws addressing computer crime have been enacted within the last two decades, these laws do not involve the development of secure software and are therefore not included in this section.

Several standards, including international standards, addressing security are important in the development of secure software. Standards that address the evaluation of secure systems such as the Common Criteria are included in a separate section.

4.2 Ethics

In defining the ethical BOK that applies to secure software development, the BOK adopts the view of ethics as a "… branch of professional ethics, which is concerned primarily with standards of practice and codes of conduct of computing professionals…" [Bynum 2001]. Furthermore, a computer professional is "… anyone involved in the design and development of computer artifacts..." [Gotterbarn 1991].

Codes of conduct and statements of ethics embody much of the ethical knowledge in developing secure software. Example codes include the Institute of Electrical and Electronics Engineers (IEEE) Code of Ethics, the Software Engineering Code of Ethics and Professional Practice,[1] Association of Computing Machinery (ACM) Code of Ethics and Professional Conduct, the British Computer Society Code of Conduct [Bynum and Rogerson 2004], and the International Information Systems Security Certification Consortium (ISC)2 Code of Ethics [ISC 2005].

The ethics and codes of conduct share common elements: (1) acting in the public interest, (2) applying duty to clients insofar as it is consistent with the public interest, (3) ensuring honesty and integrity in the practice of the profession, and (4) maintaining competence in the profession.

Controversy remains, however, over the ethics of security vulnerability disclosure (see section 12, Secure Software Sustainment).

4.3 Law

The primary legal issues surrounding the development of secure code include intellectual property and their associated copyright, patent, trade secrets, and trademarks. Principles of privacy, private and corporate liability, including the so-called "downstream" liability, are beginning to be codified in law, although these principles are often spread through various legislative acts [Bosworth and Kabay 2002], [Smedinghoff 2003], [Smedinghoff 2004]. The most explicit statement of corporate liability in the domain of secure information

[1] Available at http://onlineethics.org/codes/softeng.html

architectures includes the requirement for “internal control structure and procedures for financial reporting” in the Sarbanes-Oxley Act of 2002. Important liability principles in this area include the prudent man rule, due care, and due diligence. A summary of laws and executive orders pertaining to computer security is contained in [Moteff 2004].

4.4 Governance: Regulatory Policy and Guidance

Regulatory policies and guidance are created by government agencies to set performance levels and compel certain behaviors in corporations or industries [Harris 2003]. Privacy regulations have the most far-reaching impact on the development of secure software. Privacy protects the personal information of individuals from misuse by governments or corporations. Privacy principles include the lawful use of personal information, the accuracy of that information, and the disclosure, consent, and secure transmission of that information.

4.4.1 Policy

A policy is a statement by an authoritative body intended to determine present and future decisions, actions, or behaviors. A policy is usually general in nature, yet it requires specific standards, guidelines, and procedures to be implemented for particular situations. There are three types of policies:: regulatory, advisory, and informative [Harris 2003]. Regulatory policies are mandatory and carry the force of law; advisory policies encourage adherence to certain standards and are enforceable; informative policies are intended to make known a particular policy though they are not enforceable.

For example, *Privacy Online: OECD Guidance on Policy and Practice* is non-binding to its member nation states. Focused on the implementation of the *OECD Privacy Guidelines* online, the policy and practical guidance offered in this publication fulfill the 1998 OECD Ministerial Declaration on the Protection of Privacy on Global Networks but are nevertheless non-binding. US Federal Trade Commission’s Fair Information Practices [FTC 2000] are also relevant. For a wide-ranging but more technical slant on privacy see [Cannon 2005].

Some policies are created by governments or industry groups and affect the organizations and the software systems they employ.

4.4.2 Laws Directly Affecting Policy

In part, these security policies are often formulated in response to the requirements of law. In the US these include the following federal laws. [Moteff 2004]

- Health Insurance Portability and Accountability Act (HIPAA)
- The Sarbanes-Oxley Act of 2002 requires companies to implement extensive corporate governance policies, procedures, and tools to prevent, respond, and report fraudulent activity within the company. Effective self-policing requires companies to have the ability to acquire, search, and preserve electronic data relating to fraudulent activity within the organization.
- The Gramm-Leach-Bliley Act requires financial institutions to safeguard customer information as well as detect, prevent, and respond to information security incidents.

- California SB 1386 (California Civil Code § 1798.82) addresses how a company responds to a breach, and has important features based on cooperation with law enforcement and prompt notification to affected customers.
- The Federal Information Security Management Act (FISMA) mandates that federal agencies must maintain an incident response capability.

In the US, other state and federal laws and regulations may apply, and the number of state laws is increasing, some modeled on California SB 1386.[2]

The European Union (EU) Data Protection Directive[3] and implementing national laws may also be relevant. Indeed, for many commercial products, "internationalization" of requirements includes concern for a substantial number of countries' legal restrictions. In 2000, the US reached an agreement with the EU that organizations following a set of Safe Harbor[4] practices could import data from the EU. [Safe Harbor 2000]

These legal issues, in most cases, deserve attention from legal counsel.[5]

Typical Organizational Security Policy Areas

- Access control
- Awareness and training
- Audit and accountability
- Certification, accreditation, and security assessments
- Configuration management
- Contingency planning
- Identification and authentication
- Incident response
- Maintenance
- Media protection
- Physical and environmental protection
- Planning
- Personnel security
- Privacy
- Risk assessment
- Systems and services acquisition
- System and communications protection
- System and information integrity

NIST FIPS-200 Draft 2005

4.4.3 Standards and Guidance

Standards[6] are usually more specific statements of behavior intended to implement a policy or policies. Guidance, on the other hand, is suggestions on how one might implement a standard. International standards on information security include the Information Security Management standards ISO/IEC 17799. Other standards and guidance on computer security include the NIST 800 series special publications.

4.4.4 Organizational Security Policies

Organizations also establish security policies and possibly internal standards. These can vary from brief, limited ones to substantial ones covering much of the organization's activities. Organizational security policies normally form part of the requirements constraints that software systems must conform to by the software systems requirements and security policy.

The box on the right lists a number of areas that organizational policies may cover. For a given software system, some policies may be straightforwardly applied; others may imply subtle requirements. Thus, all must be carefully analyzed for applicability.

[2] A listing of the laws related to cyber security that was passed in 2005 in all the states is available at: http://www.cscic.state.ny.us/msisac/news/
[3] See http://www.cdt.org/privacy/eudirective/EU_Directive_.html
[4] See http://www.export.gov/safeharbor/
[5] A number of relevant links are given at http://www.cdt.org/privacy/guide/basic/fips.html
[6] Standard ... (7) An agreement among any number of organizations that defines certain characteristics, specification, or parameters related to a particular aspect of computer technology. [IEEE Std 100-1996, The IEEE Standard Dictionary of Electrical and Electronic Terms, Sixth Edition]

4.5 Further Readings

[Bishop 2003] Bishop, Matt., *Computer Security: Art and Practice*, Addison-Wesley, 2003.

[Cusumano 2005] Cusumano, Michael A., “Who is Liable for Bugs and Security Flaws in Software? Attempting to determine fault and responsibility based on available evidence”, *Communications of the ACM*, March 2004/Vol. 47, No. 3 pp. 25-27.

[Sommerville 2006] Sommerville, I., *Software Engineering*, 8th ed., Pearson Education, 2006.

Part 3: Application to Secure Software

5 Secure Software Requirements

5.1 Scope

This section addresses the knowledge needed in the activities to establish the needs and specifications for secure software and, while mentioning some of that knowledge for context, presumes the knowledge of how to do requirements other than for security and safety.

Requirements are in large part about the system's environment and the system's interactions with it to achieve the required effects – and avoid others. Secure software requirements analysts and designers pay special attention to threats in a software system's environment and to protecting system assets.

Security requirements cover the properties included in the definition of security and thus contain elements related to availability; integrity including authentication; confidentiality including anonymity, privacy, and intellectual property rights; and accountability including audit logs and non-repudiation. These issues can be accompanied by a number of the associated ones mentioned in sections 3 on fundamentals and 4 on Ethics, Law, and Governance.

Requirements identify not only what must (or should) be true about the software system for purposes of directing the producers of the system, but must also identify what should be shown by the assurance case – evidence tied together by an assurance argument.

Validation of software requirements and verification of consistency within and among its parts and representations are also of concern and are addressed in part here and in part in section 8, Verification, Validation, and Evaluation. Traceability, formality, and rigor during requirements will facilitate correctness and assurance throughout development.

The initial stimulus for a software system effort may come from a variety of sources and in a number of different forms, such as experiencing a difficulty, observing what is happening in a different industry or at a competitor, investigating market research, or exploring an idea for exploiting a technology. Formulating the software system concept that combines needs and software solutions may happen early or late in the process.

Making no presumption on the order in which these activities and decisions take place, this section presumes only some boundary has been placed on the scope of concern, and first addresses security-related needs and constraints leading to the establishment of a software system security policy. The section ends with a discussion of the need to show the specification of the external behavior of the software system meets the constraints of this policy. While aspects of verification and validation activities are mentioned several times, the main coverage of these topics is found in section 8, Verification, Validation, and Evaluation. Thus, this section addresses the knowledge needed in establishing security-related stakeholder needs and constraints, their analyses, the specification of a system's security policy and its conforming external behavior, and several related special areas.

5.2 Requirements for a Solution

Generally, at the beginning of requirements activities, no one individual or group knows the full set of needs, constraints, and relevant measures of merit. Moreover, different individuals and groups often have different viewpoints and conflicting thoughts. To address this issue, a process involving and/or considering many stakeholders normally evolves to an understanding of current and potential security and other needs in light of possible security solution approaches.

To the extent their systems share the same environment, organizations have found it effective to create "standard" inputs to security requirements, such as threat identification and security policies harmonizing external and organizational constraints and the organizations' standard procedures. These organizational standards can then be tailored as needed to particular software systems.

While a number of specific references are made throughout the section, several more general references are worthwhile. [Berg 2006, Chapter 2] provides general guidance, and [CERIAS TR 2000-01] and [Meier 2003] provide specific web- or ecommerce-related guidance. [ISO-15448], [Moffett and Nuseibeh 2003] combined somewhat redundantly with [Moffett 2004]. [NIST SP 800-64, sections 2.1, 2.2, and 2.3.1-2], [NSA 2002], and [S4EC] provide more general guidance related to requirements. From the Software Engineering Institute, [Chen 2004] describes an application of the SQUARE method, and [OCTAVE 2001] provides an eighteen-volume set of guidance for a method to "Operationally Critical Threat, Asset, and Vulnerability Evaluation^{SM}." [NSA 2004] and [Fitzgerald 2002] provide suggestive examples and discussions of security requirements.

5.2.1 Traceability

Assurance is impossible without a way to trace among requirements' artifacts and to trace later artifacts to them. Needs, features, system security policies, and other requirements artifacts therefore need to be stated in unique, single units to ease traceability to design, code, test, and other products. Software cannot be seen in isolation, so traceability must also be upwards and outwards, to any originating system requirements or other source documents' recording needs.

5.2.2 Identify Stakeholder Security-related Needs

Interactions with stakeholders result in a list of needs and preferences for security including privacy and intellectual property protection. These interactions may also identify standards and guidelines that relate to the application from which requirements can be derived. (See section 4, Ethics, Law, and Governance.) These may directly relate to the particular assets involved. Organizations may have existing enterprise architectures, including security elements.

Of course, initial stakeholder requests may or may not remain to become part of the agreed-upon set of needs and may or may not later be met by the specification..

5.2.3 Asset Protection Needs

When addressing asset protection needs, developers need to know goals, means, and processes for

- Identifying information and capabilities that might be within scope – whether the solution eventually puts them in digital form or not
- Estimating potential for damage from private and public disclosure, contamination, reduced access/capability, lack of accountability, or loss of asset

The latter may, in part, be addressed through asking questions regarding potential consequences of security violations related to assets such as those in [BSI 100-2 2005, Section 4.2] titled Defining Protection Requirements. Some discussions in [NIST SP 800-60] may lead one to relevant factors and damages. The results of estimating potential damage from violations may be reflected by categorizing data or systems by sensitivity. [Radack 2005] [CJCSM 6510.01 2004, Enclosure C Appendix E] [DoDI S-3600.2]

Assets may need to be protected in many situations across the life of the asset and any copies of it. At various times, information assets may be located in a number of places, each of which could imply security requirements including ones for:

- File system security-relevant facilities
- Database security
- Hardware protection
- Communications security

- Redundancy, backup, archives, and recovery records, including those for business continuity
- Media used for mobility, e.g., memory sticks and CDs
- Location and movement during computation including
  - Primary memory
  - Transit
  - Registers
  - Caching
  - Virtual memory paging
- Startup, including attempting to start (and operate) in a potentially hostile or damaged environment and the need for recovery
- Shutdown, including waiting for garbage collection and physical memory to be given to another process
- Logs
- Mobile computers and devices
- Disposed or reassigned equipment and media
- Lost or stolen computers, media, and devices

The assets to be protected usually can be usefully thought of being of four kinds, implying needs for:

- Data protection
- Software protection
  - In operating environment
  - Protection of software-related artifacts and supporting data throughout lifecycle
- Human and organization protection for
  - Users and operators
  - Persons involved in the engineering, supplying, and sustaining of the software, and their organizations
- Security and safety of physical and computing assets in environment may be involved.
  - Operating and user environment
  - Physical facilities, environment, and platforms in which software engineering and sustainment take place

Software protection needs are often driven by needs to

- Avoid loss of intellectual property or its revenues
- Avoid giving an adversary knowledge; for example, insight into a commercial process or a military weapons system

Special asset-related security considerations generating needs that will be covered below include

- Resolving mistakes, abuse, or failure to follow accepted procedures/use rules
- Determining threat entities' capabilities to detect deception and hiding of assets
- Helping ensure cost-effective and timely certification of system security and operational system accreditation
- Aiding administration, operation, and use in a secure fashion

Protecting humans and organizations from harm includes concern for privacy [Cannon 2005], reputation, and freedom from (cyber) stalking, abuse, and criminal acts.

In addition, environments where the software system and its assets reside may have varying levels of security and threats. The latter will be addressed next.

5.2.4 Threat Analysis

One needs to look at software from the viewpoints of the relevant threatening entities or attackers. In combination with the defensive stakeholders' valuation of the assets and the consequences of successful exploits, attackers' valuations, decisions, and capabilities will determine the security protection needs of assets.

The threat-analysis process must involve owners of the assets and other stakeholders who might be affected by the consequences of an incident. Inside attackers come from the user population, so one must consider users from both offensive and defensive perspectives. [Swiderski and Snyder 2004] [Meier 2003, Chapters 2 and 3] [Meier 2005a] [Saitta 2005]

5.2.4.1 Identification of Threat Entities

Identifying, analyzing, and forecasting threat entities or categories takes expertise that may only be available from experts, but a number of resources exist, e.g., [Meier 2005b]. Intentions and resources of threat entities need to be established. This includes their evaluation of the value of possessing, disclosing, contaminating, and denying the assets. Insider and outsider threats must be identified. For better results, validation of the threat analysis should involve stakeholders and expert reviews, including by experts in attack motivations and capabilities. See section 3, Fundamentals, Concepts, and Principles.

5.2.4.2 Threat Entities' Capabilities

Both current and future capabilities of threatening entities are of interest. The rate and directions in which threat entities will improve their capabilities is therefore of interest [Swiderski 2004].

5.2.4.2.1 Skill Available to Threat Entities

Identify or postulate attacker skills, competence, and tools, and the implications of their possession. Defensive measures need to be a match for attack capabilities now and in future. Once an attacker finds a way in, however, this originator may automate the attack and make it available to others, thereby increasing attacker effective skill level.

5.2.4.2.2 Resources

Identify or postulate attacker resources, initial access, and persistence. This identification will help forecast the size and duration of attacks.

5.2.4.2.3 Added capabilities from partial success

Requirements for tolerance and defense-in-depth can be derived by considering the effects of the possible partial successes by adversaries and the added potential this provides them. The ability to specify partial successes – either increasing privileges or control of or causing the failure of portions of the system or its environment – should evolve with the design [Stroud 2004]. Thus, security requirements continue to be created in the course of design activities.

Lessons can be learned from the extensive body of experience with fault tolerance and high-availability systems.[1]

[1] See, for example, http://lcic.org/ha.html

5.2.4.2.4 Misuse or Abuse Cases

Constructing and using misuse cases appropriately can aid in understanding the concrete nature of threats. One needs to remain aware of the limitations resulting from their being partial examples [McDermott 1999] [Sindre 2000] [McDermott 2001] [Hope 2004] [McGraw 2006, Chapter 8].

5.2.4.2.5 Attack Trees

Attack trees have the attacker's goal at the top and branches that identify alternate or combined ways (achieve subgoal A and subgoal B) that allow achievement of the goal or subgoal. These can provide a graphical way to portray potential attack paths that can then be investigated [Swiderski 2004].

5.2.4.2.6 Physical Access and Tampering

If the attacker has physical access to the system, then additional possibilities arise. These threats are explicitly documented either as concerns or non-concerns (assumed not to exist). Common cases of possessing physical access include insider attacks and stolen portable computers. Other approaches include surreptitious entry into work or equipment areas and subversion of the equipment maintenance process.

Physical access by an attacker can raise many difficulties for a defender. Physical access may aid authentication if, for example, the IP address is used as part of authentication or if users have their password written down somewhere near their computer. A hard disk might be removed and studied using forensic techniques, thereby bypassing software access control. Techniques to address this last problem are generally based on cryptology.

Tampering comes in a number of forms. Those concerning the integrity and self-protection of software and assets are covered elsewhere. Tampering with the physical equipment, however, is a source of additional threats. These threats must be addressed or explicitly assumed to be nonexistent in a particular situation [Atallah, Bryant and Sytz 2004]. See section 7, Secure Software Construction, for a discussion on anti-tampering.

5.2.4.3 Threat Entities' Intentions

In large part, security is about the issues raised by real or potential malicious intentions, so the practical importance of intentions is not surprising. In practice, they are critical, particularly for entities that are not currently exercising their capabilities to cause maximum harm. Also in practice, one extends different degrees of responsibilities, privileges, and trust to different entities based, in part, on judgments about their intentions as well as on risk estimates and resource allocation decisions.

Some common advice is, "When in doubt, one should not trust in others' intentions (or competence) at all," and "Where there is capability, the attacker may develop intent."

Motivations drive intentions. The attackers' motivations result from such factors as the value they place on the asset – knowledge, use, or denial – and their willingness to take the risks involved, including discovery and punishment. Motivations may be individual, organizational, or even secondhand, as in mercenaries hired for industrial espionage or organized crime extorting experts to perform attacks for them. See section 3, Fundamental Concepts and Principles.

A template for a threat modeling document is available from Microsoft [Meier 2003, Chapter 3] [Meier 2005c].

5.2.5 Interface and Environment Requirements

Many systems must interface with outside software whose security characteristics are uncertain or run on an operating system or other infrastructure whose security is known to be questionable. The system then may have difficult requirements to be secure in an insecure environment. External systems or domains not under trusted control should be considered potentially hostile entities. Connections to such external systems or domains must analyze and attempt to counter hostile actions originating from these entities. These areas call for careful risk analysis as well as sound technical solutions.

What are the risks to the owners or operators of the system and/or the risks related to the assets needing protection? If the system is a commercial one with effective legal protection based on a given authentication or non-repudiation involved in the commercial interactions, then fiscal risks from these transactions may be minimized with limited costs caused by the system’s insecure environment. Other situations can be significantly more difficult. Requirements need to address the need for both an acceptable (or at worst tolerable) level of risk and the technical feasibility.

Integrity issues always exist for incoming data, and integrity checks are normally a requirement. These checks include ones for acceptability of data type and size, value possible in real world, likeliness or plausibility, internal consistency and with other data, proof of origin and legitimacy of origin, and lack of corruption or interference during transit. Checks for non-repudiation may also be relevant.

Security-inspired requirements on nature and attributes of computing hardware, infrastructure, or other externally available services must be explicitly recorded as requirements or assumptions and must be assured. In some cases, these requirements will be the explicit responsibility of others in the organization, but their accomplishment and/or existence still need to be assured.

The chief security concerns about effects on environment are to:

- Not cause security problems for systems in environment
- Not release any information it should not
- Not issue any information in a risky form, e.g., often risky to send unencrypted output

Care is needed at all boundaries where the level or natures of trust or custody change, including the external boundary of the software system of concern.

5.2.6 Usability Needs

Acceptable usability is important not only to reduce user mistakes but also to ensure efficient and effective usage and acceptance of security features. Security is likely to be systematically bypassed by users if it places a perceived unacceptable impediment between them and accomplishing their tasks.

Process and task analyses and reengineering are the starting points for achieving these objectives. Beyond using sound user-system interaction techniques and design methods throughout, certain general interaction concerns have been particularized to security. [Yee 2004] identifies a small set of general needs and principles, and many more detailed issues are identified in [Cranor 2005] and [Garfinkel 2005]. IEEE Security and Privacy magazine had a September/October 2004 special issue on usability. A series of research-oriented workshops exists in this burgeoning research area that addresses practical issues [SOUPS 2005].

5.2.7 Reliability Needs

In the traditional view, the reliability of a software system depends on the distributions of inputs or the patterns of use. Normally, a seldom executed part of the software could have faults with little or no impact on reliability. Malicious attackers may, however, seek them out for exploitation (making total correctness a security issue).

Although, strictly speaking, reliability is not a security property, preserving the integrity and availability of software and data without reliability results in receiving wrong answers and would be, naturally, somewhat futile.

For these reasons and for economy, reliability's assurance case is sometimes a natural candidate to include in an overlapping assurance case with security sharing some arguments and evidence.

5.2.8 Availability, Tolerance, and Survivability Needs

Developers need to know techniques that can improve tolerance to security violations. Robustness, resilience, and automatic adaptation and recovery are possible parts of security-influenced needs. Separation of duties and "least common mechanism" might be required, as might avoidance of a single point of total vulnerability, analogous to the avoidance of a single point of failure. As an additional example, one might require secret sharing (splitting) across machines for added confidentiality. See also the subsection on Added [Threat] Capabilities from Partial Success and the Design section [Stroud 2004].

Continued availability of service, business continuity, and disaster recovery generate special security-related requirements. Survival of the capability to preserve the required security properties is always necessary regardless of system status.

DoS attacks can make it difficult to continue to serve legitimate users. Requirements can be set for acceptable, tolerable, and unacceptable sizes and natures of DoS attacks or amounts of degradation of service. These requirements must be technically and economically feasible, and limits on need may be set by attacks that would in any case deny service or communication upstream in the paths of requests or downstream for responses.

Finally, maintainability or sustainability ultimately affects availability and tolerance.

5.2.9 Sustainability (Maintainability) Needs

When sustaining software, software with unrepaired vulnerabilities is insecure software. Software that has yet to be changed to meet new security requirements is also insecure. Therefore, repair and evolution related needs, for example, ease of applying "patches," can become security needs where speed in meeting these needs is critical. See section 12, Secure Software Sustainment.

5.2.10 Deception Needs

While it is certainly not desirable to rely exclusively on obfuscation, nevertheless, one may wish to employ deception and hiding. To be effective, one should consider not only intended effects on attackers but also the measures attackers might take to detect, overcome, or exploit one's deceptions – possibly generating additional needs. For concrete examples, see subsection 6.15, Deception and Diversion.

5.2.10.1 Obfuscation and Hiding Needs

While "security through obscurity" should never be relied on exclusively to protect either information or software, in the case of software particularly obfuscation, deception, and hiding can be effective measures to increase the difficulty of reverse engineering by attackers seeking vulnerabilities in binary code or bytecode. In addition, they may decrease the accessibility of information, such as comments in browser-viewable source code, that may be exploited to more effectively target the software. In the case of information, deception and hiding techniques may discourage less skilled and casual hackers who are looking for easy targets.

For more specifics, see the Design section's subsection on Obfuscation and Hiding.

5.2.11 Validatability, Verifiability, and Evaluatability Needs

Because they can ease the diagnosis, repair, and assurance activities, concerns here include:

- Validatability

- Verifiability
- Controllability
- Observability

Particularly, one must address analyzability and testability in addition to the need to provide evidence for the assurance case on conforming to security needs, policy, and correctness of security functionality.

Needs may also be generated by other security-related aspects, including needs to:

- Ease the development and evolution of the assurance case
- Help ensure cost-effective and timely certification of software system security
- Help ensure cost-effective and timely accreditation of operational systems containing the software

If the certification of software systems or accreditations of operational situations are goals, then the needs generated by these goals should be considered and, whenever relevant, included when establishing requirements. See below and in section 8, Secure Software Verification, Validation, and Evaluation.

Testability for security or other concerns may lead to desires for additional observability, but no means should be added that would violate security (e.g., confidentiality) or unduly increase vulnerability. Desirable additional controllability also should be carefully reviewed to ensure security problems would not be produced.

5.2.12 Certification Needs

A number of needs for certification may apply particularly to government systems, but others also have certifications available. This section provides several examples:

5.2.12.1 Common Criteria Requirements

Associated with the Common Criteria are a set of Protection Profiles – standard minimal security requirements – for a number of kinds of software. These can be a required input to establishing security requirements or provide interesting examples.

Historically, the Common Criteria and associated Protection Profiles have identified the security-oriented functionality required of systems and are enumerated in [Common Criteria v.3.0 Part 2]. The Common Criteria now calls for self-protection and non-bypassability of security functionality [Common Criteria v.3.0 Part 3, pp. 96-101]. These two properties change the straightforward functionality requirements of prior versions of the Common Criteria into software system security property requirements.

5.2.12.2 Sensitive Compartmented Information

Protecting the national security information, classified information, and the highly secret Sensitive Compartmented Information (SCI) [DoDI S-3600.2], which is often intelligence secrets, within the US government is covered by [DCID 6/3 2000]. The combination of security safeguards and procedures used for SCI information systems shall ensure compliance with DCID 6/3, NSA/CSS Manual 130-1 [NSA 1990], and [DIAM 50-4 1997]. Also potentially useful is [JDCSISSS 2001, p. i][2]

5.2.12.3 Certification for Banking and Finance

The Financial Services Roundtable's BITS certification program aims to achieve a set of security functionality suitable for banking and financial services and develop a certification process simpler than that of the Common Criteria. See http://www.bitsinfo.org/c_certification.html.

[2] The *JDCSISSS* is a technical supplement to both the NSA/CSS Manual 130-1 and DIAM 50-4 and provides procedural guidance for the protection, use, management, and dissemination of SCI.

Two ISO technical reports exist related to financial services

- *ISO/TR 13569 Banking and related financial services – Information security guidelines*
- *ISO/TR 17944:2002(E) Banking – Security and other financial services – Framework for security in financial systems*

The latter includes definitions, lists of applicable standards and ISO technical reports, and mentions items lacking ISO standards in 2002 when it was published.

5.2.13 System Accreditation and Auditing Needs

Substantive and process-related ISO standards exist as part of an internationally standardized approach to systems accreditation. The US federal government has different system accreditation processes for national security materials and civilian ones.

Internationally, *ISO/IEC 17799:2005 Information technology. Code of Practice for Information Security Management*, combined with BS7799-2:2002 or ISO/IEC 27001, forms a basis for an Information Security Management System (ISMS) certification (sic) of an operational system.

The US DoD most general accreditation process is governed by DoD Instruction 5200.40, *DoD Information Technology Security Certification and Accreditation Process (DITSCAP)* [DoD1997], supplemented by the DoD8510.1-M Application Manual, and applies to all DoD entities.

The National Information Assurance Certification and Accreditation Process (NIACAP) applies to National Security Systems, and, as mentioned above, for the intelligence community protecting SCI within information systems to which DCID 6/3 applies.

On the civilian side of the US government, NIST has produced *NIST Special Publication 800-37, Guide for the Security Certification and Accreditation of Federal Information Systems*, which applies to all civilian US government Executive Branch departments, agencies, and their contractors and consultants.[3]

The first three – ISO/IEC, DITSCAP, and NIACAP – are mandatory for their communities, while the NIST SP provides guidelines for certifying and accrediting information systems supporting the executive agencies of the federal government. See section 3, Fundamental Concepts and Principles.

As part of its Federal Information Security Management Act (FISMA) Implementation Project, NIST issued a series of documents relevant to the specification and verification of security requirements in all federal information systems other than those designated as national security systems (as defined in 44 U.S.C., Section 3542). These documents include NIST Federal Information Processing Standard (FIPS) 200, “Minimum Security Requirements for Federal Information and Information Systems” (March 2006); NIST SP 800-53 Revision 1, “Recommended Security Controls for Federal Information Systems” (Public Draft, March 2006); NIST SP 800-53A, “Guide for Assessing the Security Controls in Federal Information Systems” (Second Public Draft, April 2006); and NIST 800-37, “Guide for the Security Certification and Accreditation of Federal Information Systems” (May 2004).

Many commercial firms’ systems will face audits by their accountants or others. Government organizations may also face regular audits. Any additional needs imposed by these that potentially impact security (positively or negatively) require consideration.

[3] Upon publication of SP 800-37, NIST also rescinded the older Federal Information Processing Standards (FIPS) Publication 102, Guidelines for Computer Security Certification and Accreditation (September 1983), which the new SP is intended to replace. .Note that while use of NIST SP 800-37 is not mandatory, certification of FISMA compliance is required for all non-national security information systems.

As well as anticipating the questions that will be asked during an investigation, forensic-friendly software needs to ensure that sufficient audit data is logged securely. Logs must be retained with their integrity maintained suitably for legal evidence as well as for preserving any required confidentiality and accessibility.

The implications for a software system's requirements deriving from an intention for the software to be certified or operated in an operational system that requires accreditation or auditing need to be fully identified.

5.3 Requirements Analyses

Requirements analyses are closely involved in the requirements discovery process, as they are relevant to the evolving decision process that results in agreed-upon requirements.

5.3.1 Risk Analysis

For security risks, the probabilistic approach embedded in the usual risk analyses cannot be based on technical characteristics of the system. One can, in low and possibly medium threat situations, make some probabilistic estimates based on humans and organizations that are potential threats. In high threat situations, the attacks are essentially assured eventual success if even a few vulnerabilities exist. This makes the chance of a successful attack a certainty for all but the extremely well-designed and rigorously implemented systems. In high threat situations, one must presume that some vulnerability will be found by a skilled, persistent foe. Nevertheless, probabilistic estimates may be possible regarding the time until a successful attack and other aspects.

Analyses of possibilities and risks also address the effect of attacks. The effects are not just first order such as disclosure or corruption of data but what effects this has on organizational mission success[4], reputation, stock price, recovery and repair costs, etc. A risk index (number of occurrences x effect), stakeholder utility judgments, or sensitivity level classifications of assets can be used to identify security critical concerns to influence the specifications and design and to help decide the allocation of resources and control the development process For example, Microsoft's DREAD approach (see text box) can be useful in deciding the priorities for fixes. Microsoft also addresses "VR" – Value to attacker and Reputational risk as well as having an alternative approached called ACE Threat Modeling.

Of course, software is not the only path to success for an attacker. Social engineering and subversion may often be part of the easiest attacking path leading possibly to needs for the software to somehow mitigate these sources of attack as well. Ultimately, security is a systems (or even larger scope) issue, not just a software one.

Microsoft's DREAD Acronym

Calculate risk by considering defect's or vulnerability's

- Damage potential
- Reproducibility: as successful exploit
- Exploitability: effort and expense to mount attack
- Affected users
- Discoverability: by adversary

5.3.2 Feasibility Analysis

Certain security requirements may make the proposed software product either low feasibility or infeasible from technical, economic, organizational, regulatory, marketing, or societal viewpoints. Feasibility analyses address these issues which can be partially severe given the difficulties in producing secure software.

One can benefit from a willingness to choose less ambitious goals in return for increased feasibility. Insistence on highly skilled staff and restrictions on the process, size, and complexity of the software can have quite positive affects on feasibility.

[4] In the US DoD this could include mission impact assessment classified in accordance with DoD 5200.1-R and DoD Instruction 3600.2.

5.3.3 Tradeoff Analysis

Producers and stakeholders may prioritize and perform tradeoff studies involving security and privacy requirements. Additional security may impact usability, performance, or other characteristics to make the system less desirable. A tradeoff might exist between more security – for example, multiple authentications rather than single – and acceptability, calling for analysis to resolve the problem. Separation of privilege or duties requiring multiple people may tradeoff with a single person plus the additional auditing required. As another example, privacy might benefit from increased granularity of access controls. Thus, many tradeoffs exist during establishment of requirements.

Tradeoffs occur within many activities such as establishing the security policy and specifying external behavior. This last area is a product design activity and, as such, can be fraught with tradeoffs, including ones among qualities. A number of techniques have been created for addressing these trades, such as those in [Kazman 2000], [Kazman 2002], [Prasad 1998], [Despotou 2004], and [Chung 1999]. In practice, stakeholders may tolerate a range of results.

5.3.4 Analysis of Conflicts among Security Needs

Conflicting security needs can derive from differing needs or viewpoints, or from differing policy models, capabilities, or inconsistent possibilities of configurations or settings among components of the system. Some of these derive from real-world needs and some from designs or components for the software system.

5.4 Specification

5.4.1 Document Assumptions

While they may strive for fewer and weaker assumptions, producers must ultimately identify and document the assumptions on which their results depend. These are part of the requirements, meaning they must have agreement and approval.

The following types of assumptions should be included: [ISO-15448, p. 10]

- Aspects relating to the intended usage
- Environmental (e.g. physical) protection of any part
- Connectivity aspects (e.g., a firewall being configured as the only network connection between a private network and a hostile network)
- Personnel aspects (e.g., the types of user roles anticipated, their general responsibilities, and the degree of trust assumed to be placed in the user)
- Operational dependencies; for example, on infrastructure [Meier 2003, pp. 101-105]
- Development environment dependencies – for example, correctness assumptions about tools.

Assumptions include statements predicting bad qualities or events as well as good ones. Some assumptions may be stated conditionally or probabilistically.

Rationales for the assumptions should also be documented, reviewed, and approved. One key question is, “Are the assumptions appropriate for the environments in which the software system is intended to operate – now and in the germane future?” A related goal to show the answer is yes may belong in the assurance case.

5.4.2 Specify Software-related Security Policy

The software system security policy is part of software system requirements placing constraints on system behavior. It can include policies related to confidentiality, integrity, availability, and accountability. It is derived from higher-level policies, laws and regulations, threats, the nature of the resources being protected, or other sources as described in the prior subsections of this section [Gasser 1988, Chapter 9].

Specifying the properties that the system must preserve separately from the specification of its external behavior allows the latter to be explicitly verified to be consistent with these security property constraints. In practice, what is labeled security policy may have in addition to these property constraints, a number of requirements on usages of components or techniques, e.g., cryptographic techniques. These latter requirements form parts of the requirements on how the system will be designed and constructed.

5.4.2.1 Convert Security Needs into a System Security Policy

The security protection needs of assets and the identified threats (existing, future, concrete, or postulated) must be mapped into a stakeholder understandable description of the security policy to govern the software system that the stakeholders can validate. As a basis for rigorously verifying specification, design, and code compliance, the security properties portion of this software system's security policy can then be restated using formal notation. The other portions of the policy need to be put in at least a semi-formal or structured form suitable for tracing, verifying, and evaluating compliance with them in the specifications, design, code, and possibly elsewhere.

5.4.2.2 Informally Specify

As with any critical requirement, the first step is usually creating a carefully expressed, reviewed, and approved informal or preferably semi-formal policy specification understandable by stakeholders and traced to its justifications.

The security policy may include requirements for flexibility so users may currently or in the future specify a given variety of security policies. [ISO-15448]

See section 6, Secure Software Design, for design options that may become an implicit or explicit context such as kinds of access policies, where some security policies terms are stated [ISO-15448].

To avoid excessive rework of the formal statement, developers must first analyze and stakeholders validate this user understandable description.

5.4.2.3 Formally Specify

To ensure uniform correspondence, a clear fully documented mapping must exist between the informal or semi-formal expression of the policy and the formal mathematical one [Giorgini 2005] [NRL Handbook 1995, Chapter 3][5]. To ease the demonstration that external behavior and design specifications are consistent with the security policy's constraints, the formal policy specification must be made in a notation or terms that allow reasoning about this consistency, e.g. [Giorgini 2005]. The obvious choice may be using the same notation to express the security property. Where possible, revalidation may also be appropriate.

5.4.3 Security Functionality Requirements

The Security Functionality subsections in the Fundamentals and Design sections enumerate functionalities. [SCC Part 2 2002] provides perhaps the most extensive catalog of security functionality while the most official is in [Common Criteria v.3.0 Part 2]. The requirements question is to select from the possible functionalities those that are needed for preserving the security properties in the security policy and are appropriate to the dangers and risks. [ISO-15448], [CJCSM 6510.01 2004] [S4EC], [Moffett and Nuseibeh 2003], and with some

[5] See http://chacs.nrl.navy.mil/publications/handbook/SPM.pdf (Accessed 20050917)

redundancy [Moffett 2004], all provide some guidance. The Common Criteria community has several Protection Profiles including lists of functionality for different kinds of software products. Other certification schemes may also make statements about functionality.

The portion of the security policy that is in addition to the security properties may already contain directions about some security functionality.

Security functionality specifics and locations can be strongly affected by the degree of assurance required, compatibility requirements, and architectural decisions as well as development tool choices and decisions to purchase or reuse. Of course, security functionality behavior visible as external behavior of the software system must be sufficient to provide conformance to the system's security policy.

5.4.4 High-Level Specification

The highest level of specification can detail the software system's external behavior and may do so by treating the system as a single unit or as the combined behavior of a small number of high-level parts and their interactions. Just as with any significant software, designing the external behavior of a secure software system and its required non-functional requirements is often a difficult process [Chung 1999]. Also similarly, the use of possibly abstract system state values eases writing the specification. In security, however, these state values – such as where something is stored and if it is encrypted or not – can be a central concern.

While the specification of external behavior is traditionally labeled a requirements activity, it is a product design activity and much of what is in the Design section is relevant here as well.[6] The specification of the external behavior of a system appears in a top-level specification that may treat the system as a single unit or as a small number of units composed into the system.

Skills (and methods/tools) are needed for constructing appropriate (unambiguous, complete, etc.) specifications including formal specification of the software's external behavior. [Sommerville 2004, Chapters 9 and 10] [Summerville 1997] Additional skills may be required for specifying the security property constraints and comparing them to the specifications, e.g. [Hall 2002b] who also addresses formal specification of user interfaces.

Showing that the behavior specified in the high-level specifications meets the security property constraints and the remainder of the security policy is a major step in producing the assurance case. The highest practicable assurance might come from a machine-checked, formal-methods-based logical proof of consistency combined with thorough reviews possibly including simulated attempts at attacks.

The high-level specification of the software system behavior can be the bedrock by which assurance is first gained that the system will be consistent with the required security properties and other relevant portions of the security policy. The remainder of development will then rely on this consistency with security policy, particularly the required security properties to build a system that agrees with this specification and thereby with the security policy.

The system can provide flexibility. The specification and design could be consistent with the security policy by

- Having mechanisms that allow a range of user/operator specification of security policies
- Identifying the configurations or settings for the required policy

The high-level specification must be traceable to identified needs, statements of relevant analyses, and design rationales. Of course, in addition to consistency with security policy requirements, the software system must also perform its intended purposes and provide acceptable usability, performance, and other qualities.

[6] Also remember that – as in "unsecured" systems – while external behavior may be recorded separately from internal design, the problems are intertwined and humans have legitimate trouble thinking of only one of them without the other.

5.5 Requirements Validation

In addition to verifying consistency among requirements artifacts, one needs to ensure they reflect actual needs and show that a system will work as intended under operational conditions and provide the desired benefits. As for all aspects of software systems, one must also validate security needs, the security policy, and security-relevant aspects of the high-level specifications. As mentioned in the subsections above, validation involves producers, the external stakeholders, and security experts.

Producers and relevant stakeholders should explicitly justify any exclusion of requirements deriving from laws, regulations, or standards/guides or clearly be identified by stakeholders.

Developers of secure software must know the relevant validation techniques and their applicability. See the section 8, Secure Software Verification, Validation, and Evaluation; and section 9, Tools and Methods, respectively.

5.6 Assurance Case

Assurance cases were described at some length in section 3, Secure Software Fundamental Concepts and Principles, and are also have significant coverage in section 8, Secure Software Validation, Verification, and Evaluation. The development of the assurance case begins during conception and requirements – and any related acquisition activities – and continues throughout development and sustainment. It is an integral part of technical risk management, and its required development and especially its needed contents have important effects on planning and engineering.

Security policy (particularly security properties), security risks, and assumptions play a central role in the assurance case. Making the case that the system developed meets the first, security policy, given the latter two, is the main purpose of the security assurance case. One major first step is showing the consistency of the behavior specification with the security policy, including required security properties as discussed in the High-Level Specifications subsection.

An assurance case could also address the suitability of policy and assumptions for the expectations, intentions, environment, and external requirements. The Requirements Validation subsection addresses some the relevant steps. Also see section 8, Secure Software Verification, Validation, and Evaluation.

5.7 Further Reading

[Barden 1995] Barden, Rosalind, Susan Stepney, and David Cooper, *Z in Practice*, Prentice Hall, 1995

[Bejtlich 2005] Bejtlich, Richard, *Extrusion Detection: Security Monitoring for Internal Intrusions*. Addison-Wesley Professional, 2005

[Boudra 1993] Boudra, P., Jr., *Report on rules of system composition: Principles of secure system design*. Technical Report, National Security Agency, Information Security Systems Organization, Office of Infosec Systems Engineering, I9 Technical Report 1-93, Library No.
S-240, 330, March 1993.

[Davis 1993] Davis, A.M., Software Requirements: Objects, Functions and States, Prentice Hall, 1993.

[FIPS 188] FIPS 188, Standard Security Labels for Information Transfer, September 1994.

[Fitzgerald 2002] Fitzgerald, Kevin J., "U.S. Defense Department Requirements for Information Security", *Crosstalk*, May 2002.

[Flechais 2003] Flechais, I., Sasse, M. A., and Hailes, S. M., "Bringing security home: a process for developing secure and usable systems," In *Proceedings of the 2003 Workshop on New Security*

Paradigms (Ascona, Switzerland, August 18 - 21, 2003). C. F. Hempelmann and V. Raskin, Eds. NSPW '03. ACM Press, New York, NY, pp. 49-57.

[Giorgini 2004] P. Giorgini, F. Massacci, J. Mylopoulous, and N. Zannone, "Requirements Engineering meets Trust Management: Model, Methodology, and Reasoning." *Proc. of the 2nd Int. Conf. on Trust Management (iTrust)* 2004.

[Goguen and Linde, 1993] J. Goguen and C. Linde, "Techniques for Requirements Elicitation," International Symposium on Requirements Engineering, 1993.

[Gutmann 2004] Gutmann, P., *Cryptographic Security Architecture: Design and Verification*. Springer-Verlag, 2004.

[HMAC 2002] "The Keyed-Hash Message Authentication Code (HMAC)", FIPS 198, March 2002.

[Howard 2003b] Howard, M., J. Pincus and J. Wing, "Measuring relative attack surfaces," *Proceedings of the Workshop on Advanced Developments in Software and Systems Security*, Available as CMU-TR-03-169, August 2003.

[IEEE830-98] IEEE Std 830-1998, IEEE Recommended Practice for Software Requirements Specifications, IEEE, 1998.

[Karger et al, 1990] Karger, Paul A., Mary Ellen Zurko, Douglas W. Benin, Andrew H. Mason, and Clifford E. Kahn, "A VMM Security Kernel for the VAX Architecture," *1990 IEEE Symposium on Security and Privacy,* IEEE, 1990.

[Kotonya 2000] Kotonya, G. and I. Sommerville, *Requirements Engineering: Processes and Techniques*, John Wiley & Sons, 2000.

[Manadhata and Wing 2004] Manadhata, P. and J. M. Wing, "Measuring A System's Attack Surface," CMU-TR-04-102, January 2004.
Available at: http://www-2.cs.cmu.edu/afs/cs/project/calder/www/tr04-102.pdf

[CC 2005] The National Institute of Standards and Technology, *Common Criteria v. 3.0*, July, 2005.

[NIST Special Pub 800-27] Stoneburner, Gary, Clark Hayden, and Alexis Feringa, *Engineering Principles for Information Technology Security (A Baseline for Achieving Security)*, NIST Special Publication 800-27 Rev A, June 2004.

[NIST Special Pub 800-53] Ross, Ron et al. *Recommended Security Controls for Federal Information Systems*, NIST Special Publication 800-53, Feb. 2005.

[NIST Special Pub 800-60] Barker, William C., *Guide for Mapping Types of Information and Information Systems to Security Categories*, NIST Special Publication 800-60, June 2004.

[Open Group 2004] Open Group, Security Design Patterns (SDP) Technical Guide v.1, April 2004.

[Radack 2005] Radack, Shirley, editor, *Standards for Security Categorization of Federal Information and Information Systems*, Federal Information Processing Standard (FIPS) 199, July 10, 2005.

[Riggs 2003] Riggs, S., *Network Perimeter Security: Building Defense In-Depth*, Auerbach Publications, 2003.

[Robertson 1999] Robertson S. and J. Robertson, *Mastering the Requirements Process*, Addison-Wesley, 1999.

[Rowe 2004] Rowe, Neil C. "Designing Good Deceptions in Defense of Information Systems," *ACSAC 2004* http://www.acsac.org/2004/abstracts/36.html.

[Saltzer and Schroeder 1975] Saltzer, J. H. and M. D. Schroeder, “The protection of information in computer systems,” *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278-1308, 1975. Available online at http://cap-lore.com/CapTheory/ProtInf/

[Schell 2005] Schell, Roger. “Creating High Assurance for a Product: Lessons Learned from GEMSOS.” (Keynote Talk) *Third IEEE International Workshop on Information Assurance*, College Park, MD, USA March 23-24, 2005. Available at http://www.iwia.org/2005/Schell2005.pdf.

[Schneier 1999] Schneier, Bruce, “Attack Trees: Modeling security threats,” *Dr. Dobb's Journal*, December 1999.

[SHS 2002] Secure Hash Standard (SHS), FIPS 180-2, August 2002.

[SREIS 2005] *Symposium on Requirements Engineering for Information Security* (SREIS 2005) Paris, France, August 29, 2005. See http://www.sreis.org/

[Thompson 2005] Thompson, H. H. and S. G. Chase, *The Software Vulnerability Guide*, Charles River Media, 2005.

[US Army 2003] US Army, Field Manual (FM) 3-13: *Information Operations: Doctrine, Tactics, Techniques, and Procedures*, 28th Nov., 2003. (Particularly Chapter 4 on Deception)

[US DoD 1996] Joint Chiefs of Staff, DoD JP 3-58, *Joint Doctrine for Military Deception*, 31 May 1996.

[Wyk and McGraw 2005] van Wyk, Kenneth and Gary McGraw, *After the Launch: Security for App Deployment and Operations*, Presentation at Software-related security Summit, April 2005.

[Zwicky et al, 2000] Zwicky, Elizabeth D., Simon Cooper, and D. Brent Chapman, *Building Internet Firewalls* (2nd ed.), O'Reilly, 2000.

6 Secure Software Design

6.1 Scope

Software design is the software engineering activity that takes the products of the requirements activity, particularly the specification of the system's external behavior, and decomposes the system at possibly multiple levels into components and their interactions. Design produces the:

- Description of the software's internal structure that will serve as the basis for
  - Assurance of design's agreement with the specifications
  - Its construction
- Constraints regarding aspects of design and future evolution of the design
- Rationale for the design decisions
- Assurance case portion related to design, including argument and evidence for the design's conformance to external behavior (or top-level) specifications, the system's security policy, and other relevant constraints

Design normally includes descriptions of the architecture, components, interfaces, and other characteristics of a system or component. Often the constraints mentioned in the second major bullet are embodied in an architecture description. [Gasser 1988, Chapters 3-6, 9, 11] provides an introduction to design for secure software systems, and coverage of web application design is provided by [Meier 2003, Chapter 4]. [Berg 2005] provides excellent coverage of many design issues for secure systems.

Traceability must be monitored between relevant elements of requirements, such as the high-level specification or external behavior specification and the architecture and be continued throughout design. This traceability is also reflected in the assurance case.

In addition, design must produce the detail required by the construction activity.

Design of secure software systems is similar to design of any extremely high-assurance software [Neumann 1986] but with such additional concerns as:

- Ensuring consistency with a security policy
- Incorporating necessary security functionality
- Ensuring design does what the specification calls for and nothing else
- Employing separation of mechanisms, duties, and privileges, including the principle of least privilege
- Negating the subversion or maliciousness of designers or other personnel – insiders
- Defending against outside attack on the design infrastructure and artifacts – outsiders
- Creating an assurance case with arguments and evidence that ensure these (or at least the first bullet, which implies much of the others) and addresses the danger of intelligent, malicious actions as well as security relevant random misfortunes.

Two important concerns that also exist in high-assurance systems for other purposes are

- Minimizing what must be trusted
- Reliability and correctness

Despite many commonalities where security is not a concern, these differences, along with the other differing goals raised by security, make designing secure software an exceptional problem for designers with its particular goals and characteristic design principles.

These goals and principles will be addressed first before covering the other aspects of designing secure software systems. While covering only the security-relevant elements of design, this knowledge is substantial, and, to avoid excessive document size, its enumeration leads to the use of several lists. While possibly tedious to some, the lists contain items meaningful in the design of secure software systems.

6.2 Design Objectives

Table 8: Limits to Aid Software System Security provides a number of design-relevant objectives for software systems where security is a concern. Designers of secure software systems can have a number of other distinctive objectives, including: [Gutmann 2004, Chapters 1-4] [Redwine 2005a]

- Design to defend perfectly, then assume this defense will fail and design to defend after initial security violation(s)
- Architecturally eliminate possibilities for violations – particularly of information flow policies
- Have sound:
  - Authentication
  - Authorization and access control
  - Administrative controllability
  - Manageability
  - Comprehensive accountability
  - Tamper resistance
- Design so the system does what the specification calls for and nothing else
- Do not cause security problems for systems in the environment
- Design to tolerate security violations (See subsection on Confine Damage and Increase Resilience.)
- Design for survivability [Ellison 2003]
- Avoid and workaround environment’s and tool’s security endangering weaknesses
- Design for secure operation [Berg 2005, Chapters 10 and 18] [McGraw 2006, Chapter 9]
- Make weak assumptions
- Do not rely only on obfuscation, but exploit deception and hiding
- Have a design that is open and eases traceability, verification, validation, and evaluation
- Provide predictable execution behavior
- Ease creation and maintenance of an assurance case [Berg 2005, Chapter 17]
- Help ensure cost-effective and timely certification of software system security and accreditation of the operational systems

These objectives (from section 3’s Table 8 as wells here) provide designers with a set of goals to pursue in developing their (more) secure systems.

6.3 Principles and Guidelines for Designing Secure Software

This subsection augments the principles in section 3, Fundamental Concepts and Principles.[1]

Also, see [Meier 2003, Chapters 4 and 5] for designing secure web applications.

6.3.1 General Design Principles and Guidelines for Secure Software Systems

Developers need to know secure software design principles and how they are employed in design. They also need to know ways to recognize whether the principles were used in a design and how to evaluate designs and proposed changes, including improvements. (Also, see section 3, Fundamental Concepts and Principles.)

Design includes using abstraction and decomposing the system using architecture and constraints to facilitate quality design and achievement of security requirements [Meier 2003, pp. 47-48, p. 70, and p. 100]. This includes facilitating evolution of required functionality and security policy to address future user needs, particularly changing threats. Designers may alternate between focusing on design of functionality and on examining emergent properties.

One needs a sound architecture amendable to supporting assurance arguments and evidence via review and analysis including formal proofs. To ease production of an accompanying assurance case for the security preserving correctness of compositions, care concerning composability is needed at all levels of detail [Neumann 2003] [Berg 2005, Chapter 11]. Achieving these using known security techniques can aid confidence.

Information hiding and encapsulation are well-established general design principles fundamental to the crucial activity of minimizing and simplifying the portion of the software that must be trusted. A separate trusted computing base can isolate security policy and localize reasoning and assurance arguments. This does not mean that the trusted software must be monolithic, e.g., see [Vanfleet 2005]. To help simplify and minimize what must be trustworthy, one can minimize the functionality included in the trusted parts in several ways:

- Exclude non-security relevant functionality
- Separate policy and mechanism
- Localize or constrain dependencies
- Maintain minimal retained state – e.g., do not keep startup processes or state after startup, or normal running ones during shutdown [Schell 2005]
- Virtualize roles of hardware and device drivers
- Minimize support for functionality, which is outside the trusted software component(s)

Attackers cannot compromise information that is not in the system. If software retains minimal state, attackers will have shorter and fewer opportunities to find information or to execute illegitimate actions.

Among the approaches to modularity, layering has been a crucial information hiding mechanism in secure systems –see [Gasser 1988, section 11.1] for an introduction and [Karger 1990] for an example. Layered and distributed protection can be effective. Object orientation has merit; but beware the difficulties of ensuring the

[1] This section draws heavily on an existing larger compilation from a number of sources in [Redwine 2005b]. Portions of these principles were first collected in [SDI 1992] or [Neumann 2003] as well as [NIST Special Pub 800-27] and are described in more detail there. Discussions of some appear in [Howard and LeBlanc 2003], [Viega and McGraw 2002], and [Bishop 2003].

correctness of inheritance, polymorphism, and generics. Indeed, one may need to avoid these unless adequate means of analysis and assurance for their use are in place.

The most basic approach is to reduce the possibilities for security violations. Among the techniques and principles available for this are to:

- Deny access unless explicitly authorized
- Deploy with secure initial defaults
- Check every access
- Implement least privilege

Separation can eliminate or reduce the possibilities of certain kinds of violations via:

- Least common mechanism – avoid shared mechanisms
- Separation of duties
- Separation of privilege
- Separation of roles
- Separation of trust domains [Berg 2005, Chapter 6]
- Constrained dependency

This list identifies the roles/responsibilities that, for security purposes, should remain separate. For example, they suggest it is better to have several administrators with limited access to security resources rather than one person with "super user" permissions.[2]

Isolation of software systems is a variation on this use of separation.

- Public accessibility: to the extent possible, isolate publicly accessible systems from mission-critical resources (e.g., data, processes).
- Physical isolation: no physical connection exists between an organization’s public access information resources and the organization’s critical information.
- Logical isolation: layers of security services and mechanisms should be established between public systems and secure systems responsible for protecting mission-critical resources.
- Domain isolation: use boundary mechanisms and guardian mechanisms to separate computing systems and network infrastructures to control the flow of information and access across network boundaries, and enforce proper separation of user groups.

Other ways to reduce possibilities include:

- Reducing the number of input/output points – the attack surface [Howard, Pincus, and Wing 2003] [Manadhata and Wing 2004]
- Not implementing unnecessary functionality

Unnecessary security mechanisms can add complexity and increase potential sources of additional vulnerabilities.

[2] In current practice, super-users tend to have their behavior as recorded in audit logs that are later audited/reviewed. Of course, this implies the super-user cannot tamper with the system’s capturing and recording of the data for audit logs or with the logs.

Where possible within design constraints and ensuring analyzable results, security functionality might best be based on sound, open standards that aid in evolution, portability, and interoperability, as well as assurance.

Modularity and separation enable a system to:

- Defend in depth
- Have diversity in defenses [Zwicky, et al.]

Possible lack of variety can be indicated not just by items being identical but also by common heritage of software, common involvement of persons or practices, common settings, and common components. To be most effective, combine [overlapping] physical, procedural, and software-related security [Moffett and Nuseibeh 2003]. Defense-in-depth's success could be more likely if the defenses are diverse in nature.

Designers should avoid exposing assets or weaknesses in any system state including startup, operation, shutdown, exception handling, system failure, updating, recovery, and loss or disposal.[3]

To predict execution behavior the design must be analyzable. For security concerns, this means that, at a minimum, security-relevant behavioral aspects need to be analyzable. This analysis might include sequential behavior and concurrency for the existence of the potential to violate the required security properties. Almost all concurrent structures require automated analysis for adequate confidence.

Treating a design as holistically as is practicable avoids an unreasonable combination of seemingly reasonable local design decisions. To maintain the same level of assurance, one must reuse designs and components only if they are known to be (or can be shown to be) secure in the fashion required for this system.

6.3.2 Damage Confinement and System Resilience

A designer cannot depend on a software product's defenses to be perfect. One response to this issue is to design and operate a secure software system to:

- Limit damage
- Be resilient in response to events

Aims include making the system resistant to attack, limiting damage, and having it detect and recover rapidly when attacks do occur. [Pullum 2001] provides definitive coverage of software fault tolerance, [Stavridou 2001] provides an introduction to intrusion tolerance, and [Berg 2005, Chapter 18] addresses responses to failure. This can be accomplished in a variety of related ways:

- Limit or contain vulnerabilities' impacts
- Choose safe default actions and values
- Self-limit program consumption of resources
  - Attempt to exhaust system resources (e.g., memory, processing time.) is a common attack. Add capabilities into the program to prevent overusing system resources
- Design for survivability [Ellison 2003]
- Ensure system has a well-defined status after failure, either to a secure failure state or via a recovery procedure to a known secure state [Avizienis 2004]
  - Rollback
  - Fail forward
  - Compensate

[3] "Defense in breath" is a phrase sometimes used for all encompassing defenses including across the system parts.

- Fail securely
- Eliminate “weak links”
- Implement layered security (no single point of vulnerability).
  - Especially important when COTS products are used.
- Be able to recover from system failure in any state
- Be able to recover from failure during recovery (applies recursively)
- Make sure it is possible to reconstruct events
- Record secure audit logs and facilitate periodical review to ensure system resources are functioning, confirm reconstruction is possible, and identify unauthorized users or abuse
- Support forensics and incident investigations
- Help focus response and reconstitution efforts to those areas that are most in need

Software should not just fail when a damaging problem occurs but should continue to run, at least in a restricted but secure way, degrading gracefully. A well-designed software system can provide assurance that the system is, and shall continue to be, resilient in the face of expected threats. The instructive example of the MAFTIA design is discussed in [Stroud 2004].

Software should also fail safely:

- Fail “open”: If patient “sensor” software happens to fail, should it turn off the life support machine? No!
- Fail “closed”: If a firewall dies, should the server’s network connection be shutdown? Yes!

Finally, the software is part of a system and organization which can establish their own layers of defenses including “detect and respond” capabilities, manage single points of failure in inner/lower layers, and implement a reporting and response strategy.

6.3.3 Vulnerability Reduction

A number of approaches exist to reduce vulnerability:

- Ensure proper security at system shutdown and disposal
  - Although a system may be powered down, critical information still resides on the system and could be retrieved by an unauthorized user or organization.
  - At the end of a system’s life-cycle, procedures must be implemented to ensure system hard drives, volatile memory, and other media are purged to an acceptable level and do not retain residual information.

Common errors and vulnerabilities must be identified and prevented. Many errors recur with disturbing regularity, for example

- Buffer overflows
- Format string errors
- Failing to check input for validity
- Programs/processes being given excessive privileges

Sources for lists of vulnerability categories and instances include books and websites. See section 7, Secure Software Construction.

Separation of duties and privileges, and mechanisms can aid in avoiding a "single point of total vulnerability," analogous to avoiding a "single point of failure." As an additional example, one might require secret sharing (splitting) across machines for added confidentiality or redundancy across machines as well as for improved availability and integrity. Of course, secret sharing (splitting) across machines increases the number of machines that must be accessed to access the secret.

A number of techniques, issues, and benefits result from carefully analyzing proposed designs, identifying, and reducing vulnerabilities. [Swiderski and Snyder 2004] gives substantial coverage to vulnerability analysis although they use the term threat modeling, which covers both threat analysis and vulnerability analysis. [McGraw 2006, Chapter 5 and Appendix C] address architecture risk analysis. Beyond the obvious advantages of finding vulnerabilities, vulnerability analysis can lead to better designs.

6.3.4 Viewpoints and Issues

Some viewpoints and issues are special for secure software design. These normally include:

- Design with the enemy in mind
  - Understand that subversion is the attack mode of choice e.g., subvert people, processes, procedures, testing, repair, tools, infrastructure, protocol, or systems
  - Understand and enforce the chain of trust
  - Do not invoke untrusted programs from within trusted ones.
- Test any proposed design decision against policy and ease of assurance
- Be aware of composition difficulties for security properties
- Design with the network security in mind
  - Implement security through a combination of measures distributed physically and logically
  - Associate all network elements with the security services and advantages or disadvantages they provide
- Cross (trust or custody) domain issues
  - Authenticate users and processes to ensure appropriate access control decisions both within and across domains
  - Level of trust is always an issue when dealing with cross-domain interactions
  - Formulate security measures to address multiple overlapping information domains
  - Place safeguards on information flows
- An efficient and cost-effective security capability should be able to enforce multiple security policies to protect multiple information domains without the need to (excessively) physically separate the information and the respective information systems processing the data.

These viewpoints and issues change for each system being developed, so they should be analyzed for each development effort.

6.4 Documentation of Design Assumptions

The list of assumptions made primarily about the software systems environment is one of the products of the requirements activity. These assumptions may change as the design develops, hopefully, by discovering design approaches that allow making them fewer or weaker. They may also grow if dependencies are introduced by the design – for example, by reusing a software component whose assurance case derives in part from uncheckable assertions or evidence from its supplier. See Requirements subsection on Assumptions for more discussion of contents.

6.4.1 Environmental Assumptions

Assumptions about the environment and its interfaces to the software system can aid in simplifying the design and assurance arguments but also can provide bases of building attacks.

Development environment and operational environment assumptions should be documented.

6.4.2 Internal Assumptions

Assumptions made about items inside the software system could be the result of assumptions about the environment or could be strictly internal. Conditions guaranteed to parts of the system by other parts of the system would be misidentified as “assumptions,” as they are conditions that must be provided by the software system design.

6.5 Documentation of Design Decisions and Rationales

Good practice calls for documenting design decisions made for security reasons and ensuring traceability of design items and decisions to requirements and code. Traceability is essential for the assurance case. The design rationale can often provide arguments or evidence suitable for inclusion in the assurance case.

6.6 Software Reuse

Several facts about reuse in a high-assurance system are straightforward.

- Reused software that includes defects may undermine the trust in the entire software system.
- When requiring reused software to meet the level of assurance of newly developed software is not feasible, one must still deal with the resulting uncertainty and risks.
- System operators and sustainers may have little control over software fixes and patches for software originating elsewhere.

One may be reusing much more software than one realizes as compilers and frameworks can incorporate a substantial amount of code implicitly [McGraw 2006, pp. 167-169].

To ensure the assurance case remains adequate, one may reuse prototype software in production versions only if its traceability, documentation, and evidence adhere (or is enhanced to adhere) to the levels required for the overall assurance case. This case implies all the normal production-level reviews, analysis, and tests are performed.[4]

With rare exceptions, composing systems from COTS, government off-the-shelf (GOTS), or open source products or components is fraught with security problems and existing vulnerabilities. One need only read the long lists of known defects and vulnerabilities for common COTS and open source software to realize the level of danger (even if most known ones are fixed). While the assurance issues raised by open source and COTS/GOTS are no different than those for custom software, typically, they are not accompanied by the evidence required for needed confidence.

Here, even an emphasis on

[4] Quite apart from security, good practice would call for incorporating prototypes into a production version only if the prototype was designed with production standards in mind. Otherwise one might be going “live” with something never designed for a production environment.

- Performing adequate component analysis prior to acquisition
- Analyses of the frequently awkward security-related issues in composition
- Testing and assessment of resulting composite system

can usually provide high confidence only if the individual components deserve high confidence. Identifying alternatives and performing trade-off studies on software components can therefore be critical.

If “forced” to use normal OTS software, one can consider other ways of avoiding and sharing risks as well as using defense-in-depth, damage confinement, and resilience techniques. One family of techniques used is to attempt to surround the OTS product in a “wrapper.”

When integrity is the only issue, relying primarily on high-assurance resilience and recovery may be acceptable. If the amount of risk is unacceptable, one needs to consider the option of not producing the system as it may be an insecure OTS-based system containing high-value assets or having purposes or uses with significant security risks.

For OTS-based systems with security requirements, preparing the best practicable assurance case from existing or producible information and evidence is crucial, as system owners, developers, operators, and users need to know the amount of assurance (or lack thereof) that exists to make decisions and manage risk in a potentially perilous situation. Part of the Acquisition section addresses this situation.

Subcontracting or outsourcing can reduce one’s intimate visibility and control over software production and assurance, causing increased uncertainties and risks. Whenever possible, security requirements should be defined in original contract language, with provisions for thorough testing and evaluation; and any remediation is required before acceptance or deployment. These requirements can result in the need for special arrangements and activities as described in the Acquisition section.

Acquiring software through OTS, contracting, or outsourcing (production or production support) neither removes any of the same security requirements as would hold for in-house production nor does it eliminate the need to provide the same or higher quality assurance case and evidence. They may increase the path along which integrity of the product and accompany assurance evidence must be maintained. If it is a portion of a larger system, then its assurance case must integrate properly with other assurance arguments in the remainder of the larger assurance case. The same is, of course, true of any the contractor’s suppliers and in the worst case for the entire supply chain.

6.7 Architectures for Security

Security-aware architectures need to address the concerns of high-dependability and those particular to preserving the required confidentiality, integrity, and availability under attack.[5]

Secure architectural styles or style elements include:

- Reference monitors
- System high
- Multiple Independent Levels of Security (MILS)
- Multiple Single Levels of Security (MSLS)
- Distributed access control
- Layered

[5] The International Association of Software Architects site is at http://www.iasahome.org/iasaweb/appmanager/home/home and contains relevant material.

- Tolerant – to (partial) attacker success, including “self healing” approaches
- Adaptive distributed reconfiguration responses to attacks [MAFTIA]
- Compartmentalization via–
  - Virtual machines
  - Separation via encryption
  - Physical separation
  - Separation except at point of use
  - Filters, guardians, and firewalls

[Neumann 1995], [Neumann 2000], and [Vanfleet 2005] discuss a number of relevant architectural styles. [Gasser 1988, Chapters 4, 5, 11, and 13] and [Meier 2003, p 40 and p 100] discuss general issues. (Also see subsection 6.3.2, Damage Confinement and Resilience.) {McGraw 2006, Chapter 5] addresses architecture risk analysis.

6.7.1 Access Control Issues

There are a number of access control concepts, policies, and issues. [Bishop 2003, pp. 381-406] [Gasser 1988, Chapter 6] Policies can be used individually or in combination. In addition to the generic access policies listed in the section 3, Fundamental Concepts and Principles, concepts and issues include:

- Access control process and mechanisms
- Access control in distributed systems/databases
- Disclosure by inference
- Potential exploitation of access control to create possibilities for disclosure by covert channels

See the subsection 6.8.2 on access control mechanisms.

6.7.2 Cross-Domain Control

Information crossing the boundary between security domains with different policies (or in practice ownership) raises potential problems for both the originating domain (Should it be allowed to leave? Does it need to be encrypted first?) and the receiving one (Can it be trusted? How should it be handled?). Guards may be placed on one or both sides of a boundary.

One particular problem is that information already encrypted – say to gain the benefits of end-to-end encryption – cannot be directly inspected at the boundary. Providing cross-domain solutions has become a niche area.

6.8 Security Functionality

Areas of knowledge needed include:

- Kinds of security functionality [SCC Part 2 2002] [Common Criteria v.3.0 2005, Part 2]
- Design of security functionalities
  - Individually and in combination
  - Potential conflicts
  - Centralized and decentralized security functionality

  - Planning for evolution of security functionality

- Availability, characteristics, and appropriate use of OTS or reusable security functionality

The Common Criteria call for self-protection and non-bypassability of security functionality and protection for boundaries to domains of trust. [Common Criteria v.3.0 Part 3, pp. 96-101]

Three areas of functionality important to designers are managing information about identities and authenticating them, controlling access, and using cryptographic functionality.

6.8.1 Identity Management

An identity may represent an actual user or a process or data with its own identity. [Bishop 2003, Chapter 14]

Use identities authenticated by (possibly multiple) means sufficient for the threat environment and asset value to:

- Maintain accountability and traceability of a user
- Assign specific rights to an individual user or process
- Provide for non-repudiation
- Enforce access control decisions
- Establish the identity of a peer in a secure communications path
- Prevent unauthorized users from masquerading as an authorized user.

By considering the threat environment and value of the asset being protected, the need for multiple authentications of an identity is understood.

6.8.2 Access Control Mechanisms

The rights that can be asserted over entities in that system is sometimes called its "protection state." [Bishop 2003, Section 2.1]. The ability of a system to grant, deny, or modify those rights constitute the process of access control within a system. The various methods used to control access to entities are the subject of this subsection. This subsection is only an overview of introductory highlights. [Berg 2005, Chapter 7] has 130 practitioner-relevant pages on access control concepts and a separate chapter, [Berg 2005, Chapter 8], on classification and compartmentalization.

The access control matrix is a very general framework for describing a systems protection state [Bishop 2003, Section 2.2]. Users form the rows of the matrix, while the entities in the system are the columns. Entries in a particular cell of the matrix describe the access the user from that row has over the entity from that column.

A particular access control method can be classified as either mandatory or discretionary [Bishop 2003, pp. 103-104]. A mandatory access control scheme is enforced by the system and cannot be altered or overridden by users of that system. Often these schemes are rule-based and implement organizational security policies. Discretionary access controls, on the other hand, are optional controls set by system users with the appropriate rights to the entity.

6.8.2.1 Access Control Lists

Access control lists (ACL) are typically identity based mechanisms associated with the controlled entity. In the access control matrix described above, ACLs are a column-wise partition of the matrix. An identity may be that of a particular user of the system, but can also be more general. That is, the identity may be held by several users of the system in the form of group membership or a role being played by a user.

Procedure-based access control is a form of a trusted interface mechanism. To access an entity, users must not only have been granted permission but must also use a specified procedure to accomplish that access.

A propagated access control list (PACL) provides the creator of an entity control over the use of that entity [Bishop 2003]. Any user allowed access to the object may only do so in accordance with the originator's policy. Thus, PACLs are suitable for implementing an originator-controlled (ORCON) access policy.

Access conflicts can arise if a user possesses multiple access permissions to an entity. For example, if by virtue of a user's identity, read/write access is granted and by virtue of the same user's group membership, only read access is granted, which access permissions take precedence? Normally, if an access type is held by a user, then that type of access can be exercised. However, this does not have to be the case. Interestingly, some mechanisms support the reverse: a "deny access" setting on an object for a user.

6.8.2.2 Capabilities

In contrast to ACLs, whereby access rights are associated with the controlled entity, capabilities are associated with users. Capabilities describe the objects a user is 'capable' of accessing and the manner of that access. In an access control matrix, capabilities are a row-wise partition of the matrix.

6.8.2.3 Locks and Keys

Locks and keys can be thought of as a combination of an ACL and a capability. The lock is information associated with the protected entity, while the key is information associated with or possessed by one or more users [Bishop 2003]. Sometimes cryptographic secret-sharing algorithms are used to prevent access to an entity unless a given number of users agree to that access (e.g., the Shamir secret sharing scheme).

6.8.2.4 Ring-based Access Control

Ring-based access control generalizes the concept of a super-user and an ordinary user used in many operating systems [Bishop 2003] to an arbitrary number of levels. Entities within a system operate within a certain ring level and must take certain system- or policy-directed actions to access entities in other rings.

6.8.2.5 Revocation

While revocation of access privileges may seem to be a simple task, it can become quite difficult to administer. For example, ensuring revocation when users and entities are distributed across domains, systems, or networks is problematic. Further complicating matters is whether rights granted to, say, User B by User A, should be revoked if User A rights are revoked.

6.9 Proper Use of Encryption and Encryption Protocols

In this area, experts should be consulted before establishing organizational policies or making project decisions [Meier 2003, p. 91]. To choose an encryption scheme or protocol properly, one must know the existing standards (official and de facto), their characteristics (strengths and weaknesses), and their future prospects [NIST Special Pub 800-53, Final, Appendix F], [FIPS 140], and [FIPS 180].

Computational difficulty is the fundamental measure of encryption merit, and knowledge should be possessed of the computational requirements to break, as well as to use, different forms of

- Encryption and decryption
- Calculating and reversing hashes

Hashing is an example of an area without adequate theoretical foundations where algorithms have been shown to be weaker than thought. Such areas deserve careful treatment and concern for potential future problems. For adequate assurance, only cryptographic software certified to [FIPS 140] or NSA Type 1 (or official equivalents) should be used.

For communication and storage designing with the assumption that everything of value should be strongly encrypted, including integrity checks and explicitly justifying other choices can lead to better security. (See section 3, Fundamental Concepts and Principles for more discussion of encryption.)

6.10 Frameworks

Many design efforts occur within a given framework such Sun's J2EE or Microsoft's .net [Meier 2003, Part III] [McGraw 2006, p. 150]. When this is true, designers need a thorough knowledge of the framework's security philosophy and features as well as its strengths, weaknesses, and dependences. In addition, knowledge of the approaches taken by the major frameworks and the experiences with and analyses of them can contribute to all secure software designers understanding of the field.

6.11 Design Patterns for Secure Software

The Open Group, a vendor- and technology-neutral standards consortium, has proposed several design patterns for secure software [Open Group 2004]. A good example of (retrospective) use of patterns is in [Hafiz 2004]. Materials from a number of different PLoPs are available through http://www.hillside.net/ including EuroPLoP workshops on security-related patterns. [Schumacher 2006] brings together a number of patterns.

6.12 Database Security

Databases are parts of many systems with security requirements. Many of the design issues involve establishing proper security access policies. Secure databases are their own subject. [Bertino 2005] provides an introduction to database security concepts as does [Guimaraes 2004] on teaching database security.

6.13 Specify Configurations

Specifying proper software configurations and settings can reduce vulnerabilities and facilitate security having significant impact.

- Specify database security configuration(s) [Viega 2005, p. 79] [Thuraisingham 2005]
- Specify operational configuration(s) [Viega 2005, pp. 98-99]

This includes resource-based security configurations. Configuration information is part of the contents of an operational security guide but do not rely on users to read documentation, make uniform decisions, or disable features [Schoonover 2005] .The Center for Internet Security issues a number of configuration setting guides for common software.[6]

6.14 Methods for Tolerance and Recovery

While generally prevention is to be preferred, designers of secure systems cannot assume preventive or proactive measures will always succeed. The field of fault tolerance [Pullum 2001] and the associated field of intrusion tolerance, e.g. [Stroud 2004], have developed a number of techniques. [Berg 2005, Chapter 18] addresses responses to failure. Recovery techniques have a long history as well, including disaster recovery. A

[6] The Center for Internet Security website is www.cisecurity.org

system can undertake a number of activities related to tolerance of errors or violations of correctness or legitimacy.

- Forecasting violations
- Detection of violations, possible violations, and non-violating activity
- Notification and warning
- Recording, usually via logs
- Damage isolation or confinement
- Continuing service, although possibly degraded
- Diagnosis of cause of violation
- Repair of fault or vulnerability
- Recovery of the system to a legitimate state
- Tactics that adapt to attacking (and equivalent non-malicious) actions

One might also add to the list warn, characterize, investigate root cause or causer, analyze, and learn and improve. Together, these actions attempt to make things right after something is found to be wrong. As a bonus, these actions may also set the stage for prevention of reoccurrences.

See subsection 6.3.2, Confine Damage and Increase Resilience, for related ideas.

6.15 Deception and Diversion

Design knowledge includes purposes, principles, and techniques of deception. A brief introduction with motivations appears in [Hunt 2005].

6.15.1 Purposes of Deception

Deception could be used to increase the uncertainty of the attacker or to create an impression that will lead to the attacker's disadvantage or the defender's advantage. Purposes of deception by the defense include:

- Intelligence gathering
- Diversion of illegitimate attention and resources
- Effects outside the system such as attracting others' attention, good or bad publicity, or aiding in law enforcement or mission accomplishment
- Deceptions may also sometimes be used to help conceal other deceptions

6.15.2 Purposes of Obfuscation and Hiding

Obfuscation and hiding could be used to increase the uncertainty of the attacker or to create an impression that will lead to the attacker's disadvantage or the defender's advantage. Purposes of obfuscation and hiding of software include:

- Reducing the likelihood that the attacker can locate the software artifacts in order to discover their vulnerabilities or insert malicious logic;
- Reducing the likelihood that the attacker can gain an understanding of the software's residual vulnerabilities;

- Increasing the difficulty of reverse engineering of the software (through decompilation or disassembly).

6.15.3 Principles of Deception

Rowe expands on the implications for information security deceptions of the six principles suggested by Fowler and Nesbitt [Fowler and Nesbitt 1995] [Rowe 2004b]:

- Deception should reinforce enemy expectations.
- Deception should have realistic timing and duration.
- Deception should be integrated with operations.
- Deception should be coordinated with concealment of true intentions.
- Deception realism should be tailored to the needs of the setting.
- Deception should be imaginative and creative.

6.15.4 Particular Techniques for Deception

Techniques include such approaches as honeynets, decoys, disinformation, and virtual views of system [Holz 2005], [Cheves 2005], [Hunt 2005], [Rowe 2004a], [Cohen 2001], and [Honeynet 2002].

6.16 Software Protection

A number of schemes exist to try to enforce intellectual property rights, e.g. license's files, or avoid adversaries understanding or reverse engineering a product [Atallah, Bryant and Sytz 2004]. Numerous commercial products exist to copy protect, license, encrypt, or obscure software to aid in protecting intellectual property and resisting reverse engineering. The DoD has a Software Protection Initiative [Clarke 2003]. Techniques used in relation to malware may also illuminate a number of techniques. [Szor 2005] See the section 7, Secure Software Construction, for a discussion of resisting tampering.

6.16.1.1 Anti-Tamper Technologies

Tampering occurs when an attacker modifies a program or data such that it "continue[s] to operate in a seemingly unaffected manner, but on corrupted data or in a corrupted state [Goertzel and Goguen 2005]." Anti-tamper technology can be classified using the following categories [Atallah, Bryant and Sytz 2005]: (1) hardware-based protections, (2) wrappers, (3) obfuscation, (4) watermarking, and (5) guards.

Hardware-based protection is often based on a trusted processor. In one protection scenario, a trusted processor verifies the integrity of other hardware devices in the system upon boot-up and perhaps stores cryptographic keys or other means of verifying the trustworthiness of software that will be executing on the system in trusted memory [Atallah, Bryant and Sytz 2005]. This type of anti-tamper protection will likely be suitable for only the most critical security aspects of a system. "A less drastic protection … also involves hardware, but is more lightweight such as a smart card or physically secure token. These lightweight … techniques usually require that the hardware be present for the software to run, to have certain functionality, to access a media file, etc. Defeating this kind of protection usually requires working around the need for the hardware rather than duplicating the hardware [Atallah, Bryant and Sytz 2005]."

Wrappers are a common method of incorporating "new" technology or behavior into legacy code or software libraries by intercepting calls to the legacy code and enhancing the characteristics of the legacy software within the wrapper code [Birman 1996]. In the context of anti-tamper techniques, an encryption wrapper can be used to protect part (or all) of an active process's instructions and data in memory. Only the portion being executed is decrypted, and the decryption takes place as “close” as possible to the item’s use [Atallah, Bryant and Sytz 2005].

The object of obfuscation is to render software resistant to analysis or reverse engineering. Obfuscation is added to software at various levels, including the source and object code level. Within source or object code, obfuscation techniques include altering program layout, control structures, data encoding, and data aggregation. It is generally recognized that relying on obfuscation as the sole software protection means is poor practice. However, the value of using obfuscation in conjunction with other anti-tamper techniques is recognized as well.

Watermarking “embeds information into software in a manner that makes it hard to remove by an adversary without damaging the software’s functionality [Atallah, Bryant and Sytz 2005]”. In contrast to watermarks in steganography, anti-taper watermarks need not be hidden as they have a deterrent effect [Atallah, Bryant and Sytz 2005].

“A specific type of watermarking is fingerprinting, which embeds a unique message in each instance of the software for traitor tracing [Atallah, Bryant and Sytz 2005].” This type of anti-tamper technique is sometimes called code-signing as well.

Software guards are used to detect tampering during execution. “[A] guard’s response when it detects tampering is flexible and can range from a mild response to the disruption of normal program execution through injection of run-time errors […] Generally, it is better for a guard’s reaction to be delayed rather than to occur immediately upon detection so that tracing the reaction back to its true cause is as difficult as possible and consumes a great deal of the attacker’s time [Atallah and Bryant and Sytz 2005].”

A reference monitor is a tamperproof, trusted access or interface point that mediates access to objects within a system [Bishop 2003]. Thus, a reference monitor can be considered a type of guard as well, albeit not an anti-tamper guard as discussed above. Critical services or portions of code are often secured using some type of reference monitor.

6.17 Forensic Support

The first basic support for forensics is to collect and maintain audit logs of actions and accesses. The logs can record more data or less data, but in a secure system their integrity, and depending on requirements, their confidentiality must be protected. Logs are the objects of attacks or change attempts to cover up an adversary’s actions. Confidential data needs to be protected in logs as in regular storage. In addition, designs must avoid allowing the exhaustion of log storage space to become a form of attack.

Other forensic support includes aid in identifying suspects and investigating insiders and outsiders. For insiders where the identity of the user may be known, automated recognition of use in an unusual fashion could help support identification of suspects.

6.18 User Interface Design

Because achieving a design providing usable security may need to derive from a basic reconceptualization of the solution, a user interface design should begin during process and task analyses or possibly during reengineering. Lists of principles in [Yee 2004] and [Garfinkel 2005] extend sound user-system interaction design methods to cover security issues. [Cranor 2005] has design ideas, including a number concerning privacy. A series of research-oriented workshops exists in this burgeoning research area that may address practical issues [SOUPS 2005]. [Hall 2002b] addresses how one might describe user interactions within one formal methods approach for designing to ensure security.

6.19 Assurance Case for Design

The arguments and evidence related to design are central to the assurance case [SafSec Standard] [SafSec Guidance]. These arguments necessitate assurance mappings between system descriptions. The design must agree with the external specification – doing no more and no less – and must conform to the system security policy. This position can have implications about how to structure the assurance case [Kelly 2003]. Usually, parts of the assurance case have an internal structure that mirrors the structure of the software system.

As an example approach, SafSec [SafSec Standard 2005, p. 23] calls for "modules" to have interface specifications covering dependability properties analogous to functionality in "design by contract." These modules consist of:

- "Guarantee Clause – entries define the set of dependability properties that the module provides, assuming the other four elements of the MBC hold. The guarantee will specify the responsibility that the module takes for reducing risks or supplying services unconditionally. The other clauses supply the conditions in which the guarantee will hold, such as the level of [uncertainty], limitations and assumptions. The risk assessment and mitigation processes that use the guarantees must ensure that the level of responsibility taken by the module is sufficient to mitigate risks that might arise from it – any behaviour inconsistent with the guarantee should be considered as a possible cause of increased risk in the environment using the module.
- "Rely Clause – entries define dependencies on interfaced modules, including the required dependability targets. A rely clause can represent a potential cause of loss if it cannot be met at the level of dependability expected – they should thus be treated as significant events in the risk assessment process.
- "Context Clause – assumptions made on the module's operational context.
- "Assurance Requirement – the level of [uncertainty] in the guarantees of the module that has been verified, based on the supporting evidence.
- "Counter Evidence – the residual risk for the module and its known defects and limitations."

This quote does not call for a particular level of rigor in stating these, but, clearly, the more rigorously and compatibly these are stated, the more use they will be in reasoning about composing modules together to form larger entities as well as in reasoning about decomposion.

See the section 3, Fundamental Concepts and Principles, and section 8, Secure Software Verification, Validation, and Evaluation, on assurance cases.

6.19.1 Design for Easier Modification of Assurance Argument after Software Change

Any security-relevant software changes would in turn require changes in the assurance case – if not in the argument, then certainly in the evidence. Most obviously, new testing evidence is necessary.

Designs with clean separations among parts may allow localization of assurance arguments and evidence, thereby facilitating change in the assurance case (as well as the product [Berg 2005, Chapter 17]).

6.19.2 Design for Testability

While emphasis on analysis is essential, so is testing. Three key properties affect testability [Williams 1982]: [7]

- Predictability is a measure of how difficult it is to determine what a test's outcome should be.
- Controllability is a measure of how difficult it is to provide inputs to the system to drive its execution.
- Observability is a measure of how difficult it is to capture and determine whether the test results are correct.

The complexity and consistency of an application affect its predictability in terms of testing. As complexity increases and consistency decreases, requirements should document more precisely the features and behaviors of the system to support testing. The architecture and design of an application affect its controllability and observability. Designing a system for testability eases the effort required to test it, and is often critical to supporting test automation.

6.20 Secure Design Processes and Methods

In addition to all the non-security-related design and process knowledge, security-related needs exist that affect process and methods. Creating the design must proceed hand-in-hand with constructing assurance argument for the sakes of feasibility, economy, and quality. Security-relevant components need to be traceable to security policy and needs. The composition of system components must yield the behavior called for in the external behavior specification; however, to check on this as one designs, one needs to produce (or at least the arguments plus planned evidence) the relevant parts of the assurance case.

To allow tradeoffs aimed at maximizing security while minimizing its negative impacts on the system and users, multiple alternatives should be considered within a well-structured decision process. Designing defense-in-depth may raise the tradeoff between fewer, more expensive layers of possibly stronger defenses or more layers, each possibly weaker but costing less.

Maintaining security properties is easier if during detailed design and system evolution architectural coherence is preserved and architectural constraints are never violated.

Formal notations [Hall 2002a] or combined formal and semi-formal notations (e.g., UMLsec [Jürjens 2004] [Jürjens 2005]) and accompanying methods exist to address security. Generally, design notations are abstractions and cannot ensure properties of aspects not included in the notation. See the Patterns subsection.

Modifying the design of an existing or legacy software system to be more secure may at times be relatively straightforward though tedious at the detailed design level, but architecture level changes can be severe and expensive. Security properties, being emergent properties, can lead to architecture changes being a necessity. Any redesign process must keep firmly in mind the existing artifacts and the cost involved in changing them. Because of resource and schedule issues, the changing of a design may need to take place over several releases, thereby adding another difficulty to the redesign.

For more information on the design description techniques see the section 9, Secure Software Tools and Methods, and section 10, Secure Software Processes.

[7] The "definitions" given by Williams in these bullets are not the meanings of these terms in their full generality although the general ones speaking in terms of (internal and external) behavior as opposed to test results and causing (by any means) as opposed to driving by inputs are, especially for controllability and observability, possibly even better for defining testability in the abstract.

6.21 Design Reviews for Security

Designs need to be open and amenable to verification, validation, and evaluation, including certification and accreditation. Section 8, Secure Software Verification, Validation, and Evaluation, addresses performance of these activities, and section 5, Secure Software Requirements, addresses the product needs they generate.

Design reviews should be performed by multiple persons covering each area of relevant expertise and legitimate stakeholder interests [Meier 2003, Chapter 5]. Formal techniques that exist for reviews include a scenario-based [Bass 2001] one created for architecture reviews. Reviews including security issues are essential at all levels of design. An independent outside review is recommended by [McGraw 2005]. Design-related portions of the assurance case should be reviewed as well. Since the best results occur when one develops much of the design assurance case along with the design, these parts may be best reviewed together. Using checklists can also help [Meier 2003, pp. 687-740] [Ramachandran 2002, Chapter 1].

Properly performed design reviews also allow one to gain:

- Added assurance that the system does what the specification calls for and nothing else
- Added [uncertainty] that assurance case and design map correctly to each other
- Improved likelihood of cost-effective and timely certification of software system

See section 8, Secure Software Verification, Validation, and Evaluation.

6.22 Further Reading

[Abran 2004] Abran, Alain, and James W. Moore (Executive editors); Pierre Bourque, Robert Dupuis, Leonard Tripp (Editors). *Guide to the Software Engineering Body of Knowledge*. 2004 Edition. Los Alamitos, California: IEEE Computer Society, Feb. 16, 2004. Available at http://www.swebok.org

[Barden 1995] Barden, Rosalind, Susan Stepney, and David Cooper, *Z in Practice*, Prentice Hall, 1995

[HMAC 2002] The Keyed-Hash Message Authentication Code (HMAC), FIPS 198, March 2002.

[Leveson 1986] Leveson, N. G. 1986. “Software safety: why, what, and how.” *ACM Comput. Surv.* 18, 2 (Jun. 1986), 125-163. http://doi.acm.org/10.1145/7474.7528

[Mantel 2002] Mantel, Heiko, “On the Composition of Secure Systems,” *IEEE Symposium on Security and Privacy*, p. 88, 2002.

[NIST Special Pub 800-27 Rev A 2004] Stoneburner, Gary, Clark Hayden, and Alexis Feringa. *Engineering Principles for Information Technology Security (A Baseline for Achieving Security),* Revision A, NIST Special Publication 800-27 Rev A, June 2004.

[NIST Special Pub 800-53] Ross, Ron,m et al. *Recommended Security Controls for Federal Information Systems*, NIST Special Publication 800-53, Feb. 2005.

[NIST Special Pub 800-60] Barker, William C. *Guide for Mapping Types of Information and Information Systems to Security Categories*, NIST Special Publication 800-60, June 2004.

[Peterson 2006] Pederson, Allan, Navi Partner, and Anders Hedegaard. “Designing a Secure Point-of-Sale System”, *Proceedings of the Fourth IEEE International Workshop on Information Assurance (IWIA ’06).* pp 51-65, April 2006.

[Radack 2005] Radack, Shirley, editor. *Standards for Security Categorization of Federal Information and Information Systems*, Federal Information Processing Standard (FIPS) 199, July 10, 2005.

[Riggs 2003] Riggs, S. *Network Perimeter Security: Building Defense In-Depth*, Auerbach Publications, 2003.

[Saltzer and Schroeder 1975] Saltzer, J. H. and M. D. Schroeder. "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278-1308, 1975. Available online at http://cap-lore.com/CapTheory/ProtInf/

[Schell 2005] Schell, Roger. "Creating High Assurance for a Product: Lessons Learned from GEMSOS." (Keynote Talk) *Third IEEE International Workshop on Information Assurance*, College Park, MD, USA March 23-24, 2005. Available at http://www.iwia.org/2005/Schell2005.pdf

[SHS 2002] Secure Hash Standard (SHS), FIPS 180-2, August 2002.

[Sommerville 2006] Sommerville, I. *Software Engineering*, 8th ed., Pearson Education, 2004.

[US Army 2003] US Army, Field Manual (FM) 3-13: *Information Operations: Doctrine, Tactics, Techniques, and Procedures*, 28th Nov., 2003. (Particularly Chapter 4 on Deception)

[US DoD 1996] Joint Chiefs of Staff, DoD JP 3-58, *Joint Doctrine for Military Deception*, 31 May 1996.

[Zwicky et al, 2000] Zwicky, Elizabeth D., Simon Cooper, and D. Brent Chapman. *Building Internet Firewalls* (2nd ed.), O'Reilly, 2000.

7 Secure Software Construction

7.1 Scope

Secure software construction creates working, meaningful, secure software through design, coding, verification, unit testing, integration testing, and debugging [Abran 2004]. To construct a secure software system, then, more specific security knowledge is required from a software engineer, programmer, or analyst: they must include secure design and implementation features as well as features to avoid known types of weaknesses or pitfalls. It has been said that we have long known how to build secure software; we simply don't act on what we know. This section summarizes the current knowledge and techniques available to produce secure software and briefly describes ongoing efforts to improve the tools and techniques available to those building software. An ongoing effort of note and relevance is one between government, academia, and commercial technologists to develop a comprehensive list and classification schema of common security weaknesses. When accomplished, this effort will clarify and harmonize many aspects of constructing secure code [Martin 2005], [PLOVER 2005], [SAMATE 2005], and [McGraw 2006, Chapter 12].

7.2 Common Vulnerabilities

Some vulnerabilities in software systems occur with such great frequency they have been deemed "common vulnerabilities." However, before these common vulnerabilities can be effectively discussed, it is important to clarify some terminology. The term "vulnerability" is widely overloaded in the context of secure software. It is often used to describe everything from basic weaknesses in software, in threats facing the software, in the exploitable nature of software weaknesses, and more. In this document, a vulnerability is defined as a weakness in software exploitable by an attacker. Software vulnerabilities are caused by the combination of one or more underlying weaknesses with some type of enabler during operation that makes the weaknesses exploitable. To effectively avoid vulnerabilities, one must fully understand the types of weaknesses that enable them. Rather than attempt to list specific vulnerabilities, this section focuses on the general groups of software weaknesses that allow exploitable vulnerabilities to occur in operational software and provides a short overview of how these groupings and their constituent weaknesses can be organized to facilitate understanding them. Depending on the motivation and context of the reader (e.g., viewing these weaknesses from an external attacking view, from an internal development and protection view, or from an operational impact view), various types of organization can be leveraged to provide useful insight and understanding.

Although not addressed below, there are numerous references that cover application, network, and operating system-specific vulnerabilities. Some of these include [Skoudis 2002], [Graff and Wyk 2003], [Howard and LeBlanc 2003], [Koziol. et al.. 2004], [Chirillo 2002], [Flickenger 2003], [Wheeler 2003], [Whittaker and Thompson 2004], [Meier 2003, Part IV].

However, it is also quite important to know where to find specific information on vulnerabilities, exploits, and patches for application software and systems. Some sources of such information are listed in Section 2.7.

The organization and understanding of software-related security weaknesses and the vulnerabilities they create point to ways to improve software and help find better ways to avoid introducing security weaknesses into software. It also aids in identifying security weaknesses in existing code so they can be removed. Several different ways of organizing these issues have been developed over the years. Some of the earlier ones focused on specific operating systems or languages. [Abbott 1976], [Aslam 1995], [Aslam 1996], [Bishop 1995], [Bishop 1996], [Bishop 1999], [Knight 2000], [Krsul 1997], [Lough 2001], [Landwehr 1993], and [Zitser 2005]. Others focused on a specific context such as the nature of the threat, the System Development Life Cycle (SDLC) phase of weakness introduction, the motivation of that introduction, and perhaps most meaningfully, the underlying nature of the weakness itself. [Bazaz 2005], [Firesmith 2005], [Hansman 2003], [Larus 2004], [McGraw 2005], [Piessens 2002], [Seacord 2005], [Weber 2004], [Younan 2003], and [Younan 2004].

While some modern approaches to classification and organization are a great deal more comprehensive and detailed than early efforts, they still tend to be limited by an individual technical focus or by focusing on either the low-end details of the weaknesses or the very high-end weakness groupings. This shortcoming has led to the effort to define a common weakness enumeration and classification to bridge this gap.

The remainder of this section introduces a few of the more common types of security weaknesses and classifies them by the manner in which the security of a system is compromised [Meunier 2004], [Howard 2005], [Whittaker and Thompson 2004], and [Meier 2003, p. 9]. This sampling of common weaknesses illustrates the nature and potential impact of common weaknesses, the wide diversity of contexts in which they may reside, and some measure of understanding of the dimensions along which they can be classified and organized. The aforementioned effort to define a common weakness enumeration and classification should soon bear fruit in the form of a much more comprehensive listing of these sort of groupings and their constituent, specific, and detailed weaknesses – all organized and arranged to show their interrelationships along various dimensions.

Meanwhile, Section 7.7, Appendix A. Taxonomy of Coding Errors, gives a currently published effort..

7.2.1 Buffer Overrun

Buffer overrun, also known as buffer overflow, is arguably one of the most common vulnerabilities in software [Viega and McGraw 2002]. Buffer overrun occurs when a program reads or writes outside the bounds of a storage buffer. If the overrun occurs when reading, a data leak may result or the program may malfunction due to incorrect input. If the overrun occurs during writing, the program may again malfunction or data may be corrupted. Buffer overruns are often exploited to inject attack code into a program or to induce a DOS attack. Buffer overruns can be exploited whether storage is statically allocated (i.e., at compile time) or dynamically allocated (i.e., in the stack or heap). New types of buffer overruns are emerging. It is no longer adequate to simply reference buffer overrun as a weakness. The specific type of buffer overrun should be defined to effectively recognize the weakness in existing code or to prevent its introduction into new code. Although specific exploitation techniques for buffer overruns are many and varied, prevention, detection, and protection mechanisms for the most common types of buffer overruns are well-known and can be found in various references [Foster and Osipov 2005], [Goertzel and Goguen 2005] , [Howard and LeBlanc 2003], [Viega and McGraw 2002], and [Viega and Messier 2003].

There are four basic strategies to eliminate buffer overflow vulnerabilities [Goertzel and Goguen 2005]. The first is simply to identify and avoid the various language-specific vulnerabilities, constructs, and runtime libraries while coding. Validating input to ensure it does not exploit these language-specific vulnerabilities will prevent a significant number of buffer overflow attacks if they are used.

The second strategy is to use non-executable memory locations. This prevents attackers from injecting code via buffer overflow attacks. However, this requires operating system and perhaps hardware support. Furthermore, although this will be effective against direct code injection attacks, even with non-executable code, there are ways to induce execution of code.

The third strategy uses (or modifies) the process runtime environment to perform bounds checking on all data accesses. This will have a significant, and perhaps intolerable, impact on performance.

Example Buffer-Overflow Worms

- **1998: MORRIS WORM** exploited ***buffer overflow vulnerability*** in "finger" program to connect to rsh (remote shell), and a trapdoor that put Sendmail into 'debug' mode, enabling remote execution of commands, including installation of worm code.
- **2001: CODE RED WORM** exploited a flaw in the implementation of Microsoft Internet Information Server (IIS) Version 5's random number generator, then exploited a ***buffer overflow vulnerability*** the IIS idq.dll to gain access to memory (execution stack) to either crash the system (if running on NT) or execute a worm (if running on Windows 2000) that gave the attacker root privileges.
- **2001: RAMEN WORM** exploited the ***buffer overflow and format string vulnerabilities*** in Red Hat Linux 6.2 and 7.0, enabling attackers to execute code under the root user's context.
- **2003: Structured Query Language (SQL) SLAMMER (aka Sapphire) WORM** exploited *multiple buffer overflow vulnerabilities* in Microsoft SQL Server 2000 and SQL Server Desktop Engine (MSDE), enabling the worm to flood port 1434 and cause denial of service or execute arbitrary code loaded via the overflow onto the execution heap.
- **2005: Zotob WORM** exploited a flaw in the Microsoft Plug and Play (PnP) service in Windows 2000 and XP SP1 allowed arbitrary code to be executed via a specially crafted network packet.

Finally, the compiler can perform integrity checks prior to dereferencing a pointer at a significantly lower performance penalty. This, however, does not prevent more sophisticated buffer overflow attacks.

7.2.2 Resource Exhaustion

Resource exhaustion is possible whenever there are finite resources and unrestricted access to those resources. When done with malicious intent, the exhaustion of a resource may be intended either to deny access to the resource itself, to evade traceability or accountability, or to serve as a distraction to increase the probability of succeeding in a separate attack.

Resource exhaustion “enablers” include [Meunier 2004] poor resource management due to design or implementation errors, expensive tasks such as encryption, and coding errors such as memory leaks that become vulnerabilities. Protocols and algorithms that allow anonymous or unauthenticated resource allocation and “amplification” services such as broadcasts and other distribution mechanisms can also be used to consume resources. Asymmetric attacks are a form of amplification. The cost to the attacker to request a service is much less than the cost of the service provider to respond. Thus, the attacker can easily overwhelm a service provider with service requests.

7.2.3 Operating Environment

The operating environment in which a process executes is a source of many vulnerabilities. This environment includes [Meunier 2004] the file system, the user interface, the operating system (OS), user accounts, services provided by the OS or other applications, and environment variables. Many aspects of the environment are under the control of an untrusted user(s) and therefore should be sanitized or validated prior to use. For example, most systems provide the user a capability to specify which directories are to be searched when a request for a file is submitted to the system.

Most systems provide services to create temporary files for use by processes and applications. Some provide world readable and writable directories for these temporary files as well as services to generate unique names for temporary files. Since temporary directories are available to all and the temporary file names provided by a system, though unique, are typically predictable, care must be taken when creating and using temporary files and directories. A secure temporary file has the following properties [Howard and LeBlanc 2003] [Viega and McGraw 2002]: it resides in a directory accessible only to the process that created it, it has a unique, difficult to guess name, and access control are set to prevent unauthorized access.

7.2.4 Race Conditions

When the behavior of a system depends on the order in which critical events occur, a race condition is said to exist. The most prominent example of a race condition in secure programming occurs in the so-called time-of-check, time-of-use (TOCTOU) scenario. For example, suppose a file is created (using default system access permissions) using a system call, and a subsequent call places more restrictive access controls on the file. Between the two calls, a malicious process can change the contents of the file or even replace it with a different file.

Although race conditions are commonly used to exploit file systems, any situation where multiple processes contend for resources are subject to them. Techniques to prevent race conditions usually require OS support. The general strategy to prevent race conditions ensures that access and configuration of the resource appears to be one atomic operation, even when multiple operations and disjointed periods of time are needed to complete the “atomic” operation.

7.2.5 Canonical Form

In the context of computer security, canonical form is the fundamental representation of symbols that can be interpreted without ambiguity. For example, suppose the canonical form of a file on a file system must begin with a drive letter. Then “\my_file.doc” would not be in canonical form; the location of the file is ambiguous. Depending on the number of drives in the system, there may be many files called “\my_file.doc”. Canonicalization is the process of resolving a set of symbols to their standard or canonical form.

Most resources on a system are named so a person using the system can easily remember them. The system itself, however, rarely uses this name to refer to the same resource. Rather, the system interprets the name and translates it to the internal representation the system uses. A fundamental security axiom regarding names is “Do not make any security decision based on the name of a resource… [Howard and LeBlanc 2003]” Vulnerabilities associated with interpreting non-canonical names span every level of the OSI network model, and include making decisions based on IP addresses, short forms of directory names (e.g., “dot-dot” attacks), uninterpreted symbolic file system links, and named network resources and services.

7.2.6 Violations of Trust

Trust can be defined as accepting the risk that an entity, which can harm you, will not do so. Bishop measures trust by using evidence of trustworthiness. “An entity is trustworthy if there is sufficient credible evidence leading one to believe that the system will meet a set of given requirements.” [Bishop 2003]

Common vulnerabilities related to trust span both definitions. Format strings in ‘C’ print statements that require arguments, and a compiler, in a sense, trusts that the programmer has included the correct number; this is trust in the first sense. Trust in the sense of Bishop requires evidence of trustworthiness because of methods used to build a system, or metrics gathered during testing. The authentication of an entity claiming a certain identity is an example of this type of trust.

Cross-site scripting (XSS) and SQL code injection are vulnerabilities that can lead to exploits resulting from the violation of trust.

7.3 Construction of Code

7.3.1 Language Selection

“The single most important technology choice most software projects face is which program language (or set of languages) to use for implementation” [Viega and McGraw 2002]. The term “language” is not just limited to the traditional procedural programming languages like C, C++, and Java, but it also includes query languages, shell scripting languages, and other compiled or interpreted instructions a software system may employ. The security implications of this choice extend to other technologies as well, including the host operating system and distributed system services such as Common Object Request Broker Architecture (CORBA), remote procedure call (RPC), and Java’s remote invocation procedure (RMI) service.

Specific security characteristics of languages and services are readily accessible [Wheeler 2003], [Viega and Messier 2003], [Viega and McGraw 2002]. In general, however, the following language characteristics are desirable: strong typing, safe typing, single entry/exit procedures, passing parameters by value (not by reference), and the restricted use or elimination of pointers.

In addition to Java, which implements its own security model (the Java Virtual Machine “sandbox”), and Perl (which provides the “tainting” option), some less familiar programming languages have emerged that are expressly designed to produce secure code. A noteworthy example is the E language.

Concern must go beyond traditional programming languages. The exploits and browser defenses involving JavaScript are one example. A growing attack opportunity is the rise of Asynchronous JavaScript and XML (AJAX); applications such as AJAX are continually being actively embraced by major companies.

7.3.1.1 Language Subsets, Derivatives, and Variants

These language characteristics and the requirement for analyzability lead to the use of language subsets. While organizations’ coding standards commonly exclude some dangerous language features, concerns for security and high assurance have led to excluding significant portions of programming languages.

Analyzability was the driving force behind defining the SPARK subset of Ada [Barnes 2003] and was a factor in defining the SmartCard subset of Java.

“Safe” derivatives and variants of popular languages such as C, C++, and Java have emerged that include most or all of the expressions and features of those languages, but which add safety- and/or security-enhancing features. Programs written in these derivative languages are not susceptible to some of the key attack patterns (e.g., race conditions, buffer overflows, format string attacks, double free attacks) to which programs in the languages on which the derivatives are based.

Examples of “safe” language derivatives and variants include Cyclone, a variant on C, Guava and SafeJava variants on Java, Microsoft’s C# (originally called SafeC) which incorporates features of C and Java, and Microsoft’s Vault that contains features of both C and C#, and Joyce, a secure language from the late 1970s based on Concurrent Pascal.

7.3.2 Annotations and Add-ons

While good practices for commenting code are well established, security and correctness concerns may be addressed in part by annotations that specify pre- and post-conditions, invariants, including class invariants, and information flow constraints [Barnes 2003] [Leavens 2005]. For C programs, a tool called splint (www.splint.org) uses annotations. These annotations, among other things, document assumptions about functions, variables, types, and other aspects of the code. Splint then statically examines the code to ensure the assumptions in fact hold. [Evans and Larochelle 2002]

Proof-carrying code has been advocated, but it has not come into use.

7.3.3 Using Security Principles in Secure Coding

A number of security principles for software systems have been covered in the prior sections. Using these principles is important during requirements and design continues during secure coding as well. These high-level principles lead to more concrete derivations such as design guidelines (technology-independent software design advice), coding rules (technology-specific instances of common weaknesses as discussed above), and coding practices (proactive coding advice that can be technology-independent or technology-specific in nature). Several authors have discussed security principles while emphasizing coding [Viega and McGraw 2002] [Howard and LeBlanc 2003]. They do not all cover the same set but have readable introductions and discussions of those they do cover. Also see [Bishop 2003].

7.3.4 Coding Standards for Secure Software

An essential element of secure software construction is enforcing particular coding standards. This has several benefits. It ensures programmers use a uniform set of security approaches that can be selected based on the requirements of the project and its suitability rather than on the programmers' familiarity or preference. This, in turn, increases the maintainability of the code. Finally, it reduces the techniques available for malicious developers to subvert the code.

There are numerous references available both online and in print with secure coding guidelines, best practices, suggestions, and case studies [Birman 1996], [Goertzel and Goguen 2005], [Graff and Wyk 2003], [Howard and LeBlanc 2003], [Viega and Messier 2003], [Wheeler 2003]. Many companies have internal secure coding standards. However, there seems to be a lack of public standards as such for secure programming. The community would benefit from internationally recognized secure coding standards for common programming languages. Using such coding standards may greatly reduce the number of false positives produced by vulnerability seeking static analysis tools. See Section 8, Secure Software Verification, Validation, and Evaluation.

7.3.5 Secure Coding Practices

This subsection introduces specific practices that are followed by writers of any type of secure code. Indeed, the practices described below are prudent whether the code is intended to be secure or not; however, they are essential for any code claiming some level of security. Code includes traditional procedural languages such as 'C' and 'Java,' shell scripts, and database queries using languages such as SQL, or web-based programming such as Hyper Text Markup Language (HTML).

7.3.5.1 Input Validation

Input validation is fundamental to any program claiming security. Characterizations of its importance range from the understated "Handle data with caution" [Graff and Wyk 2003] to "All input is evil!" [Howard and LeBlanc 2003]. Identifying sources of input is the critical first step in validating input.

Due to the wide range of input a program may accept, it is difficult to specify validation techniques; however, the principles and techniques below are general and widely applicable.

7.3.5.1.1 Input cleansing

Part of input validation includes bounds and type checking, which may be classified as a kind of input cleansing, but input cleansing scope is much broader. Input cleansing can be simply defined as the process of discarding input that does not conform to what is expected.

While simple to state, the definition presumes a model of expected input is known or has been defined. Based on this input model, a "white list" of acceptable input can be specified, or an algorithm can be devised that recognizes acceptable input. The common practice of specifying a "black list" of unacceptable input is strongly discouraged. Experience has confirmed intuition time and again: that blacklisting requiring the enumeration of all bad inputs for a given program can be impracticable or impossible. Therefore, validation must be approached from the perspective of determining “good” input rather than detecting “bad” input.

In many languages, such as SQL, Hypertext Transfer Protocol (HTTP), or even ‘C’ print functions, it is quite easy to insert instructions into a program for subsequent execution via input mechanisms. This is the so-called “code injection” exploit. The fundamental problem is that code and data are occupying the same input channel, and the program accepting the input must determine which is which. If code (data) can be crafted to appear as data (code), normal security controls can be bypassed and the system exploited. The remedy for this vulnerability is to separate the code input channel from the data input channel in some enforceable manner.

Input should not be declared valid until its semantic meaning (i.e., its meaning within the context of the program) has been determined. Accomplishing this validation will involve completely resolving the various meta-characters, encodings, and any link indirections of the input into its most fundamental representation (canonical form). Only at this point can the semantic meaning and proper security checks be performed.

The following has been proposed as an order for proper input validation [Meunier 2004]:

1. Interpret and render all character encodings to a common form
2. Cleanse the input using several passes if necessary
3. Validate type, range, and format of input
4. Validate the semantics of the input.

7.3.5.2 Preventing Buffer Overflow

There are four basic strategies to eliminate buffer overflow vulnerabilities [Goertzel and Goguen 2005]. The first is simply to identify and avoid the various language-specific vulnerabilities, constructs, and runtime libraries while coding. Validating input to ensure it does not exploit these language-specific vulnerabilities will prevent a significant number of buffer overflow attacks if they are used.

The second strategy is to use non-executable memory locations. This prevents attackers from injecting code via buffer overflow attacks. However, this requires OS and perhaps hardware support. Furthermore, although this strategy will be effective against direct code injection attacks, even with non-executable code, there are ways to induce code execution.

The third strategy uses (or modifies) the process runtime environment to perform bounds checking on all data accesses. This approach will have a significant, and perhaps intolerable, impact on performance.

Finally, the compiler can perform integrity checks before dereferencing a pointer at a significantly lower performance penalty. This, however, does not prevent more sophisticated buffer overflow attacks.

7.3.6 Sound Practices

Even more than knowing all the bad things that could happen, coders need to know what to do to avoid creating code that would lead to them. The followingare sound practices, several of which summarize information provided in earlier subsections, that are intended to help implementers of software achieve their security objectives. Further information on these and other sound practices appears in [Howard and LeBlanc 2003], [Goertzel 2006], [Wheeler 2003], and elsewhere.

- Coding Practices
  - Make security a criterion when selecting programming languages, subsets, and annotations to be used: this provides a better meduim for construction

  - Minimize code size and complexity, and increase traceability: this will make the code easy to analyze.
  - Code with reuse and sustainability in mind: this will make code easy to understand by others.
  - Use a consistent coding style throughout the system: this is the objective of the coding standards described in subsection 7.2.4.
  - Use consistent naming and correct encapsulation: this reduces chances for misunderstanding and eases reviews
  - Implement error and exception handling safely: this will avoid common difficulties and ease analysis and review
  - Use programming languages securely: avoid "dangerous" constructs including using compiler checks and static analysis tools to verify correct language usage and flag "dangerous" constructs, and properly leverage security features such as "taint" mode in Perl and "sandboxing" in Java.
  - Always assume that the seemingly impossible is possible: the history of increased sophistication, technical capability, and motivation of attackers shows that events, attacks, and faults that seem extremely unlikely when the software is written can become quite likely after it has been in operation for a while. Error and exception handling should be programmed explicitly to ensure required conditions for correct behavior and security exist despite as many "impossible" events as the programmer can imagine.
  - Program defensively: Use techniques such as information hiding, input validation, output verification, and anomaly awareness
  - Avoid common, well-known logic errors: use input validation, code review to ensure conformance to specification, absence of "dangerous" constructs and characters, type checking and static checking, and finally comprehensive security testing.
  - Ensure asynchronous consistency: this will avoid timing and sequence errors, race conditions, deadlocks, order dependencies, syncronization errors, etc.
  - Use multitasking and multithreading safely.
  - Implement error and exception handling safely: a failure in any component of the software should never be allowed to leave the software, its volatile data, or its resources vulnerable to attack.
  - Program defensively: Use techniques such as information hiding and anomaly awareness.

- Assembly and Integration:
  - Make security a criterion when selecting components for reuse or acquisition: before a component is selected, it should undergo the same security analyses and testing techniques that will be used on the final software system. For open source and reusable code, these can include code review. For binary components, including COTS, these will necessarily be limited to "black box" tests, as described in Section 8.4, Testing, except in the rare cases where the binary software will be used for such a critical function that reverse engineering to enable code review may be justified.
  - Analyze multiple assembly options: the combination and sequencing of components that are selected should produce a composed system that results in the lowest residual risk, because it presents the smallest attack surface. Ideally, it will require the fewest add-on countermeasures such as wrappers.

- All Development:
  - Verify the secure interaction of the software with its execution environment: this includes never trusting parameters passed by the environment, separation of data and program control, always presuming client/user hostility (thus always validating all input from the client/user), never allowing the program to spawn a system shell.

  - In addition to these sound implementation practices, there are a number of systems engineering techniques that can be used to minimize the attack surface and increase the security robustness of software in deployment. See Section 6, Secure Software Design.

7.4 Construction of User Aids

Security needs to be included in user aids such as manuals and help facilities. If user-visible security is significant or has changed considerably, users will benefit from including motivations and explicit "user mental model." Some types of user aids, including security-oriented material, include:

- Documentation
  - Normal documentation includes security aspects and use
  - Operational Security Guide [Viega 2005, p. 33, 98-00]
- Training
- Online user aids, e.g., pop-up notes and help facility
- User support
  - Help desk
  - Online support

7.5 Secure Release

In addition to sound configuration and release management, achieving secure releases requires secure configuration management facilities and cryptographic signing of artifacts or other means to ensure their proper origin and their end-to-end integrity when distributed internally or externally. End-to-end integrity can be achieved by cryptographic signing that starts with individuals producing product components and extends through deployment of the combined product, as well as possibly further through integration elsewhere with other products before final delivery to customer or user. [Berg 2005, Section 10.1] addresses designing for secure deployment.

A number of technical techniques have been tried to protect distributed software as intellectual property. Software protection is also a national security concern. These issues were covered above in Design section under anti-tampering.

7.6 Conclusion

Sound practice are intended to help implementers of software achieve their security objectives. See [Goertzel 2006] for more discussion...For software to be secure it must avoid defects in its implementation that introduce vulnerabilities regardless of whether the majority of development involves coding or assembly of acquired or reused software components. Writing secure code means not only writing correct code that meets its specifications and required security property constraints, but avoiding coding practices producing code weaknesses that could manifest as vulnerabilities and producing code that will simplify the detection and correction of.not only faults but such weaknesses.

7.7 Appendix A. Taxonomy of Coding Errors

Generalizing information about security flaws allows the developer to look at the underlying principles that result in exploitable security vulnerabilities. By organizing common types of security errors into a taxonomy, software architects and developers can more easily understand and recognize the sorts of problems that lead to vulnerabilities.

Presented below is a simple taxonomy developed by [McGraw 2006, Chapter 12] and [Tsipenyuk 2005] that organizes sets of security rules that can be used to help software developers understand the kinds of errors that have an impact on security.

By better understanding how systems fail, developers may be better equipped to analyze the software they create, more readily identify and address security problems when they see them, and ideally avoid repeating the same mistakes in the future.

- **Input Validation and Representation.** Security problems result from trusting input [McGraw 2006]. The problem of improper validation arises when data is not chekced for consistency and correctness [Bishop 2003]. Input validation and representation problems are caused by metacharacters, alternate encodings and numeric representations. The issues include:
  - **Buffer Overflows**. Buffer overflows are the principal method used to exploit software by remotely injecting malicious code into a target. The root cause of buffer overflow problems is that C and C++ are inherently unsafe. There are no bounds checks on array and pointer references, meaning a developer has to check the bounds (an activity that is often ignored) or risk encountering problems. Reading or writing past the end of a buffer can cause a number of diverse (and often unanticipated) behaviors: (1) programs can act in strange ways, (2) programs can fail completely, and (3) programs can proceed without any noticeable difference in execution. The most common form of buffer overflow, called the stack overflow, can be easily prevented. Stack-smashing attacks target a specific called the stack overflow, can be easily prevented. Stack-smashing attacks target a specific programming fault: the careless use of data buffers allocated on the program's runtime stack. An attacker can take advantage of a buffer overflow vulnerability by stack-smashing and running arbitrary code, such as code that invokes a shell in such a way that control gets passed to the attack code. More esoteric forms of memory corruption, including the heap overflow, are harder to avoid. By and large, memory usage vulnerabilities will continue to be a fruitful resource for exploiting software until modern languages that incorporate modern memory management schemes are in wider use.
  - **SQL Injection**. SQL injection is a technique used by attackers to take advantage of non-validated input vulnerabilities to pass SQL commands through a Web application for execution by a backend database. Attackers take advantage of the fact that programmers often chain together SQL commands with user-provided parameters, and the attackers, therefore, can embed SQL commands inside these parameters. The result is that the attacker can execute arbitrary SQL queries and/or commands on the backend database server through the Web application. Typically, Web applications use string queries, where the string contains both the query itself and its parameters. The string is built using server-side script languages such as ASP or JSP and is then sent to the database server as a single SQL statement.
  - **Cross-Site Scripting**. A CSS vulnerability is caused by the failure of a site to validate user input before returning it to the client’s web-browser. The essence of cross-site scripting is that an intruder causes a legitimate web server to send a page to a victim's browser that contains malicious script or HTML of the intruder's choosing. The malicious script runs with the privileges of a legitimate script originating from the legitimate web server.
  - **Integer Overflows**. Not accounting for integer overflow can result in logic errors or buffer overflow. Integer overflow errors occur when a program fails to account for the fact that an arithmetic operation can result in a quantity either greater than a data type's maximum value or less than its minimum value. These errors often cause problems in memory allocation functions, where user input intersects with an implicit conversion between signed and unsigned values. If an attacker can cause the program to under-allocate memory or interpret a signed value as an unsigned value in a memory operation, the program may be vulnerable to a buffer overflow.

- **Command Injection**. Executing commands that include unvalidated user input can cause an application to act on behalf of an attacker. Command injection vulnerabilities take two forms: (1) An attacker can change the command that the program executes: the attacker explicitly controls what the command is, and (2) An attacker can change the environment in which the command executes: the attacker implicitly controls what the command means

- **API Abuse**. An API is a contract between a caller and a callee. The most common forms of API abuse are caused by the caller failing to honor its end of this contract [McGraw 2006]. Examples of API abuse are described below:
  - **Call to Thread.run()**. The program calls a thread's run() method instead of calling start(). In most cases a direct call to a Thread object's run() method is a bug. The programmer intended to begin a new thread of control, but accidentally called run() instead of start(), so the run() method will execute in the caller's thread of control.
  - **Call to a Dangerous Function**. Certain functions behave in dangerous ways regardless of how they are used. Functions in this category were often implemented without taking security concerns into account. For example in C the gets() function is unsafe because it does not perform bounds checking on the size of its input. An attacker can easily send arbitrarily-sized input to gets() and overflow the destination buffer.
  - **Directory Restriction**. The chroot() system call allows a process to change its perception of the root directory of the file system. After properly invoking chroot(), a process cannot access any files outside the directory tree defined by the new root directory. Such an environment is called a chroot jail and is commonly used to prevent the possibility that a processes could be subverted and used to access unauthorized files. Improper use of chroot() may allow attackers to escape from the chroot jail.
  - **Use of java.io**. The Enterprise JavaBeans specification requires that every bean provider follow a set of programming guidelines designed to ensure that the bean will be portable and behave consistently in any EJB container. The program violates the Enterprise JavaBeans specification by using the java.io package to attempt to access files and directories in the file system.
  - **Use of Sockets**. The Enterprise JavaBeans specification requires that every bean provider follow a set of programming guidelines designed to ensure that the bean will be portable and behave consistently in any EJB container. The program violates the Enterprise JavaBeans specification by using sockets. An enterprise bean must not attempt to listen on a socket, accept connections on a socket, or use a socket for multicast.
  - **Authentication**. Security should not rely on DNS names, because attackers can spoof DNS entries. If an attacker are can make a DNS update (DNS cache poisoning), they can route network traffic through their machines or make it appear as if their IP addresses are part of your domain. If an attacker can poison the DNS cache, they can gain trusted status.
  - **Exception Handling.** The _alloca function allocates dynamic memory on the stack. The allocated space is freed automatically when the calling function exits, not when the allocation merely passes out of scope. The _alloca() function can throw a stack overflow exception, potentially causing the program to crash. If an allocation request is too large for the available stack space, _alloca() throws an exception. If the exception is not caught, the program will crash, potentially enabling a denial of service attack.
  - **Privilege Management**. Failure to adhere to the principle of least privilege amplifies the risk posed by other vulnerabilities. Programs that run with root privileges have caused innumerable Unix security disasters.
  - **Strings**. Functions that convert between Multibyte and Unicode strings often result in buffer overflows. Windows provides the MultiByteToWideChar(), WideCharToMultiByte(), UnicodeToBytes, and BytesToUnicode functions to convert between arbitrary multibyte (usually ANSI) character strings and Unicode (wide character) strings. The size arguments to these functions are specified in different units – one in bytes, the other in characters – making their use

prone to error. In a multibyte character string, each character occupies a varying number of bytes, and therefore the size of such strings is most easily specified as a total number of bytes. In Unicode, however, characters are always a fixed size, and string lengths are typically given by the number of characters they contain. Mistakenly specifying the wrong units in a size argument can lead to a buffer overflow.

  - **Unchecked Return Value**. Ignoring a method's return value can cause the program to overlook unexpected states and conditions. Two dubious assumptions that are easy to spot in code are "this function call can never fail" and "it doesn't matter if this function call fails". When a programmer ignores the return value from a function, they implicitly state that they are operating under one of these assumptions.

- **Security Features.** Software security features such as authentication, access control, confidentiality, cryptography, and privilege management play an important role in software security [McGraw 2006]. Discussed below are issues that lead to exploitable vulnerabilities:
  - **Least Privilege Violation**. The elevated privilege level required to perform operations such as chroot() should be dropped immediately after the operation is performed. When a program calls a privileged function, such as chroot(), it must first acquire root privilege. As soon as the privileged operation has completed, the program should drop root privilege and return to the privilege level of the invoking user. If this does not occur, a successful exploit can be carried out by an attacker against the application, resulting in a privilege escalation attack because any malicious operations will be performed with the privileges of the superuser. If the application drops to the privilege level of a non-root user, the potential for damage is substantially reduced.
  - **Hardcoded Password**. Hardcoded passwords may compromise system security in a way that cannot be easily remedied. Once the code is in production, the password cannot be changed without patching the software. If the account protected by the password is compromised, the owners of the system will be forced to choose between security and availability.
  - **Weak Cryptography**. Obscuring a password with a trivial encoding, such as base 64 encoding, but this effort does not adequately protect the password.
  - **Insecure Randomness**. Standard pseudo-random number generators cannot withstand cryptographic attacks. Insecure randomness errors occur when a function that can produce predictable values is used as a source of randomness in security-sensitive context. Pseudo-Random Number Generators (PRNGs) approximate randomness algorithmically, starting with a seed from which subsequent values are calculated. There are two types of PRNGs: statistical and cryptographic. Statistical PRNGs provide useful statistical properties, but their output is highly predictable and forms an easy to reproduce numeric stream that is unsuitable for use in cases where security depends on generated values being unpredictable. Cryptographic PRNGs address this problem by generating output that is more difficult to predict. For a value to be cryptographically secure, it must be impossible or highly improbable for an attacker to distinguish between it and a truly random value.

- **Time and State.** Distributed computation is about time and state. That is, in order for more than one component to communicate, state must be shared, and that takes time. In multi-core, multi-CPU, or distributed systems, two events may take place at exactly the same time. Defects related to unexpected interactions between threads, processes, time, and information happen through shared state: semaphores, variables, the file system, and, anything that can store information [McGraw 2006]. Time and State issues are discussed below:
  - **Race Conditions**. The window of time between when a file property is checked and when the file is used can be exploited to launch a privilege escalation attack. File access race conditions, known as time-of-check, time-of-use (TOCTOU) race conditions, occur when: (1) the program checks a property of a file, referencing the file by name, and (2) the program later performs a filesystem operation using the same filename and assumes that the previously-checked property still holds. The window of vulnerability for such an attack is the period of time between when the property is tested and when the file is used. Even if the use immediately follows the check,

modern operating systems offer no guarantee about the amount of code that will be executed before the process yields the CPU. Attackers have a variety of techniques for expanding the length of the window of opportunity in order to make exploits easier, but even with a small window, an exploit attempt can simply be repeated over and over until it is successful.

- **Insecure Temporary Files**. Creating and using insecure temporary files can leave application and system data vulnerable to attacks. The most egregious security problems related to temporary file creation have occurred on Unix-based operating systems, but Windows applications have parallel risks. The C Library and WinAPI functions designed to aid in the creation of temporary files can be broken into two groups based whether they simply provide a filename or actually open a new file. IN the former case, the functions guarantee that the filename is unique at the time it is selected, there is no mechanism to prevent another process or an attacker from creating a file with the same name after it is selected but before the application attempts to open the file. Beyond the risk of a legitimate collision caused by another call to the same function, there is a high probability that an attacker will be able to create a malicious collision because the filenames generated by these functions are not sufficiently randomized to make them difficult to guess. In the latter case, if a file with the selected name is created, then depending on how the file is opened the existing contents or access permissions of the file may remain intact. If the existing contents of the file are malicious in nature, an attacker may be able to inject dangerous data into the application when it reads data back from the temporary file. If an attacker pre-creates the file with relaxed access permissions, then data stored in the temporary file by the application may be accessed, modified or corrupted by an attacker. On Unix based systems an even more insidious attack is possible if the attacker pre-creates the file as a link to another important file. Then, if the application truncates or writes data to the file, it may unwittingly perform damaging operations for the attacker. This is an especially serious threat if the program operates with elevated permissions.
- **Session Fixation**. Authenticating a user without invalidating any existing session identifier gives an attacker the opportunity to steal authenticated sessions. Session fixation vulnerabilities occur when: (1) a web application authenticates a user without first invalidating the existing session, thereby continuing to use the session already associated with the user, and (2) an attacker is able to force a known session identifier on a user so that, once the user authenticates, the attacker has access to the authenticated session.

▪ **Errors**. Errors and error handling represent a class of API. Errors related to error handling are so common that they deserve a special classification of their own (McGraw 2006). As with API Abuse, there are two ways to introduce an error-related security vulnerability: the most common one is handling errors poorly (or not at all). The second is producing errors that either give out too much information (to possible attackers) or are difficult to handle. Errors are discussed below:

- **Empty Catch Block**. Ignoring an exception can cause the program to overlook unexpected states and conditions. Two dubious assumptions that are easy to spot in code are “this method call can never fail” and “it doesn't matter if this call fails”. When a programmer ignores an exception, they implicitly state that they are operating under one of these assumptions.
- **Catching NullPointerExceptions**. It is generally a bad practice to catch NullPointerException. Programmers typically catch NullPointerException under three circumstances: (1) the program contains a null pointer dereference. Catching the resulting exception was easier than fixing the underlying problem, (2) the program explicitly throws a NullPointerException to signal an error condition, and (3) the code is part of a test harness that supplies unexpected input to the classes under test. Of these three circumstances, only the last is acceptable.
- **Overly Broad Catch**. The catch block handles a broad swath of exceptions, potentially trapping dissimilar issues or problems that should not be dealt with at this point in the program. Multiple catch blocks can get ugly and repetitive, but “condensing” catch blocks by catching a high-level class like Exception can obscure exceptions that deserve special treatment or that should not be caught at this point in the program. Catching an overly broad exception essentially defeats the purpose of Java's typed exceptions, and can become particularly dangerous if the program grows

and begins to throw new types of exceptions. The new exception types will not receive any attention.

  – **Overly Broad Throws**. The method throws a generic exception making it harder for callers to do a good job of error handling and recovery. Declaring a method to throw Exception or Throwable makes it difficult for callers to do good error handling and error recovery. Java's exception mechanism is set up to make it easy for callers to anticipate what can go wrong and write code to handle each specific exceptional circumstance. Declaring that a method throws a generic form of exception defeats this system.
  – **Return Inside Finally**. Returning from inside a finally block will cause exceptions to be lost. A return statement inside a finally block will cause any exception that might be thrown in the try block to be discarded.

- **Code Quality**. Poor code quality leads to unpredictable behavior [McGraw 2006]. From a user's perspective that often manifests itself as poor usability. For an attacker it provides an opportunity to stress the system in unexpected ways. Code quality issues are discussed below:
  – **Expression is Always False**. An expression will always evaluate to false.
  – **Expression is Always True**. An expression will always evaluate to true.
  – **Memory Leak**. Memory is allocated but never freed. Memory leaks have two common and sometimes overlapping causes: (1) error conditions and other exceptional circumstances and, (2) confusion over which part of the program is responsible for freeing the memory. Most memory leaks result in general software reliability problems, but if an attacker can intentionally trigger a memory leak, the attacker might be able to launch a denial of service attack (by crashing the program) or take advantage of other unexpected program behavior resulting from a low memory condition.
  – **Null Dereference**. The program can potentially dereference a null pointer, thereby raising a NullPointerException. Null pointer errors are usually the result of one or more programmer assumptions being violated. Most null pointer issues result in general software reliability problems, but if an attacker can intentionally trigger a null pointer dereference, the attacker might be able to use the resulting exception to bypass security logic or to cause the application to reveal debugging information that will be valuable in planning subsequent attacks.
  – **Uninitialized Variable**. The program can potentially use a variable before it has been initialized. Stack variables in C and C++ are not initialized by default. Their initial values are determined by whatever happens to be in their location on the stack at the time the function is invoked. Programs should never use the value of an uninitialized variable.
  – **Unreleased Resource**. The program can potentially fail to release a system resource. Most unreleased resource issues result in general software reliability problems, but if an attacker can intentionally trigger a resource leak, the attacker might be able to launch a denial of service attack by depleting the resource pool. Resource leaks have at least two common causes: (1) error conditions and other exceptional circumstances and (2) confusion over which part of the program is responsible for releasing the resource.
  – **Use After Free**. Referencing memory after it has been freed can cause a program to crash. Use after free errors occur when a program continues to use a pointer after it has been freed. Like double free errors and memory leaks, use after free errors have two common and sometimes overlapping causes: (1) error conditions and other exceptional circumstances, and (2) confusion over which part of the program is responsible for freeing the memory. Use after free errors sometimes have no effect and other times cause a program to crash.
  – **Double Free**. Double free errors occur when free() is called more than once with the same memory address as an argument. Calling free() twice on the same value can lead to a buffer overflow. When a program calls free() twice with the same argument, the program's memory management data structures become corrupted. This corruption can cause the program to crash or, in some circumstances, cause two later calls to malloc() to return the same pointer. If malloc()

returns the same value twice and the program later gives the attacker control over the data that is written into this doubly-allocated memory, the program becomes vulnerable to a buffer overflow attack.

- **Encapsulation**. Encapsulation is about drawing strong boundaries [McGraw 2006]. In a web browser that might mean ensuring that your mobile code cannot be abused by other mobile code. On the server it might mean differentiation between validated data and unvalidated data, between one user's data and another's, or between data users are allowed to see and data that they are not. Encapsulation issues ar discussed below:
  - **Leftover Debug Code**. Debug code can create unintended entry points in a deployed web application. A common development practice is to add “back door” code specifically designed for debugging or testing purposes that is not intended to be shipped or deployed with the application. When this sort of debug code is accidentally left in the application, the application is open to unintended modes of interaction. These back door entry points create security risks because they are not considered during design or testing and fall outside of the expected operating conditions of the application. The most common example of forgotten debug code is a main() method appearing in a web application. Although this is an acceptable practice during product development, classes that are part of a production J2EE application should not define a main().
  - **Trust Boundary Violation**. A trust boundary can be thought of as line drawn through a program. On one side of the line, data is untrusted. On the other side of the line, data is assumed to be trustworthy. The purpose of validation logic is to allow data to safely cross the trust boundary--to move from untrusted to trusted. A trust boundary violation occurs when a program blurs the line between what is trusted and what is untrusted. The most common way to make this mistake is to allow trusted and untrusted data to commingle in the same data structure.
  - **Unsafe Mobile Code: Access Violation**. The program violates secure coding principles for mobile code by returning a private array variable from a public access method. Returning a private array variable from a public access method allows the calling code to modify the contents of the array, effectively giving the array public access and contradicting the intentions of the programmer who made it private.
  - **Unsafe Mobile Code: Inner Class**. The program violates secure coding principles for mobile code by making use of an inner class. Inner classes quietly introduce several security concerns because of the way they are translated into Java bytecode. In Java source code, it appears that an inner class can be declared to be accessible only by the enclosing class, but Java bytecode has no concept of an inner class, so the compiler must transform an inner class declaration into a peer class with package level access to the original outer class. More insidiously, since an inner class can access private fields in their enclosing class, once an inner class becomes a peer class in bytecode, the compiler converts private fields accessed by the inner class into protected fields.
  - **Unsafe Mobile Code: Public finalize() Method**. The program violates secure coding principles for mobile code by declaring a finalize()method public. A program should never call finalize explicitly, except to call super.finalize() inside an implementation of finialize(). In mobile code situations, the otherwise error prone practice of manual garbage collection can become a security threat if an attacker can maliciously invoke one of your finalize() methods because it is declared with public access. If you are using finalize() as it was designed, there is no reason to declare finalize() with anything other than protected access.
  - **Unsafe Mobile Code: Dangerous Array Declaration**. The program violates secure coding principles for mobile code by declaring an array public, final and static. In most cases an array declared public, final and static is a bug. Because arrays are mutable objects, the final constraint requires that the array object itself be assigned only once, but makes no guarantees about the values of the array elements. Since the array is public, a malicious program can change the values stored in the array. In most situations the array should be made private.
  - **Unsafe Mobile Code: Dangerous Public Field**. The program violates secure coding principles for mobile code by declaring a member variable public but not final. All public member variables

in an Applet and in classes used by an Applet should be declared final to prevent an attacker from manipulating or gaining unauthorized access to the internal state of the Applet.

7.8 Further Reading

[SWEBOK] Abran, Alain, James W. Moore (Executive editors); Pierre Bourque, Robert Dupuis, Leonard Tripp (Editors). *Guide to the Software Engineering Body of Knowledge*. 2004 Edition. Los Alamitos, California: IEEE Computer Society, Feb. 16, 2004. Available at http://www.swebok.org

[Avizienis 2004] Avizienis, Algirdas, Jean-Claude Laprie, Brian Randell, and Carl Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11-33, Jan.-Mar. 2004. Available at http://csdl.computer.org/dl/trans/tq/2004/01/q0011.pdf

[Bishop 2003] Bishop, Matt. *Computer Security: Art and Practice*. Boston, MA: Addison-Wesley Professional, 2003.

[Cohen 2004] Cohen, Lazaro Issi, and Joseph Issi Cohen, *The Web Programmer's Desk Reference*, No Starch Press, 2004.

[Gasser 1988] Gasser, M. *Building a Secure Computer System.* Van Nostrand Reinhold, 1988. Available at http://nucia.ist.unomaha.edu/library/gasser.php

[Ibrahim et al, 2004] Ibrahim, Linda, et al, *Safety and Security Extensions for Integrated Capability Maturity Models*. Washington D.C.: United States Federal Aviation Administration, Sept. 2004. Available at http://www.faa.gov/ipg/pif/evol/index.cfm

[McGraw 2006] McGraw, Gary. Software Security: Building Security In. Addison Wesley, 2006.

[Redwine 2004] Redwine, Samuel T., Jr., and Noopur Davis (Editors). *Processes for Producing Secure Software: Towards Secure Software*. vols. I and II. Washington, D.C.: National Cyber Security Partnership, 2004.
Available at http://www.cigital.com/papers/download/secure_software_process.pdf

[Sommerville 2004] Sommerville, I., *Software Engineering*, 7th ed., Pearson Education, 2004.

[Thompson 2005] Thompson, Herbert and Scott Chase. The Software Vulnerability Guide. Charles River Media, 2005.

[Hoglund 2004] Hoglund, Greg and Gary McGraw. Exploiting Software: How to Break Code. Boston, MA: Addison-Wesley Professional, 2004.

[Whittaker and Thompson 2003] Whittaker, James and Herbert Thompson. How to Break Software Security. Boston MA: Addison-Wesley Professional, 2003.

Cerven, Pavol. Crackproof Your Software: Protect Your Software Against Crackers, First Edition. No Starch Press, 2002.

Kaspersky, Kris. Hacker Disassembling Uncovered. A-List Publishing, 2003.

[Howard et al 2005] Howard, Michael, John Viega, and David LeBlanc. 19 Deadly Sins of Software Security. McGraw-Hill Osborne Media, 2005.

Viega, John and Gary McGraw: Building Secure Software: How to Avoid Security Problems the Right Way. Boston, MA: Addison-Wesley Professional, 2001.

Howard, Michael and David LeBlanc. *Writing Secure Code, Second Edition.* Redmond, WA: Microsoft Press, 2002.

Van Wyk, Kenneth R. and Mark G. Graff: *Secure Coding: Principles and Practices. Sebastopol, CA:* O'Reilly Media Inc., 2003.

[Tsipenyuk 2005] Tsipenyuk, Katrina, Brian Chess, and Gary McGraw. Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors. IEEE Security & Privacy. Vol. 4, No.2, pp 81-84, November/December 2005.

[Goertzel 2006] Goertzel, Karen Mercedes, et al. Security in the Software Lifecycle: Making Application Development Processes—and Software Produced by Them—More Secure, Appendix G, Section G.3, "Implementing Secure Software". Washington, DC: Dept. of Homeland Security, 2006.

Viega, John and Matt Messier. Secure Programming Cookbook for C and C++. Sebastopol, CA: O'Reilly Media Inc., 2003.

Seacord, Robert: Secure Coding in C and C++. Boston MA: Addison-Wesley Professional/SEI Series in Software Engineering, 2005.

University of California-Berkeley: Proof Carrying Code. Available at http://raw.cs.berkeley.edu/pcc.html.

Princeton University Secure Internet Programming: Proof-Carrying Code. Available at http://www.cs.princeton.edu/sip/projects/pcc/.

Carnegie Mellon University Fox Project: Proof-Carrying Code. Available at http://www.cs.cmu.edu/~fox/pcc.html.

8 Secure Software Verification, Validation, and Evaluation

8.1 Scope

The methods used for verification, validation, and evaluation of original development, reused and OTS software, and changes can be static or dynamic. The primary dynamic techniques are simulation and testing. [Abran 2004] states, "Testing is an activity performed for evaluating product quality, and for improving it, by identifying defects and problems. Software testing consists of the dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the expected behavior."

Static techniques may involve direct reviews, automated static analysis, and proofing or model checking. The term "formal" is used here and elsewhere to mean mathematically based. See Section 9, Tools and Methods, for a number of relevant techniques and the tools.

While the primary initial purpose for using a technique or method may be product improvement, all the techniques used should contribute to the assurance case for the software system. Those improving the development process and the developers' organizations, such as process audits and improvements, contribute indirectly.

This section first addresses assurance case concerns and then addresses a number of issues and techniques related to secure software engineering verification, validation, and evaluation.

8.2 Assurance Case

As mentioned in the Fundamentals section, the UK MoD Defence Standard 00-42 Part 3 defines a type of assurance case that they call a Reliability and Maintainability (R&M) case. Their definition of R&M Case also serves as a good definition for a security assurance case: "A reasoned, auditable argument created to support the contention that a defined system will satisfy [its security property]…requirements".

These are the security requirements, but a combined assurance case could be made for security and safety or other properties. Where they have significant impacts on the required security properties justification exists for the security assurance case including not only confidentiality, integrity, availability, etc. requirements but also reliability, and security-related usability and sustainability. Separate assurance cases may be more straightforward. [Despotou 2004]

Starting with the initial statement of requirement, the assurance case subsequently includes identified risks (perceived and actual), avoidance and mitigation strategies, and an assurance argument that refers to associated and supporting evidence. It is best developed concurrently with the software. This may include evidence and data from requirements, design, construction, verification, validation, and evaluation; management, and other sources. Evidence concerning quality results from reviews, analyses, tests, trials, in-service and field experience, process fidelity, standards conformance results, personnel qualification records, and elsewhere. The assurance case also records any changes to the case, including changes required by changes in the software system and its threat and operational environments and its assumptions.

As a reasoned, auditable argument created to support the contention that a defined software system satisfies the relevant requirements, the assurance case provides an audit trail of the engineering considerations, from requirements to full evidence of compliance. It provides the justification of why certain activities have been undertaken and how they were judged successful. It is initiated at the concept stage, progressively revised during a system life cycle to be up-to-date and available, and with its status continually tracked. It is typically summarized in Assurance Case Reports at predefined intervals or milestones. It remains with the software system throughout its life through disposal. The assurance case has, therefore, a progressively expanding body of evidence during development and responds as required to all changes during development and sustainment. [Ministry of Defence 2003b, p. 5]

The assurance argument provides a structured set of claims (or goals to assure) that progressively justify through argument – show, demonstrate, perform causal analysis, prove, quantify – their super-ordinate claim or goal. An element of the case may contain details of the initial (justified) requirement and reasons for the proposed solution.

Each goal or claim may be achieved either directly or by goal breakdown. In general, all of the things needing to be achieved by a goal are carried down and allocated to one or more of its subgoals, and hence once the decision has been made to use goal breakdown, the higher-level goal does not need to be referred to again. For this reason, many of the requirements stated in one goal are repeated in its subgoals, sometimes verbatim, sometimes slightly enhanced.

To achieve a goal directly, a number of things must be argued. The SafSec Standard divides the ways of arguing or showing into seven non-exclusive ways and calls these “frameworks.” Most important, compliance with each framework means meeting certain standards, such as the organizational roles being defined, including standard ones. In addition, they are one categorization of the rationales for use in arguments, although for a single claim or goal many of them must usually be included in its arguments and evidence. [SafSec Standard, pp 24-28] The arguments are:

1. Organizational: the goal is achieved by some organization.
2. Procedural: certain actions have been carried out.
3. Unified Risk Management: this process justifies the acceptability of residual risk against the agreed and documented top-level security or broader dependability objectives – this point is particularly relevant to assurance arguments.
4. Risk Directed Design: “document a justification for achievement, by the system, of each residual risk; and document a justification that the evidence of achievement of risk reduction is appropriate for the level and importance of the risk reduction.”
5. Modular Certification and Technical Modularity: organizational or system interfaces, particularly with external systems, need the “other” side of the interface to justifiably have the assured qualities claimed. In addition, “Module boundaries shall match the organizational boundaries.”
6. Evidence: requirements have been established for the recording, handling, and characteristics of evidence to be used.
7. Evaluation/Assessment: the project shall document a means of demonstrating the achievement, by the system, of each residual risk to a level of [uncertainty] appropriate for that risk, obtain agreements on evaluations and assessments among the parties involved, and carry them out successfully (as determined by evaluation/assessment results).

“In order to demonstrate that a system or module meets its dependability objectives, evidence for the dependability argument is prepared and presented. Evidence shall be permanent, traceable and managed in such a way as to provide later readers confidence in its source, contents and validity.” [SafSec Standard, page 9]

When judging the credence to be given to a piece of evidence, its relevance, visibility, traceability, and quality are crucial factors. Therefore, one must necessarily confirm that the evidence is generated or collected, managed, validated, and used within the constraints of acceptable practices and controls. It must also achieve the objectives claimed in the assurance argument. [MoD DefStan 00-42 Part 3, section 9.1] The body of evidence can become quite large, and for usability probably needs to be organized by some evidence framework. The structure of the assurance can make it easier or harder to create, understand, and modify. [Kelly 2003] Automated tools exist to aid in recording, maintaining, and managing assurance cases- see Section 9, Tools and Methods.

As the principal means by which the "justifiably high confidence" can be explicitly obtained, the assurance case is central to the verification, validation, and evaluation of secure software. It should not only give coherent confidence to developers, sustainers, and acquirers, but also be directly usable by others including certifiers and accreditors.

Activities involved might include

- Create top-level assurance goal from requirements
- Establish structure of argument with subordinate goals or claims including their relationships
- Create portions of assurance argument tailored for level of assurance desired
- Compile portions of argument and supporting evidence
- Verify
- Validate
- Analyze
- Establish level of confidence
- Use as input to certification

Several steps are mentioned in Sections 3 and 10 that contribute to the assurance case include.

- Perform threat analysis and assure its quality
- Provide assurance case showing that top-level specification agrees with security policy
- Develop a security test plan
- Develop an implementation of the system based on the top-level specification providing assurance case with argument and evidence (preferably including proofs) that
- Assure design
  - Agrees with top-level specification and security policy
  - Contains only items called for in top-level specification
- Assure that code is free of critical vulnerabilities, corresponds to design, and security policy and contains only items called for in design
- Perform penetration testing and test security functionality
- Provide a covert channel analysis
- Perform ongoing monitoring of security threats, needs, and environment
- Perform changes securely maintaining conformance to – possibly revised – security requirements while continuing to provide complete assurance case
- Deploy securely
- The quality and history of the people who produced it

- The characteristics and history of the kind of process used to produce it
- The quality of the environment in which it was produced
- Data on the quality and fidelity of use of the production process for this piece of software
- The realism of the assumptions made
- Characteristics of the software design including the extent to which it implements defense in depth and tolerance
- Results of reviews, tests, and analyses of it
- Data on the execution history of the software itself – particularly for post-update assurance

While certification authorities may not always consider everything relevant, every aspect having potential signification consequences for meeting security requirements or for the confidence of key stakeholders has a place in a full assurance case along with its related evidence. As all on that list are important, including everything on the list in the Fundamentals section's subsection on Assurance Case.

Thus, the assurance case is a top-level control document, normally summarized periodically thorough issuing Assurance Case Reports that record progress/changes in both arguments and linked evidence. Its purposes are to provide grounds for their confidence to producers as they proceed and, as emphasized by the Soviet gas pipeline explosion incident and its aftermath, supply the confidence in the product that acquirers, operators, and users must have to rationally acquire and use it in situations with security risks – the normal situation today.

8.3 Ensure Proper Version

Of course, one needs sound configuration and release management, but one also needs secure configuration management facilities and cryptographic signing of artifacts or other means to ensure their proper origin and their end-to-end integrity when distributed internally or externally. A process review to trace the use and non-use of cryptographic signatures could start with individuals producing them and extend all the way to execution or other external use.

8.4 Testing

8.4.1 Test Process

Many of the considerations in testing secure software are those of any quality testing process, for example, assuring the correct version is tested. Several considerations have differences that occur in activities such as:

- Independent Verification and Validation's independence add credibility to its results for some stakeholders and can bring added expertise. See subsection 8.8, Verification and Validation of User Aids.
- Unit Testing: One high-end user of formal methods has reported that measurement has shown little contribution by unit test to defect discovery and has tried eliminating it [King et al. 2000, section 5.1 and Figure 1 on pages 679-680] and [Hall 2002a, pages 21-22].
- No lightweight process user has reported similar results.
- Test Planning and Preparation must include security-related testing and a facility where potentially dangerous malicious actions can be imitated without fear of impacting others. Incorporating test results in the assurance case is eased by planning the tests to fill places requiring evidence within the assurance case's structured argument.

- Testing for Functionality Not in Specifications is important to help establish the software not just meets the specification but does nothing else.
- Conducting Test needs to be ethical as well as efficient and effective.
- Termination Criteria are unclear for testing such as penetration testing. If testing is restricted to technical attacks (e.g., no social engineering), then the interval between successes by a quality red team might be used. But how long without a break-in would be acceptable? Today, red teams seem to succeed in attacks against almost all software producers’ software with the limit often set by the willingness to continue paying the expensive red team. [Howard 2002, p. 48] gives the firm advice, “Do not ship with known exploitable vulnerabilities,” that he reports has worked for Microsoft.
- Certification Testing is a part of a certification process such as FIPS-140 for certification of cryptographic software, virus software certification by ICSA Labs, or the Common Criteria Evaluation process.
- Change and Regression Testing: Testing vulnerability fixes can be extensive because security properties are emergent properties. Any ability to confine the effort depends on the roles different tests play in the assurance case. Just as the structures of the designing and the assurance case can help in gaining assurance originally, structuring can potentially aid in restricting the amount of rework required in modifying the assurance case, including work in testing.
- Measurement: Security metrics are a difficult area, as discussed in subsection 8.9, Software Measurement and metrics attempting to measure the additional security assurance justified by a particular test result suffer the same difficulty. Successful attacks have a clear result that the system is not secure, but what confidence is justified after fixing the identified vulnerability?
- Reporting and Incorporating in Assurance Argument follow testing.

[McGraw 2006, Chapters 6 and 7] address security testing. As with most testing, security testing is risk- or consequence-driven. [McGraw 2004b] provides an introduction to risk-based security testing without explicitly addressing probability, and [Redmill 2005] addresses consequence-based (no probabilities exist) testing as well as risk-based testing.

While testing is essential in developing secure software, test-driven development where specifications are replaced by tests (specification by example) is of questionable viability, as it does not provide a serious approach to specifying such properties as non-bypassability.

8.4.2 Test Techniques

A number of testing techniques either are unique to security or have special uses for security testing.[1] [Bishop 2003, p. 533-540]

8.4.2.1 Attack-Oriented Testing

Attack testing involves having persons try to break the software. For security, this is called penetration testing and aims to violate specified or expected security usually by imitating techniques used by real-world malicious attackers. [Whitaker 2004] [Flickenger 2003] [McGraw 2006, Chapter 6] The teams that perform these tests are often called red teams, and generally include dedicated specialists. [McGraw 2006, p. 181-182] discusses an attacker’s toolkit.

Clearly, the tests are seeking vulnerabilities and likely vulnerabilities, i.e., potentially exploitable to cause harm, but testing time was not spent to prove it. One additional question to be addressed is how successful are any defensive deceptions employed.

[1] Of potential future interest, but not covered is MC/DC testing per FAA 178B avionics standard.

While often following game plans and scripts, this testing also often has an exploratory testing aspect. The SWEBOK Guide [Abran 2004, p. 5-5] defines exploratory testing as simultaneous learning, test design, and test execution; that is, the tests are not defined in advance in an established test plan, but are dynamically designed, executed, and modified.

The difficulties with deciding when to stop and what the results mean were discussed in Section 8.4.1, Test Process.

8.4.2.2 Brute Force and Random Testing

Advocates exist for testing with large volumes of structured or unstructured random data with the aim of exposing faults or vulnerabilities – sometimes called “fuzz” testing [Miller 1990]. When the frequently used criterion for failure is an abort or crash, failing these tests is a sign of quite poor software. [Redwine 2004]

Random testing to compare the performance of a new version to a different one can, however, be useful. Finally, testing with appropriate random distributions of input can be used to predict reliability and availability in the non-security context.

8.4.2.3 Fault- and Vulnerability-Oriented Testing

Testing designed to discover certain kinds of faults is known to be useful. Similarly testing aimed at a certain class of vulnerabilities can be useful particularly if one or a few kinds of faults underlie them.

While fault tolerance can mitigate the effects of faults, the characteristic of low defects is often discussed as a prerequisite for secure software, and the existence of increasing numbers of defects is usually believed to increase the chance of the existence of vulnerabilities. [Redwine 2004] Some dispute, however, exists about this [Lipner 2005b], and conclusive, generalizable evidence is hard to create.

8.4.2.4 Security Fault Injection Testing

To test the conditions where the attacker has had partial success or where part of the system has malfunctioned or ceased to exist/respond, one needs to inject faults in code or errors in data – or possibly via hardware faults. [Whitaker 2004, pp. 17-18, 159-162]

An attacker in control of part of the system would attempt to exploit its capabilities to achieve further success. A malfunctioning part of the system might be created deliberately by an attacker to achieve favorable results.

In a common case, attackers could attempt fault injection (modification) of client software, including web pages under their control in such a way to cause security violations. This can be explored by injections such as faults in an attempt to test the possibility.

Finally, an issue may exist in the possibility of failure by some element in the system’s environment, especially its infrastructure that can be tested by fault injection in the environment.

8.4.2.5 Using Formal Methods to Support Testing

Formal methods provide a formal specification of expected behavior that can be used to aid in generating test and most powerfully as a test oracle to automatically check if the behavior resulting from a test meets the specification. [Blackburn 2001] provides a security-related example. Another approach using mutation testing is described in [Wimmel 2002].

8.4.2.5.1 Test Generation

Specifications can be analyzed for a characterization of the input space and tests created heuristically (etc. partition or boundary value tests). State-machine-based specification may use modelchecking techniques to select test cases.

8.4.2.5.2 Test Oracle

Checking code derived from formal specifications can be used to check test results for correctness at any level of aggregation, but for security, the special aspect is checking if emergent system properties are violated as well as the correctness of security functionality. Most logic-based techniques presume that the specification related directly to system state and state changes, and not to the history of the system. Test oracles based on state machines may also check state sequences.

8.4.2.6 Usability Testing

Do security aspects affect usability and acceptability negatively, and, if so, how much? Usability of security features and the impacts of security such as those on convenience and performance can be tested, as can other usability issues. As mentioned in the Requirements and Design sections' subsections on Usability, creating quality solution options may require early and deep considerations. [Cranor 2005]

8.4.2.7 Performance Testing

How much do development decisions or product aspects affected by the security requirements slow performance? Performance testing can address this issue, but for best results, consideration of performance should not wait until subsystem or system testing.

8.4.2.8 Compliance Testing

Testing to demonstrate compliance with standards can be relevant for those standards relating to security. A number of standards have existing test suites or testing laboratories that can be used.

8.4.2.9 Reliability Testing

The probabilistic approach of reliability testing is not appropriate for confidentiality and integrity, but is one input to availability analysis.

Whether a software system produces correct results despite being under attack is a reliability issue, but it is also one of the central issues in security testing in general.

8.4.2.10 Evolvability or Maintainability Testing

Evolvability or maintainability testing could involve postulating changes and having them performed while measuring the time, effort, and mistakes made. The testing would include making any needed changes to the assurance case.

Among other uses, maintainability forecasts are inputs to availability analysis.

8.4.2.11 Operational Testing

The goal of operational testing is to investigate and assure successful use in a real world environment. Operational testing may include testing from early in development to after the system is in use. Kinds of operational tests include:

- Alpha and beta testing
- Field testing
- Operational test and evaluation
- Parallel with old version
- History collection of actual use after release

Security aspects need to be involved in all of these.

8.5 Dynamic Analysis

Often, dynamic analyses as opposed to dynamic testing, such as in slicing and some simulation modeling, do not actually execute the software under analysis. However, some analyses do.

8.5.1 Simulations

Simulation models of the system might be used to help verify that it conforms to the security policy or has resistance to certain kinds of denial of service attacks.

8.5.2 Prototypes

Prototypes intended to answer security-related questions play a normal prototype role. Prototypes that are intended to evolve into part of the system need to be developed so the same level of assurance case can be created for them as for the remainder of the system. Building the assurance case (or possibly a prototype thereof) during prototype development might be wise, just as it is in regular development.

8.5.3 Mental Executions

Mental execution of software has a long history and can have utility for cheaply and quickly illuminating sample execution paths, but its creditability as verification evidence is limited.

8.5.4 Dynamic Identification of Assertions and Slices

Automatically “guessing” pre- and post-conditions or invariants by observing many executions could be useful to provide candidates for existing software, but would be poor practice when developing new software. Such candidates cannot be uncritically depended upon as being the proper conditions for correctness.

Static analyses to determine a slice of a program that contains all that is relevant to some action or variable can be difficult and computationally intensive. Estimating slices by observing a number of program executions may help heuristically but generally cannot be depended upon to be complete.

8.6 Static Analysis

Static analysis methods relevant to security include formal methods and static code analysis. The tradeoff between dynamic testing and analyses, and static analyses is often a tradeoff of more accessible dynamic techniques that are only enumerating examples from a huge input space versus less accessible formal methods (mathematical but not using advanced mathematics), that cover the entire input space. See the section on Tools and Methods for more discussion of this.

8.6.1 Formal Analysis and Verification

The most powerful technical methods for assuring system descriptions’ consistency or correctness are mathematically based formal methods. Most formal techniques in use are based on either logic or model checking. See Section 9, Secure Software Methods and Tools.

Formal methods can be used for checking the existence or consistency of security-related software properties, within and across artifacts. Formal methods can show consistency among formal specifications of

- Security Policy
- Specification
- Design

- Code

Code is a formal notation as well. Some approaches lack rigorous automated support for showing correspondence between design and code.

For designs, techniques are available for both sequential and concurrent structures. Examples include [Hall 2003], [Barden 1995], and [Jürjens 2005]. Experience has shown that humans are bad at analyzing potential concurrent behaviors and that automated analysis is essential to ensure desirable properties – usually done using model checking. See subsection 9.2, Formal Methods.

Using formal methods is frequent in producing software to the highest assurance levels.

8.6.1.1 Static Code Analysis

For code in particular programming languages or subsets of languages, automated help exists to show

- Correctness (e.g., partial correctness)
- Lack of exceptions
- Information flow
- Concurrency abnormities
- Lack of potential vulnerabilities

While significant progress is being made, static analysis tools to scan source code for code practices that can result in vulnerabilities are reported often to give many false positives. It is not clear that this is the case, however, when coding standards are followed that forbid or discourage these practices. Automated style checkers can verify conformance to the bulk of such coding standards. The scalability of vulnerability-seeking static analysis tools also can be a problem though some commercial tools have analyzed software with millions of lines. [McGraw 2006, Chapter 4]

While scanning for potential vulnerabilities is a current industry emphasis, the capabilities listed in the first four bullets can also be quite useful; for an example of some of these capabilities, see [Barnes 2003]. Tools can also help locate security functionality and design features of potential interest, for example, use of cryptography, access control, authentication, native/dynamic code use, and calls opening network communications.

8.6.2 Informal Analysis, Verification, and Validation

8.6.2.1 Reviews and Audits

Reviews are prime examples of an informal but essential and effective technique. Knowledge exists on reviews with a variety of characteristics. Four primary differences exist for the concerns addressed during reviews within secure software projects:

1. Achieving security properties and policies
2. Doing only what the specification calls for
3. Avoiding software-related security pitfalls
4. Identifying possible attacker behaviors and consequences

Reviews, however, can be used to address all the topics mentioned under testing, often more cost effectively, and should therefore normally be a significant investment in testing. Security-oriented checklists exist for use in various software system reviews. For example, [Meier 2003, pp. 687-740] has a number of them.

Formalized techniques exist for reviews include scenario-based [Bass 2001], inspections and ones based on assurances of merit by experts rather identification of faults [Parnas 1985] [Laitenberger n. d.]

The last, called active or perspective reviews, might provide output in an attractive form for the assurance case.

In addition to reviews of security-oriented artifacts and concern for security as part of normal review activities, specialized security reviews and security audits may occur, as may legal or regulatory-oriented reviews. [McGraw 2006, pp. 165-167] describe “ambiguity analysis,” which is claimed to be useful to find areas of difficulty or misunderstanding in security. For full effectiveness, some of these reviews require special security-oriented expertise. For example, validation of an original threat analysis should involve threat experts.

All engineering artifacts for trusted software – requirement, design, code, etc. artifacts – must be reviewed by authors’ peers – and, as in instances mentioned above, other reviewers as needed. Good practice would, of course, call for this to be true for untrusted software as well. Indeed, without reviews, adequate assurance that software does or does not need to be trusted may not be possible.

Special or additional rounds of review or more reviewers may be employed as measures to add assurance evidence. Readiness reviews may occur before the security certification or accreditation activities.

Finally, regular reviews of the assurance case are necessary to ensure its quality. These reviews can benefit from involving

- Parties (or their surrogates) whose assurance is the goal of the assurance case
- Developers of the assurance case
- Persons who fully understand the evidence
- Persons able to judge assurance arguments rigorously

Reviews are possibly the most cost-effective verification technique and should be planned for and performed.

8.7 Usability Analysis

In addition or in combination with usability testing a fuller analysis can be done, for example [Whitten 1999].

8.8 Verification and Validation of User Aids

Because security needs to be treated correctly and effectively in user aids, such as manuals and help facilities, these need to be reviewed and probably included in usability testing. Kinds of user aids covered to ensure existence and correctness of security-oriented material include

- Documentation
  - Normal documentation covers security aspects and use
  - Additional operational security guide
- Training
- Online user aids, e.g., help facility
- User support
  - Help desk
  - Online support

8.9 Secure Software Measurement

The ultimate measure of merit question may be some variation of, “How much harm (and benefit) will be or has been caused by known and unknown security violations or vulnerabilities and how security was embodied in the software system involving it or other security-related impacts?”[2] Even if one could make decisions among all the possible variations of scope, duration, etc., of this question in defining an ultimate measure of merit, prediction or forecasting faces tremendous difficulties, including from the truth that no intellectually honest theoretical way exists to calculate risks from the technical characteristics of the software. The software measurement community is struggling with this issue and the problem of measures for software-related security, for a serious effort see [Murdock 2005].

This does not mean that useful engineering measurements cannot be made, analyzed, and used to improve. Often these measures are relative ones – change in one direction say becoming smaller, is known to be better, but “How much?” is not answerable in future experience in the field. Likewise, for an aspect of development, using a particular practice may be known to be better than using no practice at all, or one practice is better than another. In addition, doing an activity more skillfully is usually better. Over time, empirical bases may develop to make useful prediction of absolute values.

A somewhat similar problem exists with software reliability prediction, knowing the defect density of software does not allow (except at the extremes) much to be said intellectually honestly about the software’s reliability in the field as defects may or may not ever be exercised. Since we know having fewer defects is better, this has not kept us from using and benefiting from density of discovered defects as an engineering measure.

The same holds for counting the number of vulnerabilities. As with other defects, vulnerabilities can be ranked by severity and categorized for analysis and improvement. To continue the analogy, coding reviews have traditionally identified violations of the organization’s coding standards without necessarily impacting correctness. With security, coding practices that are in some circumstances potential bases for vulnerabilities can often simply be forbidden by the coding standards and counted in reviews or by automatic style checkers.

Reviews of other artifacts can do similar measurements, root cause analyses, and improvements. Counts of vulnerabilities created and discovered in each activity should be useful for analyses just as with other kinds of defects.

Measures of “attack surface” (set of points of possible attack) have been proposed. A crude one is how many ports are open. A more sophisticated measurement is described in [Manadhata and Wing 2004] and [Howard 2003a]. These are relative measures where less “surface” is better.

For projects aiming at less than highly secure software, reasonable commonality exists among the lightweight processes that have been developed, so organizations trying to improve in that range can make comfortable decisions about improvement based on community experience and expert consensus. Counting the number of activities adopted and measuring projects’ training and fidelity in following them are relevant process measurements.

In comparing high-end processes, benefit can be gained from not only the limited number of such secure systems about which information is available but the wider experience and arguments that have been

[2] Should we include future and variant versions as well as software using work products, methods, techniques, or ideas from this effort or product? What counts as harm? How insignificant a harm can be safely ignored? What about offsetting benefits, say in fewer accidents or reduced security spending for physical security?

conducted in highly safe software.[3] Measurements of analyzability and measurements resulting from automated analyses are potentially powerful and useful. Regardless of process and technology, well led, highly intelligent, skilled, and motivated people can make a big difference over merely good ones – everywhere but most noticeably at the high-end. These elements also offer opportunities from measurements.

Microsoft has adopted a set of metrics to aid in producing secure software. [Lipner 2005a] reports, "These metrics range from training coverage for engineering staff (at the beginning of the development lifecycle) to the rate of discovered vulnerabilities in software that has been released to customers.

"Microsoft has devised a set of security metrics that product teams can use to monitor their success in implementing the SDL.[4] These metrics address team implementation of the SDL from threat modeling through code review and security testing to the security of the software presented for Final Security Review (FDR). As these metrics are implemented over time, they should allow teams to track their own performance (improving, level, or deteriorating) as well as their performance in comparison to other teams. Aggregate metrics will be reported to senior product team management and Microsoft Executives on a regular basis."

Mary Ann Davidson of Oracle reports[5] similar efforts, "We also have a number of metrics in terms of measuring adherence to secure coding standards. For example, we measure and report on compliance with the mandatory secure coding class to executive management within each product team. A number of the development processes related to security are built in and reportable (since security sections are built in to functional, design, and test specifications and we enforce compliance with security section completion or projects are not approved for development). Similarly, there are requirements to run code through various checks (e.g., SQL injection detection tools) or transactions cannot be checked in. Many metrics involve "on/off switches" in that it is all or nothing: compliance is mandatory.

"There are multiple metrics around security bug handling, in terms of numbers and types of significant security vulnerabilities, who finds them (e.g., ¾ are found internally on average, though in some groups the "internal find rate" is as high as 98%).

"Most significant metrics are reported weekly, in development staff meetings."

Thus, software-related security measurement is on a slippery theoretical foundation, but much practical and useful measurement and analysis can be done.

8.10 Third-Party Verification and Validation and Evaluation

Among the relevant third-party activities are independent security testing, independent verification and validation, certification of the software system, and accreditation of an operational system.

8.10.1 Independent Verification and Validation

The phrase "independent verification and validation" implies verification and validation are done by a separate organization – not just a different team within project. For software intended to claim some degree of security,

[3] One confounding factor is that a number of such projects have been government procurements where contractors or groups of contractors have "pushed back" and possibly less than highly knowledgeable government negotiators have retreated. In other software areas, governments have been said to retreat from such arguments as, "We cannot hire enough programmers knowing <vastly superior programming language> and, if we have to train them and bring them up to speed, your whole system's tight schedule will slip." Of course, history seems to indicate that major systems have significant slippage deriving from multiple causes.

[4] Security Development Life cycle

[5] Personal communication with author (SR) September 26, 2005

an option exists to do concurrent certification. This is the seemingly preferable but rare way to do many certifications, including Common Criteria ones.

Even if this is done, independent performance of some verification and validation activities can make sense – possibly to bring added resources and expertise as well as independence. Independent security testing is possibly the most common, and is available from a number of sources. Independent verification and validation, however, appears to need changes from its traditional form to be (cost) effective for security – it certainly cannot emphasize “normal” scenarios.

Contracting for developing the assurance case – as opposed to aid and review – would appear to have the same downside that outsourcing preparation of documentation for Common Criteria certification has – it often does not influence and improve development or the product.

8.10.2 Software Certification

The Common Criteria standard is the most widely referred-to official (ISO) standard regarding software-related security. Others also exist, such as FIPS-140 certification of cryptographic software, virus software certification by ICSA Labs, and BITS in the financial sector. Protecting Sensitive Compartmented Information (SCI) within the US government is covered by [DCID 6/3 2000]. See Requirements section. [Bishop 2003, Chapter 21]

Each of these standards has its own evaluation process for certification. In the US, the Common Criteria has a two-step process, first evaluation by an approved laboratory and then validation by the government. Validation of EAL levels 1-4 is done by National Information Assurance Partnership (NIAP) and EAL 5-7 generally by NSA. [CC 2005, Part 4]

Common Criteria is an international standard and a number of countries have a reciprocity agreement for EAL 1-4, so opportunity exists at this level to consider laboratories internationally.

8.10.3 System Accreditation

The US DoD has created the DoD Information Technology Security Certification and Accreditation Process (DITSCAP). [DoD 8510.1-M] On the civilian side of the US government, the National Institute for Standards and Technology has produced NIST Special Publication (SP) 800-37, *Guide for the Security Certification and Accreditation of Federal Information Systems*. [NIST Special Pub 800-37]

ISO/IEC 17799:2005 published 15 June 2005 Information technology. Code of Practice for Information Security Management combined with BS7799-2:2002 (anticipated will be ISO/IEC 27001) form a basis for an Information Security Management System (ISMS) certification. More than 1300 certifications have been issued worldwide[6].

See Section 5, Software Requirements, for more information.

Ultimately, secure software’s goal is to be a part of a secure system providing secure services. This larger goal can have a significant impact on requirements; design; and verification, validation, and evaluation.

8.11 Assurance for Tools

Tools that have been subverted or have “naturally” occurring vulnerabilities are potential sources of problems in products and updates. The safety community has an analogous concern and, in using tools for FAA and FDA certifications, qualification is required for such tools.[7]

[6] ISMS Users Group, http://www.xisec.com/ These certifications were clearly to the old version of ISO 17799.
[7] Thanks to Mark Blackburn for pointing this out.

Automation generation of an artifact can also give confidence if confidence exists in the generation tool. For example, this is often the case for compilers. Nevertheless, an assurance case is needed for tools as well as the primary products.

In the assurance of automated support for formal methods, software to check proofs is much simpler than software to create proofs and is therefore easier to gain confidence in. .

If adequate assurance cannot be gained, then options include hardening the development network, and possibly separating it from all other networks.

8.12 Selecting among VV&E Techniques

Much of the process of selecting techniques for dealing with VV&E of secure software is unchanged. Special care can aid in ensuring that no aspect of security is covered by just one technique or only a few instances of examination.

The importance of quality requirements and design may be even more pronounced for security. The benefits of prevention and discovery of problems, while still relatively inexpensive to correct, remains true for security problems. Dangers and risk management drive much of the decision making on VV&E techniques and amounts of schedule, effort, and costs devoted to them remains unchanged.

Estimating the dangers and risks takes different techniques and can result in less certain results. See Section 11.4, Project Management, .

From a development process viewpoint, one question often asked is how likely is a defect or possibly more important a hard to discover and fix defect likely to occur if we do a development activity using certain techniques. When an activity is straightforward and the performers and reviewers have previously proven consistently proficient in their roles, then careful performance and review may be adequate. If it is less straightforward or the organization's track record is poorer, then more formal approaches to performance and VV&E would be called for.

The system may contain software that is trusted and software that is untrusted. For software, which is firmly established as not needing to be trusted, the dependability properties of interest tend toward traditional, probabilistic ones, and therefore methods such as input-distribution-based reliability testing may remain adequate.

[ISO/IEC 15443 Part 3 2004] provides a number of comparisons of official assurance or certification processes. These include ones based on ISO 13335, ISO/IEC 15408, ISO 17799, ISO/IEC 21827, ISO/IEC 15408, TCMM, X/OPEN, and FIPS-140-2.

Finally, laws, regulations, policies, and standards may call for using certain techniques. While laws and regulations may require compliance, following some standards may be discretionary. On this point and the general problem of selection, a relevant piece of advice appears in DEF STAN 00-56 [Ministry of Defence 2005b, Part 2 p. 70] that begins by mentioning the concept of making risk "as low as reasonably practicable" ALARP.[8]

"Because the ALARP criteria should be determined by the cost-effectiveness of the technique rather than the availability of the budget, the techniques recommended or mandated in authoritative sources (e.g., international or national standards) of relevant good practice should be applied whenever the costs are not grossly disproportionate to the benefits. Where there is a wide range of potential measures and techniques that could be applied, a consistent set should be adopted and the choice should be justified in the [Assurance] Case. The

[8] ALARP is a significant concept in UK law, and an excellent engineering-oriented discussion of it appears in Annex B of DEF STAND 00-56 Part 2.

selection should preferably be based on the collection of data about the performance of specific methods and techniques in similar applications, but as such data is difficult to obtain, it will probably be necessary to justify the selection using expert judgement (sic)."

8.13 Further Reading

[Avizienis 2004] Avizienis, Algirdas, Jean-Claude Laprie, Brian Randell, and Carl Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11-33, Jan.-Mar. 2004.

Available at http://csdl.computer.org/dl/trans/tq/2004/01/q0011.pdf

[Barden 1995] Barden, Rosalind, Susan Stepney, and David Cooper, *Z in Practice*, Prentice Hall, 1995.

[Chirillo 2002] Chirillo, John. Hack Attacks Revealed: A Complete Reference for UNIX, Windows, and Linux with Custom Security Toolset. Wiley Publishing, Inc., 2002.

[CSTB 2004] Committee on Certifiably Dependable Software Systems. *Summary of a Workshop on Software Certification and Dependability*. National Academies Computer Science and Telecommunications Board, National Academies Press, 2004.

[Herrmann 2001] Herrmann, Debra S. A Practical Guide to Security Engineering and Information Assurance. Auerbach, 2001.

[ISO/IEC PRF TR 19791] ISO/IEC PRF TR 19791 Information technology -- Security techniques -- Security assessment for operational systems. International Organization for Standards, February 6, 2006.

[Jackson 2005b] Jackson, David and David Cooper, "Where Do Software-related security Assurance Tools Add Value?", *NIST Workshop on Software-related security Assurance Tools, Techniques, and Metrics*, November, 2005.

[Le Grand 2005] Le Grand, Charles H. Software Security Assurance: A Framework for Software Vulnerability Management and Audit. Longwood, FL: CHL Global Associates, 2005.

[Merkow and Breithaupt 2004] Merkow, Mark S. and Jim Breithaupt, *Computer Security Assurance*, Thomson Delmar Learning, 2004.

[Ministry of Defence 2003b] Ministry of Defence. Defence Standard 00-42 Issue 2, *Reliability and Maintainability (R&M) Assurance Guidance Part 3 R&M Case*, 6 June 2003.

[Praxis 2004] Praxis Critical Systems Ltd, *EAL4 Common Criteria Evaluations Study*, September 2004. Available at: http://www.cesg.gov.uk/site/iacs/itsec/media/techniques_tools/eval4_study.pdf

[Software Tech News 2005] "Secure Software Engineering", *DoD Software Tech News*, Data Analysis Center for Software, July 2005.
Available at: http://www.softwaretechnews.com

[Srivatanakul 2003] Srivatanakul, Thitima, John A. Clark, Susan Stepney, Fiona Polack. "Challenging Formal Specifications by Mutation: a CSP security example," *apsec*, , *10th Asia-Pacific Software Engineering Conference* (APSEC'03), 2003, p. 340.

9 Secure Software Tools and Methods

9.1 Scope

Secure software development tools and methods assist a developer in creating secure software and usually encourage a specific disciplined development approach [Abran and Moore 2004]. It is assumed any secure tools and methods are used in conjunction with a secure development process (cf., Section 11: Secure Software Processes). In this section, representative tools and methodologies are presented and discussed. From the outset, it should be recognized that there is no "silver bullet" tool or technique that can deliver absolutely secure code. Each tool or technique has a particular purpose and has a particular scope of application. The section is not comprehensive in its level of detail, but does intend to cover the various types of tools and methods that can be used to produce secure software.[1] For lists of tool categories and some tools see [Jackson 2005b] and [Praxis 2004].

9.2 Formal Methods

"Formal Methods" refers to mathematically rigorous techniques and tools for the specification, design and verification of software and hardware systems. … '[M]athematically rigorous' means that the specifications used in formal methods are well-formed statements in a mathematical logic and that the formal verifications are rigorous deductions in that logic." [Langley 2005] Methods for the formal verification of security can be classified according to how security properties are specified [Sinclair 2005] or by the manner in which the security properties are established [Bishop 2003].

In general, an abstract model of a security system captures the essence of the security system under consideration. The abstract model has a well-defined semantics. Security requirements and functional requirements of the security system are then specified as logical assertions. Thereafter, a rigorous analysis is performed to show the abstract model satisfies the logical assertions. This process increases the confidence that the security system satisfies its functional and security requirements. It may well be the case that during the rigorous justification, the abstract model does not satisfy the logical assertions. This indicates either the security system does not satisfy the requirements; that the model was not a faithful abstraction; or that the functional or functional requirements were not faithfully captured. In either case, corrective measures must be taken.

A number of formalisms can achieve the process mentioned above. For example, Vienna Development Method (VDM) and Z (pronounced as Zed) are so-called axiomatic languages. Other types of languages based on state transitions model computation sequences. State transition systems include state diagrams, message sequence charts, CSP (Communicating Sequential Processes), and SCR (Software Cost Reduction). Algebraic languages implicitly define security properties by means of equivalence relations to specify functionality. Languages include the OBJ and the Larch family of languages as well as Temporal logic, CCS (Calculus of Communicating Systems), and Petri nets.

Axiomatic languages specify security properties as logical assertions that are shown to be true or false. Axiomatic language notations include the Vienna Development Method (VDM) and Zed (Z). These types of languages are also often used to specify abstract models of security systems because of their ability to model system state. State transition languages, on the other hand, not only model system state, but also explicitly model the conditions in which state transitions will occur. State transition notation systems include state

[1] Readers should be aware of the NIST SAMATE Project and its Nov. 2006 Workshop on Software-related security Assurance Tools, Techniques, and Metrics

diagrams, message sequence charts, communicating sequential processes (CSP), and calculus of communicating systems (CCS). Algebraic languages implicitly define security properties using equivalence relations to specify functionality. Languages include the OBJ and Larch family of languages. Temporal logic languages model timed or ordered sequences of events. Languages of this class of formalism include Petri nets, Temporal Logic with time Series (TLS), and state diagrams [Sinclair 2005].

Techniques for establishing security properties fall into two broad categories: proof-based techniques or model-based techniques [Bishop 2003]. Proof-based techniques are deductive and attempt to show that from a set of initial conditions, a particular set of conclusions will logically follow from a set of initial conditions. Tools that support deductive reasoning are generally called theorem provers. Examples of theorem provers include Another Logical Framework (ALF), Higher Order Logic (HOL), Prototype Verification System (PVS), Naval Research Laboratory Protocol Analyzer (NPA), and Theorem Proving System (TPS).

Some formal methods approaches are more easily used by practitioners than others. For example, based on an underlying state-machine model, the SCR technique from the US Naval Research Laboratory takes an approach that is reasonably accessible to serious practitioners, has tool support, and has been used on several projects [Heitmeyer 1998] [Heitmeyer 2005].

Model-based techniques "establish how well a specification of a system meets a set of properties." [Bishop 2003] Model checkers typically perform an exhaustive search over a finite state space [Clarke and Wing 1996]. Examples of model checkers include Concurrency Workbench (CWB), SPIN, Symbolic Model Verifier (SVM), and METAFrame. A comprehensive overview of model checking may be found in [Clarke and Grumberg 1999].

[Burrows and Martin 1989] develop a logic to reason about authentication. In their logic system, belief and knowledge can be expressed. The logic, now dubbed BAN logic, after the initials of the authors, has been successfully used to reason about authentication protocols. Several researchers have worked on this logic in an attempt to improve it.

9.3 Semi-formal Methods

Semi-formal methods, sometimes known as lightweight formal methods, often "emphasize the use of graphical representations" [Vaughn and George 2003]. Semi-formal techniques possess some formalism, but cannot be used to verify or validate most system characteristics. Moreover, another difference is that semi-formal techniques usually provide methodological guidance (since that is where these methods tend to be found) whereas formal techniques often do not [Alexander 1995]. Data Flow Diagrams (DFD), and the Universal Modeling Language (UML) can be considered semi-formal methods.

Some tools combine semi-formal and formal elements most notably extensions to UML. [Jürjens 2004] [Jürjens 2005].

9.4 Compilers

Many procedural language compilers will support some capability for compile-time checking of language-based security issues. Using the appropriate compiler invocation flags, compilers can perform strict type checking and "flag and eliminate code constructs and errors with security implications, such as pointer and array access semantics that could generate memory access errors, and bounds checking of memory references to detect and prevent buffer overflow vulnerabilities on stacks and (sometimes) heaps." [Goertzel and Goguen 2005]

Driven by the reasons C and C++ are poor choices for programming secure software, their compilers have introduced several features addressing a subset of their weaknesses of which if you must use them you should be aware.. Many C/C++ compilers can determine whether format strings used in printf or scanf statements are being used properly, as well as ensure the proper number of arguments is being used. Since these vulnerabilities are the source of many code injection and buffer overflow attacks, this compiler functionality should always be enabled.

“Code should never be compiled with debugging options when producing the production binary executable of an application or component. The reason for this is that some very popular commercial operating systems have been reported to contain a critical flaw that enables an attacker to exploit the operating system’s standard, documented debug interface … to gain control of programs accessed over the network.” [Goertzel and Goguen 2005]

Similar to so called “safe” compilers, there are also versions of safe runtime libraries. These libraries contain versions of system calls or language-specific functions that have been hardened against vulnerabilities present in the normal runtime libraries. Examples of “safe” libraries include Avaya Labs' Libsafe, which protects against buffer overflows[2] and David Leblanc's C++ SafeInt class[3].

Restricted execution and secure application environments can be used to improve security [Goertzel and Goguen 2005]. The sandboxing technique, used most notably by Java, is an example of a restricted execution environment. In the sandbox model, trusted code is given access to all resources subject to normal security policy. Untrusted code, like downloaded java applets for example, only get access to the resources made available in the sandbox. Secure application frameworks use a different technique. A secure application framework provides functionality through framework-provided libraries, applications and other built-in resources [Goertzel and Goguen 2005]. To use the services provided by the framework, applications invoke the service as prescribed by the particular framework being used. Prominent examples of these types of security services include secure sockets layer/transport security layer (SSL/TSL) services [Rescorla 2001], cryptographic libraries, HTTPS, and the .NET Windows security services.

The quality and analsys feature of compilers and libraries are important, but often the easiest and most powerful step is to select an programming language that as excellent affects on correctness and the power of the tools one has available. See section 7.3.

9.5 Static Analysis

Some static analysis techniques can be conservative, making worst case assumptions to ensure the soundness of the analysis. Dynamic analysis techniques (discussed in the next section), on the other hand, observe a program’s runtime behavior and may be precise, eliminating any need for assumptions about control flow or variable values. However, dynamic analysis may not be generalizable to all possible program inputs. Dynamic and static analysis techniques can be used together as they are in profile-direct optimization in modern compilers [Ernst 2003]. [Walden 2005]

On one hand, static analysis tools may provide support for the programmer to avoid exceptions and improper information flow or achieve correctness (e.g. the SPARK programming language with its accompanying analysis toolset). These tools can play an important part in providing assurance.

On the other hand, tool may provide useful information about the complexity, structure, style, and other directly observable characteristics of code often aimed at finding “bad” things as opposed to ensuring “goodness”. Static analysis may offer evidence of potential problems either as a result of inadequate programming and design approaches or as a result of a malicious code insertion.While most often associated with source code, Java bytecode and other binary files can also be statically analyzed. The following

[2] See http://www.research.avayalabs.com/project/libsafe/

[3] See http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure01142004.asp

paragraphs describe this latter kind of static analysis, the artifacts handled, and types of things these tools can detect.

The information in the following paragraphs draws heavily from and paraphrases [NIST SAMATE 2005]. Readers are encouraged to consult [NIST SAMATE 2005] for further information as well as a list of some of the current analysis tools and development suites available.

Source code scanning tools identify software vulnerabilities during development or after deployment. These tools detect coding flaws by examining the source code rather than scanning a target application at run time.

Because they have access to application source code, static source code scanners can identify a host of security issues, including range and type errors, calls to vulnerable library functions, buffer overflow possibilities, and unvalidated input. Runtime environmental problems can also be detected including relative path vulnerabilities, and possible sources of information leaks.

Depending on the capability of the tool, scanners can identify synchronization and timing errors, such as covert timing channels and time of check, time of use race conditions. Protocol errors, such as failure to follow chain of trust in certificate validation or key exchange without authentication, can be detected as well as general logic errors and other flaws.

While static byte code scanners are used in a similar fashion to source code scanners, they detect vulnerabilities primarily through pattern recognition. They can determine, for instance, whether a method exposes its internal representation or whether a finalizer should be protected.

Static Binary Code Scanners detect vulnerabilities through disassembly and pattern recognition. Binary code scanners examine the compiled object code directly and as a result can factor in any vulnerabilities created by the compiler itself. Furthermore, library function code (not available to a source code scanner) can be examined as well. Loadable images can also be a significant source of vulnerabilities. [Thompson 1984].

Binary code scanners analyze binary signatures within executables for things like SQL injection, cross-site scripting, buffer overflows, and missing format strings.

Database scanners are used specifically to identify vulnerabilities in database applications. In addition to performing “external” functions like password cracking, the tools also examine the internal configuration of the database for possible exploitable vulnerabilities. Database scanning tools discover vulnerabilities by checking for adequate passwords, default account vulnerabilities, role permissions, and unauthorized object owners.

Static analysis of specifications and designs are available for certain notations. Some address issues in informal descriptions, but most require formal notations. See section 9, Secure Software Tools and Methods.

9.6 Dynamic Analysis

Dynamic analysis observes entities in operation. Network scanners can detect vulnerabilities in operating systems, applications, servers and mobile code that manifest some type of observable network behavior such as generating network traffic. These types of scanners typically look for known vulnerabilities but heuristics for detecting previously unobserved malicious behavior are not unknown.

Web Application Scanners focus specifically on web applications and perform field manipulation cookie poisoning functions. In contrast, Web services scanners focus on analyzing web-based services and performing things like XML validation of received messages.

9.7 Development Tool Suites

Development software can benefit from producing secure software. A typical software requirements phase produces requirements documents that can be examined by automated tools. However, a suite of widely-used, peer-reviewed software requirements and design analysis tools does not yet exist. However, whatever software is used, it should support determining whether the specifications are:

- Complete
- Consistent
- Correct
- Modifiable
- Ranked Or Rated
- Traceable
- Unambiguous
- Understandable
- Verifiable

Suites covering the entire life cycle are available from IBM Rational and by composing suites from multiple vendors.

9.8 Selecting Tools

Few tool selection efforts have appeared in the literature, and, as always, one should beware of simple-minded decision-making processes such as counting the numbers of features. Andrew J. Kornecki has performed a series of studies of tools for high-assurance software for the US Federal Aviation Administration.[4] His work has been driven in part by the requirements of DO 178B, the aviation safety standard. An example is [Kornecki 2005]. A report resulted from Praxis High-Integrity Systems study for the UK government on code analysis tools, [Jackson 2005a] as part of an investigation on improving the evaluation at the Common Criteria EAL4.

Advice from practical experience comes from a Working Group member, Mary Ann Davidson of Oracle:, “It’s worth noting that the state of a number of vulnerability finding tools is so bad at present that some of them make the problem worse, because a) it takes a scarce resource (a security-aware developer) to go through 300 pages of tool output only to find ¼ page of real issues and b) a bad tool will turn developers off from ever using them again. I recommend that people put together cross-functional teams to vet products competitively before buying, including scalability, proof of claims (proof that product works as advertised), extensibility and so on. I believe we are in the hype phase of the product life cycle: there are beginning to be a critical mass of tools, but many or most of them do not work as advertised, or do not work at an acceptable cost for value of information provided. There are a few – very few – good tools, and many bad ones. This is, however, one of the "right problems to solve in security," and at such time as these tools are proven to work with high scalability and very low false positive rate, it ought to become standard development practice to use them.”

This subsection gives some insight into differences in evaluating tools for high-assurance software, but also shows that the selection processes have the same problems and merits as those to select tools for other software – the stakes and the need for care are just higher.

[4] See http://faculty.erau.edu/korn/publ05.html

9.9 Further Reading

[Abran 2004] Abran, Alain, James W. Moore (Executive editors); Pierre Bourque, Robert Dupuis, Leonard Tripp (Editors). *Guide to the Software Engineering Body of Knowledge*. 2004 Edition. Los Alamitos, California: IEEE Computer Society, Feb. 16, 2004. Available at http://www.swebok.org.

[Barden 1995] Barden, Rosalind, Susan Stepney, and David Cooper, *Z in Practice*, Prentice Hall, 1995.

[Ibrahim et al, 2004] Ibrahim, Linda, et al, *Safety and Security Extensions for Integrated Capability Maturity Models*. Washington D.C.: United States Federal Aviation Administration, Sept. 2004. Available at http://www.faa.gov/ipg/pif/evol/index.cfm

[Lipton 2002] Lipson, H.F., N.R. Mead, A.P. Moore. "Assessing the Risk of COTS Usage in Survivable Systems." *Cutter IT Journal* 15:5. May 2002.

[Redwine 2004] Redwine, Samuel T., Jr., and Noopur Davis (Editors). *Processes for Producing Secure Software: Towards Secure Software*. vols. I and II. Washington, D.C.: National Cyber Security Partnership, 2004.
Available at http://www.cigital.com/papers/download/secure_software_process.pdf

[Sheyner 2002] Sheyner, Oleg, Somesh Jha, and Jeannette M. Wing, “Automated Generation and Analysis of Attack Graphs,” *Proceedings of the IEEE Symposium on Security and Privacy*, 2002.

[Sommerville 2006] Sommerville, I., *Software Engineering*, 8th ed., Pearson Education, 2006.

[Viega 2005] Viega, J., *The CLASP Application Security Process*, Secure Software, 2005. Available at http://www.securesoftware.com.

10Secure Software Processes

This Secure Software Engineering Process section has two themes:

1. The process arrangements for technical and managerial activities within the secure software life cycle
2. Definition, implementation, assessment, measurement, management, change, and improvement of the these secure software processes

The processes described cover a major part or all of secure software acquisition, development, sustainment, and retirement. They vary from ones based around mathematical formalism to those merely trying to make modest improvements on a legacy product.

This guide presumes the reader has knowledge of software engineering activities and processes that do not have security or safety as a concern. A number of activities are new or significantly modified for secure software. The other sections of this guide cover these changes in activities. This section emphasizes the changes in overall life cycle process by describing several classes and examples of secure software processes. Following [Boehm 2003], it uses the terms “heavyweight” and “lightweight”.

In practice, these processes vary across a wide range, but they will be grouped here into three categories:

1. Heavyweight development processes [Hall 2002a] [NSA 2002, Chapter 3]
2. Lightweight processes, [Viega 2005] [Lipner 2005a] [Howard 2002] [Meier 2003]
3. Processes especially for the problems of legacy software [Viega 2005, p. 40] [Meier 2003, p. lxxxi].

Generally, the kinds of activities done in heavyweight processes are a modest superset of lightweight processes but with many performed with considerably more rigor. The activities initially used remedially for legacy software often consist of a subset of the lightweight ones. Note that legacy software may eventually require more serious rework such as major re-architecting.

Introducing secure software-oriented changes in processes within ongoing organizations and projects – development and sustainment – has all the usual problems and solutions of organizational change and software process improvement.

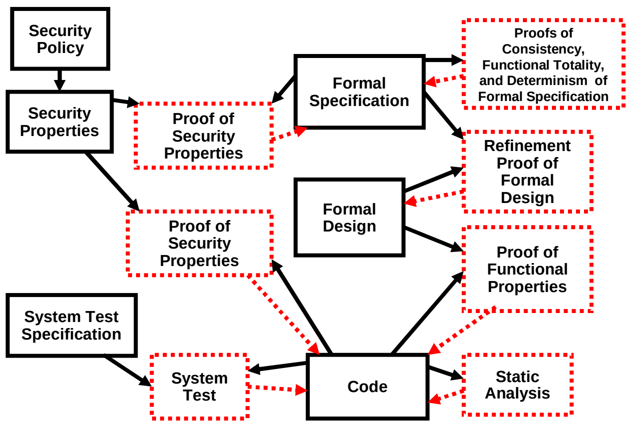
10.1 Heavyweight Processes

Heavyweight processes are often built around extensive use of reviews and formal methods. Formal notations used include Z, ACT2, extensions to UML such as UMLsec, and code. [Hall 2002b] [Jűrjens 2004] [Barnes 2003]

Figure 3 shows an example of how formal techniques can be used to increase assurance by adding evidence to the assurance case in proofs. Using proofs does not result in the complete elimination of testing. Demonstrating that the system handles all possible input – the functions are total – and that they are functions – e.g., deterministic – aids in showing security properties. Not shown explicitly are proofs needed to deal with concurrency.

One of the three processes mentioned in [Redwine 2004] as a foundation for producing secure software uses fully explicit formal methods – Correctness-by-Construction. The two other processes – Cleanroom and TSP-Secure – use mathematically based methods, such as state machines, but with more reliance on reviews and informal proofs and less on formal proofs. The SCR approach with its toolset also has its proponents (see Section 9, Software Tools and Methods).

Figure 3: Formal Methods Assurance (Dashed) – Modified from a diagram by Praxis Critical Systems


Security Policy
Security Properties
Proof of Security Properties
Formal Specification
Proofs of Consistency, Functional Totality, and Determinism of Formal Specification
Refinement Proof of Formal Design
Formal Design
Proof of Security Properties
Proof of Functional Properties
System Test Specification
System Test
Code
Static Analysis

At least one exception exists to heavyweight processes using formal methods. While aimed at safety, the aviation industry's heavyweight process based on DO-178B uses extensive systematic testing, although builders have chosen at times to use formal methods also. Unmodified, this style of testing effectiveness for security has not been accepted by DoD for the newest US military aircraft.

10.2 Lightweight Processes

Lightweight secure software processes may or may not be used with processes having CMM levels. A CMM-style addition has been proposed for safety and security and incorporated into the FAA's iCMM but not the SEI's CMMI. [Ibrahim et al, 2004]

Microsoft's Secure Development Life cycle is an example of a lightweight process that builds on a number of existing processes of varying degrees of restrictiveness. [Lipner 2005a] Covering engineer education, metrics, and accountability; and presuming a centralized security group, this process incorporates:

Planning and Requirements

- Assign "security buddy" from central group
- Incorporate security activities into plans
- Identify key security objectives
- Consider security feature requirements
- Consider need to comply with industry standards and by certification processes such as the Common Criteria
- Consider how the security features and assurance measures of its software will integrate with other software likely to be used together with its software

Design

- Define security architecture and design guidelines
- Document the elements of the software attack surface
- Conduct threat modeling
- Define supplemental ship criteria for security

Implementation

- Apply coding and testing standards
- Apply security-testing tools including fuzzing tools
- Apply static-analysis code scanning tools
- Conduct code reviews

Verification

- While the software is undergoing beta testing, the product team conducts a "security push" that includes security code reviews beyond those completed in the implementation phase as well as focused security testing

- Final Security Review or FSR

Support

- Prepare to evaluate reports of vulnerabilities and release security advisories and updates when appropriate
- Conduct a post-mortem of reported vulnerabilities and take action as necessary
- Improve tools and processes to avoid similar future vulnerabilities

[Howard 2006] provides a book length management-oriented view of this Microsoft process.

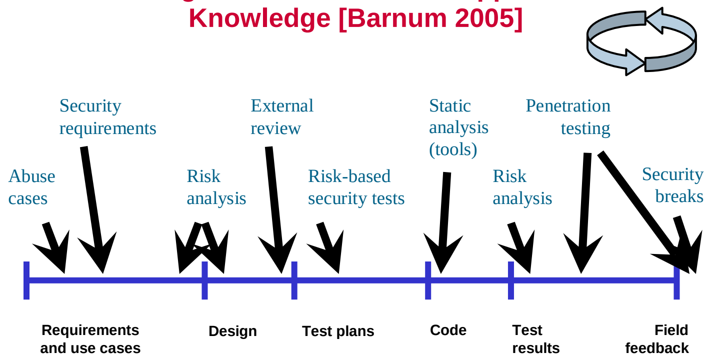
CLASP adds 30 activities to existing software processes without explicit concern for the previously existing processes. [Viega 2005] These activities are:

- Institute security awareness program
- Monitor security metrics integrator
- Manage certification process
- Specify operational environment
- Identify global security policy
- Identify user roles and requirements
- Detail misuse cases
- Perform security analysis of requirements
- Document security design assumptions
- Specify resource-based security properties
- Apply security principles to design
- Research and assess security solutions
- Build information labeling scheme
- Design UI for security functionality
- Annotate class designs with security properties
- Perform security functionality usability testing
- Manage System Security Authorization Agreement
- Specify database security configuration
- Perform security analysis of system design
- Integrate security analysis into build process
- Implement and elaborate resource policies
- Implement interface contracts
- Perform software-related security fault injection testing
- Address reported security issues
- Perform source-level security review
- Identify and implement security tests
- Verify security attributes of resources
- Perform code signing
- Build operational security guide
- Manage security issue disclosure process

[McGraw 2005] and [McGraw 2006] take an artifact-oriented approach and propose seven “touch points” to add to existing processes. These are:

- Abuse cases
- Good security requirements
- Code review
- Risk analysis is a necessity
- Penetration testing is also useful, especially if an architectural risk analysis is driving the tests
- Testing security functionality with standard functional testing techniques, and risk-based security testing based on attack patterns
- Monitoring software behavior is an essential defensive technique. Knowledge gained by understanding attacks and exploits should be cycled back into software development.

Figure 4: Activities and Applied Knowledge [Barnum 2005]



He also provides a bonus “touch point” – external analysis, meaning from outside the design team. This is often a necessity in security. He also makes a point that at the design and architecture level, a system must be coherent and present a unified security front.

RUPSec, which so far has extended the Business Modeling and Requirements disciplines of Rational Unified Process for developing secure systems, provides a way for users of RUP to initially address security for new and possibly legacy systems. [Pooya 2005]

Lightweight processes have found favor in industry as a way to produce more secure software. Unlike some heavyweight processes, their ability to produce highly secure software is not established.

An area of controversy is the abilities of various “agile” software processes to produce secure software. The central role of architecture in establishing and maintaining security over the evolution of the software as well as other issues raise a number of questions. [Beznosov 2004] provides an enumeration of the areas where agile processes tend to be consistent with achieving security and one where they have unresolved conflicts, and the issues are discussed at some length in [Berg 2005, Chapter 19].[1]

10.3 Legacy Upgrade Processes

In practice, the lightweight processes discussed in the prior subsection have been applied to legacy software. Two authors suggest processes or activities especially for approaching the problems of legacy software [Viega 2005, p. 40] and to a much lesser extent [Meier 2003, p. lxxxi].

Viega lists a variant of the CLASP activities in a “Legacy Roadmap.” These are

- Institute security awareness program
- Specify operational environment
- Identify recourses and trust boundaries
- Document security-relevant requirements

[1] An mp3 recording of a discussion with Clifford Berg on security and agile processes is available at http://agiletoolkit.libsyn.com/index.php?post_id=53799 (200603) within which he describes a number of key issues to an experience agile dévote.

- Identify attack surface
- Perform security analysis of system requirements and design (threat modeling)
- Address reported security issues
- Perform source-level security Reviewer Identify, implement, and perform security tests
- Verify security attributes of resources
- Build operational security guide
- Manage security issue disclosure process

[Meier 2003, p. lxxxi] states, "Threat modeling and security assessment (specifically the code review and deployment review …) apply when one builds a new web applications or when one reviews an existing applications."

Steve Lipner of Microsoft has suggested[2] the following for legacy software:

- Go back and modify legacy code to make in closer to [security cognizant] standard for new code
- Deprecate functionality that is dangerous or unneeded
- Reduce its attack surface
- Reduce the privilege level of operating system mode or environment that it runs in

Microsoft has also had success using fuzz testing.

Also directly relevant is Section 10.5.1, Introducing Secure Software Engineering Processes, below.

10.4 Concern for Security of Developmental Process

As mentioned elsewhere, a secure environment and organizational security processes are needed. A special problem requiring prevention and detection measures in the secure software development process is subversion.

Subversion must be overcome by observation, verification, and validation with levels of independence, skill, and intensity sufficient to reduce the risk of the inclusion of malicious code in the product (or malicious elements in non-code artifacts) to an acceptable level. This may include automated analysis and testing, but currently only human reviewers contain the intelligence needed.

The concept of separation of duties is a central one in countering maliciousness. The rule that everything must be known by at least two people is mandatory as first level protection against subversion. This rule could extend thorough reviews at all levels. An example is the review of the "proposed final" version by persons other than the author before final approvals. Proper configuration management ensures the version reviewed is the version approved and delivered.

Verification, validation, and evaluation activities should be controlled to prevent deliberate overlooking or mislabeling of defects.

Operations that touch the code (e.g., doing builds or preparing releases) must also be controlled and verified.

[2] Microsoft Academic Days on Trustworthy Computing 7 April 2006

10.5 Improving Processes for Developing Secure Software

This Guide presumes knowledge related to processes for software systems where security or safety are concerns. Introducing changes in software engineering processes to deal with the added requirements for secure software resembles making other software process changes. Organizational change and software process improvement are fields in which a substantial body of knowledge and expertise exist; [Redwine 2004, Chapter 6] lists a number of references including the SEI site on technology transition, [SEI TT], and the definitive book on technology transfer, [Rogers 1995]. [McGraw 2006, Chapter 10] discusses introducing security-oriented practices. The general characteristics of good process definitions and characteristics have also been enumerated [ISO 15288] [Redwine 2004, Chapters 3 and 5]. The subsections below enumerate some issues or special considerations for processes for secure software.

10.5.1 Introducing Secure Software Engineering Processes

Readiness for the needed process changes involves the motivation, general familiarity, and particular skills required for the changes being instituted. Security awareness training has frequently been used initially to address the first two of these. [Viega 2005] [Lipner and Howard 2003]. While sharing some features with computer security awareness training given to users and system administrators, this training also provides an introduction to security-related virtues and "sins" in software, including common kinds of exploits. This possibly includes a demonstrated breaching the security in the software product of the project or a team receiving training.

The eventual success of significant change is aided by its early stages showing immediate payoff. These might include changing configuration settings to leave only universally (or say 80 percent) used or essential services available by default; eliminating uniform default passwords; and ensuring all external inputs are validated.

What to do first, however, may be driven by an expert security audit of the product. [Campara 2005] [Viega 2005] While this may require obtaining outside expert services, the properly explained results can directly motivate not only changes to the product but also changes to the process. Some process changes have been done first in a form aimed at finding and fixing certain kinds of vulnerabilities, for example problems deriving from unauthenticated users, but also change can be motivated to avoid future problems similar to ones currently being reported.

Repeated auditing could be used to motivate yet more changes. [Campara 2005]

10.5.2 Improving Secure Software Engineering Processes

Special considerations in secure software engineering process improvement include those deriving from the forces driving change and the mathematical skills required.

Identified or exploited threat capabilities, exploit techniques, and reported vulnerabilities may drive changes in processes to counter them. These stimulating troubles may be identified by internal reviews, tests, or audits or may arise in the external world. Repeated product audits were mentioned in the case of initiating a security effort, but they can be a driver of established efforts as well. Process audits or assessments also can be a driver.

External stimulus events may be particular to the organization's product(s) or a general development. Scanning the environment for relevant items can provide early warnings of possible future problems.

Driven by security concerns, one major software vendor has adopted the policy of increasing the absolute requirements on its projects' software engineering processes every six months. Efforts to improve have shown results. For example, Mary Ann Davidson of Oracle, another major vendor, states, "Oracle finds fully 3/4 of significant security vulnerabilities ourselves, via our development processes and tools such as code reviews, QA, security-specific tests (e.g., for SQL injection attacks), ethical hacking, and the like."[3]

[Ibrahim et al, 2004] defines proposed extensions to the Software Engineering Institute's CMMI for combined coverage of safety and security. These have been adopted and incorporated into the US Federal Aviation Administration's iCMM capability maturity model. It includes additional activities for the development process and concern for the work environment, including its security. Thus, these extensions have four development goals and a work environment goal:

- Goal 1 – An infrastructure for safety and security is established and maintained.
- Goal 2 – Safety and security risks are identified and managed.
- Goal 3 – Safety and security requirements are satisfied.
- Goal 4 – Activities and products are managed to achieve safety and security requirements and objectives.
- Additional Goal: A work environment that meets stakeholder needs is established and maintained.

The 16 practice areas in the development portion are shown in Table 9 below under the goal they support.

Table 9: SSE-CMM Practice Areas

• Goal 1 – An infrastructure for safety and security is established and maintained.
• AP01.01. Ensure safety and security awareness, guidance, and competency.
• AP01.02. Establish and maintain a qualified work environment that meets safety and security needs.
• AP01.03. Establish and maintain storage, protection, and access and distribution control to assure the integrity of information.
• AP01.04. Monitor, report and analyze safety and security incidents and identify potential corrective actions.
• AP01.05. Plan and provide for continuity of activities with contingencies for threats and hazards to operations and the infrastructure.
• Goal 2 – Safety and security risks are identified and managed.
• AP01.06. Identify risks and sources of risks attributable to vulnerabilities, security threats, and safety hazards.
• AP01.07. For each risk associated with safety or security, determine the causal factors, estimate the consequence and likelihood of an occurrence, and determine relative priority.
• AP01.08. For each risk associated with safety or security, determine, implement, and monitor the risk mitigation plan to achieve an acceptable level of risk.
• Goal 3 – Safety and security requirements are satisfied.
• AP01.09. Identify and document applicable regulatory requirements, laws, standards, policies, and acceptable levels of safety and security.
• AP01.10. Establish and maintain safety and security requirements, including integrity levels, and design the product or service to meet them.
• AP01.11. Objectively verify and validate work products and delivered products and services to assure safety and security requirements have been achieved and fulfill intended use.
• AP01.12. Establish and maintain safety and security assurance arguments and supporting evidence throughout the life cycle.
• Goal 4 – Activities and products are managed to achieve safety and security requirements and objectives.
• AP01.13. Establish and maintain independent reporting of safety and security status and issues.
• AP01.14. Establish and maintain a plan to achieve safety and security requirements and objectives.

[3] Personal communication with author (SR) September 26, 2005.

- AP01.15. Select and manage products and suppliers using safety and security criteria.
- AP01.16. Measure, monitor, and review safety and security activities against plans, control products, take corrective action, and improve processes.

Note that at this high level, all the items combine safety and security. They are treated differently at lower levels of the document.

A System Security Engineering CMM exists with technical and organizational parts [SSE-CMM 3.0]. The box below lists technical areas.

SSE-CMM Technical Areas

- Administer Security Controls
- Assess Impact
- Assess Security Risk
- Assess Threat
- Assess Vulnerability
- Build Assurance Argument
- Coordinate Security
- Monitor Security Posture
- Provide Security Input
- Specify Security Needs
- Verify and Validate Security

The highest EAL-levels of the Common Criteria and much of the advanced practice in high-confidence systems call for using mathematically based formal methods. These may require a limited amount of knowledge of discrete mathematics – often centered on set theory, predicate logic, and finite state machines – but may require sophistication in performing proofs or model checking by some on the project.

Except possibly for a few approaches, the mathematics involved is much less sophisticated than other fields of engineering – albeit discrete rather than continuous mathematics.[4] Nevertheless, many US developers and managers appear to possess a reluctance to learn (in the majority of the cases, probably relearn) the relatively simple mathematics involved to apply it to software. As important, organizations may possess a reluctance to pay for the training and learning. One does see, however, an increased acceptance of particular techniques, such as formally expressed "contracts" for invocations. [Myers 84]

While use of formal methods may be motivated by concerns for security, since the early 1990s their practical use has been driven more by concerns for correctness and safety. A mixed history exists concerning their introduction and use in industry. Events related to the industrial use of formal methods include the decade old annual International Workshop on Formal Methods for Industrial Critical Systems (FMICS)[5] and a related industry association is the Formal Techniques Industrial Association (FotTIA).

10.6 Further Reading

[Breu 2003] Breu, R., K. Burger, M. Hafner, J. Jürjens, G. Popp, G. Wimmel, and V. Lotz, "Key Issues of a Formally Based Process Model for Security Engineering", *Proceedings of the 16th International Conference on Software & Systems Engineering and their Applications (ICSSEA03)*, 2003.

[Jacky 1996] Jacky, Jonathan, *The Way of Z: Practical Programming with Formal Methods*, Cambridge University Press, 1996.

[Kolawa 2005] Kolawa, Adam, "Hold the Line against App Attacks," *Software Test and Performance*, November 2005.

[Mead 2003] Mead, Nancy R. "Lifecycle Models for High Assurance Systems," Proc. of Software Engineering for High Assurance Systems: Synergies between Process, Product, and Profiling (SEHAS 2003), Software Engineering Institute, p. 33, 2003.
Available at: http://www.sei.cmu.edu/community/sehas-workshop/

[4] Remember this document does not address developing cryptographic software.
[5] See http://www.inrialpes.fr/vasy/fmics/

[Saitta 2005] Saitta, Paul, Brenda Larcom and Michael Eddington, "Trike v.1 Methodology Document," [Draft], 20 June 2005.
Available at: http://www.hhhh.org/trike/papers/Trike_v1_Methodology_Document-draft.pdf

[Srivatanakul 2003] Srivatanakul, Thitima, John A. Clark, Susan Stepney, Fiona Polack. "Challenging Formal Specifications by Mutation: a CSP security example," p. 340, *10th Asia-Pacific Software Engineering Conference* (APSEC'03), 2003.

11Secure Software Project Management

11.1 Introduction

Secure Software Project Management is the “systematic, disciplined, and quantified” application of management activity to include the “planning, coordinating, measuring, monitoring, controlling, and reporting” that ensures the software being developed conforms to security policies and meets security requirements [Abran 2004].

Two issues cause differences in project management:

- High-assurance (i.e. low uncertainty about meeting a software system’s requirements being highly assured, which might or might not include security)
- Security

Almost no literature exists that directly addresses differences in project management deriving from these differences. Therefore, the author of this section sought out experts and experienced managers in these areas.

In correspondence with one of this guide’s authors (SR), John McDermid of the University of York, an experienced expert on high assurance systems, stated the following concerning managing high-assurance software projects:

“In running projects, managers have essentially four “variables” to control:

- Scope – the functions and facilities included in the system
- Quality – the reliability, robustness, usability, etc., of the system
- Resources – the quantity and capability of the staff and computational tools used on the project
- Time – the schedule for the project.

“For secure systems, where assurance and possibly certification or accreditation are essential goals, both scope and quality are constrained. The system will not be acceptable without certain non-bypassable security functionality, nor will it achieve certification or accreditation unless it is of sufficient quality. This reduces the options available to project managers.

“It is becoming widely accepted that incremental or evolutionary development is an appropriate way to address the problems of complex IT systems. The above analysis of options available to project managers suggests two problems with these approaches:

- The first releasable increment has to include all core security functions, hence it may be a ‘large’ increment
- Quality cannot be compromised, and the design must be such that the quality of the “first increment” cannot be undermined by later increments.

“Thus, managers need to define, as early as possible, the architecture for the system, identifying the key security properties and mechanisms and ensuring that they are designed and developed with an architecture such that they cannot be compromised by adding later increments.

“The project needs to be managed viewing the architecture and the quality of the software as non-negotiable plus gaining senior management acceptance that timescales and resources, hence costs, have to be variable (may need to increase) so as not to compromise security.

“There is little published literature on this topic. However, whilst they do not address security in any depth, Boehm and Turner [Boehm 2003] give a very readable account of the issues in balancing agility and discipline, which is highly relevant to achieving software assurance.

“The competence of personnel is very important. Competencies include:

- Technical skills, especially knowledge of security
- Knowledge of techniques for achieving low defect density
- Domain knowledge, i.e., understanding of the application area
- Communication skills
- Personal attributes, such as integrity

There has been an attempt in the UK to codify such skills for safety of programmable electronic systems [IEE 1999], but so far as I am aware no systematic approach exists in the area of security.

“There are few management techniques focused solely on low defect/high assurance software, although there has been some positive experience of using approaches such as the SEI's Team Software Process on projects in the UK. Perhaps the most crucial management technique is in process measurement of errors and defects, analysis of the root cause of such defects, and using this information to help manage the project and to improve processes."

If one has successfully managed a high-assurance project, say for safety, then a number of security issues remain, but much of the general project management approach can be the same. Martin Croxford of Praxis High-Integrity Systems Ltd wrote one of the authors (SR) that, “From a project management perspective, I'm not aware of anything that is different/additional with respect to any other high integrity development. When I managed the Mondex project [Hall 2002a], I didn't do anything differently because it was a security project - simply applied the usual Praxis planning and delivery approach (plus lessons learned from my previous projects, of course). The fact that it was a security project obviously affected the risk assessment, technical approach, quality planning, etc., but the project management approach was the same.”

11.2 Start Up

Project initiation has all the usual concerns, but with security and the need for high assurance impacting almost all of them in some way. Security is an essential issue in establishing secure physical, communications, and computing facilities. Personnel need to be not only highly skilled and knowledgeable about security concerns within their roles but also trustworthy. [PMBOK] has a enumeration of start up activities. For considerations related to process and organizational change, see Section 10, Secure Software Processes.

11.3 Scoping Project

A number of differences in activities have been covered in prior sections. These must be planned and performed. The levels of organizational and individual experience and proficiency can mean that the result is anything from a substantial increase in hours needed if new or a low to modest increase if highly experienced and skilled. This and the acceptable risk levels of the producer, users, and other stakeholders may make some requirements essentially unobtainable by some organizations. The size of the product can have an impact on its number of vulnerabilities possibly varying with the square of the size [McGraw 2006, p. 13]. Questions of feasibility and the wisdom and usefulness of instead attempting a more modest system or more modest security and assurance goals dominate the scoping issues for many organizations. Project management should realistically balance the project’s capabilities against the initial goals of the project and be willing to decide they are unlikely to be achieved and explore realistic goals.

11.4 Project Risk Management

As in scoping the project, possible consequences or risks are a constant concern throughout secure software projects. [McGraw 2006, Chapter 2] Product-affecting ones are explicitly addressed in the assurance case, but project-oriented ones not directly affecting the product but rather such items as schedule or costs must also be addressed, including attacks on the project and subversion of products. One essential difference within many secure software projects is the lack of or irrelevancy of probabilities. This means that consequences rather than risks are what must be managed.

One attempt to bridge the gap between possible consequences and probability analysis is [Baskerville 2003]. Theoretically, project managers might benefit from knowledge of game theory with its techniques that do not depend on knowing the probabilities.

11.5 Selecting a Secure Software Process

The Secure Software Processes section grouped relevant processes into three groups: heavyweight, lightweight, and those especially for legacy software. Most software organizations are facing the potential need for organizational change to better address software-related security. In addressing the process selection, decision management might consider the process' suitability and fit, and costs and benefits; and constraints on the project.

In selecting a process, many factors are relevant as covered within Section 10.5, Improving Processes for Developing Secure Software, and Section 9.8, Selecting Tools. Four factors management probably will need to consider are:

1. The nature, amount, and quality of any existing artifacts
2. The required levels of software system dependability and assurance
3. The project's (and organization's) goals, its environment's expectations, and the factors affecting its success
4. The organization's, project's, and individuals' readiness for particular processes

For many ongoing projects with substantial existing products, their legacy system and artifacts may place a heavy burden on the project as it tries to produce (more) secure software. This and other limitations may cause many managers to decide essentially to draw from the lightweight and legacy-oriented processes' activities to, "Do as well as we can within our constraints and keep trying to improve." While this is understandable, most software producers' projects needing (not all will need) to produce high-assurance or secure software will require a revolutionary increase in the rigor of their approach to doing software. While in the short term, decisions on initial incremental changes may be relatively straightforward, in the longer term these managers will face this need for radical change, and they may need considerable courage and skill to eventually succeed in producing high-assurance secure software.

11.6 Security Management

Secure software would best be done by trustworthy, skilled personnel in an environment with at least the level of security as required of the product.

11.6.1 Personnel Management

People who are intelligent, highly skilled, and knowledgeable about secure software may be hard to find and recruit, and require careful management to ensure retention. In addition, care needs to be taken to avoid personnel security problems.

More than routine background checks on personnel producing software where security is an important concern occur in commercial as well as government organizations. Personnel need to be resistant to succumbing to attempts at corruption or recruitment, to conceal problems, or to disclose sensitive information. Different levels of background checks may be desirable depending on a person's role. For example, one might have additional levels of checks on personnel who do ethical hacking.

Always important, properly staffing a project with requisite skills is even more important and difficult for secure software projects. A central group with in-depth security expertise may help stretch this scare resource across all the projects in need. [Lipner 2005a]

Continuing communications about the importance of security, security procedures, and what to do if approached or suspect others are necessary to maintain a proper level of attention and discipline among personnel.

Vigilance and proper processes can reduce the chances of successful subversion. This means implementing separation of duties and privileges, as well as the principle of least privilege. The rule that at least two persons are fully aware of – and completely understand – any given thing on the project must be rigorously planned and ensured. The concept of separation of duties should extend to thorough reviews at all levels, including review of the "final" version resulting in final approvals by other than the author and accompanied by rigorous (and secured) configuration management throughout, including ensuring the version reviewed and approved is the version delivered.

Project managers need to supply the environment and resources the staff needs to do high quality work and the needed information, guidance, inspiration, motivation, emotional support, perseverance, and discipline to achieve the highly demanding level of rigor. Additionally, the manager preferably will do this in a fashion that is suitable to the professionals involved and allows everyone to benefit while experiencing an acceptable quality of life.

11.6.2 Development Work Environment

A development or sustainment environment with at least the level of security required of the product is best for developing secure software. is Much of the material in Section 12, Secure Software Sustainment, intended to address concern for the sustainment and operation of the developed or acquired software applies as well to managing the computing environment and software tools for development. [Ibrahim et al, 2004] contains a section on the work environment. The work environment needs to not only be secure but also contain the proper tools and equipment for the approach taken to secure software. For more, see Section 9, Secure Software Tools and Methods.

Physical security is also a concern, as good physical security is essential to maintaining information (and industrial) security. Commercial needs or customer requirements may call for a higher than normal level of operational security.

11.6.3 Using Software from Outside the Project

Make versus acquire decisions are compounded by security requirements and needs for assurance. Section 13, Secure Software Acquisition, deals at length with these issues and managing contracts and outsourcing. Showing that security properties are preserved during the composition of parts from inside and outside the project is, of course, also an obligation.

11.7 Assuring Security Level of Software Shipped

Producers need the equivalent of an assurance case to assure themselves of the adequacy of the software they ship. In practice, one of the key differences in managing secure software development is the increased rigor of the final assessment and approval process. Additional reviews and testing may be used. [Lipner 2005] In particular, some development organizations are adhering to a strict policy of never shipping a known serious vulnerability – but what "serious" or "severe" means can vary.

Secure distribution, installation, and deployment require care with the most common techniques involving crytogrsphicly signed hashes. Some attention is given to this issue in [Howard 2006]. As discussed in Section 7, Secure Software Construction, section, one needs to ensure that the software received and installed is the software shipped.

11.8 Secure Configuration Management

Diligent configuration management (CM) of software artifacts and supporting data artifacts is critical to ensure the trustworthiness of those artifacts throughout the development lifecycle, and to eliminate opportunities for malicious developers to sabotage the security of the software. By contrast, inaccurate or incomplete CM may enable malicious developers to exploit the shortcomings in the CM process to make unauthorized or undocumented changes to the software. Lack of proper software change control, for example, could allow rogue developers to insert or substitute malicious code inserted, introduce exploitable vulnerabilities, or remove or modify security controls implemented in the software.

By tracking and controlling all of the artifacts of the software development process, CM helps ensure that changes made to those artifacts cannot compromise the trustworthiness of the software as it evolves through each phase of the process. For example, establishing a configuration baseline has a significant security implication in CM because it represents a set of critical observations and data about each development artifact, information that can then be used to compare known baseline versions with later versions, to help identify any unauthorized substitutions or modifications.

As described in NCSC-TG-006, A Guide to Understanding Configuration Management in Trusted Systems (known as the "Amber Book") and in Section B.2. of NIST SP-800-64, Security Considerations in the Information System Development Life Cycle (see Appendix B), CM should establish mechanisms to will help ensure software security, including:

- Increased accountability for the software by making its development activities more traceable;
- Impact analysis and control of changes to software and other development artifacts;
- Minimization of undesirable changes that may affect the security of the software.

Access control of software and associated artifacts are essential in providing reasonable assurance that the security of the software has not been intentionally compromised during the development process. Developers and testers should have to authenticate to the CM/version control system using strong credentials (e.g., PKI certificates, one-time passwords) before being allowed to check out or check in an artifact.

Without such access controls, developers will be able to check in and check out the development artifacts haphazardly, including those that have already undergone review and/or testing. In such an environment, the insider threat becomes a real possibility: a malicious or nefarious developer could insert spurious requirements into or delete valid requirements from the requirements specification, introduce security defects into the design, inject malicious code into the source code, and modify test plans or results to remove evidence of such sabotages. To reduce further the risk of such insider threat activities, the CM system should be one that can automatically create a digital signature and time stamp for each artifact upon check-in, so that any later unauthorized changes to the artifact can be detected easily.

Another effective countermeasure to the insider threat from developers is requiring that every configuration item be checked into the CM system as a baseline before it is reviewed or tested. In this way, as changes are made based on findings of the review/test, the new configuration item that results can easily be compared against the pre-review/pre-test baseline to determine whether those changes also included unintentional vulnerabilities or malicious elements. Two closely related principles that should be applied to CM are separation of roles and separation of duties. The development, testing, and production environments, and their corresponding personnel, should be assigned different, non-contiguous roles with separate access rights in the CM system. In practical terms, this means that developers will never have access to code that is in the testing or production phase of the lifecycle.

Further information on configuration management and change control within the secure sustainment phase of the lifecyle appears in sections 12.5.2 and 12.5.3.

11.8.1 Using CM to Prevent Malicious Code Insertion During Development

Uncontrolled software development lifecycle activities are susceptible to malicious software developers, testers, or intruders surreptitiously inserting malicious code, or backdoors that can later be used to insert malicious code, into the software. For this reason, all source code, binary executables, and documentation should be kept under strict configuration management control.

Developers and testers should be required to authenticate themselves to a version control system using strong credentials, such as digital certificates or strong passwords, before checking out or submitting source code, executables, or documentation. Use only configuration management/version control software that can apply a digital signature to all software and documentation files, so that the configuration manager, other developers, and testers can quickly detect any unauthorized changes.

Diligent configuration control of the software development and testing processes is critical to ensure the trustworthiness of code. The software development lifecycle offers multiple opportunities for malicious insiders to sabotage the integrity of the source code, executables, and documentation.

A basic principle that should be applied is the one of separation of duties. Different environment types—development, testing, and production—and their corresponding personnel need to be kept separate, and the associated functionality and operations should not overlap. Developers should never have access to the software that is in production.

During testing, from-scratch code should be examined for exploitable defects such as buffer overflows and format string errors. Software developers should be given awareness training in how common vulnerabilities manifest in software, and how to avoid them.

Malicious developers who purposely plant such defects have plausible deniability: they can always claim that the defects were simple errors. Work to prevent such problems by making sure that at least one other set of eyes besides the developer's peer reviews all code before it moves on to testing, and use a "multilevel commit" approach to checking code and documents into the version control system.

The developer should make sure code and documentation is checked in to the version control system before it goes out for review/testing. This prevents a malicious developer from being able to surreptitiously insert changes into code (or documentation) before the approved version is checked in to the CM system. Instead, the approved version is the version already in the CM system, i.e., the same version the reviewers/testers examined. Ideally, every file would be digitally signed by the developer when submitted to the CM system to provide even stronger defense-in-depth against unauthorized changes.

Software testing even of code developed from scratch should include black box analysis to verify that the software does not manifest any unexpected behaviors in execution. The software quality assurance process should also include white box analysis (code review), focusing on hunting down any extra, and unexpected logic branches associated with user input, which could be a sign of a backdoor planted in the code during the development process.

All issues addressed and specific solutions to problems encountered during all phases of the lifecycle need to be properly documented. Security training among software developers and testers must be encouraged. Security background checks should be performed on all non-cleared software developers and testers.

11.9 Software Quality Assurance and Security

In a secure software development process, quality assurance practitioners must always have security in mind. They must be very skeptical of the accuracy and thoroughness of any security-related requirements in the software's specification. In short, they must be willing to adapt their requirements-driven mentality to introduce some risk-driven thinking into their verification processes. The quality assurance process, then, will necessarily incorporate some risk-management activities focusing on "secure in deployment" objectives:

- **Configuration management of patches (patch management):** Must extend to security patches, to ensure that they are applied in a timely manner (both to the commercial and open source software in the software system itself and to its execution environment), and that interim risk analyses are performed to determine the impact the patch will have and to identify any conflicts that may be caused by applying the patch, particularly those impacts/conflicts with security implications, to mitigate those effects to the extent possible (which, in some cases, may mean not installing the patch because its impact/conflicts may put the software at greater risk than the vulnerability the patch is meant to address). See section 12.5.3 for more information on this topic.
- **File system clean-ups:** All server file systems are reviewed frequently, and extraneous files are removed, to prevent avoidable future conflicts (resulting from patching or new component releases);
- **Security "refresh" testing:** The software's continued correct and secure operation is verified any time any component or configuration parameter of its execution environment or infrastructure changes;
- **Security auditing:** The software's security configuration is periodically audited, to ensure that the file permissions, user account privileges, configuration settings, logging and auditing, etc., continue to be correct and to achieve their security objectives, considering any changes in the threat environment. See sections 8.11 and 12.5.4 for further discussion of security audits for software.

In addition to these activities, quality assurance practitioners should periodically audit the correctness of the performance of these procedures.

11.10 Further Reading[1]

11.10.1 Secure Software Engineering Management

[PMBOK 2004] Bolles, D., and Fahrenkrog, S. *A Guide to the Project Management Body of Knowledge (PMBOK*— ANSI/PMI 99-001-2004. Third Edition. Newton Square, PA.: Project Management Institute, Inc. 2004

Gilb, Tom. Principles of Software Engineering Management. Boston: Addison-Wesley, 1988.

Futrell, Robert T., Donald F. Shafer and Linda I. Shafer. Quality Software Project Management, First Edition. Upper Saddle River, NJ: Prentice Hall Professional Technical Reference, 2002.

[1] For more material see the management section of the Build-Security-In website http://buildsecurityin.us-cert.gov

Thayer, Richard H., editor, and Edward Yourdon. Software Engineering Project Management, Second Edition (Paperback) Hoboken, NJ: Wiley-IEEE Computer Society Press, 2000.

Williams, S. and P. Fagan, “Secure Software: Management Control Is a Key Issue”. London, UK: Institute o Electrical Engineers (IEE) Colloquium on Designing Secure Systems, June 1992.

[Baskerville 2005a] Baskerville, R., and Portougal, V. “Possibility Theory in Protecting National Information Infrastructure.” In K. Siau (Ed.), *Advanced Topics in Database Research* (Vol. 4). Idea Group, 2005.

[Baskerville 2005b] Baskerville, R., and Sainsbury, R. “Securing Against the Possibility of an Improbable Event: Concepts for Managing Predictable Threats and Normal Compromises.” *European Conference on Information Warfare and Security*, Glamorgan University, UK, 11-12 July 2005.

[Feathers 2005] Feathers, M.C., *Working Effectively with Legacy Code*, Prentice Hall, 2005.

11.10.2 Secure Configuration Management

Leon, Alexis. Software Configuration Management Handbook, Second Edition. Norwood, MA: Artech House Publishers, 2004.

Wheeler, David A., “Software Configuration Management (SCM) Security”. May 2005. http://www.dwheeler.com/essays/scm-security.html

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

Devanbu, P., M. Gertz and Stuart Stubblebine. Security for Automated, Distributed Configuration Management. Proceedings, ICSE 99 Workshop on Software Engineering over the Internet, 1999. http://www.stubblebine.com/99icse-workshop-stubblebine.pdf

NCSC-TG-006-88. A Guide to Understanding Configuration Management in Trusted Systems. National Computer Security Center, 1988.

Mell, Peter, Tiffany Bergeron and David Henning, “NIST Special Publication 800-40, Creating a Patch and Vulnerability Management Program, Version 2.0”. NIST, November 2005.

11.10.3 Software Quality Assurance and Security

Godbole, Nina S. Software Quality Assurance: Principles And Practice. Oxford, UK: Alpha Science International, Ltd., 2004.

Dustin, Elfriede, Jeff Rashka and Douglas McDiarmid. Quality Web Systems: Performance, Security, and Usability, First Edition. Boston, MA: Addison-Wesley Professional, 2001.

[OWASP 2006] Open Web Application Security Project. “Software Quality Assurance” chapter in A Guide to Building Secure Web Applications and Web Services, 2.1 (Draft 3). OWASP Foundation, February 2006.
http://www.owasp.org/index.php/Software_Quality_Assurance

12Secure Software Sustainment

12.1 Introduction

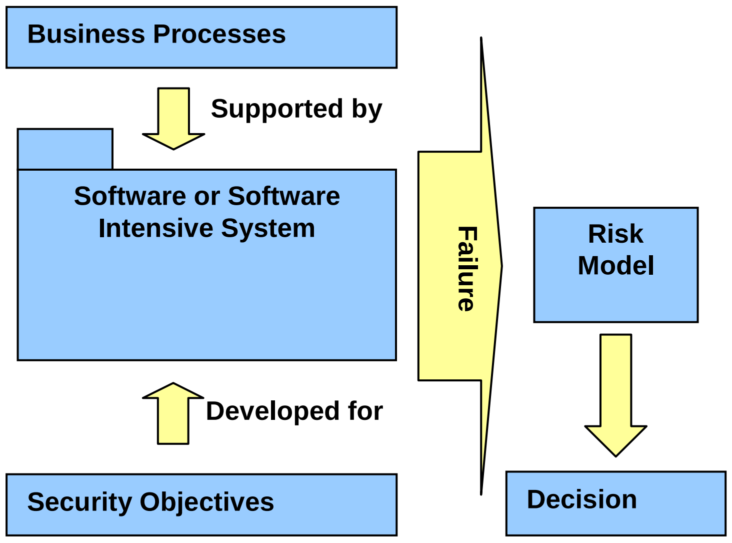
This section covers software assurance activities when software is put into an operational environment and the unique software assurance activities after initial deployment. Sustainment involves processes that continue to assure that software satisfies its intended purpose after initial deployment and during operations. [Berg 2005, Chapter 10]

Software is written to facilitate organizational goals and each goal has security requirements. If the software does not meet those requirements, the organization must realign the functioning of the software, or alter the goal to bring it into alignment. The security environment is continuously changing. Environmental factors that could impact security include:

- People connecting to the system's endpoints and the motivations of those people
- Systems interconnected with the software
- The type of data flowing to, through, or from the software, or system
- The way the organization does business, or the type of business that is conducted
- The rigor, or extent of the security objectives
- The organizational risk model/risk tolerance

Software must always satisfy security objectives. If that is not the case, then the organization's risk mitigation process must dictate the steps necessary to change the software, or change the objectives.

Figure 5: The Environment of Software or Software Intensive Systems



Certain aspects of sustainment are also related to Section 13, Acquisition of Secure Software. Those sections will be cross-referenced in this section. The following is an outline of the contents of this section:

- Background
- Operational Assurance (Sensing)

- Analysis
- Response Management (Responding)
- Infrastructure Assurance

12.2 Background

Because environment constantly changes, confidence in the security of the software must be continuously renewed. In addition, because software is increasingly inter-connected, an appropriate level of confidence must also be established for the entire portfolio of software.

Typically, sustainment monitors the system's (and portfolio of software's) ability to effectively sustain confidence in the security of the software. Sustainment monitors the system's ability to effectively deliver services, identify and record problems, analyze those problems, take the appropriate corrective, adaptive, perfective, or preventive action and confirm the restored capability. Sustainment activities also encompass the migration and retirement of the software. The process ends when the software or software-intensive system is retired. Also see [ISO/IEC 15288] and [IEEE 12207].

Sustainment is either proactive or reactive. Proactive activities include identifying of threats and vulnerabilities; creating, assessing, and optimizing security solutions (within a generalized security architecture); and implementing controls to protect the software and the information that it processes. Reactive activities include threat response and the detection and reaction to external or internal intrusions or security violations.

Both proactive and reactive security activities are supported by sensing, analyzing, and responding functions. The sensing function involves monitoring, testing, and assessment. The goal of sensing is to identify intrusions (e.g., a break in), violations (e.g., an inappropriate access) and vulnerabilities (a weakness). The analyzing function facilitates understanding the risk. It assesses the frequency of occurrence and impact. It considers and evaluates all risk avoidance, risk mitigation, and risk transfer options. The responding function selects and authorizes the remediation option. It monitors and assures the change and ensures the correct re-integration of the altered code, system settings, or policies. Management must then make certain that the necessary resources are available to ensure the authorized level of risk mitigation.

12.2.1 Types of Response

With proactive assurance, decision makers have the option of authorizing a preventive, perfective, or adaptive response. With reactive assurance, decision makers have the option of authorizing corrective, or emergency action.

Corrective change involves identifying and removing vulnerabilities and correcting actual errors. Preventive change involves the identification and detection of latent vulnerabilities (e.g., the software or software-intensive system is found to be vulnerable to a particular class of intrusion). Perfective change involves the improvement of performance, dependability and maintainability.

Adaptive change adapts the software to a new or changed environment (e.g., a new operating system with enhanced security functionality is available). Emergency change involves unscheduled corrective action (e.g., an intrusion or violation has taken place).

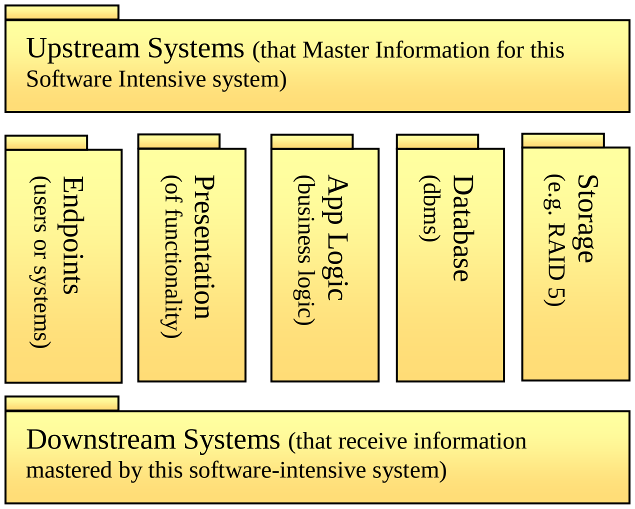
12.2.2 Representation

Sustainment presupposes complete, accurate, and unambiguous representation of the software, or system at its most basic level of functioning (atomistic representation). In addition, the interconnection between the software and the entire portfolio of software of systems must also be fully understood (holistic representation).

The software's responsibility for receiving, processing, or transmitting information must be known as well as the status of all of the attached users (or other pieces of software).

Figure 6 shows (at the highest level) how this is represented. Drilling down into each element of the diagram would produce a detailed understanding of the actual nodes with all of their installed components and inter-relationships.

Figure 6: High level representation of a system


Upstream Systems (that Master Information for this Software Intensive system)
Endpoints (users or systems)
Presentation (of functionality)
App Logic (business logic)
Database (dbms)
Storage (e.g. RAID 5)
Downstream Systems (that receive information mastered by this software-intensive system)

12.3 Operational Assurance (Sensing)

Operational assurance is primarily a proactive sustainment function(sensing) that encompasses using defined policies, procedures, tools, and standards to monitor, test, and review the software or system. This is done continuously within the operating environment. Standard operational assurance activities are specified within the larger context of the Operation Primary Process [e.g., ISO/IEC 12207, 5.4 – Operation Process].

Operational assurance also identifies and resolves security and control vulnerabilities within the software, the system, the data, the policies, and the users. Because vulnerabilities can be associated with application software, operating system software, network or device configuration, policies and procedures, security mechanisms, physical security, and employee usage, operational assurance is not limited to software alone, but extends into the operating environment that surrounds the software.

Technical assurance practices include intrusion detection, penetration testing, and violation processing using clipping levels. Reviewing is a periodic activity that evaluates the software, the system, the policies and procedures, and the users' usage against established standards. Reviews may consist of walkthroughs, inspections, or audits. They can be both managerial and technical [IEEE 1028].

Identified threats, vulnerabilities, and violations are recorded and reported using a defined problem reporting and problem resolution process [e.g., ISO/IEC 12207 section 6.8 – Problem Resolution Process]. The problem resolution process should be tailored to allow decision makers to authorize multi-dimensional responses based on their assessment of risk.

The policies, procedures, tools, and standards used for operational assurance are also continuously assessed so that recommendations can be made to improve or enhance them [e.g., ISO/IEC 12207 section 7.3 – Improvement process].

12.3.1 Initiation

Software must be in a secure state and that state must be understood and documented to carry out operational assurance. Upon initiation of operational assurance process, it is necessary to validate the security of the installation and configuration of the software and that all security functions are enabled. It is a requisite of good practice to:

- Identify feasible security perimeter(s) and defense in depth [Riggs 2003]
- Document an overall concept of operations.
- Prepare an operational testing plan.
- Prepare a policy to ensure appropriate response to unexpected incidents.
- Prepare a secure site plan.
- Prepare a Business Continuity Plan and a Disaster Recovery Plan (BCP/DRP) with Recovery Time Objectives (RTO), Network Recovery Objectives (NRO) and Recovery Point Objectives (RPO) fully established for every item within the secure perimeter.
- Ensure the system staff is adequately trained in secure operation.
- Ensure the system staff is capable of utilizing all embedded security functionality.
- Identify a valid security accreditation process and obtain certification of security of the operational system.

12.3.2 Operational Testing

It is necessary to monitor the ongoing functioning of the software, or system, within the operational environment because threats can arise at any point in the process and can represent a range of unanticipated hazards. This must be done on a disciplined and regularly scheduled basis. Therefore, a requisite of good practice is to deploy a continuous operational testing process to identify security threats and vulnerabilities and control violations in software and software-intensive systems [e.g., ISO/IEC 12207 section 5.4.2 - Operational testing].

12.3.3 Environmental Monitoring

Environmental threats exist in the context in which the software functions. In that respect, the environment represents the place where early warning of impending hazards or attacks can be best obtained. Therefore, a requisite of good practice is to continuously monitor the operating environment that surrounds the software to identify and respond to security threats, exposures, vulnerabilities, and violations as they arise (threat identification).

12.3.4 Incident Reporting

Incidents must be reported through a standard and disciplined process. The aim is to respond as quickly as possible to trouble arising from vulnerabilities, malfunctions, or incidents that might occur as a result of attempts to exploit those vulnerabilities or malfunctions. The process must be both standard in its procedure and fully documented.

The process also must be well understood within the organization. Therefore, a requisite of good practice is to institute a systematic procedure to document and record threat exposures and vulnerabilities (trouble reporting).

12.3.5 Reporting Vulnerabilities

Producers who do not actively seek to identify and repair vulnerabilities in their products, or who do not report vulnerabilities that they know about, increase the chances that customers and users will suffer serious harm with accompany harm to the producer’s reputation or business. On the other hand, restricting early access to vulnerability information may be needed in practice to prevent dangerous leakage to attackers.

Therefore, a requisite of good practice is to identify and report all vulnerabilities possibly to a central entity, but not necessarily wide disclosure: Section 12.5.1, Responding to Known Vulnerabilities, discusses reporting and response issues at more length. Identification occurs in operational assurance (sensing). The classification, prioritization, and development of remediation options occurs in analysis.

12.3.6 Operational Process Assurance

It is necessary to ensure that operational assurance is carried out in the most effective and efficient manner. As such, the functioning of the operational assurance process itself must be continuously monitored. The aim is to identify and report any deviation from proper practice to management for remediation. As such, it is a requisite of good practice to:

- Assess and audit the policies, procedures, tools, and standards used for operational assurance.
- Document assessments and audits and make recommendations for improvement to the designated approving authority [e.g., ISO/IEC 12207 section 7.3 – Improvement Process].

12.3.7 Assurance Case Evidence for Operational Assurance

A variety of evidence is relevant to creating and maintaining an assurance case for operational assurance, including:

- Evidence of an organizationally standard operational procedure manual that details the required steps for every activity in operational assurance, including expected results and some way to determine that they have been achieved.
- Evidence of a tangible set of organizationally sanctioned actions, procedures, or protocols invoked when anticipated hazards occur.
- Evidence of a tangible set of organizationally sanctioned actions, procedures, or protocols invoked when unforeseen hazards occur.
- Documenting the specific method for incident reporting, or requesting change and the procedures for responding to each report.
- The process for ensuring that the Business Continuity Plan is up to date.
- Evidence that all of relevant members of the organization know precisely what activities they have to carry out in Sustainment and the timing requirements for performing them.
  - Documenting the precise steps taken to build awareness of correct practice, including a formal employee education and training program
  - Documenting each employee’s specific education, training, and awareness activities
  - Documenting the explicit enforcement requirements and consequences for non-compliance for every job title
  - Specification and evidence of personal agreement to the consequences for non-compliance
- Evidence that enforcement was practiced on a continuous basis and as an organization wide commitment

12.4 Analysis

The analysis section evaluates the consequences of an identified threat or violation along with the impact of recommended changes. All software, software-intensive systems, policies, processes, or objectives impacted by the change must be included in the evaluation.

This is necessary to ensure a coordinated response. Analysis entails identification of the affected software and systems (to include cascading or ripple effects) along with affected policies or processes.

Affected software and systems elements are studied to determine impacts of a prospective change. Impacts on existing software and systems, as well as any interfacing systems and organizational functioning are characterized. Security and safety impacts of the change must be fully examined and documented and communicated to the Response Management Function for authorization.

In addition, to determining impacts, the results of the analysis are formally recorded and maintained in a permanent repository. This increases understanding of the content and structure of the portfolio of software. In addition, this supports retrospective causal analysis that could be undertaken to understand security and control issues associated with the change.

12.4.1 Understanding

To implement a change properly, it is essential to understand all of the components and the consequences of change to them. That degree of understanding requires knowledge of all aspects of the design architecture and the affected code. To support this change, it is good practice to:

- Document the problem/modification request and capture all requisite data in standard form as specified by [IEEE/EIA 12207.1, 6.2].
- Replicate or verify the existence of the reported problem – for the sake of resource coordination confirm that the problem really exists.
- Verify the violation, exposure, or vulnerability – understand the precise nature and implications of the vulnerability and develop an overall response strategy.
- Identify elements to be modified in the existing system – identify all system components that will be changed – develop a specific response strategy for each element using good development practices.
- Identify interface elements affected by the modification.
- Estimate the impact of change to the software on system interfaces – perform impact analysis on affected interfaces and, from that, design a specific response strategy for each interface using good development practices.
- Identify documentation to be updated.
- Identify relevant security policies – validate recommended response strategy against relevant security and safety policy.
- Identify relevant legal, regulatory, and forensic requirements – validate the recommended response strategy against relevant legal, regulatory, or forensic requirements.

These steps help ensure proper change implementation decisions for a component or system.

12.4.2 Impact Analysis

As indicated in the previous subsection, to implement a specifically tailored response, it is necessary to know what the implications of a particular response strategy or action might be. That level of knowledge requires a comprehensive and detailed impact analysis. This should be based on a formal methodology ensuring a comprehensive and unambiguous understanding of all operational implications for the software, its requirements, and its associated architecture. Therefore, for each remediation option:

- Identify the impact of change on the assurance case.
- Identify the violation, exposure, or vulnerability type – the threat is explicitly classified by type.

- Identify the scope of the violation, exposure, or vulnerability – the extent or boundary of the threat is fully and explicitly itemized.
- Provide a formal statement of the criticality of the violation, exposure, or vulnerability.
- Document all feasible options for analysis.
- Perform a comprehensive risk identification – identification of the type and extent of risk for each option.
- Perform a detailed risk evaluation – assess the likelihood and feasibility of each identified risk for each option.
- Estimate safety and security impacts if change is implemented – based on likelihood percentages and feasibility for each option.
- Estimate the safety and security impacts if change *is not* implemented – based on the likelihood of occurrence of financial and operational impacts of each identified option.
- Assess the impact of change on security and control architecture.
- Perform software understanding and design description exercise for all automated security and control features.
- Estimate and assess the implications of change as they impact the policy and procedure infrastructure.
- Estimate the impact of change on the Business Continuity/Disaster Recovery strategy.
- Specify feasible recovery time, NRO, and recovery point impact estimates for each option.
- Estimate the return on investment for each option, including total cost of ownership and marginal loss percentage.
- Estimate the level of test and evaluation commitment necessary for verification and validation.
- For each option, prepare a testing program – sample test cases and methods of administration.
- Estimate the resource requirements, staff capability, and feasibility of administration of tests.
- Estimate the financial impacts where appropriate for each option.
- Estimate the feasibility and timelines for implementing each option.
- Prepare a project plan for each option if detailed level of understanding required.

Once the options have been investigated, a basis for decision-making exists.

12.4.3 Reporting

To support explicit decision-making on the response, the body-of-evidence (developed in the analysis phase) must be communicated in an understandable fashion to the designated approving authority (DAA) for authorization of the change. For each change requested, the nominal correct practice is to:

- Determine or designate the appropriate decision maker – This identification may be based on the results of the analysis phase, or carried out as a result of pre-designation.
- Decision-making may also be done through a pre-selected control board composed of the appropriate decision makers.
- If the decision is significant enough, it may also be decided through a singular process instituted at the conclusion of the analysis phase.

The results of the analysis are reported to the DAA with a full explanation of the implementation requirements for each remediation option.

- This report must clearly outline impacts of each option and it must be plainly and explicitly understandable to lay-decision makers.
- The feasible remediation options must be itemized.
- These must be expressed in a manner that is understandable to lay-decision makers and each option recommended must be fully and demonstrably traceable to the business case.

All of this prepares for the responses discussed in the next subsection.

12.5 Response Management (Responding)

Response management entails processes specified within the larger context of the change process [e.g., ISO/IEC 12207 section 5.5, Maintenance Process]. Response management involves coordination and assurance of the remediation option. The development process or the acquisition process is the actual agent of change. Also see the Development and Acquisition sections and [ISO12207].

Policies, tools, and standards that are employed for response management are also continuously assessed so that recommendations can be made to improve or enhance them, e.g.,[ISO/IEC 12207 section 7.3 – Improvement process] and [Krutz 2004].

12.5.1 Responding to Known Vulnerabilities

12.5.1.1 Responding to Known Vulnerabilities: Patching

The duty of response management is to maintain the security and integrity of the software throughout its useful lifetime. The reality is that over that lifetime, a significant number of new vulnerabilities, which might threaten that security will be discovered.

Vulnerabilities might be discovered through explicit investigation by security professionals, software vendors, white-hat hackers, internal members of an organization, or any other interested party, including exploits by the black-hat community [Havana, 2003]. Whatever the source, any vulnerability that has been discovered requires risk management decisions on patching or other risk mitigations.

To ensure against revealing the existence of a vulnerability before a patch or remediation option is developed, it is a requirement of the process that the finder and vendor communicate securely with each other throughout all phases [OIS, 2004]. The discovery process will entail the following stages [OIS, 2004]:

- Discovery – Finder discovers a security vulnerability.
- Notification- Finder notifies vendor to advise the potential vulnerability. or flaw. Vendor confirms the receipt of the notification
- Investigation – Vendor investigates the finder’s reports to verify and validate the vulnerability. in collaboration with the finder
- Resolution – If the vulnerability is confirmed, the vendor will develop a remediation option (software patch or change procedure) to eliminate the vulnerability.
- Release – Vendor and finder will coordinate and publicly release information about the vulnerability and remedy, including the requisite patch (if applicable).

12.5.1.2 Responding to Known Vulnerabilities: Reporting

If a vulnerability is discovered, it is the duty of the response management function (of the organization that has discovered it) to disclose its existence in a way that will ensure users of the software are not harmed. The risk is that, if the vulnerability is simply reported, or reported too soon, the black hat community will exploit the problem before the vendor, or finder, can discover a way to patch, or mitigate it. As such, it is not correct practice to provide full and immediate public disclosure of a vulnerability without fully exploring the risk mitigation options. That understanding is obtained through the Impact Analysis (0.4.2)

Depending on the risk mitigation situation, there are three ways to disclose the existence of a new vulnerability [Havana, et al. 2003]. All three of these entail risks, which must be considered; however, partial disclosures are considered to be more appropriate [Havana, 2003].

- Wide public disclosure. This involves publishing a full report to a wide audience where system security experts decide on actions to take. This approach is subject to exploitation by criminals.
- Limited public disclosure. Only one organization is informed either because the remediation is only required there – or to research appropriate responses. That limits the potential for exploitation.
- Total limited public disclosure. Only a selected group is informed, for example, only the security group in a particular organization, until an appropriate response is developed.

12.5.1.3 Responding to Known Vulnerabilities without Fixes

There might be operational justification for not fixing, or reporting a vulnerability or for not fixing it or reporting it as soon as possible. These justifications might include such reasons as the lack of obvious harm, or the amount of time and resources required to develop the fix will outweigh any potential cost should the threat occur. It might also include lack of current resources; difficulty or infeasibility of the repair, or an unwillingness to take down a critical operational system.

However, it is essential that known vulnerabilities are monitored and managed. Therefore, a requisite of good practice is to:

- Maintain continuous records of publicly known vulnerabilities..
- Maintain a continuous record of privately known vulnerabilities.
- Monitor operational behavior of the system to detect and recognize the signature of any attempt to exploit a known vulnerability.
- Set automated alarms to inform of an attempt to exploit a known vulnerability.
- Maintain a systematic and well-defined response to any expected attempt to exploit a known vulnerability.
- Ensure that the system staff understands the proper response to an attempt to exploit a known vulnerability.

12.5.2 Change Control

The agent performing a fix must understand all of the requirements and restrictions involved in making the change. Thus, a process must be established to unambiguously convey all of the technical and contextual specifications of the remediation option to the change agent. Organizationally persistent controls must be put in place to ensure that this is done in a standard and disciplined fashion. Therefore. it is good practice to: [ISO04, p.17, ISO96, p. 25]

- Identify the appropriate change agent – this may be either an acquisition or a development entity.

- Develop and document a Statement of Work (SOW) and specify the precise method for communicating this to the change agent.
- Develop and document criteria for testing and evaluating the software or software-intensive system to ensure successful remediation.
- Communicate these to the change agent prior to instituting the change process.
- Develop and document criteria for ensuring that elements and requirements that should not be modified remain unaffected.

12.5.3 Post-Change Analysis

Changes to software can eliminate vulnerabilities. However, change can also create new ones. As a consequence, attackers typically examine changed code to identify any new or different methods of exploitation. Therefore, the system staff must also continue to monitor and analyze the impact of a change. This must occur even after the change has been made. The aim is to understand all of the long-term consequences. As such, it is a requisite of good practice to:

- Perform an analysis of the changed code within its operational environment to identify potential points of future exploitation.
- Execute pen testing, load testing, or other stress-testing exercises to identify any new points of weakness, or failure.
- Continue to update the countermeasure set to enforce the security status and requirements of the changed software.
- Modify the assurance case as appropriate.
- Ensure that all required operational integration and testing processes have been executed on the changed code.
- Ensure that all test results are reported to the appropriate DAA for resolution.

12.5.4 Change Assurance

Although the change is implemented through the agency of another process (either Acquisition, or Development) the change assurance process ensures that the change meets established criteria [ISO04, p.17, ISO96, p. 25]. This is a continuous process, involving the joint review and problem resolution processes as specified by [IEEE/EIA 12207]. Therefore, it is good practice to:

- Monitor the change through joint reviews as specified in the SOW.
- Ensure that all reviews specified by the SOW are conducted at their scheduled check points.
- Ensure that action items issuing out of each review are recorded for further action.
- Ensure that closure criteria are specified.
- Perform audits as specified in the SOW.
- Ensure any audit specified in the SOW is properly resourced and performed.
- Ensure that any audit or review findings involving non-concurrence are resolved.
- Monitor service levels as agreed to by contract (see the Acquisition section on SLAs).
- Oversee the execution of the contract to ensure that required service levels are maintained (see Section 13, Acquisition of Secure Software).

- Ensure that problems identified through joint reviews and audits are resolved.
- Baseline and track the number and type of vulnerabilities over time to verify that audit and remediation programs are demonstrating positive return on investment.

The DAA certify that any non-concurrences issuing out of the review process are addressed and that all closure criteria have been satisfied.

Following delivery of the change, it is a requisite of good practice to:

- Perform tests to ensure correctness in accordance with test and evaluation criteria [ISO96, p.29].
- Conduct a verification and validation program to sufficiently ensure the correctness and integrity of operation of the change as delivered.
- Ensure functional completeness against requirements [ISO96, p.29].
- Ensure that the change satisfies all function requirements specified in the SOW.
- Ensure physical completeness against technical description [ISO96, p.29]. Validate correctness of the change against technical description documents.

Documenting the assurance case requires an audit trail that provides traceability between the change authorization and the delivered product [ISO96, p.29].

12.5.5 Assurance Case Evidence for Response Management

A variety of evidence is relevant to creating and maintaining an assurance case for response management, including:

- Evidence that resource considerations are factored into impact analyses and change authorizations.
- Evidence that every change has been authorized.
- Documentation of a formal schedule, or timetable for each change.
- Evidence that a formal configuration management plan exists that itemizes the change management, baseline management, and verification management functions, as well as documents how the configuration identification scheme will be formulated and ensured.
- Evidence of a capable status accounting function comprising established baselines for each software item that documents the current state of the software at all times.
- Documentation that a baseline management ledger (BML) account exists for each controlled entity in the software asset base.

12.5.6 Change Re-integration

Changes must be re-integrated into the operational system. The decision-maker who authorized a change must provide the approval to perform the final re-integration. This approval must be underwritten by the findings of the Change Assurance Process (0.4.4).

Once authorization is granted, the change is re-integrated into the operational system. It is then necessary to conduct a technically rigorous process to assure that this re-integration has been done correctly.

This re-integration is supported by a comprehensive testing program. The re-integration testing program is specified at the point where the change agent prepares the plan to perform the work (see 0.4.2 Change Control). The testing certifies that the re-integration is satisfactory and that all interfaces are functioning properly [ISO96, p. 30].

12.5.7 Configuration Management

In addition to certifying of the correctness of the re-integration, it is necessary to fully document the new software's baseline configuration and then maintain it under strict configuration control.

The documentation is kept as a current baseline configuration description. That baseline is stored in an organizationally designated repository [ISO04, p. 21]. Therefore, it is good practice to:

- Confirm reintegration maintains correct level of integrity and security [ISO04, p.21]. This confirmation is certified by the results of the testing program.
- Ensure documentation is updated at the requisite level of integrity and security (including the assurance case) [ISO04, p.21]. This is assured by confirming that the new baseline has been satisfactorily represented in a controlled baseline repository.
- Ensure changed items maintain backward and forward traceability to other baselined states [ISO04, p.21]. This is assured by maintaining a repository of prior baseline states that is typically called a static library, or archive.
- Ensure the configuration record is securely maintained throughout the lifecycle and archived according to good practice. [ISO04, p.21]

12.5.8 Recertification and Accreditation

To ensure confidence, software systems must be assessed and authorized by an appropriate third-party agent using a commonly accepted process. The findings of that process are typically accredited by formal certification.

Re-accreditation of the results of a certification process should be obtained periodically to ensure continuing confidence. Intervals for re-accreditation are typically specified by organizational policy and/or external regulation. Therefore:

- A legitimate third-party agency must be employed to conduct certification audits – the audit standard should be established by regulation or contract.
- Assessments for certification/re-certification of accreditation must be performed by properly certified lead auditors.
- Adequate resources must be provided to ensure the effectiveness of the audit process.
- The independence of the auditing/accrediting agency must be assured.
- Consistent use of a standard audit method must be assured.
- Re-certification audits must be performed timely enough to ensure continuing confidence.

12.5.9 Secure Migration, Retirement, Loss, and Disposal

Migration and retirement of software and systems must be controlled by organizationally standard and rigorous processes [ISO96, pp.25-26, ISO04, pp.37-38, pp.47-48]. This ensures that major changes are controlled using a rational and secure process. It also ensures that the overall portfolio of software will continue to meet its intended purpose. Therefore, it must be ensured that:

- Migration and retirement risks and impacts must be known [ISO96, pp.25-26] – –that knowledge is developed through the agency of the analysis function.
- A transition strategy must be developed that assures system services will be maintained during transition [ISO04, p.33].

- The safety and security aspects of the transition must be fully and explicitly studied, characterized, and understood.

A DAA must authorize migration or retirement strategy.

Tests and inspections must be executed to explicitly assure:

- Transition of the software or system [ISO04, p.33] is accomplished in a safe and secure fashion.
- The transition process is confirmed effective and correct.
- The proper functioning of the software, or system after transition [ISO04, p.33].
- The effectiveness of the transition is confirmed and certified by a satisfactory verification and validation procedure.
- The results of the transition process are documented and retained.
- The integrity of the software or system is confirmed after transition [ISO04, p.33].
- All software operation and data integrity is confirmed by an appropriate set of measures.
- The results of the program and data integrity checking process are documented and retained.
- Software or system documentation accurately reflects the changed state.

12.6 Infrastructure Assurance

Infrastructure assurance involves processes that apply, coordinate, and sustain operational assurance, analysis, and response management. Infrastructure assurance ensures that the organization has a planned and documented assurance case and security architecture as well as tangible policies, processes, and methodologies that establish operational assurance, analysis, and response management.

The organization's security architecture is the composite of all solutions the organization devised to provide tangible preventing, sensing, analysis and response to threats. Security architecture must be holistic and complete. Establishing and maintaining a security architecture also establishes and maintains the interrelationships among those solutions.

Standard policies, processes, and methodologies are necessary to ensure that the actions of the organization will be appropriate. They are altered to maintain a correct response to changes in the contextual situation (i.e., to maintain alignment between security objectives and the software). These policies, processes and methodologies constitute the tangible elements of the Sustainment process.

Besides creating a tangible sustainment infrastructure, policies, processes, and procedures also ensure continuous coordination between the Sustainment function and the Acquisition and Development functions.

12.6.1 Security Architecture

Security architecture involves the development, deployment, and continuous maintenance of the most appropriate and capable security solutions, tools, frameworks, and components. The aim of security architecture is to maintain a dynamic security response to threats and changes. To accomplish this:

- Standard security and control processes must be planned, designed, administered, and maintained. The aim is to ensure that effective leadership vision, expertise, and the correct technology solutions are available to assure security and control of applications and infrastructure.
- Standard procedures are in place to assess, integrate, and optimize security architecture – the security architecture must be assessed regularly and systematically to ensure that it remains effective.

- Organization-wide evaluations are done to ensure the continuous state of the security architecture within the enterprise. The security architecture must be evaluated to ensure that it remains effective.
- Future security trends are evaluated to define security architecture strategy. Security trends are assessed as they impact the evolution of the security architecture of the organization.
- Consultation and strategy resources are made available to ensure the effectiveness of the security architecture. Expertise is provided to lay-practitioners to ensure a minimum acceptable awareness of the implications and requirements of security architecture.
- Directions are evaluated as they relate to the acquisition of security architecture components. From this evaluation, rational decisions are made on the most effective purchase of products and services and the in-house development of tools.
- The security aspects of the acquisition process must be continuously ensured to be in conformance with all assurance practices defined by the organization.
- On-call technical support must be provided for security architecture. Any questions or concerns raised in the day-to-day maintenance of the security architecture must be responded as Do-It-Now (DIN) requests.
- Enterprise on security and control must be maintained as a readily available knowledge base. This must be kept in an information store that will share, as well as cascade security architecture knowledge and information to all levels in the organization.

12.6.2 Policy, Process, and Methodology Assurance

To ensure security, the most appropriate and capable set of policies, processes, and methodologies must be developed, deployed, and sustained. The aim is to maintain a dynamically effective security and controls architecture. To ensure this:

- The most current security methodologies, processes, and associated documentation are developed and deployed and kept in an organizationally standard and accessible repository.
- Cross-organization collaboration must be established to communicate security practices by developing and implementing training materials. These materials must be organizationally standard. Their coordination must be carried out by a formal process.
- Security metrics must be developed and collected. Security metrics must be standard. They must be used for causal analysis to optimize the ongoing security process.
- Formal teams must be established and coached in how to apply the organizationally standard methodologies and processes. This item also assists with assessing compliance.
- Metrics to improve methodology and process efficiency, usage, and results must be defined and analyzed.
- Modeling of application and infrastructure must be done from a security perspective.
- Ongoing security and control certifications of applications and infrastructure must be enforced.
- Security and control awareness, knowledge of policies, procedures, tools, and standards must be championed and promoted.

Altogether, to minimize risks the infrastructure needs careful attention to its security aspects.

12.6.3 Assurance Case Evidence for Infrastructure Assurance

A variety of evidence is relevant to creating and maintaining an assurance case for infrastructure assurance, including:

- Evidence of Sustainment function’s role in formulating strategic security requirements
- Evidence that Sustainment operational plan exists and is current
  - Documentation of assumptions about current, known risks and threats
  - Documentation of organization-wide standards, or standard practices
  - Specification of the technologies and products that will be utilized during the planning period, along with the method for installing, maintaining, and operating them on a secure basis
- Evidence that all security updates have been verified, tested, and installed in a timely fashion
- Evidence of an organization-wide information sharing process
  - Evidence that the information sharing process is revised and updated as the security situation changes.

12.7 Further Reading

12.7.1 General

[Pethia] Pethia, Richard, "Congressional Testimony: Attacks on the Internet in 2003", accessed 8/27, http://usinfo.state.gov/journals/itgic/1103/ijge/gj11a.htm.

[Oberndorf] Oberndorf, P and E Wrubel, “Transformation of a Software Development Organization Using Software Acquisition Principles, A Case Study”, Software Engineering Institute, Carnegie Mellon University, 2006

[Lipson] Lipson, Howard, “Evolutionary Systems Design: Recognizing Changes in

Security and Survivability Risks”, CERT, Software Engineering Institute, Carnegie Mellon University, 2006

[Hissam] Hissam, Scott, Charles B. Weinstock, Daniel Plakosh, Jayatirtha Asundi

Perspectives on Open Source Software,” Software Engineering Institute, Carnegie Mellon University, November 2001

[McDermid] McDermid, John A, “Trends in system safety: a European view?” Conferences in Research and Practice in Information Technology Series; Vol. 139, Proceedings of the seventh Australian workshop conference on Safety critical systems and software 2002 - Volume 15, Australian Computer Society, 2002

[Novak] Novak William E (Editor): “Software Acquisition Planning Guidelines

Software Engineering Institute, Carnegie Mellon University, December 2005

CMU/SEI-2005-HB-006

[Smith] Smith, Jim, “An Alternative to Technology Readiness Levels for Non-Developmental Item (NDI) Software” Software Engineering Institute, Carnegie Mellon University, April 2004

[Wilde 93] Norman Wilde, Paul Matthews, and Ross Huitt, "Maintaining Object-Oriented Software”, IEEE Software Volume 10, Issue 1 (January 1993) Pages: 75 - 80

[Wohlin 94] Claes Wohlin and Per Runeson, "Certification of Software Components," *IEEE Transactions on Software Engineering*, v 20, n 6, June 1994, 494-499. riented Software," *IEEE Software*, January 1993, 75-80.

12.7.2 Operational Assurance

[BS ISO/IEC 17799] BS ISO/IEC 17799:2000, Information Technology - Code of Practice for Information Security Management.

[Carter 2004] Carter, Earl, Cisco Systems Inc., *CCSP Self-Study: Cisco Secure Intrusion Detection System*, Cisco Press, 2004.

[CCIMB-2004-01-001] CCIMB-2004-01-001, Common Criteria for Information Technology Security Evaluation, 2004.

[DoD 5200.28-STD 1985] DOD 5200.28-STD, Department of Defense Trusted Computer System Evaluation Criteria, 1985.

[Escamilla 1996] Escamilla, T. “Intrusion Detection: Network Security Beyond the Firewall,” Wiley, 1998, Chap. 5.Jones, Capers, Software Defect Removal Efficiency, *Computer*, April 1996, Vol.29, #4.

[IEEE 1008] IEEE 1008 Standard for Software Unit Testing.

[IEEE 1012] IEEE 1012 1986 Software Validation and Verification Plan.

[IEEE 1012a] IEEE 1012a-1998 Content Map to IEEE/EIA 12207.1-1997.

[IEEE 1028] IEEE 1028-1997 Standard for Software Reviews.

[IEEE 1042] IEEE Standard 1042-1987, Guide to Software Configuration Management.

[IEEE 1045] IEEE 1045-1992 IEEE Standard for Software Productivity Metrics.

[IEEE 1059] IEEE 1059 1993 Guideline for SVV Planning.

[IEEE 12207.2] IEEE/EIA 12207.2-1997, IEEE/EIA Guide: Industry Implementation of International Standard ISO/IEC 12207:1995, 1998.

[IEEE 730.1] IEEE 730.1 1995 Software Quality Assurance Planning.

[IEEE 730] IEEE 730 1998 Software Quality Assurance Plans.

[IEEE 828] IEEE 828 1998 Software Configuration Management Plan.

[IEEE 829] IEEE 829 1998 Software Test Documentation.

[IEEE/EIA 12207.1] IEEE/EIA 12207.1-1997, IEEE/EIA Guide: Industry Implementation of International Standard ISO/IEC 12207:1995, 1998.

[Ingalsbe 2004] Ingalsbe, Jeffery A. “Supporting the Building and Analysis of an Infrastructure Portfolio of software Using UML Deployment Diagrams,” *UML Satellite Activities 2004*, pp 105-117, 2004.

[ISACA 2004-CobiT] “CobiT in Academia,” *IT Governance Institute*, 2004, available from Information Systems Audit and Control Association: http://www.isaca.org, Accessed: July 20, 2005.

[ISACA 2005-COBIT] "COBIT 3rd Edition Executive Summary", *IT Governance Institute*, Available from Information Systems Audit and Control Association: http://www.isaca.org, Accessed: July 2005.

[ISO/IEC 12207] ISO/IEC 12207: 1995, Information Technology - Software life cycle process.

[Kairab 2005] Kairab, S. *A Practical Guide to Security Assessments*, Auerbach Publications, 2005.

[Krutz 2004] Krutz, R., R. Vines, *The CISSP Prep Guide: Mastering the CISSP and ISSEP Exams*, Second Edition, John Wiley & Sons, Chap. 5, 6, 10, 2004.

[Lee 1997] Lee, E. “Software Inspections: How to Diagnose Problems and Improve the Odds of Organizational Acceptance,” *Crosstalk,* Vol.10 #8 1997.

[Lukatsky 2003] Lukatsky, *Protect Your Information with Intrusion Detection.*, A-LIST Publishing, Chap. 1, 4, 6, 2003.

[Northcutt 2003] Northcutt, S. “Computer Security Incident Handling: An Action Plan for Dealing with Intrusions, Cyber-Theft, and Other Security-Related Events,” SANS Institute, 2003.

[Peltier 2003] Peltier, T., J. Peltier, and J. Blackley, *Managing a Network Vulnerability Assessment*, Auerbach Publications, 2003.

[Riggs 2003] Riggs, S., *Network Perimeter Security: Building Defense In-Depth*, Auerbach Publications, Chap. 9, 12, 2003.

[Swiderski 2004] Swiderski, F. and W. Snyder, *Threat Modeling*, Microsoft Press, 2004.

12.7.2.1 Operational Testing

[Andrews 2006] Andrews, James H., Lionel C. Briand, Yvan Labiche, Akbar Siami Namin: Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. IEEE Trans. Software Eng. 32(8): 608-624 (2006)

[Arisholm 2006] Arisholm, Erik, Lionel C. Briand: Predicting fault-prone components in a java legacy system. ISESE 2006: 8-17

[Besal 2001] Besal, R.E. and Steven K. Whitehead, “Operational Testing: Redefining Industry Role,” National Defense, Saturday, September 1 2001

[Briand 1993] Briand, Lionel C, Victor R. Basili, Christopher J. Hetmanski: Developing Interpretable Models with Optimized Set Reduction for Identifying High-Risk Software Components. IEEE Trans. Software Eng. 19(11): 1028-1044 (1993)

[Briand 1994] Briand, Lionel C, Victor R. Basili, Yong-Mi Kim, Donald R. Squier: A Change Analysis Process to Characterize Software Maintenance Projects. ICSM 1994: 38-49

[Briand 1996] Briand, Lionel C, Sandro Morasca, Victor R. Basili: Property-Based Software Engineering Measurement. IEEE Trans. Software Eng. 22(1): 68-86 (1996)

[Briand 1997] Briand, Lionel C, Christian Bunse, John W. Daly, Christiane Differding: An Experimental Comparison of the Maintainability of Object-Oriented and Structured Design Documents. Empirical Software Engineering 2(3): 291-312 (1997)

[Briand 1998] Briand, Lionel C, Khaled El Emam, Oliver Laitenberger, Thomas Fussbroich: Using Simulation to Build Inspection Efficiency Benchmarks for Development Projects. ICSE 1998: 340-349

[Briand 2000] Briand, Lionel C, Bernd G. Freimut, Ferdinand Vollei: Assessing the Cost-Effectiveness of Inspections by Combining Project Data and Expert Opinion. ISSRE 2000: 124-135

[Briand 2002] Briand, Lionel C, Walcélio L. Melo, Jürgen Wüst: Assessing the Applicability of Fault-Proneness Models Across Object-Oriented Software Projects. IEEE Trans. Software Eng. 28(7): 706-720 (2002)

[Briand 2004a] Briand, Lionel C, Yvan Labiche, Yihong Wang: Using Simulation to Empirically Investigate Test Coverage Criteria Based on Statechart. ICSE 2004: 86-95

[Briand 2004b] Briand, Lionel C, Bernd G. Freimut, Ferdinand Vollei: Using multiple adaptive regression splines to support decision making in code inspections. Journal of Systems and Software 73: 205-217 (2004)

[Cohen 1998] Cohen, Michael I, John E. Rolph, and Duane L. Steffey (Eds.); “Statistics, Testing, and Defense Acquisition: New Approaches and Methodological Improvements,” Panel on Statistical Methods for Testing and Evaluating Defense Systems, Committee on National Statistics, National Research Council, 1998

[Frankl 1998] Frankl, Phyllis G, Richard G. Hamlet, Bev Littlewood and Lorenzo Strigini, “Evaluating Testing Methods by Delivered Reliability”, IEEE Transactions on Software Engineering, August 1998, Vol. 24, No. 8, pp. 586+601

[Freimut 2005] Freimut, Bernd G, Lionel C. Briand, Ferdinand Vollei: Determining Inspection Cost-Effectiveness by Combining Project Data and Expert Opinion. IEEE Trans. Software Eng. 31(12): 1074-1092 (2005)

[Gilliam 2003] Gilliam, David, John Powell, Eric Haugh and Matt Bishop, “Addressing Software Security and Mitigations in the Life Cycle”, Proceedings 28th Annual NASA Goddard Software Engineering Workshop (SEW'03), 2003

[Gokhale 2003] Gokhale, Swapna S, “Optimal Software Release Time Incorporating Fault Correction” Proceedings 28th Annual NASA Goddard Software Engineering Workshop (SEW'03), 2003

[Grottke 2002] Grottke, Michael, “A Markov Model for Software Code Construct Coverage and Fault Detection”, Communications of Third International Conference on Mathematical Methods in Reliability, Trondheim, Norway 2002

[Hochstein 2003] Hochstein, Lorin and Mikael Lindvall, “Diagnosing Architectural Degeneration”, Proceedings 28th Annual NASA Goddard Software Engineering Workshop (SEW'03), 2003

[Kamavaram 2003] Kamavaram, Sunil and Katerina Goseva-Popstojanova, “Sensitivity of Software Usage to Changes in the Operational Profile,” Proceedings 28th Annual NASA Goddard Software Engineering Workshop (SEW'03), 2003

[Morasca 1997] Morasca, Sandro, Lionel C. Briand: Towards A Theoretical Framework For Measuring Software Attributes. IEEE Metrics 1997: 119-126

[Mouchawrab 2005] Mouchawrab, Samar, Lionel C. Briand, Yvan Labiche: A measurement framework for object-oriented software testability. Information & Software Technology 47(15): 979-997 (2005)

[NRC 2003] Panel on Operational Test Design and Evaluation of the Interim Armored Vehicle, “Improved Operational Testing and Evaluation: Better Measurement and Test Design for the Interim Brigade Combat Team with Stryker Vehicles, Phase I Report,” National Research Council, Washington DC 2003

[NRC 2004] Panel on Operational Test Design and Evaluation of the Interim Armored Vehicle, “Improved Operational Testing and Evaluation and Methods of Combining Test Information for the Stryker Family of Vehicles and Related Army Systems: Phase II Report,” National Research Council, Washington DC. 2004

[Rivers 1998] Rivers, Anthony T. and M. A. Vouk "Resource Constrained Non-Operational Testing Of Software," Proceedings ISSRE 98, 9th International Symposium on Software Reliability Engineering, Paderborn Germany, Nov. 4-7, 1998

[VonMayrhauser 1993] Von Mayrhauser A and A.M. Vans "From Program Comprehension to Tool Requirements for an Industrial Environment" CASE '93: The Sixth International Conference on Computer-Aided Software Engineering, Singapore; July 19-23, 1993

[Vouk 1992] Vouk, M. A, "Using Reliability Models During Testing with Non-Operational Profiles" Proceedings 2nd Bellcore/Purdue Workshop on Issues in Software Reliability Estimation, October, 1992, Oct. 1992

[Vouk 1993] Vouk, M. A. and K.C. Tai, "Some Issues in Multi-Phase Software Reliability Modeling," Proceedings International Conference on Computer Science and Software Engineering CASCON, October 1993

[Walker 2003] Walker, Robert J,. Lionel C. Briand, David Notkin, Carolyn B. Seaman, Walter F. Tichy: Panel: Empirical Validation-What, Why, When, and How. ICSE 2003: 721-722

12.7.3 Analysis

[Babich 1986] Babich, W., Software Configuration Management, Addison-Wesley, 1986.

[Berry 2001] Berry, John, "IT ROI Metrics Fall Into Four Groups," Internet Week, July 16, 2001.

[Bersoff 1980] Bersoff, E., V. Henderson, and S. Siegel., Software Configuration Management, Prentice-Hall, 1980.

[Dart 1996] Dart, Susan A., "Achieving the Best Possible Configuration Management Solution," Crosstalk, September 1996.

[Dorofee 1997] Dorofee A.J., JA Walker, and RC Williams. "Risk Management in Practice," Crosstalk, Volume 10 #4, April 1997.

[Feiler 1990] Feiler, P., "Software Process Support in Software Development Environments", Fifth International Software Process Workshop, ACM Press, October 1990.

[Feiler 1991] Feiler, P., Configuration Management Models in Commercial Environments, Tech. report CMU/SEI-91-TR-7, ADA235782, Software Engineering Institute, Carnegie-Mellon University, April 1991.

[Han 1997] Han, Jun. "Designing for Increased Software Maintainability," International Conference on Software Maintenance (ICSM, 97), January 1, 1997.

[Hatton 1999] Hatton, L. (1999) "Repetitive failure, feedback and the lost art of diagnosis," Journal of Systems and Software, 1999.

[Hatton 2001] Hatton, L. "Exploring the role of Diagnosis in Software Failure", IEEE Software, July 2001.

[Hatton 2002] Hatton, L. "Safer Language Subsets: an overview and a case history," MISRA C, Information and Software Technology, June 2002.

[IEEE/ANSI 1042] IEEE Guide to Software Configuration Management, IEEE/ANSI Standard 1042-1987,1987.

[IEEE/ANSI 828] IEEE Standard for Software Configuration Management Plans, IEEE/ANSI Standard 828-1998, 1998.

[IEEE/EIA 12207.1] IEEE/EIA 12207.1-1997, IEEE/EIA Guide: Industry Implementation of International Standard ISO/IEC 12207:1995, 1998.

[IEEE/EIA 12207.2] IEEE/EIA 12207.2-1997, IEEE/EIA Guide: Industry Implementation of International Standard ISO/IEC 12207:1995, 1998.

[ISO/IEC 12207] ISO/IEC Std. 12207:1995, Information Technology - Software Life Cycle Processes, International Standards Organization, 1995.

[ISO/IEC 15288] ISO/IEC Std. 15288:2002, E, Systems Engineering – System Lifecycle Processes, International Standards Organization, 2002.

[ISO/IEC 15846] ISO/IEC 15846: 1998, Information technology - Software life cycle processes - Configuration Management, May 5, 1998.

[Jones 2004] Jones, Capers, Software “Project Management Practices: Failures versus Success,” Crosstalk, pp, 5-9, October 1, 2004.

[Lee 1997] Lee, E. “Software Inspections: How to Diagnose Problems and Improve the Odds of Organizational Acceptance,” Crosstalk, Vol.10 #8 1997.

[OMB 1999] Office of Management and Budget, Evaluating Information Technology Investments, 1999.

[Pfleeger 1997b] Pfleeger, S. and L. Hatton, “Do formal methods really work," IEEE Computer, Jan 1997.

[SEI 1990] “Configuration Management: State of the Art,” SEI Bridge, Software Engineering Institute, Carnegie-Mellon University, March 1990.

[Violino 1997] Violino R, “Measuring Value: Return on Investment,” Information Week, No. 637, pp. 36-44, June 30, 1997.

[Whitgift 1991] Whitgift, D., Methods and Tools for Software Configuration Management, John Wiley and Sons, England, 1991.

[Zimmerman 1997] Zimmerman, Michael, “Configuration Management, Just a Fashion or a Profession,” White Paper, usb GmbH, 1997.

Available at: at http://www.itmweb.com.

12.7.3.1 Threats Identification and Analysis

[Arnold 1996] Arnold, Robert and Shawn Bohner, “Software Change Impact Analysis,” Wiley-IEEE Computer Society Press, July 1996

[Beattie 2002] Beattie, S., Arnold, S., Cowan, C., Wagle, P. and Wright,”Timing the application of security patches for optimal uptime”, Proceedings: Large Installation Systems Administration LISA '02: 16th Systems Administration Conference. (2002)

[Bharti 2006] Bharti, Nitin, “Threat Modeling Key to Proactive Security,” Software Quality News, March 1, 2006

[Bohner 2003] Bohner, Shawn A and Denis Gracanin, “Software Impact Analysis in a Virtual Environment”, Proceedings 28th Annual NASA Goddard Software Engineering Workshop (SEW'03), 2003 p. 143

[Burns 2005] Burns, Steven F, “Threat Modeling: A Process To Ensure Application Security”, SANS-GSEC , January 5, 2005

[Chen 2002] Chen, M., E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem Determination in Large, Dynamic, Internet Services," Proceedings:. International Conf. on Dependable Systems and Networks (IPDS Track), 2002

[Cobb 2007] Cobb, Michael, “Improve Web Application Security with Threat Modeling,” SearchSecurity.com, January 11, 2007, accessed 8/31/2007, http://searchsecurity.techtarget.com/tip/0,289483,sid14_gci1237571,00.html

[Dacey 2003] Dacey, Robert F, “Effective Patch Management is Critical to Mitigating Software Vulnerabilities,” Testimony before the Subcommittee on Technology Information Policy, Intergovernmental Relations, and the Census, House Committee on Government Reform, United States General Accounting Office, September 10, 2003

[Das 2002] Das, M., S. Lerner, and M. Seigle, “ESP: Path-sensitive Program Verification in Polynomial Time. Proceedings: ACM Conf. on Programming Language Design and Implementation, pages 57-68, Berlin, June 2002

[Ensel 2002] Ensel C. and A. Keller, “An approach for managing service dependencies with xml and the resource description framework”, Journal of Network and Systems Management, Special Issue on Selected Papers of IM 2001, 10(2):27--34, June 2002

[Heckman 2007] Heckman, Rocky, “Application Threat Modeling v2,” Builder. Au, 2006/03/07, Accessed 8/30/2007, http://www.builderau.com.au/blogs/intothebreach/soa/Application-Threat-Modeling-v2/0,339027621,339237280,00.htm

[Herman 2006] Hernan, Shawn, Scott Lambert, Tomasz Ostwald and Adam Shostack, “ Uncover Security Design Flaws Using The STRIDE Approach,” MSDN Magazine, November 2006

[Larsson 2001] Larsson M. and I. Crnkovic, "Configuration Management for Component-based Systems," Proceedings International Conf. on Software Engineering (ICSE), May 2001

[Lippmann 2002] Lippmann, Richard, Seth Webster and Douglas Stetson, “The Effect of Identifying Vulnerabilities and Patching Software on the Utility of Network Intrusion Detection”, Recent Advances in Intrusion Detection, 5th International Symposium, RAID 2002, Zurich, Switzerland, in Luca Deri, Giovanni Vigna, Andreas Wespi, (Eds.) Springer-Verlag, Lecture Notes in Computer Science, New York (2002)

[Murphy 2004] Murphy, Paul, “Software Vulnerabilities and the Future of Liability Reform,” LinuxInsider, January 22, 2004, Accessed 8/31/2007 http://www.linuxinsider.com/

[Rus 2003] Rus, Ioana, Forrest Shull and Paolo Donzelli, Decision Support for Using Software Inspections”, Proceedings 28th Annual NASA Goddard Software Engineering Workshop (SEW'03), 2003

[Swiderski 2004] Swiderski, Frank and Window Snyder, “Threat Modeling”, Microsoft Press, 2004

[Thompson 2005] Thompson, Herbert H and Scott G. Chase ,“The Software Vulnerability Guide,” (Charles River Media, 2005

[Wang 2003] Wang, Y, M Verbowski, Chad Dunagan, J., Chen, Y., Wang, H.J., Yuan, C., and Zhang, Z., "STRIDER: A Blackbox, State-based Approach to Change and Configuration Management and Support," Proceedings: Usenix Large Installation Systems Administration (LISA) Conference, pp. 159-171, October 2003

[Wyatt 2003] Wyatt, Valerie, Justin DiStefano, Mike Chapman and Edward Aycoth, “A Metrics Based Approach for Identifying Requirements Risks, “Proceedings 28th Annual NASA Goddard Software Engineering Workshop (SEW'03), 2003

[Xu 2006] Xu, Dianxiang and Kendall E. Nygard, “Threat-Driven Modeling and Verification of Secure Software Using Aspect-Oriented Petri Nets,” IEEE Transactions on Software Engineering, April 2006 (Vol. 32, No. 4) pp. 265-278

12.7.3.2 Reporting

[Berinato 2007] Berinato, Scott, “The Chilling Effect, How the Web Makes Creating Software Vulnerabilities Easier, Disclosing them More Difficult and Discovering them Possibly Illegal, CSO, January 2007

[DoD 1986] DODI 5215.2, “Computer Security Technical Vulnerability Reporting Program (CSTVRP),” U.S. Department of Defense (DoD), September 2, 1986

[Leyden 2006] Leyden, John, “Report Security Vulnerabilities at your Peril,” The Register, Thursday 25th May 2006, Accessed, 8/14/2007, http://www.theregister.co.uk/2006/05/25/security_vuln_reporting_risk/

[Meunier 2006] Meunier Pascal, “CERIAS Weblogs » Reporting Vulnerabilities is for the Brave”, 2006, Accessed 8/17/2007, http://www.cerias.purdue.edu/weblogs/pmeunier/policies-law/post-38/

[OIS 2004] Organization for Internet Safety, ”Guidelines for Security Vulnerability Reporting and Response, Version 2.0, OIS, September 2004

[Verghese 2003] Varghese, Sam, “Researchers Critical of Vulnerability Reporting Process Draft,” The Age, June 11 2003, Accessed 8/14/2007, http://www.theage.com.au/articles/2003/06/11/1055220630601.html

[Zarra 2006] Zarra, Marcus, "The Morals of Security: Reporting Vulnerabilities". Dev Source. Ziff Davis Media, August 2006.

12.7.4 Response Management

[Bastani 96] F. Bastani, B. Cukic, V. Hilford, and V. Jamoussi, "Toward Dependable Safety-critical Software," Proceedings of WORDS '96 Second Workshop on Object-Oriented Real-Time Dependable Systems, IEEE Computer Society Press, Los Alamitos, Calif. 1996. 86 - 92

[Bernot 91] Giles Bernot, M. C. Gaudel, and B. Marre, "Software Testing based on Formal Specifications: a Theory and a Tool," Software Engineering Journal, v 6, n 6, November 1991, 387-405.

[Binder 96c] Robert V. Binder, "Off-the-Shelf Test Automation for Objects," Object Magazine, v 6, n 2, April 1996, 26-30

[BS 15000-1] BS 15000-1: 2000, *Specification for Service Management*.

[BS 15000-2] BS 15000-2, 2000, Code of Practice for Service Management.

[Cunningham] Cunningham, Ward, "Finding and Exploiting Potent Abstractions FAST," Smalltalk Solutions '96, SIGS Conferences, New York, March 1996

[DeLooze 2004] DeLooze, L. “Classification of computer attacks using a self-organizing map,” *Proceedings from the Fifth Annual IEEE SMC*, 10-11 June 2004, Pages: 365-369, 2004.

[FDA 2002] U.S. Food and Drug Administration, Center for Devices and Radiological Health, January 11, 2002

[Hsia 96] Pei Hsia, Xiaolin Li, Kung, D.C. "A History-based Approach for Early Faulty State Detection," Proceedings of 20th International Computer Software and Applications Conference: COMPSAC '96, IEEE Computer Society Press, Los Alamitos, Ca 1996,321 - 326,

[IEEE/EIA 12207.1] IEEE/EIA 12207.1-1997, IEEE/EIA Guide: Industry Implementation of International Standard ISO/IEC 12207:1995, 1998.

[IEEE/EIA 12207.2] IEEE/EIA 12207.2-1997, IEEE/EIA Guide: Industry Implementation of International Standard ISO/IEC 12207:1995, 1998.

[ISO/IEC 12207] ISO/IEC Std. 12207:1995, *Information Technology - Software Life Cycle Processes,* International Standards Organization, 1995.

[ISO/IEC 15288] ISO/IEC 15288: 2002 Systems Engineering - System Life Cycle Processes.

[ISO/IEC 17799] ISO/IEC Std. 17799:2000, *Information Technology - Code of Practice for Information Security Management*, International Standards Organization, 2000.

[ISO/IEC 27001] ISO/IEC 27001: 2005, Information Security Management - Specification With Guidance for Use, 2005.

[ITIL 1999] IT Infrastructure Library – "ITIL v2: 1999 Best Practice in IT Service*,*" *Management*, 1999.

[Kornecki] Kornecki, Andrew J and Janusz Zalewski, "Design Tool Assessment for Safety-Critical Software Development", Proceedings 28th Annual NASA Goddard Software Engineering Workshop (SEW'03), 2003

[Mann 1999] Mann, D and D. Christey, *Towards a Common Enumeration of Vulnerabilities*, The MITRE Corporation, Bedford MA, 1999.

[Martin 2003] Martin, R. "Integrating your information security vulnerability management capabilities through industry standards (CVE&OVAL)," *Systems, Man and Cybernetics, IEEE International Conference*, Volume 2, 5-8 Oct. 2003 Page(s):1528-1533, 2003.

[Mattern 2006] Mattern, Steven F, "Increasing the Likelihood of Success of a Software Assurance Program", Crosstalk, Sep 2006 Issue

[NASA 1993] NASA-STD-2201-93 "Software Assurance Standard", National Aeronautics and Space Administration, November 10, 1992

[Overbeck 1995] Jan Overbeck, "Testing Object-Oriented Software and Reusability -- Contradiction or Key to Success" Proceedings, 8th Annual Software Quality Week May 1995, Software Research, Inc. San Francisco

[Rajlich] Rajlich, Vaclav, "Software Change and Evolution", Conference on Current Trends in Theory and Practice of Informatics, SOFSEM'99: Milovy, Czech Republic, 1999

[Riggs 2003] Riggs, S., *Network Perimeter Security: Building Defense In-Depth*, Auerbach Publications, Chap. 9, 12, 2003.

[Wallace] Wallace Dolores R. Wendy W. Peng and Laura M. Ippolito, "NISTIR 4909

"General Principles of Software Validation; Final Guidance for Industry and FDA Staff",

Software Quality Assurance: Documentation and Reviews" U.S. Department of Commerce, Technology Administration, National Institute of Standards and Technology

[YCC 2005] Web site on viruses: www.ycc.com/security/details/virus.htm (December 20, 2005).

12.7.4.1 Responding to Known Vulnerabilities

[Mell 2007] Mell, Peter, Karen Scarfone and Sasha Romanosky, Common Vulnerability Scoring System (CVSS-SIG), A Complete Guide to the Common Vulnerability Scoring System Version 2.0, Forum of Incident Response and Security Teams (FIRST), June 2007

[Myerson 2005] Myerson, Judith, "Use SLAs in a Web services context, Part 7: Mitigate risk for vulnerability with a SLA Guarantee, IBM, 28 Jan 2005, Accessed 8/12/2007, http://www.ibm.com/developerworks/library/ws-sla7/

12.7.4.2 Secure Migration, Retirement, Loss, and Disposal

[Intel 2006] Intel Information Technology, "Integrated Software Enhances Enterprise Security",

IT@Intel Brief, November 2006, Accessed 8/31/07, http://www.intel.com/it/pdf/integrated-sw-enhances-security.pdf

12.7.5 Infrastructure Assurance

[Ashton 2001] Ashton, Gerry. "Cleaning up your Security Act for Inspection," *Computer Weekly*, Jan 18, 2001.

[Berg 2005] Berg, Clifford J, High-Assurance Design: Architecting Secure and Reliable Enterprise Applications, Addison Wesley, 2005.

[BS 15000-1] BS 15000-1: 2000, *Specification for Service Management*.

[Byfield 2005] Byfield, Bruce, "Nine principles of Security Architecture," Linux.com, November 22, 2005, Accessed 8/17/2007, http://www.linux.com/articles/49803?tid=78

[CCIMB-2004-01-001] CCIMB-2004-01-001, Common Criteria for Information Technology Security Evaluation", 2004.

[DoD 5200.28-STD 1985] DOD 5200.28-STD, Department of Defense Trusted Computer System Evaluation Criteria, 1985.

[DOE 2007] National Energy Technology Laboratory (NETL), "Key Issues & Mandates, Critical Infrastructure Assurance," Department of Energy, 2007, Accessed 8/31/2007 http://www.netl.doe.gov/KeyIssues/critical_infra.html

[Dorofee 1997] Dorofee A.J., JA Walker, and RC Williams. "Risk Management in Practice", *Crosstalk*, Volume 10 #4, April 1997.

[DTI UK 2004] Price, Waterhouse and Coopers, *Information Security Breaches 2003*, Department of Trade and Industry (DTI), U.K., 2004.

[GAO 1999] General Accounting Office, *GAO Internal Control Standard*, 1999.

[GASSP 1999] GASSP, "Generally Accepted System Security Principles", *International Information Security Forum*, June 1999.

[IEEE/EIA 12207.1] IEEE/EIA 12207.1-1997, IEEE/EIA Guide: Industry Implementation of International Standard ISO/IEC 12207:1995, 1998.

[IEEE/EIA 12207.2] IEEE/EIA 12207.2-1997, IEEE/EIA Guide: Industry Implementation of International Standard ISO/IEC 12207:1995, 1998.

[ISACA (AS) 1999] ISACA, "Audit Sampling," *IS Auditing Guideline*, 1999.

[ISACA (CM) 2004] IT Governance Institute, *CobiT Mapping*, ISACA, 2004.

[ISACA (CRSA) 2003] ISACA, "Control Risk Self Assessment," *IS Auditing Guideline*, 2003.

[ISACA (DPC) 1999] ISACA, "Due Professional Care," *IS Auditing Guideline*, 1999.

[ISACA (ICO) 1999] IT Governance Institute, *IT Control Objectives for Enterprise Governance*, ISACA, 1999.

[ISACA (ID) 2003] ISACA, "Intrusion Detection," *IS Auditing Guideline*, 2003.

[ISACA (ISC) 2003] IT Governance Institute, *IT Strategy Committee*, ISACA, 2003.

[ISACA (SDLC) 2003] ISACA, "SDLC Reviews," *IS Auditing Guideline*, 2003.

[ISACA (SO) 2004] IT Governance Institute, *IT Control Objectives for Sarbanes-Oxley*, ISACA, 2004.

[ISACA (URA) 2000] ISACA, "Use of Risk Assessment," *IS Auditing Guideline*, 2000.

[ISACA 2004-CobiT] "CobiT in Academia", *IT Governance Institute*, 2004, available from Information Systems Audit and Control Association: http://www.isaca.org, Accessed: July 20, 2005.

[ISACA 2005-COBIT] "COBIT 3rd Edition Executive Summary", *IT Governance Institute*, Available from Information Systems Audit and Control Association: http://www.isaca.org, Accessed: July 2005.

[ISO/IEC 12207] ISO/IEC Std. 12207:1995, *Information Technology - Software Life Cycle Processes*, International Standards Organization, 1995.

[ISO/IEC 15288] ISO/IEC Std. 15288:2002, *E, Systems Engineering – System Lifecycle Processes*, International Standards Organization, 2002.

[ISO/IEC 27001] ISO/IEC 27001: 2005, Information Security Management - Specification With Guidance for Use, 2005.

[ISO/IEC 27003] ISO/IEC 27003: 2005, Information Security Management – Implementation of ISO 27001, 2005.

[ISO/IEC 27004] ISO/IEC 27004: 2005, Information Security Management – Information Security Metrics and Measurement, 2005

[ITIL 1999] IT Infrastructure Library – "ITIL v2: 1999 Best Practice in IT Service," *Management*, 1999.

[Kairab 2005] Kairab, S. *A Practical Guide to Security Assessments*, Auerbach Publications, 2005. Chap. 6,7.

[Kim 2007] Kim, Gene, and Rob Warmack, "Proving Control of the Infrastructure," Whitepaper, Tripwire, 2007

[King 2001] King, C., C. Dalton and E. Osmanoglu, *Security Architecture: Design Deployment and Operations*, McGraw-Hill/Osborne, 2001.

[Kolodgy 2004] Kolodgy, Charles J., "Infrastructure Integrity", White Paper, Tripwire, October 2004

[Krutz 2004] Krutz, R., R. Vines, *The CISSP Prep Guide: Mastering the CISSP and ISSEP Exams*, Second Edition, John Wiley & Sons, Chap. 5, 6, 10, 2004.

[Nelson 2007] Nelson, Mike, "Complying with the Federal Information Security Management Act," Whitepaper, Tripwire, 2007

[NIST 800-26] National Institute of Standards and Technology (NIST 800-26), *Security Self-Assessment Guide for Information Technology Systems*, November 2001.

[Purser 2004] Purser, S., *A Practical Guide to Managing Information Security*, Artech House, Chap. 1, 8, 2004.

[Qualys 2007] Qualys, "7 Essential Steps to Achieve, Measure and Prove Optimal Security Risk Reduction" Whitepaper, Qualys, Redwood Shores, CA Jun 11, 2007

[Swanson ND] Swanson, Marianne, Security Self Assessment Guide for Information Technology Systems, ND.

[Tripwire 2007a] Tripwire, “Control IT with Configuration Audit and Control, Enforcing Change Policy to Reach Compliance, Security, and Availability Goals,” Whitepaper, Tripwire, 2007

[Tripwire 2007b]Tripwire, “Configuration Audit and Control in a Layered Security Strategy, Providing the Essential Foundation for Data and Network Security, “Whitepaper, Tripwire, 2007

[Tripwire 2007c] Tripwire, “Managing Change in IT Infrastructures”, Whitepaper, Tripwire, 2007

[Van Grembergen 2004] Van Grembergen, W., *Strategies for Information Technology Governance*, Idea Group Publishing, Chap.11, 2004.

[White House 1995] “White Paper on Information Infrastructure Assurance,” Security Policy Board, Assistant to the President for National Security, White House, December 1995

Part 4: Using the Software Assurance Common Body of Knowledge

13Acquiring Secure Software

13.1 Introduction

This section covers use of existing software or software externally produced for an organization including several kinds of sources, commercial and otherwise. It is intended for readers involved in software reuse or software acquisition education and training, as well as standards developers who need additional knowledge on acquiring secure software. Notional examples or illustrations are provided to facilitate understanding.

At this point in developing this section, most of the references cited are to U.S .federal government documents because they are more readily available to the working group. While international and UK Ministry of Defense standards have had significant influence on the content, the material in this section reflects a US federal government perspective. Although this section may require more effort and some "translation" by international and the commercial readers due to sometimes unfamiliar terminology, the content is still universally relevant. To help this section evolve into a more general perspective and terminology, we hope that readers and reviewers will help us with additional relevant information from the commercial and international arenas.

Models of acquisition in general are defined in the Federal Acquisitions Regulation [FAR] and [DoDI 5000.2], and several standards address the software acquisition process and practices [ISO/IEC 12207, p. 10, para. 5.1]; [IEEE/EIA 12207]; IEEE 1062-1998]. A NIST Special Publication more directly addresses the acquisition of secure information systems [NIST Special Pub 800-64, para 2.3.2, para 2.3.2.9, Appendix A, and Appendix B]. The acquisition activities that cross most models include initiation, request for proposals, proposal (tender) preparation, contract preparation and update, supplier monitoring, and acceptance and completion. This section uses a generic acquisition framework that can be easily related to the ISO/IEC, civilian US federal government, and DoD models. This section also provides the acquirer and, to a lesser extent, supplier perspectives.

The SwA Acquisition guide is recommended in Sep 2007 Report of the Defense Science Board (DSB) Task Force on "Mission Impact of Foreign Influence on DoD Software" -- Under the Recommendations on Risk-Based Acquisition (starting on page 64):

"…the mere fact of asking what vendors do to engineer security and quality into their lifecycle puts the vendor community on notice that it is important to DoD."

- "The DoD/DHS software assurance forum has been working on a procurement guide focused on software assurance, which helps procurement officers glean (through a series of questions) what vendors have done (and not done) as part of their secure development process, how they handle vulnerabilities, and so on."

"Such a document, when reviewed by a larger audience and finalized, could be used as part of IT procurement cycles to help DoD better evaluate risk."

- "As long as this is sensible, the questions are phrased to allow expository answers, and the benefit derived is commensurate with the cost of vendors completing it, this is one way for DoD both to know what they are getting and to put vendors on notice that quality and security-worthiness has become a purchasing criteria for DoD."

Certain aspects of acquisition are also related to the Development and Post Release sections. Those sections will be cross-referenced in this section. The following is an outline of the contents of this section:

- Concepts, Terms, and Definitions
- Program Initiation and Planning – Acquirer

- Acquisition and Software Reuse – Acquirer/Supplier
- Request for Proposals – Acquirer
- Preparation of Response – Supplier
- Source Selection – Acquirer
- Contract Negotiation and Finalization – Acquirer
- Project/Contract Management – Acquirer/Supplier

13.2 Concepts, Terms, and Definitions

13.2.1 Acquisition

"Acquisition" as used in this section means acquiring software development services or software products, whether by contract or by other means (e.g., downloading open source software from the Internet, etc.) [For the US federal government also see the FAR Subpart 2.101(b)(2) definition of acquisition].

13.2.2 Off the Shelf Software (OTS)

Off-the-shelf software includes COTS (Commercial off the Shelf Software) and other OTS (Off the Shelf Software). This may also include (for governments) government off-the-shelf (GOTS) software and Non-developmental Items (NDI) [Also see FAR Subpart 2.1 for a US federal government definition of commercial items].

13.2.3 Information Assurance Architecture

Information Assurance Architecture is an abstract description (used among others by the US Department of Defense (DoD)) of a combination of information assurance (IA) solutions for a system or set of systems that assigns and portrays IA roles, identifies behavior among a set of information technology assets, and prescribes rules for interaction and interconnection to ensure security and taking advantage of supporting IA infrastructures [DoD Instruction 8500.2, Enclosure 2].

13.2.4 US NIAP

National Information Assurance Partnership is a combined initiative of the US government's National Institute of Standards and Technology (NIST) and National Security Agency (NSA) initiative originated to meet the security testing needs of both information technology (IT) consumers and producers [http://www.niap.nist.gov/]. Currently, its efforts center on the Common Criteria and Protection Profiles [CC 2005, Part 1, Annex B].

13.2.5 Security Accreditation

"Security accreditation" means the official management decision given to authorize operation of an information system and to explicitly accept the risk to an organization's (and by implication interconnecting organizations') operations (including mission, functions, image, or reputation), assets, or individuals based on implementing of an agreed-upon set of security controls [NIST Special Publication 800-37].

13.2.6 Security Certification

"Security certification" may apply to a software system as in the case of the Common Criteria or FIPS-140, or may mean a comprehensive assessment of the management, operational, and technical security controls in an information system, made in support of security accreditation, to determine the extent to which the controls are implemented correctly, operating as intended, and producing the desired outcome with respect to meeting the security requirements for the system [NIST Special Publication 800-37].

13.3 Program Initiation and Planning–Acquirer

13.3.1 Scope

Program initiation and planning begins with developing the concept of the need to acquire, develop, or enhance a software capability. During the initiation stage, the acquirer is concerned with setting boundaries about the project. Of these, cost, schedule, and quality are most easily seen; more difficult is the definition of the required capability, and even more difficult than that is the definition of the required level of software assurance.

Program initiation and planning may include the following activities:

- Determine the need (requirements) and solutions approaches. Key issues during this activity relative to software assurance include acquiring public domain software; purchasing COTS software; developing the software internally; acquiring the developing software by contract; enhancing existing software; modifying legacy software; or any combination of these options.
- Make the decision to contract (The terms outsourcing and contracting out are often used interchangeably. However, outsourcing implies that the work is being done within the acquirer's organization and a subsequent decision is made to contract out the work to an outside organization.) [FAR 2005, Part 10: Market Research].
- Identify responsibilities of organizations.
- Determine (maximum) potential risk from software system and its use and values to acquirer of achieving different levels of reduced risk, e.g., [NIST Special Pub 800-30; FAR 2005, Subpart 39.102: Management of Risk].
- Determine type of contract based on the level of cost, schedule, and performance risks that are acceptable between the parties of the contract, e.g., choose among those described in [FAR 2005, Part 16: Types of Contracts].
- Define software assurance requirements within contract language and establish process for testing and evaluation, and remediation before final acceptance.
- Develop an acquisition strategy and/or plan, e.g., see [http://akss.dau.mil/DAG/Guidebook/IG_c2.3.asp] and [FAR 2005, Part 7: Acquisition Planning] that incorporates key strategies for software assurance. In particular, FAR 7.105(b)(17) requires that plans discuss how agency information security requirements are to be met.

For additional relevant material see [ISO/IEC 12207, para 5.1.1.], [ANSI/PMI 99-001-2004 , Ch2], [Schwalbe 2006, p. 80], and [Meredith 2000, Part 1],

13.3.2 Determining the Need (Requirements) and Solution Approaches

Similar to software needs and requirements, software assurance needs and requirements at this point are usually broad statements of management and technical capabilities and constraints. Constraints may be specified in risk mitigation requirements that result from a risk assessment (See Risk Management below). Other capabilities may be specified in EAL [CC 2005, Part 3]. In addition, Common Criteria Protection Profiles [CC 2005, Part 1, Annex A] may also be another alternative for describing capabilities and constraints related to secure software. Also see Section 5, Requirements for Secure Software, for guidance on technical software assurance requirements [also see NIST Special Pub 800-64, p. 12-13].

See the “Acquisition from the Public Domain” and “OTS Acquisition” subsections for issues software assurance issues related to off-the-shelf software, software code reuse, and public domain software.

13.3.3 Making the Decision to Contract

One decision that the acquirer must make is whether (and how much) to either acquire OTS or contract out (e.g., outsourcing) the effort versus doing the work with in-house resources. Conducting market research [e.g., see FAR 2005, Part 10: Market Research] can assist in determining if the capabilities are available through suppliers. “Engage in software outsourcing only when you understand the pitfalls” [Haddad 2004] is one conclusion of a recent study in contracting out software-intensive projects.

Software-related security risks must be considered and mitigated when making decisions to contract out the software effort and in conducting a market search for qualified software suppliers. Some considerations include:

- Supplier qualification in developing and/or integrating secure software
- Supplier history of good software engineering practices. Good software engineering practices can reduce faults and vulnerabilities [Seacord and Householder 2005, p. 2] and [Seacord 2005]
- Supplier employment of known “hackers”
- Foreign influence of the supplier and its personnel

13.3.4 Risk Management

An essential part of project management is risk management. Risk management is the process of planning, assessing risk, mitigating risks, monitoring risk mitigation activities, and adjusting the risk mitigation activities, as appropriate, based on the results of the monitoring activity. [NIST Special Pub 800-30] provides guidance on information systems risk management. Software assurance risks should be managed as part of the overall information systems risk management.

Software assurance risk mitigation strategies may translate into specific program specifications or requirements (e.g., “The software shall be capable of National Information Assurance Partnership (NIAP) certification at EAL-Level 6”). Other strategies can translate into actions that the acquirer needs to take during the life of the project (e.g., ongoing risk management activities, ongoing monitoring of software assurance testing, etc.). Some may require action during the actual acquisition process (e.g., a FOCI investigation, inclusion of software assurance criteria in the best-value calculation, inclusion of individuals qualified to evaluate software assurance issues on the proposal evaluation board, etc.).

The overall cost of risk management activities over the entire life cycle must be considered in the total cost of the contract effort. NIST Special Publication 800-30 (paragraph 4.5) provides some insights into cost-benefit analysis.

This subsection addresses software assurance-related considerations in risk assessment, risk mitigation (identifying risk reduction activities), and risk monitoring.

13.3.4.1 Software Assurance Risk Assessment

In general, a software assurance risk assessment (see Section 2, Dangers, for a list of risks) is based on the criticality and sensitivity of the information processed by the software as well as vulnerability and threat information. An initial step in risk assessment is identifying the risk. Using a risk-based categorization scheme can facilitate software assurance risk identification and is useful in standardizing the results of assessing potential security risks for software-intensive systems. Such a security categorization scheme is based on a software-intensive system's criticality to the organization's mission and the sensitivity of the information that it processes. –See Section 5, Secure Software Requirements, for more detailed coverage. These security categories can then be used in conjunction with vulnerability and threat information in assessing risk to an organization. See the Threat Analysis subsection in Section 5, Secure Software Requirements, under Requirements. The following covers are several risk based categorization schemes. While not perfect, their efforts in this area are suggestive of methods others might use for categorization.

- The civilian US federal government's method defines three levels or potential impact on organizations or individuals should there be a breach of security (i.e., loss of confidentiality, integrity, or availability) [FIPS Pub 199]. Confidentiality, integrity, and availability are determined by these levels.
- The DoD's method defines three mission assurance categories that reflect the importance of information relative to achieving mission goals and objectives [DoDI 8500.2,para E2.1.38 and Enclosure 4]. Integrity and availability levels are determined by these categories.
- The DoD also has several levels of sensitivity classifications for information, including unclassified, confidential, secret, top secret, and SCI [DoDI S-3600.2]. Confidentiality levels are determined by these classifications [DoDI 8500.2, para E4.1.1].
- The ISO 17799 Newsletter,:News & Updates for ISO 27001 and ISO17799 in Issue 9, suggests categories to "cover most eventualities" – Top Secret, Highly Confidential, Proprietary, Internal Use Only, and Public Documents – giving brief definitions for each.1
- (Asset classification is covered by Section 5 of the ISO17799 standard.)
- The Information Assurance Task Force Framework document [NSA 2002] enumerates five levels of consequences of a violation (as value of data protected) and seven levels of threat.

13.3.4.2 Software Assurance Risk Mitigation Strategies

Once software assurance risks have been assessed, the acquirer should identify corresponding risk mitigation strategies. These risk mitigation strategies provide a baseline level of software assurance and become part of the acquisition requirement. The US government's method for mapping risks to mitigation strategies or techniques provides a useful way for standardizing on specific safeguards based on a categorization scheme as mentioned above. While not perfect, the following are suggestive mapping methods that others might use:

- [NIST Special Pub 800-53] and [FIPS 200] define assurance levels by the required security measures – primarily network-oriented security measures – related to the sensitivity or potential impact levels in FIPS 199.
- DoD Instruction 8500.2 defines assurance levels by assurance conditions achieved through applying specific safeguards. Minimum safeguards are provided for each mission assurance category. Several are relevant to software assurance [as examples, see DoDI 8500.2, pp. 58, 69, and 79 Software

[1] See http://17799-news.the-hamster.com/issue09-news1.htm (accessed 2005/09/17)

Quality; pp. 60 and 71 Software Development Change Control; p. 67 Mobile Code; p. 68 and 78, Public Domain Software].

- The Information Assurance Task Force Framework document [NSA 2002] maps each combination of its five levels of value or consequences of a violation and seven levels of threat to recommendations on strength of mechanism level (SML) and evaluation assurance level (EAL). SMLs are discussed as they related to several security properties.
- The US Director of Central Intelligence Directive [DCID 6/3 2000] places assurance requirements on systems protecting Sensitive Compartmented Information (SCI) within the US government.
- Outside the intelligence community, FIPS 140 certifications for cryptographic software appear a universally accepted minimum for assurance, but NSA Type 1 certification is required for intelligence information – some other nations have similar certifications.
- While not using as simple a mapping as in some schemes, the US DoD accreditation process governed by DoD Instruction 5200.40, DoD Information Technology Security Certification and Accreditation Process (DITSCAP). [DoD1997] supplemented by the DoD8510.1-M Application Manual uses a combined system security assessment and risk analysis process to assist the certifier in determining if the system provides adequate protection for the information handled in the system.

13.3.4.3 Software Assurance Risk Monitoring

Lastly, the acquirer should implement a means for monitoring the software risk mitigation strategies to assess whether those strategies result in acceptable protection [NIST Special Pub 800-30, table 2-1, paragraph 3.9, and Chapter 5].

13.4 Acquisition and Software Reuse – Acquirer/Supplier

13.4.1 Scope

This section covers the acquisition of reusable software , which includes open source software and other reusable software code (hereinafter collectively referred to as free software). Currently, more than 50 types of recognized open source licenses exist with varying obligations on acquirers. Open source products are also sold with value-added products or services (e.g., Linux by Red Hat). The implications and choices are often not simple even before one considers security.

13.4.2 Reusable Software in the Acquisition Process

Reusable software applications and code pose significant software-related security risks. Acquirers and suppliers should have controls in place to ensure that reusable software does not compromise the security of the software-intensive system.

With rare exceptions, composing systems of reusable software is fraught with security problems and existing vulnerabilities. One needs only read the long lists of known defects and vulnerabilities of reusable software to realize the level of danger (even if most known ones are fixed). The risks in reusable software components may or may not be greater than software that costs more money. Indeed, products with source code available can benefit from review by more people knowledgeable in software-related security, and the customers with the skills and resources have the opportunity to fix problems as soon as thy are identified (versus “closed source” software where customers must rely on the vendor to diagnose and fix any problems). The question is not whether free or commercial software is “superior” from a security perspective but the situation with particular choices for software and their pedigrees. Acquiring reusable software does not remove any of the same security requirements as would hold for in-house production or the need to provide the same or higher quality assurance case and evidence. Also, see Section 13.4.4, Software Reuse as Part of Requirer's Solution. in this body of knowledge.

Because the acquisition of reusable software may not require formal contracts, visibility and control over software production and assurance can be significantly reduced, causing increased uncertainties and risks. Therefore, special arrangements are needed to control the use and integration into the software-intensive system [see DoDI 8500.1, para 4.19]. Use of reusable software should be thoroughly assessed for risk [see DoDI 8500.1, para 4.19]. The process in place to deal with vulnerability reports, and to construct and deploy patches or new versions needs special care and scrutiny. Analysis commensurate with the required confidence or assurance level must be performed before (and after) integration into a larger software-intensive system. [Goertzel 2005, section on security issues associated with acquired or reused components]. Where possible, this ideally would include financial analysis of life cycle costs.

If one is willing to potentially spend what may be large amounts for analysis and review of source code, on the plus side for open source software versus software whose source is not available is the malicious code issue. While not specifically mentioning it, a reverberation of incidents similar to the Soviet pipeline explosion of 1982 appeared in *IST Features* in 2004 while discussing open source software. "During the Cold War, for example, the Soviet Union bought a lot of commercial software from the United States. US intelligence, however, had put spies in the software so they could track what the Soviets were doing – and Moscow knew nothing about it," the coordinator [Antonis Ramfos] says. "The same problem exists today for governments and corporations around the world, and that is why using open source is starting to become more widespread." [Antonis Ramfos 2004] This again emphasizes the need for confidence as well as product.

13.4.3 Acquirer Only

Approval and/or limitations on software resuse as part of the supplier's solution should be clearly stated in the contract. Part of this approval should include risk mitigation strategies and testing. In addition, conditions for approval may rely on the security history and posture of the software, including pedigree, vulnerability, and patch management system [Goertzel 2005, section on security issues associated with acquired and reused components]. Approvals may also be contingent on negotiated liabilities for loss or damage between the acquirer and supplier. See [NIST Special Pub 800-65, Appendix B] and [Rasmussen 2004, p. 2].

The amount and kind of information available about the software and its quality and credibility impact the amount of investigation, including testing required to produce the equivalent of an assurance case for the software that under other circumstances would mainly be supplied by the supplier.

13.4.4 Supplier Software Reuse as Part of Acquirer's Solution

The supplier may also be an acquirer of reuseable software. In that event, the supplier should also refer to Section 13.4, Acquisition and Software Reuse–Acquisition/Supplier. As part of good software development practices, the supplier should also have strict controls over software reuse in a contract solution.

13.4.5 Evaluating Reusable Software

A Software Assurance Case is a good mechanism for providing a reasoned, auditable argument to support the contention that reusable software will satisfy security (and safety) requirements. See Section 3, Fundamental Concepts and Principles, [MOD Def Std 00-42, Part 3, 2003] and [Goertzel 2005] for evaluation criteria and sources of evidence that might be included in a Software Assurance Case for reusable software components. Also Section 8, Secure Software Verification, Validation, and Evaluation for more on assurance cases.

If reusable software is a portion of a larger system, then its assurance case must integrate properly with other assurance arguments in the remainder of the larger assurance case. The same is, of course, true of any the contractor's suppliers and in the worst case for the entire supply chain.

13.5 Request for Proposals – Acquirer

13.5.1 Scope

This section includes the first formal stage of requesting work from suppliers to satisfy a software integration or development need. When an organization needs to enter into a contract for the acquisition of secure software, normally a written request is issued to prospective developers/offerors/vendors (herein after called suppliers). This request is sometimes referred to as a request for proposal (RFP) [FAR Subpart 15.203]. The acquirer should consider incorporating the following in the RFP:

- Special terms and conditions related to software assurance. Also see [NIST Special Pub 800-64, Appendix B]
- Software assurance needs and requirements (capabilities and constraints defined in the Program Initiation stage), including measures/metrics in SOW as well as the requirements for software assurance case. Also see Section 3, Fundamental Concepts and Principles and “Development” sections [FAR 15.204-2(c)]
- Instructions to suppliers on the information they must submit to be evaluated, including software assurance case and related deliverables [FAR Subpart 15.204-5(b)]
- Appropriate evaluation criteria for software assurance [FAR 15.204-5(c) and FAR 15.304]. See the section “Source Selection – Acquirer” for evaluation criteria.

In addition to the request for proposal, the organization should develop other documents to manage the selection of the supplier. In the US federal government one such document is called the Source Selection Plan [FAR 15.300] that would include guidance on what is to be evaluated, how the information should be evaluated, and who should be involved in the evaluation. See Section 13.7, Source Selection-Acquirer for the source selection plan.

13.5.2 Software Assurance Terms and Conditions

Terms and conditions should be considered to identify software assurance responsibilities assigned to parties under a contract even to the extent that such obligations survive after the period of performance [NIST Special Pub 800-64, p.21]. In addition, terms and conditions on the right to audit (security review of the code and other security-relevant engineering artifacts), the use of secure coding practices, and security warranties should also be considered [Rasmussen 2004, pp. 1-2]. The following is a discussion on recommended terms and conditions taken from the NIST Special Publication 800-64. The NIST publication divides terms and conditions into 10 categories. See [NIST Special Pub 800-64, Appendix B]. The following is an expanded discussion on terms and conditions relevant to software assurance and is illustrative of what should be considered:

- Legal responsibilities may include the following:
  - Security violations. These can even be caused when the software is performing correctly. Some examples include backdoors (software companies may include these to assist customers), malicious code, etc.
  - Allocating contractual risk and responsibility. Acquirers may wish to include clauses to address allocating responsibility for integrity, confidentiality, and availability. Consideration should also be given to using guarantees, warranties, and liquidated damages (i.e., providing for the supplier to compensate the acquirer for losses or damage resulting from security issues).
  - Remediation. As used here, this would be the process of tracking and correcting software-related security flaws by the supplier. The terms and conditions should require the supplier to have a procedure acceptable to the acquirer for acting on reports of software-related security flaws [NIST Special Pub 800-64; Rasmussen 2004].

- Software Assurance Training. Security violations are often a result of the poorly trained supplier personnel. Consideration should be given in requiring a software assurance training program. [NIST Special Pub 800-50] provides a suggested model for establishing security training and awareness programs that could be used for a software assurance training program.
- FOCI. When appropriate, consideration should be given to requiring investigation into the chain of ownership. This may be appropriate if foreign ownership, control or influence are concerns, or if, in a business context, possible conflicts of interest, or rival control of a potential supplier is an issue.
- Information Security Features in Software-Intensive Systems. These refer to specific functions that can be incorporated into or those integral to the software. Consideration should be given to including terms and conditions that require certain controls for:
  - Access, identification and authentication, auditing, cryptography (data authentication, digital signature, key management, security of cryptographic modules [FIPS PUB 140-2], cryptographic validations), software integrity, software architecture, and media sanitation
  - Non-bypassibility and self-protection of security functionality
- Software-related security Acceptance Criteria – COTS. The acquirer should consider terms/conditions requiring the supplier to configure the security features as specified by the acquirer, to require the supplier to demonstrate the software is fully functional on hardened platforms, to require that software updates not change any configuration settings without the permission of the acquirer, and to deliver vulnerability test reports.

Example

The following is an example of a particular term/condition on the security of commercial software. [This is work that is currently in progress by the US Chief Information Officer's Council to be incorporated into the FAR.]

Securely configuring commercial software

(a) In the performance of this contract, any commercial off the shelf (COTS) software delivered by [contractor] shall be configured in conformance with applicable system configuration requirements established by [acquirer] pursuant to [44 USC 3544 (b) (1) (D)] (iii) as enumerated in Exhibit [X].

(b) [Supplier] shall maintain such COTS software so as to assure that it conforms to the most current version of all applicable [acquirer] system configuration requirements throughout its operational life cycle.

(c) For both custom and COTS software, with initial delivery and each subsequent release, [supplier] shall provide written assurance that the software operates satisfactorily on systems configured in accordance with section (a) above

(d) Prior to acceptance of the software by [acquirer] [supplier] shall deliver a complete vulnerability test report of both system and application vulnerability of the application running on the operating system and platform proposed to be used in production prior to initial acceptance and for each subsequent software release.

(e) Any exceptions to conformance with the [acquirer's] system configuration requirements must be approved in advance and in writing.

13.5.3 Software Assurance and the Common Criteria (CC) in the Acquisition Process

Use of the CC does not necessarily answer questions of software assurance. However, IA and IA-enabled software products that have a designated CC Evaluation Assurance Level (EAL) may be incorporated into a final solution. For more IA and IA-enabled products see [NSTISSP No. 11, para (5) through (11)] and [NSTISSAM INFOSEC/2-00]. The DoD includes software components in this category [DoDD 8500.1, para 4.17].

Assuming that a Software Assurance Case is the primary mechanism for establishing the measure of acceptable confidence, one can use the CC EAL as a comparison in asserting a level of assurance. The delta is the change that is needed to attain the appropriate level of confidence stated in the assurance case argument.

Protection Profiles are implementation independent statements of security requirements that address threats that exist in a specified environment. Protection Profiles could be used in creating a specification for secure software as a basis of acquisition. Also see [DoDI 8500.2, para 5.6.3] and [CC 2005, Part 1, section 8 and Annex B]. The Software Assurance Case uses the specification (among other items) in forming its arguments and identifying the supporting evidence [CC, 1999, pp. 11-12].

13.5.4 Software Assurance Measures and Metrics in the Acquisition Process

Measures and metrics in the acquisition process usually are used to determine whether the supplier is performing adequately under a contract. There are no known predictive measures for software assurance. See Section 8, Verification, Validation, and Evaluation, for more on measures. Suggestive measures include but are not limited to, counts of vulnerabilities created and discovered and process measures (i.e., measuring the supplier's training and fidelity in following them), etc. The acquirer needs to select measures based on the level of trust that is needed for different parts of the software. The measures and metrics should be determined in part by the Software Assurance Case.

In that measures and metrics used for software assurance may be a subset of a contract's measurement program, the acquirer/supplier should incorporate them into the larger acquisition measurement method/system used by the acquirer's organization. The following are among the methods that one might explore for use in the measurement of contract performance:

- Service Level Agreements (SLA) are suggestive of a method for expressing and contractually agreeing to specific measures of performance. [Gaines & Micahel 2005] suggests using SLAs for software development and [Rasmussen 2005] suggests SLAs for software assurance contract requirements.
- Earned Value Management Systems (EVMS) are defined in [ANSI/EIA 748-1998], an industry standard that the US federal government requires internally and of its suppliers. EVM is a tool that provides visibility into contract technical, cost, and schedule planning, performance, progress [DoD EVMS 2005, pp. 1-3]. The US federal government requires using EVMS in many of its contracts [OMB A11, Part 7, pp. 3, 5, 14, 26, and Exhibit 300]. If EVMS is used, the acquirer should ensure that software assurance is included.
- For other suggestive measurement models that may be relevant to software assurance see [NIST Special Pub 800-55], [CJCSI 3401.03A], [MOD Def Std 00-42, Part 3, 2003], and [Murdoch 2005, sections 6, 7, and 8].

13.5.5 Software Assurance Language for a Statement of Work to Develop Secure Software, Including Incentives

The written request for proposals should include a statement of requirements. The statement of requirements is often called a Statement of Work (SOW), Statement of Objectives (SOO), Work Statement (WS), or Performance-Based Work Statement (PBWS). The work statement could include requirements for software certification and accreditation (see "System Accreditation and Auditing Needs," "Design Reviews for Security," "System Accreditation," and "Recertification and Accreditation" sections), personnel software assurance education and training programs, software assurance case, verification and validation for software-related security, software assurance plan, software architecture (with security controls included), and minimum software-related security measures and countermeasures, etc. The statement of work might also specify required standards [See FAR 11.102 as an example of required use of Federal Information Processing Standards Publications.] As another example, FAR 39.101(d) requires US federal agencies to include "appropriate information technology security policies and requirements" when acquiring information technology. Appendix A includes notional language for a statement of work. The sample is not intended to be all inclusive. Rather, the sample is intended to motivate thought on what might be included.

Also see [FAR Subpart 11.2] and [MIL-HDBK-245D, figure 1 and para 3.6].

13.5.6 Develop Software Assurance Language for a Statement of Work to Acquire COTS or Commercial Items

Acquirers (and suppliers) are motivated to use COTS software (e.g., cost savings, time savings, etc.). This motivation may be counter to software assurance – so when given a choice and if the risk of COTS integration is not acceptable, software assurance should take precedence over COTS. Purchasing COTS or commercial software items or products exposes an organization's information infrastructure to malicious attacks from intentional or unintentional vulnerabilities (e.g., backdoors, malicious code, etc.). Also, the acquirer does not normally have access to the COTS source code, resulting in acquirer's inability to evaluate the COTS in the same manner in which organically developed software is evaluated for security vulnerabilities. To minimize the potential of damage because of poor software assurance practices, provisions can be included in the statement of work or in terms and conditions (see "Software Assurance Terms and Conditions" in this section) that requires the software vendor to include statements of assurance against unintentional and intentional vulnerabilities and provide warranties to support those assurances. Also, see [NIST Special Pub 800-64, Rev. 1, Appendix B].

In cases where commercial software products are security enabled [NSTISSP No. 11], countries or organizations may establish their own security evaluation process. An example of this is the US government's NIAP for the Evaluation of Commercial Off-The-Shelf (COTS) Security Enabled Information Technology Products. See [NIST Special Pub 800-23], [NSTISSP No. 11, para (5) through (11)], and [NSTISSAM INFOSEC/2-00]. This program evaluates and accredits security-enabled information technology products (including software products) at licensed/approved evaluation facilities in the US or in other countries participating in the Common Criteria Recognition Arrangement (CCRA) for conformance to the Common Criteria for IT Security Evaluation (ISO Standard 15408). The Common Criteria certificates issued to those software products apply only to the specific versions and releases of those products. These certificates do not endorse the "goodness" of a product, but, rather, represent the successful completion of a validation that the product met Common Criteria requirements for which it was evaluated/tested [CCEVS 2005].

13.5.7 Software Assurance Language for the Instructions to Suppliers

In addition to the statement of work, the organization should also include instructions on the information the supplier must submit to enable the organization to evaluate the supplier's proposal for developing the (secure) software [FAR 15.204(b)]. These instructions may include what to submit to answer foreign ownership, control or influence (FOCI) concerns, the content of the initial Software Assurance Case and the initial Software Assurance Plan, and the content for the initial software architecture.

See Appendix B for notional language. The sample is not intended to be all inclusive. Rather, the sample is intended to motivate thought on what can be included.

13.6 Preparation of Response--Supplier

13.6.1 Scope

The response to a request for proposals should reflect the software assurance capabilities that have been specified in the terms and conditions, statement of work, and evaluation factors, including the incorporation of an assurance case.

The supplier must submit adequate information in response or risk being eliminated from the competition. If any questions exist on the meaning of the instructions, suppliers should not hesitate to request clarification. The request for clarification must be clear and comprehensive to minimize subsequent requests for clarification.

Also, as a word of caution, the acquirer may request software assurance-related information that is evaluated as a “go or no-go” factor. This means that the supplier must adequately address software assurance in the first round or face elimination from the competition. Acquirers often use this kind of factor as a first order of eliminating suppliers. If this technique is used, suppliers must take extreme care in providing adequate information regarding software assurance in the initial. Also, see Section 13.7, Source Selection – Acquirer. and the references noted in Section 13.5, Request for Proposals-Acquirer.

13.6.2 Initial Software Architecture

A software architecture is a design plan that assigns and portrays roles and behavior among all IT assets. Software-related security-related roles and behaviors must be integrated into the overall software architecture in a manner that facilitates the software assurance case. Also, see relevant software architecture references. See [Hofmeister 2000] and [Bass 1998] for software architecture concepts. [Hofmeister 2000, para 1.2.2] identifies a suggestive model for creating different views (conceptual, module, and execution) and suggests engineering questions that should be answered by each view. Software assurance questions that need to be answered by the software architecture can be added to the list for each view.

13.6.3 Initial Software Assurance Plan

During the acquisition/supply of software, there are a myriad of plans required to ensure quality, security/safety, timely delivery within cost, and appropriate engineering practices are used. These plans include the Software Development Plan (reflects security requirements and assurance-oriented activities including assurance case); Project Risk Management Plan; Software Test Plan (software-related security vulnerability testing, user testing, stress testing and/or IV&V); Systems Engineering Plan; Project Quality Assurance Plan; Project Configuration Management Plan; Project Training Plan; Project Information Assurance or Security Plan; Project Incident Response Plan; and Software Contingency Plan. Also, see ISO 15026.

Acquirers/suppliers may opt to include appropriate software assurance items in those plans or opt to develop a stand-alone plan. The acquirer may require a supplier to submit an initial Software Assurance Plan as a part of the proposal. If this is the case, the winning supplier will likely be expected to refine the initial plan throughout the life cycle of the software effort. The contents of a Software Assurance Plan may include but is not limited to the following:

- Plan Purpose and Scope.
- Software Systems Identification and Description.
- Software Assurance Manager: Name of person responsible for software assurance.
- Software Staff Qualifications: Names, education, experience related to software development and software-related security.

- Software-related security Risks and Mitigation: Identification of vulnerabilities and risks associated with the development and deployment of software and the steps that will be taken to mitigate those risks. These include risks across the entire lifespan of the software and address the software and its environment, including the security of and within its development environment.
- Software Assurance Case (security and possibly other dependability-related) goals, structure, arguments, and evidence).
- Software Assurance Case Management: Include a discussion on the contents of the case, how the case will evolve and be managed throughout the software development life cycle, configuration management of the case, verification, validation, and evaluation of software-related security.
- Software Assurance Life Cycle: Include software assurance tasks, products, and documentation/records by phase.
- Software Assurance Case Management: Include a discussion on the contents of the case, how the case will evolve and be managed throughout the software development life cycle, configuration management of the case, verification, validation, and evaluation of software-related security.
- Software Requirements and Traceability: Include a discussion on traceability management to include protection of the software.
- Tool Support (special or unique software assurance modeling and simulation).
- Subcontract Management (special or unique software assurance requirements).
- Process and Product Certification.

13.7 Source Selection–Acquirer

13.7.1 Scope

This subsection includes considerations and issues related to software assurance in selecting a supplier. See [FAR Subpart 15.3] for notional source selection processes and techniques.

13.7.2 Develop Software Assurance Evaluation Criteria

Evaluation criteria are included in the Request for Proposals and are used to determine the best supplier for the job. Therefore, to distinguish clearly the best fit for the job, only significant factors and significant subfactors should be included [see FAR Subpart 15.204(c) and FAR Subpart 15.304]. When software-related security or assurance is an issue, ensuring their serious consideration during evaluation implies including a software assurance criterion that is no less than a significant subfactor. The software assurance evaluation criterion might include (but not be limited to) the quality of the software assurance case, quality of the software assurance plan and architecture, experience in developing secure software (or experience in integrating secure software components), past software assurance performance under similar acquisition efforts, training program for secure software, quality of personnel and software-related security expertise, and quality of software assurance integrated within other management and technical considerations. Any element of an appropriate assurance case is relevant here – see section 3.3.6.

Another consideration would be to include critical software assurance factors as “go no-go” factors. In other words, the supplier’s proposal for software assurance considerations is either acceptable or not acceptable at the outset. If not acceptable, then the supplier is no longer considered in the competition.

For addition information in constructing evaluation criteria see [ANSI/PMI 99-001-2004, section 12.2.3.2], [[DAU (SEF) 2001, p. 197-200] and [DoD PMI 2003, section 12.2.3].

13.7.3 Software Assurance in the Source Selection Plan

A Source Selection Plan guides the source selection process. The plan includes roles and responsibilities of the source selection team, a description of the process to be followed, and the evaluation criteria with specific instructions on how to perform the evaluation against the criteria. Also, see [DAU (SEF) 2001, p. 197-200] for general information on source selection plans.

Assuming that one of the significant factors or subfactors is software assurance considerations, the acquirer should ensure that an individual experienced in software assurance as it relates to the specific acquisition is part of the source selection team.

13.8 Contract Negotiation and Finalization

13.8.1 Scope

This subsection includes the discussions between acquirer and supplier and their finalizing contract requirements, terms, and conditions [see FAR 15.306].

13.8.2 Contract Negotiations

During negotiations, both the acquirer and supplier give and take on requirements, terms, and conditions. It is important that the give and take on software assurance requirements, terms, and conditions do not compromise the ultimate assurance goals or the specific critical assurance goals for the software intensive system.

The acquirer may find that contractors will push back on the software assurance requirements because the contractor may not be fully competent to do the job or willing to take the risk. The acquirer may find that suppliers may over bid because of the perceived risk and doing something they have never done. The acquirer should consider share-in-savings (savings as a result of implementing software assurance requirements as stated). The sharing includes not only costs and benefits but also the willingness to afford the supplier more time to engage in the education and training that is needed. An alternative would be to consider a contract type that shifts the burden of some of the risk to the acquirer and/or provide additional cost or performance incentives [see FAR Subpart 16.1 and FAR Subpart 16.3].

13.9 Project/Contract Management – Acquirer/Supplier

13.9.1 Scope

The subsection discusses the acquirer and supplier considerations and issues in integrating and managing software assurance during the project/contract.

Also see [ANSI/PMI 99-001-2004, Chapter 4] and [DoD PMI 2003, Chapter 4] for general project integration tips. These tips would also apply to integrating software assurance into the overall software-intensive system project.

13.9.2 Project/Contract Management

The acquirer and supplier should consider a separate plan for overseeing software assurance-related reviews and audits, including oversight of any sub-contract management activities. In addition, the acquirer must ensure that competent software assurance professional(s) oversee the supplier's delivery of software assurance. The supplier must ensure that competent software assurance professional(s) oversee their software assurance requirements delivery as well as oversee the competence of software professionals who provide software assurance capabilities under the contract. In addition, software assurance management should play a significant role in overall project management.

The following should be considered in overseeing software assurance delivery:

- How frequently will the supplier provide assurance statements and an updated Software Assurance Case? How will the updated Software Assurance be evaluated?
- If a software assurance SLA is used (see Section 13.5.4, Software Assurance Measures and Metrics Process), how will performance be evaluated?
- If an EVMS is used, how is software assurance incorporated?
- What role does software assurance play in software certification and accreditation?
- How will the software system's architecture be managed and what will be reviewed from a software assurance perspective?
- How often will the software risk management plan be updated and software assurance risks evaluated?
- What is the mechanism for elevating software assurance issues? Is there an issues resolution plan/process?
- Under what circumstances will software assurance intelligence updates (FOCI review) be conducted?
- If corrective actions are needed in software assurance, how will these be monitored?
- If a share-in-savings is negotiated, how will software assurance savings be measured and the supplier rewarded?
- If software assurance experience is important, how will the experience level be monitored? Will key software personnel be required and how will this be monitored? If a software assurance training program is required, how will this be monitored?
- What will be the involvement in testing?

13.10 Further Reading

13.10.1 General

[Abadi 2003] M. Abadi, "Built-in Object Security", Proceedings: European Conference on Object Programming (ECOOP) 2003, Darmstadt, Germany, July 2003

[Anderson 2001] R. Anderson. Security Engineering, "A Guide to Building Dependable Distributed Systems," Wiley Computer Publishing, 2001.

[Berg 2007] Berg, Ryan, "Secure at the Source:: Implementing Source Code Vulnerability Testing in the Software Development Life Cycle" Ounce Labs, 2007, Accessed 8/31/2007 http://www.ouncelabs.com/abstracts/Software-Security-Testing-SDLC.asp

[Boehm 2000] Boehm, B. and Basili, V.R., "Gaining Intellectual Control of Software Development." IEEE Computer Vol. 33, No. 5, May 2000, pp. 27-33

[CASIS3 2004] *Third Annual Conference on the Acquisition of Software-Intensive Systems*, sponsored by the Software Engineering Institute (SEI) and the Office of the Under Secretary of Defense (Acquisition, Technology, and Logistics), Defense Systems, Software-Intensive Systems, January 26-28, 2004. Available at http://www.sei.cmu.edu/products/events/acquisition/2004-presentations/

[DACS API] DACS Gold Practice, *Acquisition Process Improvement*. Available at http://www.goldpractices.com/practices/api/

[Falcarin 2003] Falcarin, Paolo and Maurizio Morisio, “Developing Secure Software and Systems” Special Issue on security software, Software Practice and Experience, vol. 33, issue 5, 2003

[Francic 2005] Francis, Bob, “Security Vendors Respond to Heightened Concerns,” InfoWorld, June 20, 2005, Accessed 8/14/2007, http://akamai.infoworld.com/article/05/06/28/HNsecprod_1.html

[Gamble 2005] Gamble, M. T, R. Gamble and M. Hepner. “Understanding Solution Architecture Concerns,” Proceedings: 2nd international workshop on Models and processes for the evaluation of off-the-shelf components, ACM Press, New York, NY, 2005

[GAO 2004] GAO, *Defense Acquisitions: Stronger Management Practices Are Needed to Improve DoD’s Software-Intensive Weapon Acquisitions*, GAO Report GAO-04-393, March 2004. Available at http://www.gao.gov/new.items/d04393.pdf.

[Giorgini 2006] P. Giorgini, F. Massacci, J. Mylopoulos, and N. Zannone. Requirements Engineering for Trust Management: Model, Methodology, and Reasoning. International Journal of Information Security, 2006

[Jacobson 1998] Jacobson, Ivar, Martin Griss, and Patrik Jonsson, “Software Reuse: Architecture, Process, and Organization for Business Success”, Addison Wesley, 1998.

[Lodderstedt 2002] Lodderstedt, T, D. A. Basin, and J. Doser, “SecureUML: A UML-based Modeling Language for Model-Driven Security” Proceedings: UML 5th International Conference, Dresden, Germany, 2002

[Massacci 2006] Massacci, F. and N. Zannone, “Detecting Conflicts between Functional and Security Requirements with Secure Tropos: John Rusnak and the Allied Irish Bank” Technical Report DIT-06-002, University of Trento, 2006

[NASA 1989] NASA Software Assurance guidebook, NASA GSFC MD, Office of Safety and Mission Assurance, 1989

[NCSC 1994] National Computer Security Center, “A Guide to Procurement of Single and Connected Systems,” NCSC Technical Report-004, July 1994

[Turner 2002] Turner, R.G., *Implementation of Best Practices in U.S. Department of Defense Software-Intensive System Acquisitions*, Ph.D. Dissertation, George Washington University, 31 January 2002. Available at http://www.goldpractices.com/survey/turner/index.php.

13.10.2 OTS

[Anderson 2002] Anderson, Tom, Mei Feng, Steve Riddle, Alexander Romanovsky, “Protective Wrapper Development: A Case Study, School of Computing Science, University of Newcastle upon Tyne, Newcastle upon Tyne, 2002

[Bishop 2001] Bishop, Peter, Robin Bloomfield, Tim Clement, Sofia Guerra, “Software Criticality Analysis of COTS/SOUP, Adelard, London, 2001

[Butertre 1997] Dutertre, B, “A Study of COTS and Component Integration for Safety Critical Systems”, Technical report, UK Ministry of Defense (MOD) Task Report, August 1997

[C. Guerra 2003] P. A. de C. Guerra, C. Rubira, A. Romanovsky, R. de Lemos, “Integrating COTS Software Components into Dependable Software Architectures”, Proceedings: 6th IEEE International

Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'03). Hokaido, Japan. May 2003

[Cooper 2005] Cooper, Kendra and Lawrence Chung “Managing Change in an OTS-aware Requirements Engineering Approach,” Proceedings: 2nd international workshop on Models and processes for the evaluation of off-the-shelf components, ACM Press, New York, NY, 2005

[Dawkins 2000] Dawkins, S., and S. Riddle, “Managing and supporting the use of COTS”, In F. Redmill, T. Anderson (eds.), Lessons in System Safety: Proc. 8th Safety-Critical Systems Symposium, 2000

[Estublier 2001] Estublier, J., H. Verjus and P. Cunin, “Designing and Building Software Federations,” Proceedings: 1st Conference on Component Based Software Engineering (CBSE), Varsovie, Poland, 2001

[Fenelon 1994] Fenelon, Peter and McDermid, John, “New Directions in Software Safety: Causal Modeling as an Aid to Integration,” ASAM-II, High Integrity Systems Engineering Group, Department of Computer Science, University of York, York, U.K., 1994

[Franch 2005] Franch, Xavier and Marco Torchiano, “Towards a Reference Framework for COTS-based Development: A Proposal,” Proceedings: 2nd international workshop on Models and processes for the evaluation of off-the-shelf components, ACM Press, New York, NY, 2005

[Gorton 2003] Gorton, Ian, Anna Liu and Paul Brebner, “Rigorous Evaluation of COTS Middleware Technology” IEEE Computer Society, March 2003

[Goseva-Popstojanova 2001] Goseva-Popstojanova K and K.S. Trivedi, "Architecture-based approach to reliability assessment of software systems" Performance Evaluation, v45 n2-3, July 2001

[Graland 1995] Garland, David, Robert Allen, and John Ockerbloom, “Architectural Mismatch or Why it's hard to build systems out of existing parts.” Proceedings: 17th International Conference on Software Engineering, ACM SIGSOFT, pages 179--185, Seattle, Washington, April 1995.

[IEEE 1998] IEEE Computer, “Special issue on COTS”, IEEE Computer, 31(6), 1998

[Jaccheri 2002] Jaccheri, L. and Torchiano, M, “Classifying COTS Products”, Proceedings: European Conference on Software Quality, Helsinki, 2002

[Jilani 1998] Jilani, Labed L. and A. Mili, “Estimating COTS Integration: an Analytic Approach”, Software Engineering Research Center (SERC), Ball State University, 1998

[Jones 2001] Jones, C, R E Bloomfield, P K D Froome and P G Bishop, “Methods for assessing the safety integrity of safety-related software of uncertain pedigree (SOUP), UK Health and Safety Executive, Adelard, London,2001

[Kapfhammer 2000] Kapfhammer, G., C. Michael, J. Haddox, R. Coyler, “An Approach to Identifying and Understanding Problematic COTS Components,” Presentation: ISACC 2000, The Software Risk Management Conference

[Kim 2001] Kim, Wook K. and Jongmoon Baik, “Dynamic Model for COTS Glue Code Development and COTS Integration,” Raymond J. Madachy, Editor, “Software Process Dynamics” P.274, Wiley-IEEE Press, 2001

[Kontio 1995] Kontio J. and R. Tesoriero, “A COTS Selection Method and Experiences of its Use”, Proceedings, the Twentieth Annual Software Engineering Workshop, Greenbelt, Maryland, November 1995

[Kontio 1996] Kontio, Jyrki, “A Case Study in Applying a Systematic Method for COTS Selection”. In H. Dieter Rombach, editor, Proc. 18th Int'l Conf. on Software Engineering (ICSE'96), Berlin, pages 201--209. ACM/IEEE-CS Press, N.Y., March 1996

[Kunda 1999] Kunda Douglas and Brooks Laurence, "Applying a Social-technical Approach for COTS Selection," Proceedings: 4th UKAIS Conference, University of York, McGraw Hill. April 1999

[Lutz 1993] Lutz, R, "Analyzing Software Requirements Errors in Safety-Critical, Embedded Systems," Proceedings of the IEEE International Symposium on Requirements Engineering, Jan 1993, pp. 126-133

[McIlroy 1968] McIlroy, M.D, “Mass Produced Software Components,” In P. Naur and B. Randel, editors, NATO Conference on Software Engineering. NATO Science Committee, October 1968

[Meyer 1992] Meyer B, “Programming by Contract” In D. Mandrioli, B. Meyer (eds.), Advances in Object-Oriented Software Engineering. Prentice Hall, 1992

[Mohamed 2005] Mohamed, Abdallah, Guenther Ruhe, Armin Eberlein, “Decision Support for Customization of the COTS Selection Process,” Proceedings: 2nd international workshop on Models and processes for the evaluation of off-the-shelf components, ACM Press, New York, NY, 2005

[Morisio 2000] Morisio, M, Seaman, C. B., Parra, A. T., Basili, V. R., Kraft, S. E., and Condon, S. E., “Investigating and Improving a COTS-Based Software Development Process” Proceedings: 2000 International Conference on Software Engineering. ACM, New York, 2000, 32-41.

[Morisio 2002] Morisio, M, C. B. Seaman, V. R. Basili, A. T. Parra, S. E. Kraft, and S. E. Condon. “COTS-based Software Development Processes and Open Issues”, The Journal of Systems and Software, 61(3):189--199, 2002

[Murray 2005] Murray, John F, “Guidance for Industry - Cybersecurity for Networked Medical Devices Containing Off-the-Shelf (OTS) Software,” U.S. Department of Health and Human Services, January 14, 2005

[Ochs 2000] Ochs, M.A, Pfahl, D.; Chrobok-Diening, G.; Nothhelfer-Kolb, B, “A COTS Acquisition Process: Definition and Application Experience,” Proceedings of the 11th ESCOM Conference, Shaker, Maastricht, 2000. Pp.335-343

[Réquilé-Romanczuk 2005] Réquilé-Romanczuk, Annya, Alejandra Cechich, Anne Dourgnon-Hanoune and Jean-Christophe Mielnik, “Towards a Knowledge-based Framework for COTS Component Identification ,” Proceedings: 2nd international workshop on Models and processes for the evaluation of off-the-shelf components, ACM Press, New York, NY, 2005

[Solberg 2001] Solberg, Håkon and Karl Morten Dahl, "COTS Software Evaluation and Integration issues" Norwegian University of Technology and Science, SIF8094, Software Engineering, Project, November 2001 Accessed 8/31/2007 http://citeseer.ist.psu.edu/cache/papers/cs/24138/http:zSzzSzwww.idi.ntnu.nozSzgrupperzSzsuzSzsif8094-reportszSzp14.pdf/solberg01cots.pdf

[Stavridou 2007] Stavridou, Victoria, “COTS, Integration and Critical Systems,” SRI International, Accessed 8/31/2007, http://coblitz.codeen.org:3125/citeseer.ist.psu.edu/cache/papers/cs/823/http:zSzzSzwww.csl.sri.comzSzdsazSz.zSzpubliszSziee.pdf/cots-integration-and-critical.pdf

[Torchiano 2002] Torchiano, Morisio, M. "Definition and Classification of COTS: a proposal" Proceedings, International Conference on COTS Based Software Systems (ICCBBS), Orlando (FL), February 2002

[Torchiano 2002a] Torchiano, Marco, Letizia Jaccheri, Carl-Fredrik Sørensen and Alf Inge Wang, "COTS Products Characterization" Proceedings" 14th International Conference on Software Engineering and Knowledge Engineering (SEKE'02), 2002

[Voas 1996] Voas, J, F. Charron, K. Miller, "Robust software interfaces: Can COTS--based systems be trusted without them?" Proceedings: 15th International Conference, Computer Safety, Reliability, and Security (SAFECOMP'96), 1996, pp. 126-135

[Voas 1998] Voas, J, "Certifying Off-The-Shelf Software Components", IEEE Computer, 31(6), 1998, 53-59.

[Voas 1998a] Voas, J, "The Challenges of Using COTS Software in Component-Based Development," IEEE Computer, 31(6):44-45, June 1998. 21

[Yacoub 2000] Yacoub, Sherif, Ali Mili, Chakri Kaveri and Mark Dehlin, "A Model for Certifying COTS Components for Product Lines" Proceedings: First Software Product Line Conference, Denver, August, 2000

13.10.3 Software Security Accreditation and Certification

[Casey 1988] Casey, T., Vinter, S., Weber, D., Varadarajan, R., and Rosenthal, D, "A Secure Distributed Operating System," Proceedings: Symposium on Security and Privacy, April 1988, IEEE Computer Society, pp. 27—38

[Denning 1977] Denning, D.E, and P.J. Denning,"Certification of Programs for Secure Information Flow. Communications of the ACM, 20(7):504–513, 1977

[DoD 1985] DoD 5200.28-STD, "Department of Defense Trusted Computer System Evaluation Criteria," December 26, l985

[Feiertag 1977] Feiertag R. J., K. N. Levitt, and L. Robinson, "Proving Multi-Level Security of a System Design, Proceedings: Sixth ACH Symposium on Operating Systems Principles, November 1977, 57-65.

[Feiertag 1979] R. J. Feiertag and P. G. Neumann, "The Foundations of a Provably Secure Operating System (PSOS)," Proceedings, American Federation of Information Processing Societies (AFIPS), National Computing Conference (NCC 79), pages 329--334, New York, NY, USA, June 1979

[Neumann 2003] Neumann, P.G, and R.J. Feiertag, "PSOS revisited," Proceedings: 19th Annual Computer Security Applications Conference (ACSAC 2003), Las Vegas, December 2003

[Neumann 2003a] Neumann P.G, "Principled Assuredly Trustworthy Composable Architectures," Final report, SRI Project 11459, Computer Science Laboratory, SRI International, Menlo Park, California, 2003

[NIST 1981] Federal Information Processing Standard (FIPS) Publication 74, "Guidelines for Implementing and Using the NBS Data Encryption Standard," National Institute of Standards and Technology, April, 1981

[NIST/NSA 1992] NIST/NSA, "Federal Criteria for Information Technology Security: Volume 1: Protection Profile Development,". National Institute of Standards and Technology & National Security Agency, Fort Meade, MD, December 1992.

[Pfleeger 1988] Pfleeger, C., and Pfleeger, S, "A Transaction Flow Approach to Software Security Certification for Document Handling Systems", Computers and Security 7, 5 (October 1988), 495—502

[Rushby 1993] Rushby, John, “Formal Methods and the Certification of Critical Systems”, Technical Report, Computer Science Laboratory, SRI International, Menlo Park CA, December 1993

[Zheng 2005] Zheng, Jiang, Brian Robinson, Laurie Williams and Karen Smiley, “A Process for Identifying Changes when Source Code is not Available ,” Proceedings: 2nd international workshop on Models and processes for the evaluation of off-the-shelf components, ACM Press, New York, NY, 2005

13.10.4 Secure Requirements Approaches

[Alberts 2001] Alberts, Christopher J, Audrey J. Dorofee and Julia H. Allen, “OCTAVE Catalog of Practices, Version 2.0,” Technical Report, CMU/SEI-2001-TR-020, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa. October 2001

[Arnab 2004] Arnab Ray, Bikram Sengupta, and Rance Cleaveland, “Secure Requirements Elicitation through Triggered Message Sequence Charts”, Lecture Notes in Computer Science Springer Berlin / Heidelberg, 2004

[Biskup 1998] Biskup, J, U. Flegel, and Y. Karabulut, “Secure Mediation: Requirements and Design,” Proceedings: 12th Annual IFIP WG 11.3 Working Conference on Database Security, Chalkidiki, Greece, July 1998

[Ellison 1997] Ellison, R.J.; Fisher, D.; Linger, R.C.; Lipson, H.F.; Longstaff, T, and Mead, N.R. “Survivable Network Systems: An Emerging Discipline”, Technical Report (CMU/SEI-97-TR-013): Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 1997
[Linger 2002] Linger, Richard C. Howard F. Lipson, John McHugh, Nancy R. Mead and Carol A. Sledge, “Life-Cycle Models for Survivable Systems” Technical Report, CMU/SEI-2002- Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, October 2002

[Ellison 2002] Ellison, Robert J. and Andrew P. Moore, “Trustworthy Refinement Through Intrusion-Aware Design (TRIAD),” Technical Report, CMU/SEI-2003-TR-002, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, October 2002 (Revised March 2003)

[Ellison 2004] Ellison, Robert J, Andrew P. Moore, Len Bass, Mark Klein and Felix Bachmann, “Security and Survivability Reasoning Frameworks and Architectural Design Tactics”, Technical Note, CMU/SEI-2004-TN-022, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, September 2004

[Hartson 1980] Hartson, H. Rex, “Architectural Approaches to Secure Databases”, ACM SIGSMALL Newsletter, Volume 6, Issue 1, June July 1980, pp. 16-24

[Leveson 1994] Leveson, N. G, M. P. E. Heimdahl, H. Hildreth, and J. D. Reese, “Requirements Specification for Process Control Systems” IEEE Transactions on Software Engineering, 20(9):684--707, September 1994

[Linger 1997] Linger, R.; Mead, N.; Lipson, H, “Requirements Definition for Survivable Network Systems”, 1997, Accessed 8/31/2007, http://www.cert.org/research

[Mead 2003] Mead, Nancy R, “Requirements Engineering for Survivable Systems”, Technical Note, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, CMU/SEI-2003-TN-013, September 2003

[Rosum 1992] Rozum, James A, “Software Measurement Concepts for Acquisition Program Managers” Technical Report, CMU/SEI-92-TR-11, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa. 1992

[Serpanos 2002] Serpanos, D.N and Voyiatzis, A.G,"Secure Network Design: A Layered Approach, Proceedings: The 2nd International Workshop on Autonomous Decentralized System, Volume, Issue, 6-7 Nov. 2002 Page(s): 95 - 100

[Thompson 1999] Thompson, J. M., M.P.E. Heimdahl, and S. P. Miller, "Specification-Based Prototyping for Embedded Systems", Proceedings: 7th Embedded Systems Expo (EXEC/FSE), September 1999

13.10.5 Acquisition Risk Management

[Ketchenham 1996] Kitchenham, Barbara, Pfleeger, Shari Lawrence, Software Quality: The Elusive Target, IEEE Software 13, 1 (January 1996) 12-21

[Lutz 2000] Lutz, R, "Software Engineering for Safety: A Roadmap," in A. Finkelstein, editor, "The Future of Software Engineering." ACM Press, New York, 2000

[Mazzanti 1995] Mazzanti, F, "Coding Regulations for Safety Critical Software Development," 2nd IEEE Software Engineering Standards Symposium, 1995, p. 134

[Rosenberg 1998] Rosenberg, Linda H, Theodore Hammer and Jack Shaw, "Software Metrics and Reliability," Proceedings: 9th International Symposium on Software Reliability Engineering, Paderborn Germany, 1998

[Rosenberg 1999] Rosenberg, Linda H, Theodore Hammer and Albert Gallo,"Continuous Risk Management at NASA," Proceedings: Quality Week Conference, San Francisco, California, May 1999

13.10.6 Reusable Software in the Acquisition Process

[Eisner 1995] Eisner, H., "Reengineering the Software Acquisition Process using Developer off-the-shelf Systems (DOTSS)", IEEE International Conference on Intelligent Systems for the 21st Century, Volume 5, Issue, 22-25 Oct 1995 Page(s):3971 - 3976

[Giaria 2000] Giaria, A.J, "Codifying ATS Software Components for Acquisition and Procurement Proceedings: AUTOTESTCON 2000, IEEE 2000 Page(s):56 - 67

[Kontio 1995] Kontio, Jyrki, "OTSO: A Systematic Process for Reusable Software Component Selection", University of Maryland, Technical report, December 1995

[SEI 2007] Software Acquisition Pilot Studies, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2007, Accessed 9/1/2007 http://www.sei.cmu.edu/programs/acquisition-support/pilot-intro.html

13.11 Appendices

13.11.1 APPENDIX A: *NOTIONAL* Language for the Statement of Work

1.0 “1.0 Secure Software”[2]

1.1 Key definitions:

1.1.1 “Secure software” means “highly secure software realizing – with justifiably high confidence but not guaranteeing absolutely – a substantial set of explicit security properties and functionality including all those required for its intended usage.” [Redwine 2004, p. 2] One can also state this in a negative way as “justifiably high confidence that no software-based vulnerabilities exist that the system is not designed to tolerate.” That definition incorporates the appropriate software-related security controls for a software intensive system’s security category to meet software-related security objectives.

1.1.2 “Software-related security controls” mean the management, operational, and technical controls (i.e., safeguards or countermeasures) prescribed for a software information system to protect the confidentiality, integrity, and availability of the system and its information.

1.1.3 “Security category” means the characterization of information or an information system based on an assessment of the potential impact that a loss of confidentiality, integrity, or availability of such information or information system would have on organizational operations, organizational assets, or individuals.

1.1.4 “Software-related security objectives” means confidentiality, integrity, availability, authenticity, accountability, and non-repudiation.

1.1.5 “Software assurance case” means a reasoned, auditable argument created to support the contention that the defined software-intensive system will satisfy software-related security requirements and objectives.

1.1.6 Include other appropriate definitions--

1.2 Security Category [NOTE: This is an example, also see FIPS Pub 199 and DoDI 8500.2, Enclosure 4.]:

1.2.1 This software system is used for large procurements in a contracting organization and contains both sensitive and proprietary supplier information and routine administrative information. For the sensitive supplier information, the potential impact from a loss of confidentiality is moderate (e.g., the loss may result in a significant financial loss), the potential impact from a loss of integrity is moderate (e.g., the loss may result in the effectiveness of the contracting mission is significantly reduced and there is significant damage to the information asset), the potential impact from a loss of availability is low (e.g., the loss may result in downtime, but there is backup). For the routine administrative information, the potential impact from a loss of confidentiality is low, the impact from a loss of integrity is low, and the impact from a loss of availability is low.

1.2.2 Based on 2.1, the resulting security category of the software system is {(confidentiality, moderate), (integrity, moderate), (availability, low)}

1.3 Software-related security Requirements. Based on the security category for the software system, the minimum security requirements specified in [NOTE: Reference the external document(s)] are required.

[2] This language is provided to provoke thought and has not been reviewed by attorneys for its efficacy.

(NOTE: Minimum security controls may be specified in this paragraph or in an external document similar to FIPS Pub 200; NIST SP 800-53; and DoDI 8500.2, Enclosure 4].

1.4 Software Assurance Case. The contractor shall refine the Software Assurance Case throughout the development process. This assurance case should be based on the software-related security requirements. The contractor shall submit the case for review – [NOTE: Specify when the case should be reviewed, such as when submitting the software design, etc.] Lastly, the successful execution of the Software Assurance Case shall be a condition for final acceptance of the software.

1.5 Auditing the Code. The supplier shall have an independent verification and validate (V&V) performed on the code to determine the security posture of the code. This verification and validation shall be performed by a qualified [NOTE: specify what "qualify" means] software assurance V&V entity. [NOTE: Also see "Secure Software Verification, Validation, and Evaluation" section in this CBK]

1.6 Software Assurance Practices. The supplier shall use software assurance practices in accordance with [NOTE: either explain those practices or provide a reference document].

1.7 Software Assurance Plan. The supplier shall refine, throughout the life cycle of this software development work, the Software Assurance Plan that was submitted with the supplier's proposal. The Software Assurance Plan shall be submitted to the acquirer [XX] days after each development milestone for review. [NOTE: Include how often this should be delivered. As a suggestion, the revisions to this plan should be submitted at key milestones. Such milestones might be after requirements analysis, after software architectural design, after detailed software design, after coding and testing. See the "Development" section in this CBK. Also, see ISO/IEC 12207, 5.3.] This plan shall include but not be limited to: [State what is to be included. See Section 13.6, Preparation of Response – Supplier in this section.]

1.8 Software Assurance Risk Management. The supplier shall maintain a formal software assurance risk management program. Within [XX] days of the award of the contract, the supplier shall deliver a Software Assurance Risk Management Plan to the acquirer for review. [NOTE: This could be a section in the Software Assurance Plan.]

13.11.2 APPENDIX B: *NOTIONAL*[3] Language for Instructions to Suppliers

2.0 Foreign ownership, control or influence (FOCI) is a concern. For any software product that the supplier intends to acquire or develop, the supplier shall answer the following questions:

2.1 Need to develop questions

3.0 Software Assurance Case

3.1 In order for the acquirer to evaluate the proposed software assurance capabilities, the offeror must submit an initial System Description and Software Assurance Case that addresses the required security properties and functionality and the arguments and evidence (existing or proposed) that the properties are preserved and all relevant laws, regulations, standards and other legal or societal requirements are met, the organizational and system security policies are adhered to, and security objectives are met for the software intensive system as specified in the statement of work and other characteristics as specified in paragraph 2.2 below. This initial Software Assurance Case will subsequently become a part of the contract and be used by the acquirer as an acceptance condition.

3.2 A software assurance case should present a convincing argument that the software intensive system will operate in an acceptably secure manner. The case should present definitive evidence, supported by process, procedure, and analysis, that a system and its software will be acceptably secure throughout its life cycle, including termination. [See Fundamentals section's subsection on Assurance Case for a list of kinds of evidence might explicitly call for here.] The case should demonstrate that within the totality of the environment in which the software will operate, any problems created by the software itself failing to operate as required, have been identified and assessed and that any necessary amelioration has been made. Subsequent to contract award, the contractor shall modify the case as development progresses, and knowledge of the system and its software increases. All the lifecycle activities, resources, and products, which contribute to or are affected by the security of the software, also need to be covered by the case.

Similar to a risk analysis, the following questions are examples of what should be of interest, be examined, and be analyzed through the use of the software assurance case:

a) How can a software-related security violations or failure occur?
b) What might the causes and conditions of the security violation or failure be?
c) What are the consequences of security violations or failure?
d) What is the level of criticality of the consequences?
e) How often is it likely that the security violation or failure will occur?

Software assurance cases are likely to contain significant numbers of complex inter-dependencies that typically result from a wide range of related analyses. They often rest upon a number of explicit, as well as implicit, assumptions and can have a long history, going through multiple versions in the course of their production. Both product, process, and resource issues need to be addressed in the case; it must be shown both that the system meets its software assurance requirements and that the processes for deriving the requirements, constructing the system, and assessing the system are of appropriate depth, breadth, and integrity. The analyses crucially depend upon the formulation of suitable models of the system, at various levels of abstraction, produced during the development process. Given these characteristics, it should be noted that software assurance cases are resource intensive to create and support.

4.0 Initial Software Assurance Plan.

4.1.1 The supplier shall submit an initial Software Assurance Plan.

[3] This language is provided to provoke thought and has not been reviewed by attorneys for its efficacy.

4.1.2 The Plan shall include [NOTE: See “Preparation of Response – Supplier” in this section of the CBK.]

5.0 Initial Software Description

5.1.1 The supplier shall submit an initial Software Architecture and such other description as needed to provide a structure for the assurance case.

The Software Architecture shall include an initial description of the software components and connector, including software-related security-related aspects. [NOTE: Include additional explanation.]

14Tips on Using this Body of Knowledge

14.1 Purpose and Scope

This section provides some suggestions on how one can use this document. The advice given is tentative, as the document has not been widely circulated yet.

This section is not intended to replace professional knowledge or experience in curriculum development, instruction, self-study, standards development, product evaluation or other areas of application. Rather, it provides a high-level introduction, and combined with the references contained herein, can serve as a guide.

This document can be used by

- Educators – to influence and support curricula
- Trainers – to extend/improve contents of training of current workforce
- Acquisition personnel – as an aid in acquiring (more) secure software
- Evaluators and testers – to supplement content beyond what is covered in their current evaluations and consider its implications for evaluations of persons, organizations, and products
- Standards developers – to encourage and facilitate inclusion of security-related items in standards
- Experts – to solicit feedback by way of suggested modifications or validation
- Practitioners – as a high-level introduction to field and a guide for learning
- Program managers – to understand software-related security and assurance, including alternative approaches, risks, prioritization, and budget

After covering general considerations applicable to most users, educators and trainers are addressed specifically. Because self-study shares many of the concerns of instructor-led learning, it is covered next. Potential uses for developing curriculum, professional and product standards and guidelines are also covered. The section closes with coverage of evaluation and testing, acquisition, and some final remarks encouraging readers to share information about their experiences using this document.

While a number of experts in various areas may be quite capable of using this document without help, this section is intended to provide ideas to anyone who wishes to use it.[1]

14.2 General Considerations

Users of this document have three audiences to consider: themselves, associates, and the users of their products or services. Each audience has their own backgrounds and existing knowledge, capabilities, and motivations, and each must start from where they are, which may or may not match the knowledge presumed in this document. So let us first consider how to gain an adequate background in software engineering and security.

Some users of this document may simply need a refresher; some may come from backgrounds that do not include software engineering. Some who work in network security or information assurance may fall into this group. Even though many of these may have had instruction or experience in programming and software engineering, it may have become dated. Some may have experience with deficient software production

[1] Items in this section often reflect generally held opinions or individual experiences as selected by the author and editor. As with all contents of this document, users must exercise their own judgment concerning them. See disclaimer on back of title page.

processes. Unfortunately, learning the basics of good software engineering is not trivial. While none of the introductory software engineering textbooks currently reflect the complexities of real software production or adequately cover software-related security, the seventh edition of [Sommerville 2006]has extensive coverage of critical systems and has a subsection on security. Thus, it is a good place to start.

If on the other hand one needs an initial understanding of the wider field of computer security, [Anderson 2001] is probably the best first book for self-study (not a textbook).[2] While it has more than 600 pages, it is quite readable. If [Anderson 2001] it is simply too large, there are a number of shorter introductions to computer security. Many references given in this document would be suitable as well.

Associates, such as fellow professors or standards developers, may not have the motivation or time to acquire the needed background on their own. They may, on the other hand, already know some of what is covered in this document and may simply need to review it.

Of course, preparing end-users of products and services such as students or standards users will vary. In some cases, special effort may be needed to identify their background. For example, the amount of explanatory material included in a product should vary depending on how knowledgeable the audience is.

The second important consideration is what the objective is in using this document. The scope – breadth and depth – of the relevant knowledge heavily depends on these objectives and needs careful consideration, particularly when motivation or resources are limited.

The third consideration is how to proceed from your current knowledge to the desired knowledge. Much of the remainder of this section addresses this consideration for the different kinds of intended users of this document.

14.3 Use for Learning

This subsection covers the use of this document by educators in higher education, trainers of software professionals as well as self-study by practitioners. Instructors should also refer to Subsections 14.3.3, Some Topics that May Require Special Attention from Instructors, and 14.3.5 Education and Training Literature.

14.3.1 Use in Higher Education Instruction

Two approaches are suggested for incorporating knowledge areas described in this document:

1. Offering courses specific to software-related security and assurance
2. Covering software-related security and assurance throughout existing courses as it relates to the particular subject being covered

Because of the nature of institutions of higher education and the need for learning on the part of instructors or professors, the easiest first steps for most institutions and interested individuals would be to take the first approach – establish separate courses. An early offering could be a course in secure software engineering with an existing software engineering course as a prerequisite or a co-requisite.

Computer security courses have proven popular, but usually contain little of the software-oriented knowledge identified in this document. Efforts to include an introduction to this material within such courses would broaden students beyond network security and the assumption that software is simply a black box as well as motivate students to take courses in secure software engineering. In any case, prerequisite (or, if properly synchronized, co-requisite) introductory computer security courses can remove the need to teach basic security concepts within secure software courses and provide students with a useful context.

[2] Textbooks exist such as [Pfleeger 2003].

Likewise, introducing secure software engineering in traditional software engineering or programming courses could motivate students to take specialized courses and give them a realization that in the “real-world” software-related security is a common or “normal” concern. If using [Sommerville 2006]seventh edition, an initial step might be to include the section on requirements for critical systems followed by the section on security engineering.

If one takes the second approach of spreading security coverage throughout the curriculum, then the problem of limited instructional time and student study time must be addressed. If security is added, what subject matter will be removed? In addition, textbooks often do not explicitly cover it. This may not be as severe a problem as one might think, particularly in beginning courses.

For instance, in programming courses much of what needs to be taught early is what not to do. This means covering a smaller portion of languages such as C/C++ (better not to use early) or Java to avoid known dangerous features or practices and to restrict use as much as possible to analyzable subsets. This could fit well into existing approaches of teaching good style. A stronger step might be to use SPARK [Barnes 2003] or Alloy [Jackson 2006] so tools can emphasize and aid correctness.

Concern for correctness in early (and, indeed, all) courses could be reinforced with an insistence that students must be able to provide an argument or case for their program being correct that goes beyond evidence from tests. These could include arguments about understandability, design, lack of dangerous features or poor practices, explicitly knowing all the exceptions possible and avoiding or handing them, static analysis and metrics, arguments analyzing explicitly what can be concluded from the tests, proof-like arguments or program proofs, results of reviews, student history of correctness, and so on. More and more of these kinds of arguments might be expected as students progress. See section 3.3.6 for other ideas.

Design courses could potentially benefit from two new books, [Berg 2005] (which has questions at ends of chapters) and [Schumacher 2006]. Network programming courses might use an instructive article analyzing a rewrite of sendmail in patterns and principles, [Hafiz 2004].

14.3.1.1 Graduate

Graduate programs face a variety of incoming students that can be addressed using the guidance for training entry-level practitioners below. James Madison University, for example, has graduate students take an introductory course on the software engineering life cycle including secure, software aspects resulting in a common foundation for subsequent learning.

14.3.2 Using in Training

Training shares similar concerns as education but is more likely to focus on particular job roles or skills. Thus, those interested in training would do well also to read the prior subsections in 14.3 Use for Learning.

14.3.2.1 General

Training is, of course, not just about the training sessions, but about proper tailoring to client and student needs as well as ensuring successful on-the-job use of material, including follow-up. Beyond the training itself, key factors in successful training includes fit to the organization, availability of on-the-job help and reinforcement/enforcement, and support from co-workers and management.

One difference between education and training is less coverage of theory and abstract concepts in training. Section 0Principles, contains a number of abstractions, but these probably all deserve coverage in training, although depth of coverage might vary. Several books emphasizing coding practices give substantial coverage to software system security principles (3.4), indicating the authors felt these are important for practitioners. Among the fundamentals covered, however, the concrete aspects of items may not need to be covered if they are not immediately relevant to the context of the training.

The organization being trained might not use formal methods. This does not mean that individuals should not be made aware of its existence and potential. In addition, arguments for agreement among descriptions, such as specifications and design as well as the absence of certain information flows, are needed whether formal methods are used as evidence or not.

The Secure Software Design, Section 6, also contains a significant number of more abstract concepts and principles. Judgment is required in dividing instructor and student time among these and more concrete items such as the security features of the framework the students use for their development, e.g. .net. Key architects and designers, however, need to know and properly apply the design principles as well as the fundamental principles mentioned above.

Most organizations producing substantial systems are integrating parts produced elsewhere. This means that persons deciding what to obtain and use need to be aware of the issues covered in Section 13 Acquiring Secure Software, and able to deal with the relevant issues that apply to free or open source and commercial software.

Instructional content that motivates and inspires is discussed below to emphasize the differences in considerations between training entry-level and experienced practitioners and between existing and new employees. Users of all subsections should also refer to subsections 14.3.3 Some Topics that May Require Special Attention from Instructors, and 14.3.5 Education and Training Literature.

14.3.2.2 Entry-Level Practitioners versus Experienced Practitioners

Entry-level practitioners seldom have any significant knowledge or skill in secure software engineering or assurance. Indeed, some computer science graduates may not have had even a single course of any kind in software engineering, and most have no substantial education in computer security, although the latter is rapidly improving for network security. Information systems graduates, and to a greater extent software engineering graduates, may have more of a background in producing software systems, but these students add to the variation in student backgrounds. Thus, training may need to begin with relevant aspects of software engineering and most if not all training in **secure** software engineering must cover this in the beginning. On the other hand, most will not have to unlearn incorrect notions about software-related security. Some misconceptions may, however, exist, such as security simply being a matter of avoiding buffer overflows or that poor software quality is inevitable.

Variation in the knowledge and skill of incoming students in software engineering for systems where safety and security are not serious concerns is not a new problem. What is new is the variety of knowledge and skill in security and secure software. In many instances, any variety in secure software knowledge can be adequately dealt with by assuming students know nothing.

Entry-level practitioners are often new employees, and the next subsection will, therefore, be relevant. Entry-level practitioners – as do new employees – generally have a greater willingness to learn and change. Training entry-level practitioners is part of initiating them into the profession, and this is doubly so when training new entry-level employees for an organization. Initiation rites (or rites of passage) generally have powerful effects – good or bad – on future expectations and behavior and therefore need to be crafted with particular care.

Entry-level practitioners may have some software production job experience, such as being a summer intern. Depending on the experience, this can be helpful, but it may also mean that the student now believes “real” software development organizations are not rigorous about security or have misconceptions about what being truly serious about security means.

The potentially wide variation among entry-level students makes the instructor’s job of establishing a common foundation difficult. As mentioned above, facing a similar problem with students entering its MS in secure software engineering, James Madison University has graduate students take an introductory course on the software engineering life cycle, including secure software aspects resulting in a common foundation for subsequent learning. To conserve resources, industry might chose to provide additional training only to those who need it.

While training experienced practitioners benefits from their more extensive knowledge and skill derived from real-world experience, this same experience may cause them to have more difficulty seeing the need for new knowledge or skills and be less open to learning and change since much of industry today does not vigorously address software-related security. While entry-level personnel may be willing to assume the trainer or organizational management knows best what they need to learn, experienced personnel often are not willing to make that assumption. This may require instructors to spend even more time on motivation and relevance. Luckily, many examples exist to help with motivation. While others appear throughout the document, see particularly Sections 2, Dangers and Damage, and 7, Secure Software Construction, as well as the trade press and the places mentioned in 14.3.5.

Students will often know little about security or secure software. Prospective students need clear information about not only the required prerequisite levels of experience, knowledge, and skill, but also the motivations, benefits, goals, and content of the training. Even when prerequisite background and course objectives are clearly stated, students may arrive with inadequate backgrounds or incorrect expectations. The instructor must decide how to prepare for this. Techniques trainers already use for addressing this variety in other topics should be applicable here as well. These problems may be more acute when training experienced practitioners who are not all from the same specific organizational setting.

As we will see, some of the same factors mentioned in this subsection apply as well in the next subsection, New versus Existing Employees.

14.3.2.3 New versus Existing Employees

Generally, new employees are more open to change than existing employees are. They are more accepting of "This is the way we do things here." In contrast, training existing employees is often in the context of trying to change the organizations' processes and face the problems of organizational change efforts – see Section 10.5, Improving Processes for Developing Secure Software.

Instructor activities needed for successful training in an organizational context are described in [Delahaye 1998, p. 55],[3] and will not be enumerated here. However, a substantial number of activities before and after training sessions are just as essential to successful training as the training itself. As mentioned above, fit to the organization, availability of on-the-job help and reinforcement/enforcement, and support from co-workers and management are among the key factors. In addition, training is more effective when the participant can immediately apply the training rather than waiting a long time before applying it. This is also true when the entire team is trained at the same time, preferably following appropriate management orientation or training. Support prior to and during training from management and lack of opposition (and even better active support) from influential, technical personnel are critical for organizational change and acceptance of training.

Technical and manage personnel need to leave the training with a favorable opinion, as well as an enthusiasm and willingness to use the training. Therefore, motivation and indoctrination are generally critical elements in the training. Examples can help. While others appear throughout the document, see particularly Sections 2, Dangers and Damage, and 7, Secure Software Construction, as well as the trade press and the places mentioned in 14.3.5. Most effective for existing employees, however, is an example involving their team's product. Several organizations create and present such examples to their teams as part of training.

[3] [Kelly 1994] further addresses some of the organizational issues, e.g. on page 55.

14.3.3 Some Topics that May Require Special Attention from Instructors

This subsection highlights topics instructors may be less familiar with or which require significant effort for the instructor to learn or teach. These topics include the assurance case, analyzability, formal methods, penetration testing, laws and policy, and professional ethics including student recognition of his/her limitations.

14.3.3.1 Learning and Teaching Assurance Cases

Together, including section 3.3.6 on assurance cases and section 8.2 under Secure Software Verification, Validation, and Evaluation provide an initial text to read in this document to learn about assurance cases. An article giving context, background, and a gentle introduction is [Despotou 2004]. [ISO TR 15443 - 1] includes good security-oriented introductory material and a generally applicable ISO standard for a computer security assurance case. [SafSec Standard] is applicable and addresses safety and security,[4] and example safety cases are available at http://www.esafetycase.com.

14.3.3.2 Teaching Analyzability

The need for analyzability is the direct result of the need for predictability and verifiability and addresses the inherent weaknesses of testing, which does only point verification.

Establishing the state of the art or practice in analyzability is difficult because the answer depends on the power and availability of the techniques that exist at a given moment. For example, what can be predicted about the various aspects programs written in Java or various subsets of it? On the other hand, notations may be designed to permit certain kinds of analyses, for example the SPARK programming language with its accompanying analysis toolset.

The most common use of the term analyzability means a design or program can be proven to conform to its specification. A familiar example is program proofs showing agreement with pre- and post-conditions. But even for these, the state of the practice is not adequate to address the full breadth of the programming languages in common use.

Specifications and designs can use notations for which

- A number of properties can be shown – mostly mathematically or state-machine-based ones
- Little can be proved – most of UML

The latter has led to the existence of OCL within UML [OCL 2.0] and various UML extensions for security, e.g., [Jürjens 2004].

14.3.3.3 Customizing Instruction about Construction

The intent of the Secure Software Construction chapter is to identify elements of secure software construction that span the construction of software whatever particular language or environment is chosen. Therefore, it is necessarily general in its treatment. It is assumed that the reader will map topics in this section to the particulars of an operating system such as Microsoft Windows or Linux or to the particular programming language being used. No matter which operating system or language is assumed, each will have "common vulnerabilities" it must address or particular security-related strengths and weaknesses.[5]

[4] A tutorial on using SafSec for security is planned for IEEE International Symposium on Secure Software Engineering, March 13, 2006, Washington D.C. area.
[5] A construction-relevant body of knowledge is [Pomeroy-Huff 2005] for the Personal Software Process.

14.3.3.4 Teaching Formal Methods

As mentioned below in subsection 14.3.5 Education and Training Literature, several articles exist describing experiences in teaching formal methods. An older book also exists on the subject [Dean 1996] as well as a recent workshop [Duce 2003] and symposium [Dean 2004].

While formal methods or techniques of equivalent power are needed for highly secure software, formal methods in isolation are not a “silver bullet” or are appropriate for general application in such situations as those involving masses of legacy code. The Secure Software Tools and Methods chapter’s contents can help teaching how to identify criteria for selecting appropriate tools and the strengths of particular methods and choose a specific disciplined development approach as well as to understand different approaches to verifying the security of software. Furthermore, it identifies key characteristics of tools that can aid in producing secure software.

14.3.3.5 Teaching Reuse and Composability

Incorporating “reusable” software, including OTS and open source software, can have difficulties because of the quality or characteristics of the reused software or from verifying that the composition of the parts has the properties desired. Addressing the first can lead to discussions of what categories of software have fewer vulnerabilities. Long discussions on this subject are seldom beneficial – better to insist it is a case-by-case question of the particular software involved. Students may also have a hard time accepting that their favorite software has a significant number of defects and vulnerabilities. This seems to be particularly difficult if the software is open source.

Composing pieces and reasoning about the composition’s security properties from those of its parts and their interconnections can be quite difficult once any element of complexity is involved [Bishop 2002]. Students need to understand the hazards and the care that needs to be exercised.

14.3.3.6 Learning and Teaching Penetration Testing

Penetration testing is a subject whose techniques are continually evolving. A readable place to start, however, is [Whittaker and Thompson 2003]. An introduction to the hacking world can be found in the latest of the “Hacking Exposed” series of books. If one wants the latest up-to-date information, no substitute exists for learning from professionals in the field. Of course, only teach “ethical hacking.” A number of books exist on this subject; one is [EC-Council 2003] – the same organization offers an ethical hacking exam [EC-Council CEH 312-50].

14.3.3.7 Teaching Laws and Policy

This is another constantly changing field. A number of references are given in section 4 Ethics, Law, and Governance, but these can never be entirely up to date. On an advanced level or when desiring to be entirely up to date, this topic will need to be taught by or with the benefit of knowledgeable lawyers. Since most software products have or intend to have international sales or use, covering just the laws and regulations of the home country will likely not be enough.

14.3.3.8 Teaching Professional Ethics

Professional ethics need to be addressed early, before specific attack techniques are taught even if these are taught as part of teaching defense. To emphasize that students must behave ethically and to aid in protecting their institution from liability, Julie and Dan Ryan of George Washington University have produced an article and a statement for students to sign [Ryan 2002]. Requiring the signing of such statements is not unusual in information/network security education. Also, see [Endicott-Popovsky 2003]. Finally, student recognition of his or her limitations and knowledge of what to do if a problem lies outside their expertise are also important for professional practice.

14.3.3.9 Hands on Learning, Student Projects, and Industrial Pilot Projects

Meaningful examples and demonstrations are useful, but for a person to be a member of the workforce producing (more) secure software, practice and feedback are needed. In addition, many people learn best by doing. Achieving good exercises or projects for students or employees to do as part of learning can be a major effort, and they may require revision to reflect what happens when initially used. Organizations might establish a pilot project on which people learn, practice, receive feedback, and gain skill with ongoing, expert coaching. Learners also need exposure to failures before working on a production product to realize what may happen. Finally, practitioners need to have time to blend the proper techniques and concepts into their everyday work.

14.3.4 Training Educators and Trainers

In the summer of 2004, a month long intensive seminar was held at James Madison University to provide a foundation for teaching secure software engineering – although significant teacher preparation might still be required beyond this foundation to teach a specific course. The five participants were software engineering faculty with little or no background in computer security or secure software. Attendees rated the seminar well, and two successfully taught secure software engineering courses the following fall. For a copy of the syllabus, contact Sam Redwine (http://www.cs.jmu.edu/faculty/redwine.htm).

Microsoft has also held events to teach instructors, mainly from higher education: one in December 2005 and another in April 2006.

14.3.5 Education and Training Literature

Literature on teaching is available in ACM Special Interest Group on Computer Science Education (SIGCSE) publications and proceedings and in the Proceedings of the IEEE Conference on Software Engineering Education and Training (CSEET). The IEEE has an Education Society and Transactions on Engineering Education. Also of note is an IEEE Workshop on Secure Software Engineering Education and Training (WSSEET), April 2006. A Colloquium on Information Security Education (CISSE) is held every year.[6] The *Training and Development Journal* contains articles on training. In addition, relevant publications on education and training and computer and software-related security exist scattered throughout the computing literature and worldwide web.

Some articles address specific topics. Several items were referenced earlier in this section. Several articles describe experiences in teaching formal methods to undergraduates, for example [Sobel 2000] – also see references above in 14.3.3. [Mead 2005] reports an experience related to teaching security requirements.

Microsoft has sponsored developing the curriculum. Microsoft sites of interest that are not cited elsewhere include

- Curricula items – http://www.msdnaacr.net/curriculum/repository.aspx
- Archived event presentations – http://www.microsoft.com/industry/publicsector/eventpresosarchive.mspx
- Archived web casts – http://www.microsoft.com/technet/security/eventswebcasts/default.mspx
- 2003 Software-related security Summer Institute – http://research.microsoft.com/projects/SWSecInstitute/schedule.htm#Monday,%20June%2016

Educational materials are a part of the Nebraska University Consortium for Information Assurance site – http://nucia.ist.unomaha.edu/. James Madison University has a description of its graduate secure software engineering program at http://www.cs.jmu.edu/sse/. The Department of Defense has information assurance training and awareness materials, including CDs at http://iase.disa.mil/eta/prod-des.html. Finally, while not

[6] For CISSE see http://www.ncisse.org/conferences.htm

specifically addressing education and training, one should not forget the material available at https://buildsecurityin.us-cert.gov/.

14.3.6 Use by Practitioners for Professional Development

A practitioner can use this document in the following ways

1. Learn the general, widely applicable terminology, concepts, and principles
2. Learn a comprehensive overview of terminology, concepts, and issues in general and for a given major sub-area
3. Obtain an overview understanding of secure software
4. Learn the details of a sub-area
5. Thoroughly learn secure software engineering

A good beginning on the first can be obtained by reading sections 2 through 4 and preferably through 5, constituting the general sections plus the requirements section. While not as basic, the issues surrounding programming are the subject of much of the discussions in industry, so practitioners may also wish to read the section on construction. This can be done without concern for reading any references, although they should consider [Davis 2004], which is a good initial overview article for practitioners.

The second can be done by reading sections 2 through 4 and the section(s) related to the sub-area. These sections can be identified from the Table of Contents and the Index. These sections may refer to other sections as well.

For the third, a comprehensive overview, the reader needs to cover the text in at least sections 2-11 and preferably 12 and 13 as well.

The fourth item is similar to the second but requires using the references to learn details. One should consider in each reference what is relevant, new to the reader, or important and concentrate on these. Some skimming may be appropriate. In the end, the reader may find some gaps in the details, and search the digital libraries at www.acm.org and www.computer.org, or general search engines to find additional material. The same is true of the fifth, thorough understanding of secure software engineering, but the reader needs to cover at least sections 2-11 and preferable 12 and 13 as well.

Practitioners should keep the purpose and scope of this document firmly in mind. It presumes learners already know the relevant knowledge identified in the SWEBOK Guide [Abran 2004]. In addition, while some text and references may permit the most skillful to successfully perform some task, this document's purpose is to identify knowledge and not to teach skills. Finally, one should not forget the material available at https://buildsecurityin.us-cert.gov/ and elsewhere.

14.4 Using to Develop Standards and Guidelines

This document can aid standards and guideline developers to identify the scope of the high-level topics within their area of interest and provide guidance in seeking more information. The first may be achieved by reviewing the topics covered in this document within the area of interest. If this area corresponds to section(s) within this document, this is easy. If not, the index is available as well as document search functions when using an electronic version. Ensuring one knows enough about the area of interest may require using the references and the "Further Reading" items that appear at the end of each major section.

Standards and guideline developers need to recognize that this is neither a normative document nor a document identifying skills or competences. They are responsible for establishing what is "right" or "best" for their purposes and users.

Four areas are briefly addressed:

1. Curriculum
2. Professional Personnel
3. Professional Practice
4. Product Evaluation

14.4.1 Curriculum

With its increased importance, a number of curricula "standards" in higher education and elsewhere need to be revised to include software-related security. Of course, a team developing curricula standards should include experts in the relevant areas of secure software. This document can help such experts or other team members to identify and understand possible topics and possibly help in deciding what topics would be best to include.

Because many curricula have courses whose scope somewhat mirrors the structure of this document, mapping possible topics into existing course curricula may be relatively straightforward by topic. What may be less easy to ascertain is classifying topics as beginning, intermediate, or advanced; or by difficulty or importance. This is not surprising because supplying such a classification was not a goal of this document and may vary by kind of role or curricula. Experts on the team can make substantial contributions to these classification decisions.

14.4.2 Professional Personnel

This document can help identify topics that need to be covered. The key difficulty may be in mapping to a professional role, as this document did not have this as an objective. The developer of a certification exam for a certain role will need to classify topics and aspects of topics by professional level of competency and by importance to role.

One should be aware that persons in a number of roles do not currently use knowledge of secure software engineering or acquisition but would be better able to perform their role if they had competencies within these areas. In other words, perhaps they should have these knowledge or competencies, and professional standards should expand to include them.

14.4.3 Professional Practice and Processes

Many existing standards and guidelines are "process" oriented and constrain how something should be done. This document is organized mainly by processes and sub-processes, so some mappings may be relatively easy.

Currently, many do not adequately address software-related security. For example, some "assurance" standards maintainers may desire to expand their scope beyond safety to include security. Standards developers might wish to maintain consistency with existing standards such as ISO 15443 (including its emerging third part) or the emerging revision of ISO 15026.

Using terminology, however, may be a problem. For example, what are currently termed "integrity levels" in safety standards will need to be changed to something like "assurance levels" as "integrity" is used in certain ways in the security community and the "integrity level" usage will be awkward or confusing in that community.

This document may also help standards developers to recognize limitations either theoretical or in the state of art or practice that make standards in certain areas problematic.

14.4.4 Product Evaluation

A common shortcoming of product evaluations that this document can help in overcoming is using only a few kinds of evidence rather than a fuller set as listed in section 3.3.6. While not all of these may be significant in all areas, a great majority might be relevant for a particular standard. In addition, unless clear justification with substantial evidence exists, justifying another approach, good engineering or risk management would appear to require something in the nature of an assurance case to establish a system's real condition or justify rational confidence.

14.5 Use in Evaluation and Testing

What was stated in the prior subsection on product evaluation standards applies here as well – good engineering or risk management requires something similar to an assurance case to establish a system's real condition and justify rational confidence.

Section 8, Secure Software Verification, Validation, and Evaluation, is dedicated to this topic, but relevant knowledge is identified elsewhere in this document as well. Most major sections contain a subsection on reviews and an assurance case, and the Secure Software Requirements section has a number of relevant subsections – 5.2.10.1, 5.2.13, 5.3, 5.5, 5.6, and 5.2.12. Assurance cases are introduced and first explained in subsection 3.3.6.

This document can help evaluators and testers identify techniques. Deciding, for example, which testing techniques to use requires judgment and knowledge of local conditions as well as the merits and costs of the techniques.

Because a topic is covered in this document does not mean it should be used (indeed, wide usage does not guarantee merit). For example, there is widespread agreement exists that the Common Criteria process needs substantial changes. In addition, many believe Common Criteria results below EAL5 offer little evidence relevant to justified high confidence. This is doubly true when, as is often the case today, the production and evaluation of Common Criteria required artifacts are not integrated into the actual production process.

One testing technique commonly used is attack or penetration testing – see subsections 8.4.2.1 and 14.3.3.6. Brute force testing is also common, but proves little, since any software that fails (i.e., crashes) is simply poor software. Subsection 8.4.2.4 discusses Security Fault Injection Testing, which is a technique that is needed to test "defense in depth."

If the software producers restricted themselves to using (adequately) analyzable notations or descriptions (for example, by using formal methods), an evaluator has the potential to predict or verify the producer's prediction of security-relevant aspects of behavior. While this does not mean that reviews and testing should not also be performed, it can provide a level of assurance that can substantially contribute toward rational confidence that the software system meets its security-relevant policy and specifications.

In practice, other factors also can combine to give high levels of assurance; these appear to always include adequate time and resources, high-quality people with the right expertise (and experience), extensive review, a specification whose ambiguities have been resolved, and a clean, relatively stable architecture. Evaluators of systems where these are not true should be suspicious.

14.6 Tips on Using the Acquisition Section

14.6.1 Introduction

The primary intended users of Section 0 are those involved in software acquisition education and training, as well as standards developers, who need additional knowledge on acquiring secure software. While educators, trainers and standards developers are the primary users, buyers and suppliers may also find the knowledge in this section useful. The advice in this section provides ideas for anyone who wishes to use the Acquisition of Secure Software section.

14.6.2 About the Sample Language

The notional examples and illustrations that provide sample language are intended to provoke thought. Atorneys have not reviewed them for efficacy. The sample language should be modified and expanded to fit the users' particular acquisition environment. As an example, the sample statements of work may contain language that may be more appropriate in other sections of a request for proposals or contract as "terms and conditions." Likewise, any sample "terms or conditions" may be more appropriate in the statement of work.

14.6.3 Software Acquisition Education and Training

In using this section, the educator or trainer should first establish a specific or generic acquisition process description that is best suited for their educational environment. As an example, educators and trainers of US Department of Defense acquisition would use the process established in the 5000 regulation series, while other educators and trainers might use a process that is identified in a generally accepted IEEE of international standard. In addition, major sub-processes may be imbedded within the larger acquisition process, such as the (software) systems engineering process. In this case, the educator or trainer may wish to integrate acquisition within the context of a major sub-process. The educator or trainer may also wish to select the larger software lifecycle process.

Once the process is defined, relevant ideas presented in the Acquisition of Secure Software can be mapped and integrated into the selected process. Student materials can then be created by expanding the information using the references provided within each section and relating the acquisition of secure software knowledge to the selected process.

In addition to providing a single repository of knowledge on the acquisition of secure software, this section also provides subject matter for expanded research and other scholarly work. Risk-based approaches, pedigree management, and incentives in the acquisition process for providing secure software are examples of areas that need further exploration.

14.6.4 Standards Developers

Standards developers can use the knowledge in this section to establish new or modify existing standards. As an example, standards on recommended practice for software acquisition [IEEE 1062] or software life cycle [IEEE 12207] may be modified to incorporate ideas presented in the Acquisition of Secure Software section.

14.6.5 Buyers and Suppliers

Buyers and suppliers can use the sample statement of work or other language provided in this section to the extent that it is applicable to their acquisition. However, refer to the cautions on using the language.

14.7 Final Remark

Feedback from readers and users is needed to improve this section and the document as a whole. We welcome feedback of all kinds about the document. It will be greatly appreciated.

14.8 Further Reading

[Bell 2005] Bell, David Elliot. “Looking Back at the Bell-La Padula Model,” *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC ’05).* pp 337-351, December 2005.

[Epstein 2005] Epstein, Jeremy, Scott Matsumoto, and Gary McGraw. “Software-related security and SOA: Danger, Will Robinson!” *IEEE Security & Privacy*. Vol. 4, No.1, pp 80-83, January/February 2006.

[Fernandez 2005] E.B.Fernandez and Maria M. Larrondo-Petrie, “Using UML and security patterns to teach secure systems design,” *Proceedings of the American Society for Engineering Education Annual Conference (ASEE 2005)*. American Society for Engineering Education, 2005

[McGraw 2006] McGraw, Gary. *Software-related security: Building Security In*. Addison Wesley, 2006.

[Microsoft Security Regulatory Compliance Site] http://www.microsoft.com/technet/security/learning/compliance/all/default.mspx.

[Peterson 2006] Pederson, Allan, Navi Partner, and Anders Hedegaard. “Designing a Secure Point-of-Sale System,” *Proceedings of the Fourth IEEE International Workshop on Information Assurance (IWIA ’06).* pp 51-65, April 2006.

[Schlesinger 2004] Schlesinger, Rich (ed.). Proceedings of the 1st annual conference on Information security curriculum development. Kennesaw, Georgia, ACM, October 08 - 08, 2004.

[Snow 2005] Snow, Brian. “We need Assurance!” Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC ’05). pp 3-10, December 2005.

[Verton 2005] Verton, Dan. *The Insider: A True Story*. Llumina Press, 2005.

[Vizedom 1976] Vizedom, Monika, Rites and Relationships: *Rites of Passage and Contemporary Anthropology*, Beverly Hills, CA: Sage Publications, 1976.

15Mapping to the Security principles of software assurance

This matrix illustrates the relationship between the Security principles of software assurance and the sections of this document. This is a scaled-down version of the matrix aimed to provide the reader with a general understanding of how the concepts relate to one another. A full version of the matrix will be available on the Build Security In Web site at https://buildsecurityin.us-cert.gov.

The Security principles of software assurance

	1 The Adverse	1.1 Adversaries Intelligent and Malicious	1.2 Limit, Reduce, or Manage Benefits to Violators or Attackers	Think like an attacker	1.3 Increase Attacker Losses	1.3.1 Increase expense of attacking	1.3.2.1 Adequate detection and forensics	1.4 Increase Attacker Uncertainty	1.5 Limit, Reduce, or Manage Set of Violators	1.5.1 Users	1.5.2.1 Limit, remove, and discourage aspiration to be attacker	1.6 Limit, Reduce, or Manage Attempted Violations	1.6.1 Discourage violations	1.6.1.1.3 Psychological Acceptability	2 The System	2.1 Limit, Reduce, or Manage Violations	2.1.1 Limit, reduce, or manage origination or continuing existence of opportunities or possible ways for performing violations throughout system's lifecycle/lifespan	2.1.1.1 Accurate Identification	Separate Identity from Privilege	Positive Authorization	Least Exposure	2.1.1.4.1.3 Complete Mediation of Accesses	2.1.1.4.1.5 Least Privilege	2.1.1.4.1.6 Tamper Proof or Resistant
2 DANGERS AND DAMAGE 15																								
2.3 ATTACKERS18	■	■																						
2.4 METHODS FOR ATTACKS20		■																						
2.5 NON-MALICIOUS DANGERS TO SOFTWARE 23																								
2.6 ATTACKS ACROSS LIFECYCLE24		■																						
2.7 INFORMATION ABOUT KNOWN VULNERABILITIES AND EXPLOITS29																								

	1 The Adverse	1.1 Adversaries Intelligent and Malicious	1.2 Limit, Reduce, or Manage Benefits to Violators or Attackers	Think like an attacker	1.3 Increase Attacker Losses	1.3.1 Increase expense of attacking	1.3.2.1 Adequate detection and forensics	1.4 Increase Attacker Uncertainty	1.5 Limit, Reduce, or Manage Set of Violators	1.5.1 Users	1.5.2.1 Limit, remove, and discourage aspiration to be attacker	1.6 Limit, Reduce, or Manage Attempted Violations	1.6.1 Discourage violations	1.6.1.1.3 Psychological Acceptability	2 The System	2.1 Limit, Reduce, or Manage Violations	2.1.1 Limit, reduce, or manage origination or continuing existence of opportunities or possible ways for performing violations throughout system's lifecycle/lifespan	2.1.1.1 Accurate Identification	Separate Identity from Privilege	Positive Authorization	Least Exposure	2.1.1.4.1.3 Complete Mediation of Accesses	2.1.1.4.1.5 Least Privilege	2.1.1.4.1.6 Tamper Proof or Resistant
3 FUNDAMENTAL CONCEPTS AND PRINCIPLES 33																								
3.3 BASIC CONCEPTS 35																								
3.3.1 Dependability 35																								
3.3.2 Security 36																								
3.3.3 Software and other Security-related Concerns 37																								
3.3.4 Assets 37																								
3.3.5 Security-Violation-related Concepts38																■	■	■	■	■	■	■	■	■
3.3.6 Assurance 38																								
3.4 BASIC SOFTWARE SYSTEM SECURITY PRINCIPLES45																								
3.4.1 Least Privilege46																			■				■	
3.4.2 Complete Mediation 46																						■		
3.4.3 Fail-Safe Defaults.46																				■				
3.4.4 Least Common																								

	1 The Adverse	1.1 Adversaries Intelligent and Malicious	1.2 Limit, Reduce, or Manage Benefits to Violators or Attackers	Think like an attacker	1.3 Increase Attacker Losses	1.3.1 Increase expense of attacking	1.3.2.1 Adequate detection and forensics	1.4 Increase Attacker Uncertainty	1.5 Limit, Reduce, or Manage Set of Violators	1.5.1 Users	1.5.2.1 Limit, remove, and discourage aspiration to be attacker	1.6 Limit, Reduce, or Manage Attempted Violations	1.6.1 Discourage violations	1.6.1.1.3 Psychological Acceptability	2 The System	2.1 Limit, Reduce, or Manage Violations	2.1.1 Limit, reduce, or manage origination or continuing existence of opportunities or possible ways for performing violations throughout system's lifecycle/lifespan	2.1.1.1 Accurate Identification	Separate Identity from Privilege	Positive Authorization	Least Exposure	2.1.1.4.1.3 Complete Mediation of Accesses	2.1.1.4.1.5 Least Privilege	2.1.1.4.1.6 Tamper Proof or Resistant
Mechanism46																								
3.4.5 Separation of Privilege46																			■					
3.4.6 Psychological Acceptability 47														■										
3.4.7 Work Factor.47						■																		
3.4.8 Economy of Mechanism 47																								
3.4.9 Open Design47																								
3.4.10 Analyzability 47																								
3.4.11 Recording of Compromises.47																								
3.4.12 Defense in Depth 47					■	■	■																	
3.4.13 Treat as Conflict 48				■																				
3.4.14 Tradeoffs49																								
3.5 SAFETY AND SECURITY 49																								
3.5.1 Probability versus Possibility 49																								
3.6 SECURE SOFTWARE ENGINEERING50																								

	1 The Adverse	1.1 Adversaries Intelligent and Malicious	1.2 Limit, Reduce, or Manage Benefits to Violators or Attackers	Think like an attacker	1.3 Increase Attacker Losses	1.3.1 Increase expense of attacking	1.3.2.1 Adequate detection and forensics	1.4 Increase Attacker Uncertainty	1.5 Limit, Reduce, or Manage Set of Violators	1.5.1 Users	1.5.2.1 Limit, remove, and discourage aspiration to be attacker	1.6 Limit, Reduce, or Manage Attempted Violations	1.6.1 Discourage violations	1.6.1.1.3 Psychological Acceptability	2 The System	2.1 Limit, Reduce, or Manage Violations	2.1.1 Limit, reduce, or manage origination or continuing existence of opportunities or possible ways for performing violations throughout system's lifecycle/lifespan	2.1.1.1 Accurate Identification	Separate Identity from Privilege	Positive Authorization	Least Exposure	2.1.1.4.1.3 Complete Mediation of Accesses	2.1.1.4.1.5 Least Privilege	2.1.1.4.1.6 Tamper Proof or Resistant
3.6.4 Security-Related Architectural Concepts 53																								
3.6.5 Secure Software Development Activities57																								
3.6.6 Security Functionality 60																				■				
3.6.8 Security Risk Management for Software60																								
3.7 SECURITY PROPERTIES ELABORATED63																								
3.7.1 Confidentiality63																								
3.7.2 Integrity 65																								
3.7.3 Availability65																								
3.7.4 Accountability 66																								
4 ETHICS, LAW, AND GOVERNANCE 71																								
4.2 ETHICS71										■	■													
4.3 LAW.71											■													

	1 The Adverse	1.1 Adversaries Intelligent and Malicious	1.2 Limit, Reduce, or Manage Benefits to Violators or Attackers	Think like an attacker	1.3 Increase Attacker Losses	1.3.1 Increase expense of attacking	1.3.2.1 Adequate detection and forensics	1.4 Increase Attacker Uncertainty	1.5 Limit, Reduce, or Manage Set of Violators	1.5.1 Users	1.5.2.1 Limit, remove, and discourage aspiration to be attacker	1.6 Limit, Reduce, or Manage Attempted Violations	1.6.1 Discourage violations	1.6.1.1.3 Psychological Acceptability	2 The System	2.1 Limit, Reduce, or Manage Violations	2.1.1 Limit, reduce, or manage origination or continuing existence of opportunities or possible ways for performing violations throughout system's lifecycle/lifespan	2.1.1.1 Accurate Identification	Separate Identity from Privilege	Positive Authorization	Least Exposure	2.1.1.4.1.3 Complete Mediation of Accesses	2.1.1.4.1.5 Least Privilege	2.1.1.4.1.6 Tamper Proof or Resistant
5 SECURE SOFTWARE REQUIREMENTS 77																								
5.2 REQUIREMENTS FOR A SOLUTION 77																								
5.2.3 Asset Protection Needs.78																								
5.2.4 Threat Analysis80																								
5.2.5 Interface and Environment Requirements82																								
5.2.13 System Accreditation and Auditing Needs 85																								
5.3 REQUIREMENTS ANALYSES 86																								
5.3.1 Risk Analysis86																								
5.3.2 Feasibility Analysis87																								
5.3.3 Tradeoff Analysis87																								
5.4 SPECIFICATION 88																								
5.5 REQUIREMENTS VALIDATION 90																								

	1 The Adverse	1.1 Adversaries Intelligent and Malicious	1.2 Limit, Reduce, or Manage Benefits to Violators or Attackers	Think like an attacker	1.3 Increase Attacker Losses	1.3.1 Increase expense of attacking	1.3.2.1 Adequate detection and forensics	1.4 Increase Attacker Uncertainty	1.5 Limit, Reduce, or Manage Set of Violators	1.5.1 Users	1.5.2.1 Limit, remove, and discourage aspiration to be attacker	1.6 Limit, Reduce, or Manage Attempted Violations	1.6.1 Discourage violations	1.6.1.1.3 Psychological Acceptability	2 The System	2.1 Limit, Reduce, or Manage Violations	2.1.1 Limit, reduce, or manage origination or continuing existence of opportunities or possible ways for performing violations throughout system's lifecycle/lifespan	2.1.1.1 Accurate Identification	Separate Identity from Privilege	Positive Authorization	Least Exposure	2.1.1.4.1.3 Complete Mediation of Accesses	2.1.1.4.1.5 Least Privilege	2.1.1.4.1.6 Tamper Proof or Resistant
5.6 ASSURANCE CASE91																								
6 SECURE SOFTWARE DESIGN 95																								
6.3 PRINCIPLES AND GUIDELINES FOR DESIGNING SECURE SOFTWARE.97																								
6.3.2 Damage Confinement and System Resilience 99			■	■																				
6.3.3 Vulnerability Reduction100																								
6.4 DOCUMENTATION OF DESIGN ASSUMPTIONS101																								
6.4.1 Environmental Assumptions102																								
6.7 ARCHITECTURES FOR SECURITY 103																								
6.8 SECURITY FUNCTIONALITY104																								
6.8.1 Identity Management 105																		■	■					

	1 The Adverse	1.1 Adversaries Intelligent and Malicious	1.2 Limit, Reduce, or Manage Benefits to Violators or Attackers	Think like an attacker	1.3 Increase Attacker Losses	1.3.1 Increase expense of attacking	1.3.2.1 Adequate detection and forensics	1.4 Increase Attacker Uncertainty	1.5 Limit, Reduce, or Manage Set of Violators	1.5.1 Users	1.5.2.1 Limit, remove, and discourage aspiration to be attacker	1.6 Limit, Reduce, or Manage Attempted Violations	1.6.1 Discourage violations	1.6.1.1.3 Psychological Acceptability	2 The System	2.1 Limit, Reduce, or Manage Violations	2.1.1 Limit, reduce, or manage origination or continuing existence of opportunities or possible ways for performing violations throughout system's lifecycle/lifespan	2.1.1.1 Accurate Identification	Separate Identity from Privilege	Positive Authorization	Least Exposure	2.1.1.4.1.3 Complete Mediation of Accesses	2.1.1.4.1.5 Least Privilege	2.1.1.4.1.6 Tamper Proof or Resistant
6.8.2 Access Control Mechanisms 105																			■	■				
6.9 PROPER USE OF ENCRYPTION AND ENCRYPTION PROTOCOLS106																								
6.14 METHODS FOR TOLERANCE AND RECOVERY108																	■	■	■	■	■	■	■	■
6.15 DECEPTION AND DIVERSION108								■																
6.16 SOFTWARE PROTECTION109																								■
6.17 FORENSIC SUPPORT 110							■																	
6.18 USER INTERFACE DESIGN110														■										
6.19 ASSURANCE CASE FOR DESIGN111																								
7 SECURE SOFTWARE CONSTRUCTION115																								
7.4 CONSTRUCTION OF USER AIDS.124																								

	1 The Adverse	1.1 Adversaries Intelligent and Malicious	1.2 Limit, Reduce, or Manage Benefits to Violators or Attackers	Think like an attacker	1.3 Increase Attacker Losses	1.3.1 Increase expense of attacking	1.3.2.1 Adequate detection and forensics	1.4 Increase Attacker Uncertainty	1.5 Limit, Reduce, or Manage Set of Violators	1.5.1 Users	1.5.2.1 Limit, remove, and discourage aspiration to be attacker	1.6 Limit, Reduce, or Manage Attempted Violations	1.6.1 Discourage violations	1.6.1.1.3 Psychological Acceptability	2 The System	2.1 Limit, Reduce, or Manage Violations	2.1.1 Limit, reduce, or manage origination or continuing existence of opportunities or possible ways for performing violations throughout system's lifecycle/lifespan	2.1.1.1 Accurate Identification	Separate Identity from Privilege	Positive Authorization	Least Exposure	2.1.1.4.1.3 Complete Mediation of Accesses	2.1.1.4.1.5 Least Privilege	2.1.1.4.1.6 Tamper Proof or Resistant
8 SECURE SOFTWARE VERIFICATION, VALIDATION, AND EVALUATION135																								
8.2 ASSURANCE CASE 135																								
8.10 THIRD-PARTY VERIFICATION AND VALIDATION AND EVALUATION147																								
9 SECURE SOFTWARE TOOLS AND METHODS151																								
10 SECURE SOFTWARE PROCESSES.157																								

	2.1.1.4.2.1 Secure defaults	2.1.1.4.2.2 Secure Failure	2.1.1.4.2.7 Trusted Communication Channels	2.1.1.4.2.10 Least Common Mechanism	2.1.2 Hierarchical Protection	2.1.3 Learn, Adapt, and Improve	2.1.4 Limit, reduce, or manage undetected violations	2.1.5 Limit, reduce, or manage lack of accountability	2.1.6 Limit, reduce, or manage violations unable to respond to acceptably or learn from	2.1.7 Defense in Depth	2.2 Avoid Adverse Effects on System Benefits	2.2.1 Authorizations Fulfill Needs and Facilitate User	2.2.2 Encourage and ease use of security aspects	Quickly Mediated Access	Ease secure operation	2.2.3 Articulate the desired characteristics and tradeoff among them [jabir 1998]	2.2.4 Economic Security	Efficiently Mediated Access	2.2.5 High-Performance Security	2.2.6 Provide Privacy Benefit	2.2.7 Provide Reliability	2.3 Limit, Reduce, or Manage Security-related Costs	2.3.1 Limit, reduce, or manage security-related adverse consequences	Exclusion of Dangerous Assets
2 DANGERS AND DAMAGE 15																								
2.3 ATTACKERS18																								
2.4 METHODS FOR ATTACKS20																								
2.5 NON-MALICIOUS DANGERS TO SOFTWARE 23																								
2.6 ATTACKS ACROSS LIFECYCLE24																								
2.7 INFORMATION ABOUT KNOWN VULNERABILITIES AND EXPLOITS29																								
3 FUNDAMENTAL CONCEPTS AND PRINCIPLES 33																								
3.3 BASIC CONCEPTS 35																								
3.3.1 Dependability 35																					■			
3.3.2 Security 36																								
3.3.3 Software and other Security-related Concerns 37																								

	2.1.1.4.2.1 Secure defaults	2.1.1.4.2.2 Secure Failure	2.1.1.4.2.7 Trusted Communication Channels	2.1.1.4.2.10 Least Common Mechanism	2.1.2 Hierarchical Protection	2.1.3 Learn, Adapt, and Improve	2.1.4 Limit, reduce, or manage undetected violations	2.1.5 Limit, reduce, or manage lack of accountability	2.1.6 Limit, reduce, or manage violations unable to respond to acceptably or learn from	2.1.7 Defense in Depth	2.2 Avoid Adverse Effects on System Benefits	2.2.1 Authorizations Fulfill Needs and Facilitate User	2.2.2 Encourage and ease use of security aspects	Quickly Mediated Access	Ease secure operation	2.2.3 Articulate the desired characteristics and tradeoff among them [jabir 1998]	2.2.4 Economic Security	Efficiently Mediated Access	2.2.5 High-Performance Security	2.2.6 Provide Privacy Benefit	2.2.7 Provide Reliability	2.3 Limit, Reduce, or Manage Security-related Costs	2.3.1 Limit, reduce, or manage security-related adverse consequences	Exclusion of Dangerous Assets
3.3.4 Assets 37																								■
3.3.5 Security-Violation-related Concepts38	■	■	■	■	■	■	■	■	■	■														
3.3.6 Assurance 38																								
3.4 BASIC SOFTWARE SYSTEM SECURITY PRINCIPLES45																								
3.4.1 Least Privilege46																								
3.4.2 Complete Mediation 46																								
3.4.3 Fail-Safe Defaults.46	■	■																						
3.4.4 Least Common Mechanism46				■																				
3.4.5 Separation of Privilege46																								
3.4.6 Psychological Acceptability 47													■	■	■									
3.4.7 Work Factor.47																								
3.4.8 Economy of Mechanism 47														■	■			■						
3.4.9 Open Design47																								
3.4.10 Analyzability 47																								
3.4.11 Recording of Compromises.47																								

	2.1.1.4.2.1 Secure defaults	2.1.1.4.2.2 Secure Failure	2.1.1.4.2.7 Trusted Communication Channels	2.1.1.4.2.10 Least Common Mechanism	2.1.2 Hierarchical Protection	2.1.3 Learn, Adapt, and Improve	2.1.4 Limit, reduce, or manage undetected violations	2.1.5 Limit, reduce, or manage lack of accountability	2.1.6 Limit, reduce, or manage violations unable to respond to acceptably or learn from	2.1.7 Defense in Depth	2.2 Avoid Adverse Effects on System Benefits	2.2.1 Authorizations Fulfill Needs and Facilitate User	2.2.2 Encourage and ease use of security aspects	Quickly Mediated Access	Ease secure operation	2.2.3 Articulate the desired characteristics and tradeoff among them [jabir 1998]	2.2.4 Economic Security	Efficiently Mediated Access	2.2.5 High-Performance Security	2.2.6 Provide Privacy Benefit	2.2.7 Provide Reliability	2.3 Limit, Reduce, or Manage Security-related Costs	2.3.1 Limit, reduce, or manage security-related adverse consequences	Exclusion of Dangerous Assets
3.4.12 Defense in Depth 47										■														
3.4.13 Treat as Conflict 48																								
3.4.14 Tradeoffs49																■								
3.5 SAFETY AND SECURITY 49																								
3.5.1 Probability versus Possibility 49																■	■							
3.6 SECURE SOFTWARE ENGINEERING50																								
3.6.4 Security-Related Architectural Concepts 53																								
3.6.5 Secure Software Development Activities57																								
3.6.6 Security Functionality 60																								
3.6.8 Security Risk Management for Software60																								
3.7 SECURITY PROPERTIES ELABORATED63																								

	2.1.1.4.2.1 Secure defaults	2.1.1.4.2.2 Secure Failure	2.1.1.4.2.7 Trusted Communication Channels	2.1.1.4.2.10 Least Common Mechanism	2.1.2 Hierarchical Protection	2.1.3 Learn, Adapt, and Improve	2.1.4 Limit, reduce, or manage undetected violations	2.1.5 Limit, reduce, or manage lack of accountability	2.1.6 Limit, reduce, or manage violations unable to respond to acceptably or learn from	2.1.7 Defense in Depth	2.2 Avoid Adverse Effects on System Benefits	2.2.1 Authorizations Fulfill Needs and Facilitate User	2.2.2 Encourage and ease use of security aspects	Quickly Mediated Access	Ease secure operation	2.2.3 Articulate the desired characteristics and tradeoff among them [jabir 1998]	2.2.4 Economic Security	Efficiently Mediated Access	2.2.5 High-Performance Security	2.2.6 Provide Privacy Benefit	2.2.7 Provide Reliability	2.3 Limit, Reduce, or Manage Security-related Costs	2.3.1 Limit, reduce, or manage security-related adverse consequences	Exclusion of Dangerous Assets
3.7.1 Confidentiality63																								
3.7.2 Integrity 65																								
3.7.3 Availability65																								
3.7.4 Accountability 66								■																
4 ETHICS, LAW, AND GOVERNANCE 71																								
4.2 ETHICS71																								
4.3 LAW.71																								
5 SECURE SOFTWARE REQUIREMENTS 77																								
5.2 REQUIREMENTS FOR A SOLUTION 77																■								
5.2.3 Asset Protection Needs.78																■								
5.2.4 Threat Analysis80																■								
5.2.5 Interface and Environment Requirements82																■								
5.2.13 System Accreditation and Auditing Needs 85																■								
5.3 REQUIREMENTS ANALYSES 86																								

	2.1.1.4.2.1 Secure defaults	2.1.1.4.2.2 Secure Failure	2.1.1.4.2.7 Trusted Communication Channels	2.1.1.4.2.10 Least Common Mechanism	2.1.2 Hierarchical Protection	2.1.3 Learn, Adapt, and Improve	2.1.4 Limit, reduce, or manage undetected violations	2.1.5 Limit, reduce, or manage lack of accountability	2.1.6 Limit, reduce, or manage violations unable to respond to acceptably or learn from	2.1.7 Defense in Depth	2.2 Avoid Adverse Effects on System Benefits	2.2.1 Authorizations Fulfill Needs and Facilitate User	2.2.2 Encourage and ease use of security aspects	Quickly Mediated Access	Ease secure operation	2.2.3 Articulate the desired characteristics and tradeoff among them [jabir 1998]	2.2.4 Economic Security	Efficiently Mediated Access	2.2.5 High-Performance Security	2.2.6 Provide Privacy Benefit	2.2.7 Provide Reliability	2.3 Limit, Reduce, or Manage Security-related Costs	2.3.1 Limit, reduce, or manage security-related adverse consequences	Exclusion of Dangerous Assets
5.3.1 Risk Analysis86																								
5.3.2 Feasibility Analysis87																	■							
5.3.3 Tradeoff Analysis87																■								
5.4 SPECIFICATION 88																								
5.5 REQUIREMENTS VALIDATION 90																								
5.6 ASSURANCE CASE91																								
6 SECURE SOFTWARE DESIGN 95																								
6.3 PRINCIPLES AND GUIDELINES FOR DESIGNING SECURE SOFTWARE.97																								
6.3.2 Damage Confinement and System Resilience 99																								
6.3.3 Vulnerability Reduction100																								
6.4 DOCUMENTATION OF DESIGN ASSUMPTIONS101																								
6.4.1 Environmental Assumptions102																								

	2.1.1.4.2.1 Secure defaults	2.1.1.4.2.2 Secure Failure	2.1.1.4.2.7 Trusted Communication Channels	2.1.1.4.2.10 Least Common Mechanism	2.1.2 Hierarchical Protection	2.1.3 Learn, Adapt, and Improve	2.1.4 Limit, reduce, or manage undetected violations	2.1.5 Limit, reduce, or manage lack of accountability	2.1.6 Limit, reduce, or manage violations unable to respond to acceptably or learn from	2.1.7 Defense in Depth	2.2 Avoid Adverse Effects on System Benefits	2.2.1 Authorizations Fulfill Needs and Facilitate User	2.2.2 Encourage and ease use of security aspects	Quickly Mediated Access	Ease secure operation	2.2.3 Articulate the desired characteristics and tradeoff among them [jabir 1998]	2.2.4 Economic Security	Efficiently Mediated Access	2.2.5 High-Performance Security	2.2.6 Provide Privacy Benefit	2.2.7 Provide Reliability	2.3 Limit, Reduce, or Manage Security-related Costs	2.3.1 Limit, reduce, or manage security-related adverse consequences	Exclusion of Dangerous Assets
6.7 ARCHITECTURES FOR SECURITY 103																								
6.8 SECURITY FUNCTIONALITY104																								
6.8.1 Identity Management 105																								
6.8.2 Access Control Mechanisms 105																								
6.9 PROPER USE OF ENCRYPTION AND ENCRYPTION PROTOCOLS106			■																					
6.14 METHODS FOR TOLERANCE AND RECOVERY108	■	■	■	■	■	■	■	■	■	■														
6.15 DECEPTION AND DIVERSION108																								
6.16 SOFTWARE PROTECTION109																								
6.17 FORENSIC SUPPORT 110																								
6.18 USER INTERFACE DESIGN110																								
6.19 ASSURANCE CASE FOR DESIGN111																								

	2.1.1.4.2.1 Secure defaults	2.1.1.4.2.2 Secure Failure	2.1.1.4.2.7 Trusted Communication Channels	2.1.1.4.2.10 Least Common Mechanism	2.1.2 Hierarchical Protection	2.1.3 Learn, Adapt, and Improve	2.1.4 Limit, reduce, or manage undetected violations	2.1.5 Limit, reduce, or manage lack of accountability	2.1.6 Limit, reduce, or manage violations unable to respond to acceptably or learn from	2.1.7 Defense in Depth	2.2 Avoid Adverse Effects on System Benefits	2.2.1 Authorizations Fulfill Needs and Facilitate User	2.2.2 Encourage and ease use of security aspects	Quickly Mediated Access	Ease secure operation	2.2.3 Articulate the desired characteristics and tradeoff among them [jabir 1998]	2.2.4 Economic Security	Efficiently Mediated Access	2.2.5 High-Performance Security	2.2.6 Provide Privacy Benefit	2.2.7 Provide Reliability	2.3 Limit, Reduce, or Manage Security-related Costs	2.3.1 Limit, reduce, or manage security-related adverse consequences	Exclusion of Dangerous Assets
7 SECURE SOFTWARE CONSTRUCTION115																								
7.4 CONSTRUCTION OF USER AIDS.124																								
8 SECURE SOFTWARE VERIFICATION, VALIDATION, AND EVALUATION135																								
8.2 ASSURANCE CASE 135																								
8.10 THIRD-PARTY VERIFICATION AND VALIDATION AND EVALUATION147																								
9 SECURE SOFTWARE TOOLS AND METHODS151																								
10 SECURE SOFTWARE PROCESSES.157																								

	Retain minimal state	Tolerate Security Violations	2.3.1.3.1 • Limit damage	2.3.1.3.7.1 • Ensure system has a well-defined status after failure, either to a secure failure state or via a recovery procedure to a known secure state [Avizienis 2004]	2.3.1.4.1.4 • Make sure it is possible to reconstruct events	2.3.1.4.1.5 o Record secure audit logs and facilitate periodical review to ensure system resources are functioning, confirm reconstruction is possible, and identify unauthorized users or abuse	2.3.1.4.1.6 o Support forensics and incident investigations	2.3.1.4.1.7 o Help focus response and reconstitution efforts to those areas that are most in need	Avoid Single-Point Security Failure	2.3.1.5.2 Avoid Multiple Losses from Single Attack Success	2.3.1.5.3 Separation of Privilege	2.3.1.5.4 Defense in Depth	Allocation of Defenses according to Consequences	2.3.2 Limit, reduce, or manage security-related developmental and operational expenses	2.3.2.1.2 Ease (cost-effective and timely) Certification and Accreditation	2.4 Limit, Reduce, or Manage Security-related Uncertainties	2.4.1 Limit, reduce, or manage security-related unknowns	2.4.2 Limit, reduce, or manage security-related assumptions	2.4.3 Limit, reduce, or manage unpredictability of system behavior	Analyzability	2.4.4 Limit, reduce, or manage consequences or risks not addressed in assurance case	2.4.5 Limit, reduce, or manage consequences or risks related to uncertainty	Risk Sharing	2.4.6 Increase Assurance Regarding Product
2 DANGERS AND DAMAGE 15																								
2.3 ATTACKERS18																								
2.4 METHODS FOR ATTACKS20																								
2.5 NON-MALICIOUS DANGERS TO SOFTWARE 23																								
2.6 ATTACKS ACROSS LIFECYCLE24																								
2.7 INFORMATION ABOUT KNOWN VULNERABILITIES AND EXPLOITS29																								
3 FUNDAMENTAL CONCEPTS AND PRINCIPLES 33																								
3.3 BASIC CONCEPTS 35																								
3.3.1 Dependability 35																								
3.3.2 Security 36																								
3.3.3 Software and other Security-related Concerns 37																								

	Retain minimal state	Tolerate Security Violations	2.3.1.3.1 • Limit damage	2.3.1.3.7.1 • Ensure system has a well-defined status after failure, either to a secure failure state or via a recovery procedure to a known secure state [Avizienis 2004]	2.3.1.4.1.4 • Make sure it is possible to reconstruct events	2.3.1.4.1.5 o Record secure audit logs and facilitate periodical review to ensure system resources are functioning, confirm reconstruction is possible, and identify unauthorized users or abuse	2.3.1.4.1.6 o Support forensics and incident investigations	2.3.1.4.1.7 o Help focus response and reconstitution efforts to those areas that are most in need	Avoid Single-Point Security Failure	2.3.1.5.2 Avoid Multiple Losses from Single Attack Success	2.3.1.5.3 Separation of Privilege	2.3.1.5.4 Defense in Depth	Allocation of Defenses according to Consequences	2.3.2 Limit, reduce, or manage security-related developmental and operational expenses	2.3.2.1.2 Ease (cost-effective and timely) Certification and Accreditation	2.4 Limit, Reduce, or Manage Security-related Uncertainties	2.4.1 Limit, reduce, or manage security-related unknowns	2.4.2 Limit, reduce, or manage security-related assumptions	2.4.3 Limit, reduce, or manage unpredictability of system behavior	Analyzability	2.4.4 Limit, reduce, or manage consequences or risks not addressed in assurance case	2.4.5 Limit, reduce, or manage consequences or risks related to uncertainty	Risk Sharing	2.4.6 Increase Assurance Regarding Product
3.3.4 Assets 37																								
3.3.5 Security-Violation-related Concepts38		■	■	■																				
3.3.6 Assurance 38																								■
3.4 BASIC SOFTWARE SYSTEM SECURITY PRINCIPLES45																								
3.4.1 Least Privilege46																								
3.4.2 Complete Mediation 46																								
3.4.3 Fail-Safe Defaults.46																								
3.4.4 Least Common Mechanism46																								
3.4.5 Separation of Privilege46											■													
3.4.6 Psychological Acceptability 47																								
3.4.7 Work Factor.47													■											
3.4.8 Economy of Mechanism 47																								
3.4.9 Open Design47																								
3.4.10 Analyzability 47																				■				
3.4.11 Recording of Compromises.47					■	■	■																	

	Retain minimal state	Tolerate Security Violations	2.3.1.3.1 • Limit damage	2.3.1.3.7.1 • Ensure system has a well-defined status after failure, either to a secure failure state or via a recovery procedure to a known secure state [Avizienis 2004]	2.3.1.4.1.4 • Make sure it is possible to reconstruct events	2.3.1.4.1.5 o Record secure audit logs and facilitate periodical review to ensure system resources are functioning, confirm reconstruction is possible, and identify unauthorized users or abuse	2.3.1.4.1.6 o Support forensics and incident investigations	2.3.1.4.1.7 o Help focus response and reconstitution efforts to those areas that are most in need	Avoid Single-Point Security Failure	2.3.1.5.2 Avoid Multiple Losses from Single Attack Success	2.3.1.5.3 Separation of Privilege	2.3.1.5.4 Defense in Depth	Allocation of Defenses according to Consequences	2.3.2 Limit, reduce, or manage security-related developmental and operational expenses	2.3.2.1.2 Ease (cost-effective and timely) Certification and Accreditation	2.4 Limit, Reduce, or Manage Security-related Uncertainties	2.4.1 Limit, reduce, or manage security-related unknowns	2.4.2 Limit, reduce, or manage security-related assumptions	2.4.3 Limit, reduce, or manage unpredictability of system behavior	Analyzability	2.4.4 Limit, reduce, or manage consequences or risks not addressed in assurance case	2.4.5 Limit, reduce, or manage consequences or risks related to uncertainty	Risk Sharing	2.4.6 Increase Assurance Regarding Product
3.4.12 Defense in Depth 47									■			■												
3.4.13 Treat as Conflict 48																								
3.4.14 Tradeoffs49																								
3.5 SAFETY AND SECURITY 49																								
3.5.1 Probability versus Possibility 49													■											
3.6 SECURE SOFTWARE ENGINEERING50																								
3.6.4 Security-Related Architectural Concepts 53																								
3.6.5 Secure Software Development Activities57																								
3.6.6 Security Functionality 60																								
3.6.8 Security Risk Management for Software60																								
3.7 SECURITY PROPERTIES ELABORATED63																								
3.7.1 Confidentiality63																								

	Retain minimal state	Tolerate Security Violations	2.3.1.3.1 • Limit damage	2.3.1.3.7.1 • Ensure system has a well-defined status after failure, either to a secure failure state or via a recovery procedure to a known secure state [Avizienis 2004]	2.3.1.4.1.4 • Make sure it is possible to reconstruct events	2.3.1.4.1.5 o Record secure audit logs and facilitate periodical review to ensure system resources are functioning, confirm reconstruction is possible, and identify unauthorized users or abuse	2.3.1.4.1.6 o Support forensics and incident investigations	2.3.1.4.1.7 o Help focus response and reconstitution efforts to those areas that are most in need	Avoid Single-Point Security Failure	2.3.1.5.2 Avoid Multiple Losses from Single Attack Success	2.3.1.5.3 Separation of Privilege	2.3.1.5.4 Defense in Depth	Allocation of Defenses according to Consequences	2.3.2 Limit, reduce, or manage security-related developmental and operational expenses	2.3.2.1.2 Ease (cost-effective and timely) Certification and Accreditation	2.4 Limit, Reduce, or Manage Security-related Uncertainties	2.4.1 Limit, reduce, or manage security-related unknowns	2.4.2 Limit, reduce, or manage security-related assumptions	2.4.3 Limit, reduce, or manage unpredictability of system behavior	Analyzability	2.4.4 Limit, reduce, or manage consequences or risks not addressed in assurance case	2.4.5 Limit, reduce, or manage consequences or risks related to uncertainty	Risk Sharing	2.4.6 Increase Assurance Regarding Product
3.7.2 Integrity 65																								
3.7.3 Availability65																								
3.7.4 Accountability 66					■	■	■																	
4 ETHICS, LAW, AND GOVERNANCE 71																								
4.2 ETHICS71																								
4.3 LAW.71																								
5 SECURE SOFTWARE REQUIREMENTS 77																								
5.2 REQUIREMENTS FOR A SOLUTION 77																								
5.2.3 Asset Protection Needs.78																								
5.2.4 Threat Analysis80																								
5.2.5 Interface and Environment Requirements82																								
5.2.13 System Accreditation and Auditing Needs 85						■	■								■									
5.3 REQUIREMENTS ANALYSES 86																								
5.3.1 Risk Analysis86													■											

	Retain minimal state	Tolerate Security Violations	2.3.1.3.1 • Limit damage	2.3.1.3.7.1 • Ensure system has a well-defined status after failure, either to a secure failure state or via a recovery procedure to a known secure state [Avizienis 2004]	2.3.1.4.1.4 • Make sure it is possible to reconstruct events	2.3.1.4.1.5 o Record secure audit logs and facilitate periodical review to ensure system resources are functioning, confirm reconstruction is possible, and identify unauthorized users or abuse	2.3.1.4.1.6 o Support forensics and incident investigations	2.3.1.4.1.7 o Help focus response and reconstitution efforts to those areas that are most in need	Avoid Single-Point Security Failure	2.3.1.5.2 Avoid Multiple Losses from Single Attack Success	2.3.1.5.3 Separation of Privilege	2.3.1.5.4 Defense in Depth	Allocation of Defenses according to Consequences	2.3.2 Limit, reduce, or manage security-related developmental and operational expenses	2.3.2.1.2 Ease (cost-effective and timely) Certification and Accreditation	2.4 Limit, Reduce, or Manage Security-related Uncertainties	2.4.1 Limit, reduce, or manage security-related unknowns	2.4.2 Limit, reduce, or manage security-related assumptions	2.4.3 Limit, reduce, or manage unpredictability of system behavior	Analyzability	2.4.4 Limit, reduce, or manage consequences or risks not addressed in assurance case	2.4.5 Limit, reduce, or manage consequences or risks related to uncertainty	Risk Sharing	2.4.6 Increase Assurance Regarding Product
5.3.2 Feasibility Analysis87													■											
5.3.3 Tradeoff Analysis87													■											
5.4 SPECIFICATION 88																								
5.5 REQUIREMENTS VALIDATION 90																								
5.6 ASSURANCE CASE91																								
6 SECURE SOFTWARE DESIGN 95																								
6.3 PRINCIPLES AND GUIDELINES FOR DESIGNING SECURE SOFTWARE.97																								
6.3.2 Damage Confinement and System Resilience 99		■	■	■	■	■	■	■		■														
6.3.3 Vulnerability Reduction100																								
6.4 DOCUMENTATION OF DESIGN ASSUMPTIONS101																								
6.4.1 Environmental Assumptions102																								
6.7 ARCHITECTURES FOR SECURITY 103																								

	Retain minimal state	Tolerate Security Violations	2.3.1.3.1 • Limit damage	2.3.1.3.7.1 • Ensure system has a well-defined status after failure, either to a secure failure state or via a recovery procedure to a known secure state [Avizienis 2004]	2.3.1.4.1.4 • Make sure it is possible to reconstruct events	2.3.1.4.1.5 o Record secure audit logs and facilitate periodical review to ensure system resources are functioning, confirm reconstruction is possible, and identify unauthorized users or abuse	2.3.1.4.1.6 o Support forensics and incident investigations	2.3.1.4.1.7 o Help focus response and reconstitution efforts to those areas that are most in need	Avoid Single-Point Security Failure	2.3.1.5.2 Avoid Multiple Losses from Single Attack Success	2.3.1.5.3 Separation of Privilege	2.3.1.5.4 Defense in Depth	Allocation of Defenses according to Consequences	2.3.2 Limit, reduce, or manage security-related developmental and operational expenses	2.3.2.1.2 Ease (cost-effective and timely) Certification and Accreditation	2.4 Limit, Reduce, or Manage Security-related Uncertainties	2.4.1 Limit, reduce, or manage security-related unknowns	2.4.2 Limit, reduce, or manage security-related assumptions	2.4.3 Limit, reduce, or manage unpredictability of system behavior	Analyzability	2.4.4 Limit, reduce, or manage consequences or risks not addressed in assurance case	2.4.5 Limit, reduce, or manage consequences or risks related to uncertainty	Risk Sharing	2.4.6 Increase Assurance Regarding Product
6.8 SECURITY FUNCTIONALITY104																								
6.8.1 Identity Management 105																								
6.8.2 Access Control Mechanisms 105																								
6.9 PROPER USE OF ENCRYPTION AND ENCRYPTION PROTOCOLS106																								
6.14 METHODS FOR TOLERANCE AND RECOVERY108		■	■	■	■	■	■	■																
6.15 DECEPTION AND DIVERSION108																								
6.16 SOFTWARE PROTECTION109																								
6.17 FORENSIC SUPPORT 110							■																	
6.18 USER INTERFACE DESIGN110																								
6.19 ASSURANCE CASE FOR DESIGN111																								
7 SECURE SOFTWARE CONSTRUCTION115																								
7.4 CONSTRUCTION OF USER AIDS.124																								

	8 SECURE SOFTWARE VERIFICATION, VALIDATION, AND EVALUATION135	8.2 ASSURANCE CASE 135	8.10 THIRD-PARTY VERIFICATION AND VALIDATION AND EVALUATION147	9 SECURE SOFTWARE TOOLS AND METHODS151	10 SECURE SOFTWARE PROCESSES.157
Retain minimal state					
Tolerate Security Violations					
2.3.1.3.1 • Limit damage					
2.3.1.3.7.1 • Ensure system has a well-defined status after failure, either to a secure failure state or via a recovery procedure to a known secure state [Avizienis 2004]					
2.3.1.4.1.4 • Make sure it is possible to reconstruct events					
2.3.1.4.1.5 o Record secure audit logs and facilitate periodical review to ensure system resources are functioning, confirm reconstruction is possible, and identify unauthorized users or abuse					
2.3.1.4.1.6 o Support forensics and incident investigations					
2.3.1.4.1.7 o Help focus response and reconstitution efforts to those areas that are most in need					
Avoid Single-Point Security Failure					
2.3.1.5.2 Avoid Multiple Losses from Single Attack Success					
2.3.1.5.3 Separation of Privilege					
2.3.1.5.4 Defense in Depth					
Allocation of Defenses according to Consequences					
2.3.2 Limit, reduce, or manage security-related developmental and operational expenses					
2.3.2.1.2 Ease (cost-effective and timely) Certification and Accreditation					
2.4 Limit, Reduce, or Manage Security-related Uncertainties					
2.4.1 Limit, reduce, or manage security-related unknowns					
2.4.2 Limit, reduce, or manage security-related assumptions					
2.4.3 Limit, reduce, or manage unpredictability of system behavior					
Analyzability					
2.4.4 Limit, reduce, or manage consequences or risks not addressed in assurance case					
2.4.5 Limit, reduce, or manage consequences or risks related to uncertainty					
Risk Sharing					
2.4.6 Increase Assurance Regarding Product					

	System Assurability	Reduce Danger from other software or systems	2.4.6.2.1 Avoid and workaround environment's security endangering weaknesses	2.4.6.2.2 System does what the specification calls for and nothing else	Reduce Complexity	2.4.6.3.1 Make Small	2.4.6.3.1.1 Minimized Security Elements	2.4.6.3.2 Simplify	2.4.6.3.2.1 Control complexity with multiple perspectives and multiple levels of abstraction	2.4.6.3.2.1.1 Use information hiding and encapsulation	2.4.6.3.2.1.2 Clear Abstractions	2.4.6.3.2.1.3 Partially Ordered Dependencies	2.4.6.3.3 Straightforward Composition	2.4.6.3.3.1 Trustworthy Components	2.4.6.3.3.2 Self-reliant Trustworthiness	2.4.6.3.4 To improve design study previous solutions to similar problems [jabir 1998]	2.4.6.3.4.1 Use known security techniques and solutions	2.4.6.3.4.2 Use standards	Change Slowly	2.4.6.4.1 Use a stable architecture	2.4.6.4.1.1 To eliminate possibilities for violations – particularly of information flow policies	2.4.6.4.1.2 To facilitate achievement of security requirements and evolution	2.4.6.4.1.3 Amendable to supporting assurance arguments and evidence	Assure Security of Product
2 DANGERS AND DAMAGE 15																								
2.3 ATTACKERS18																								
2.4 METHODS FOR ATTACKS20																								
2.5 NON-MALICIOUS DANGERS TO SOFTWARE 23																								
2.6 ATTACKS ACROSS LIFECYCLE24																								
2.7 INFORMATION ABOUT KNOWN VULNERABILITIES AND EXPLOITS29																								
3 FUNDAMENTAL CONCEPTS AND PRINCIPLES 33																								
3.3 BASIC CONCEPTS 35																								
3.3.1 Dependability 35																								
3.3.2 Security 36																								
3.3.3 Software and other Security-related Concerns 37																								

	System Assurability	Reduce Danger from other software or systems	2.4.6.2.1 Avoid and workaround environment's security endangering weaknesses	2.4.6.2.2 System does what the specification calls for and nothing else	Reduce Complexity	2.4.6.3.1 Make Small	2.4.6.3.1.1 Minimized Security Elements	2.4.6.3.2 Simplify	2.4.6.3.2.1 Control complexity with multiple perspectives and multiple levels of abstraction	2.4.6.3.2.1.1 Use information hiding and encapsulation	2.4.6.3.2.1.2 Clear Abstractions	2.4.6.3.2.1.3 Partially Ordered Dependencies	2.4.6.3.3 Straightforward Composition	2.4.6.3.3.1 Trustworthy Components	2.4.6.3.3.2 Self-reliant Trustworthiness	2.4.6.3.4 To improve design study previous solutions to similar problems [jabir 1998]	2.4.6.3.4.1 Use known security techniques and solutions	2.4.6.3.4.2 Use standards	Change Slowly	2.4.6.4.1 Use a stable architecture	2.4.6.4.1.1 To eliminate possibilities for violations – particularly of information flow policies	2.4.6.4.1.2 To facilitate achievement of security requirements and evolution	2.4.6.4.1.3 Amendable to supporting assurance arguments and evidence	Assure Security of Product
3.3.4 Assets 37																								
3.3.5 Security-Violation-related Concepts38																								
3.3.6 Assurance 38	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■
3.4 BASIC SOFTWARE SYSTEM SECURITY PRINCIPLES45																								
3.4.1 Least Privilege46																								
3.4.2 Complete Mediation 46																								
3.4.3 Fail-Safe Defaults.46																								
3.4.4 Least Common Mechanism46																								
3.4.5 Separation of Privilege46		■	■	■																				
3.4.6 Psychological Acceptability 47																								
3.4.7 Work Factor.47																								
3.4.8 Economy of Mechanism 47					■	■	■	■	■	■	■	■	■	■	■	■	■	■						
3.4.9 Open Design47																								
3.4.10 Analyzability 47																								
3.4.11 Recording of Compromises.47																								

	System Assurability	Reduce Danger from other software or systems	2.4.6.2.1 Avoid and workaround environment's security endangering weaknesses	2.4.6.2.2 System does what the specification calls for and nothing else	Reduce Complexity	2.4.6.3.1 Make Small	2.4.6.3.1.1 Minimized Security Elements	2.4.6.3.2 Simplify	2.4.6.3.2.1 Control complexity with multiple perspectives and multiple levels of abstraction	2.4.6.3.2.1.1 Use information hiding and encapsulation	2.4.6.3.2.1.2 Clear Abstractions	2.4.6.3.2.1.3 Partially Ordered Dependencies	2.4.6.3.3 Straightforward Composition	2.4.6.3.3.1 Trustworthy Components	2.4.6.3.3.2 Self-reliant Trustworthiness	2.4.6.3.4 To improve design study previous solutions to similar problems [jabir 1998]	2.4.6.3.4.1 Use known security techniques and solutions	2.4.6.3.4.2 Use standards	Change Slowly	2.4.6.4.1 Use a stable architecture	2.4.6.4.1.1 To eliminate possibilities for violations – particularly of information flow policies	2.4.6.4.1.2 To facilitate achievement of security requirements and evolution	2.4.6.4.1.3 Amendable to supporting assurance arguments and evidence	Assure Security of Product
3.4.12 Defense in Depth 47																								
3.4.13 Treat as Conflict 48																								
3.4.14 Tradeoffs49																								
3.5 SAFETY AND SECURITY 49																								
3.5.1 Probability versus Possibility 49																								
3.6 SECURE SOFTWARE ENGINEERING50																								
3.6.4 Security-Related Architectural Concepts 53																								
3.6.5 Secure Software Development Activities57																								
3.6.6 Security Functionality 60																								
3.6.8 Security Risk Management for Software60																								
3.7 SECURITY PROPERTIES ELABORATED63																								
3.7.1 Confidentiality63																								

	System Assurability	Reduce Danger from other software or systems	2.4.6.2.1 Avoid and workaround environment's security endangering weaknesses	2.4.6.2.2 System does what the specification calls for and nothing else	Reduce Complexity	2.4.6.3.1 Make Small	2.4.6.3.1.1 Minimized Security Elements	2.4.6.3.2 Simplify	2.4.6.3.2.1 Control complexity with multiple perspectives and multiple levels of abstraction	2.4.6.3.2.1.1 Use information hiding and encapsulation	2.4.6.3.2.1.2 Clear Abstractions	2.4.6.3.2.1.3 Partially Ordered Dependencies	2.4.6.3.3 Straightforward Composition	2.4.6.3.3.1 Trustworthy Components	2.4.6.3.3.2 Self-reliant Trustworthiness	2.4.6.3.4 To improve design study previous solutions to similar problems [jabir 1998]	2.4.6.3.4.1 Use known security techniques and solutions	2.4.6.3.4.2 Use standards	Change Slowly	2.4.6.4.1 Use a stable architecture	2.4.6.4.1.1 To eliminate possibilities for violations – particularly of information flow policies	2.4.6.4.1.2 To facilitate achievement of security requirements and evolution	2.4.6.4.1.3 Amendable to supporting assurance arguments and evidence	Assure Security of Product
3.7.2 Integrity 65																								
3.7.3 Availability65																								
3.7.4 Accountability 66																								
4 ETHICS, LAW, AND GOVERNANCE 71																								
4.2 ETHICS71																								
4.3 LAW.71																								
5 SECURE SOFTWARE REQUIREMENTS 77																								
5.2 REQUIREMENTS FOR A SOLUTION 77																								
5.2.3 Asset Protection Needs.78																								
5.2.4 Threat Analysis80																								
5.2.5 Interface and Environment Requirements82																								
5.2.13 System Accreditation and Auditing Needs 85																								
5.3 REQUIREMENTS ANALYSES 86																								
5.3.1 Risk Analysis86																								

	System Assurability	Reduce Danger from other software or systems	2.4.6.2.1 Avoid and workaround environment's security endangering weaknesses	2.4.6.2.2 System does what the specification calls for and nothing else	Reduce Complexity	2.4.6.3.1 Make Small	2.4.6.3.1.1 Minimized Security Elements	2.4.6.3.2 Simplify	2.4.6.3.2.1 Control complexity with multiple perspectives and multiple levels of abstraction	2.4.6.3.2.1.1 Use information hiding and encapsulation	2.4.6.3.2.1.2 Clear Abstractions	2.4.6.3.2.1.3 Partially Ordered Dependencies	2.4.6.3.3 Straightforward Composition	2.4.6.3.3.1 Trustworthy Components	2.4.6.3.3.2 Self-reliant Trustworthiness	2.4.6.3.4 To improve design study previous solutions to similar problems [jabir 1998]	2.4.6.3.4.1 Use known security techniques and solutions	2.4.6.3.4.2 Use standards	Change Slowly	2.4.6.4.1 Use a stable architecture	2.4.6.4.1.1 To eliminate possibilities for violations – particularly of information flow policies	2.4.6.4.1.2 To facilitate achievement of security requirements and evolution	2.4.6.4.1.3 Amendable to supporting assurance arguments and evidence	Assure Security of Product
5.3.2 Feasibility Analysis87																								
5.3.3 Tradeoff Analysis87																								
5.4 SPECIFICATION 88																								
5.5 REQUIREMENTS VALIDATION 90																								
5.6 ASSURANCE CASE91																								
6 SECURE SOFTWARE DESIGN 95																								
6.3 PRINCIPLES AND GUIDELINES FOR DESIGNING SECURE SOFTWARE.97																								
6.3.2 Damage Confinement and System Resilience 99																								
6.3.3 Vulnerability Reduction100																								
6.4 DOCUMENTATION OF DESIGN ASSUMPTIONS101																								
6.4.1 Environmental Assumptions102																								
6.7 ARCHITECTURES FOR SECURITY 103																				■	■	■	■	

	System Assurability	Reduce Danger from other software or systems	2.4.6.2.1 Avoid and workaround environment's security endangering weaknesses	2.4.6.2.2 System does what the specification calls for and nothing else	Reduce Complexity	2.4.6.3.1 Make Small	2.4.6.3.1.1 Minimized Security Elements	2.4.6.3.2 Simplify	2.4.6.3.2.1 Control complexity with multiple perspectives and multiple levels of abstraction	2.4.6.3.2.1.1 Use information hiding and encapsulation	2.4.6.3.2.1.2 Clear Abstractions	2.4.6.3.2.1.3 Partially Ordered Dependencies	2.4.6.3.3 Straightforward Composition	2.4.6.3.3.1 Trustworthy Components	2.4.6.3.3.2 Self-reliant Trustworthiness	2.4.6.3.4 To improve design study previous solutions to similar problems [jabir 1998]	2.4.6.3.4.1 Use known security techniques and solutions	2.4.6.3.4.2 Use standards	Change Slowly	2.4.6.4.1 Use a stable architecture	2.4.6.4.1.1 To eliminate possibilities for violations – particularly of information flow policies	2.4.6.4.1.2 To facilitate achievement of security requirements and evolution	2.4.6.4.1.3 Amendable to supporting assurance arguments and evidence	Assure Security of Product
6.8 SECURITY FUNCTIONALITY104																								
6.8.1 Identity Management 105																								
6.8.2 Access Control Mechanisms 105																								
6.9 PROPER USE OF ENCRYPTION AND ENCRYPTION PROTOCOLS106																								
6.14 METHODS FOR TOLERANCE AND RECOVERY108																								
6.15 DECEPTION AND DIVERSION108																								
6.16 SOFTWARE PROTECTION109																								
6.17 FORENSIC SUPPORT 110																								
6.18 USER INTERFACE DESIGN110																								
6.19 ASSURANCE CASE FOR DESIGN111																								
7 SECURE SOFTWARE CONSTRUCTION115																								
7.4 CONSTRUCTION OF USER AIDS.124																								

	System Assurability	Reduce Danger from other software or systems	2.4.6.2.1 Avoid and workaround environment's security endangering weaknesses	2.4.6.2.2 System does what the specification calls for and nothing else	Reduce Complexity	2.4.6.3.1 Make Small	2.4.6.3.1.1 Minimized Security Elements	2.4.6.3.2 Simplify	2.4.6.3.2.1 Control complexity with multiple perspectives and multiple levels of abstraction	2.4.6.3.2.1.1 Use information hiding and encapsulation	2.4.6.3.2.1.2 Clear Abstractions	2.4.6.3.2.1.3 Partially Ordered Dependencies	2.4.6.3.3 Straightforward Composition	2.4.6.3.3.1 Trustworthy Components	2.4.6.3.3.2 Self-reliant Trustworthiness	2.4.6.3.4 To improve design study previous solutions to similar problems [jabir 1998]	2.4.6.3.4.1 Use known security techniques and solutions	2.4.6.3.4.2 Use standards	Change Slowly	2.4.6.4.1 Use a stable architecture	2.4.6.4.1.1 To eliminate possibilities for violations – particularly of information flow policies	2.4.6.4.1.2 To facilitate achievement of security requirements and evolution	2.4.6.4.1.3 Amendable to supporting assurance arguments and evidence	Assure Security of Product
8 SECURE SOFTWARE VERIFICATION, VALIDATION, AND EVALUATION135																								
8.2 ASSURANCE CASE 135																								■
8.10 THIRD-PARTY VERIFICATION AND VALIDATION AND EVALUATION147																								
9 SECURE SOFTWARE TOOLS AND METHODS151																								
10 SECURE SOFTWARE PROCESSES.157																								

	2.4.6.5.1 Create and Maintain an Assurance Case	2.4.6.5.2 Ensure security preserving composition at all levels of detail	2.4.6.5.3 Secure Distributed Composition	2.4.6.5.4 Ease production of an accompanying assurance case for the security preserving correctness of compositions	2.4.6.5.5 Design to ease traceability, verification, validation, and evaluation	2.4.6.5.6 Analyzability	2.4.6.5.7 Chain of Trust	62.4..6 Use Production Process and Means that Ease and Increase Assurance	2.4.6.6.1 Ease creation and maintenance of an assurance case	2.4.6.6.2 Use Repeatable, Documented Procedures	2.4.6.6.3 Procedural Rigor	2.4.6.6.4 Engineering Rigor	2.4.6.6.5 Open Design	2.4.6.6.5.1 Review for use of design principles (and guidelines	2.4.6.6.6 Chose notations and tools that facilitate achieving security and its assurance	2.4.6.6.7 Have expertise in technologies being used and application domain	2.4.6.6.8 Avoid Known Pitfalls	2.4.6.6.8.1 Avoid common errors and vulnerabilities	2.4.6.6.8.2 Avoid and workaround tools' security endangering weaknesses	2.4.6.6.8.3 Avoid non-malicious pitfalls	Continuous Risk Management	2.4.6.7.1 Consider security or assurance risks together with other risks	3 The Environment	3.1 Nature of Environment
2 DANGERS AND DAMAGE 15																								
2.3 ATTACKERS18																								
2.4 METHODS FOR ATTACKS20																								
2.5 NON-MALICIOUS DANGERS TO SOFTWARE 23																								
2.6 ATTACKS ACROSS LIFECYCLE24																								
2.7 INFORMATION ABOUT KNOWN VULNERABILITIES AND EXPLOITS29																								
3 FUNDAMENTAL CONCEPTS AND PRINCIPLES 33																								
3.3 BASIC CONCEPTS 35																								
3.3.1 Dependability 35																								
3.3.2 Security 36																								
3.3.3 Software and other Security-related Concerns 37																								

	2.4.6.5.1 Create and Maintain an Assurance Case	2.4.6.5.2 Ensure security preserving composition at all levels of detail	2.4.6.5.3 Secure Distributed Composition	2.4.6.5.4 Ease production of an accompanying assurance case for the security preserving correctness of compositions	2.4.6.5.5 Design to ease traceability, verification, validation, and evaluation	2.4.6.5.6 Analyzability	2.4.6.5.7 Chain of Trust	62.4..6 Use Production Process and Means that Ease and Increase Assurance	2.4.6.6.1 Ease creation and maintenance of an assurance case	2.4.6.6.2 Use Repeatable, Documented Procedures	2.4.6.6.3 Procedural Rigor	2.4.6.6.4 Engineering Rigor	2.4.6.6.5 Open Design	2.4.6.6.5.1 Review for use of design principles (and guidelines	2.4.6.6.6 Chose notations and tools that facilitate achieving security and its assurance	2.4.6.6.7 Have expertise in technologies being used and application domain	2.4.6.6.8 Avoid Known Pitfalls	2.4.6.6.8.1 Avoid common errors and vulnerabilities	2.4.6.6.8.2 Avoid and workaround tools' security endangering weaknesses	2.4.6.6.8.3 Avoid non-malicious pitfalls	Continuous Risk Management	2.4.6.7.1 Consider security or assurance risks together with other risks	3 The Environment	3.1 Nature of Environment
3.3.4 Assets 37																								
3.3.5 Security-Violation-related Concepts38																								
3.3.6 Assurance 38	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■		
3.4 BASIC SOFTWARE SYSTEM SECURITY PRINCIPLES45																								
3.4.1 Least Privilege46																								
3.4.2 Complete Mediation 46																								
3.4.3 Fail-Safe Defaults.46																								
3.4.4 Least Common Mechanism46																								
3.4.5 Separation of Privilege46																								
3.4.6 Psychological Acceptability 47																								
3.4.7 Work Factor.47																								
3.4.8 Economy of Mechanism 47																								
3.4.9 Open Design47								■	■	■	■	■	■	■	■	■	■	■	■	■				
3.4.10 Analyzability 47						■																		
3.4.11 Recording of Compromises.47																								

	2.4.6.5.1 Create and Maintain an Assurance Case	2.4.6.5.2 Ensure security preserving composition at all levels of detail	2.4.6.5.3 Secure Distributed Composition	2.4.6.5.4 Ease production of an accompanying assurance case for the security preserving correctness of compositions	2.4.6.5.5 Design to ease traceability, verification, validation, and evaluation	2.4.6.5.6 Analyzability	2.4.6.5.7 Chain of Trust	62.4..6 Use Production Process and Means that Ease and Increase Assurance	2.4.6.6.1 Ease creation and maintenance of an assurance case	2.4.6.6.2 Use Repeatable, Documented Procedures	2.4.6.6.3 Procedural Rigor	2.4.6.6.4 Engineering Rigor	2.4.6.6.5 Open Design	2.4.6.6.5.1 Review for use of design principles (and guidelines	2.4.6.6.6 Chose notations and tools that facilitate achieving security and its assurance	2.4.6.6.7 Have expertise in technologies being used and application domain	2.4.6.6.8 Avoid Known Pitfalls	2.4.6.6.8.1 Avoid common errors and vulnerabilities	2.4.6.6.8.2 Avoid and workaround tools' security endangering weaknesses	2.4.6.6.8.3 Avoid non-malicious pitfalls	Continuous Risk Management	2.4.6.7.1 Consider security or assurance risks together with other risks	3 The Environment	3.1 Nature of Environment
3.4.12 Defense in Depth 47																								
3.4.13 Treat as Conflict 48																								
3.4.14 Tradeoffs49																								
3.5 SAFETY AND SECURITY 49																								
3.5.1 Probability versus Possibility 49																								
3.6 SECURE SOFTWARE ENGINEERING50																								
3.6.4 Security-Related Architectural Concepts 53																								
3.6.5 Secure Software Development Activities57								■	■	■	■	■	■	■	■	■	■	■	■	■				
3.6.6 Security Functionality 60																								
3.6.8 Security Risk Management for Software60								■	■	■	■	■	■	■	■	■	■	■	■	■	■	■		
3.7 SECURITY PROPERTIES ELABORATED63																								
3.7.1 Confidentiality63																								
3.7.2 Integrity 65																								

	2.4.6.5.1 Create and Maintain an Assurance Case	2.4.6.5.2 Ensure security preserving composition at all levels of detail	2.4.6.5.3 Secure Distributed Composition	2.4.6.5.4 Ease production of an accompanying assurance case for the security preserving correctness of compositions	2.4.6.5.5 Design to ease traceability, verification, validation, and evaluation	2.4.6.5.6 Analyzability	2.4.6.5.7 Chain of Trust	62.4..6 Use Production Process and Means that Ease and Increase Assurance	2.4.6.6.1 Ease creation and maintenance of an assurance case	2.4.6.6.2 Use Repeatable, Documented Procedures	2.4.6.6.3 Procedural Rigor	2.4.6.6.4 Engineering Rigor	2.4.6.6.5 Open Design	2.4.6.6.5.1 Review for use of design principles (and guidelines	2.4.6.6.6 Chose notations and tools that facilitate achieving security and its assurance	2.4.6.6.7 Have expertise in technologies being used and application domain	2.4.6.6.8 Avoid Known Pitfalls	2.4.6.6.8.1 Avoid common errors and vulnerabilities	2.4.6.6.8.2 Avoid and workaround tools' security endangering weaknesses	2.4.6.6.8.3 Avoid non-malicious pitfalls	Continuous Risk Management	2.4.6.7.1 Consider security or assurance risks together with other risks	3 The Environment	3.1 Nature of Environment
3.7.3 Availability65																								
3.7.4 Accountability 66																								
4 ETHICS, LAW, AND GOVERNANCE 71																								
4.2 ETHICS71																								
4.3 LAW.71																								
5 SECURE SOFTWARE REQUIREMENTS 77																								
5.2 REQUIREMENTS FOR A SOLUTION 77																								
5.2.3 Asset Protection Needs.78																								
5.2.4 Threat Analysis80																							■	■
5.2.5 Interface and Environment Requirements82																							■	■
5.2.13 System Accreditation and Auditing Needs 85																								
5.3 REQUIREMENTS ANALYSES 86																								
5.3.1 Risk Analysis86																								
5.3.2 Feasibility Analysis87																								

	2.4.6.5.1 Create and Maintain an Assurance Case	2.4.6.5.2 Ensure security preserving composition at all levels of detail	2.4.6.5.3 Secure Distributed Composition	2.4.6.5.4 Ease production of an accompanying assurance case for the security preserving correctness of compositions	2.4.6.5.5 Design to ease traceability, verification, validation, and evaluation	2.4.6.5.6 Analyzability	2.4.6.5.7 Chain of Trust	62.4..6 Use Production Process and Means that Ease and Increase Assurance	2.4.6.6.1 Ease creation and maintenance of an assurance case	2.4.6.6.2 Use Repeatable, Documented Procedures	2.4.6.6.3 Procedural Rigor	2.4.6.6.4 Engineering Rigor	2.4.6.6.5 Open Design	2.4.6.6.5.1 Review for use of design principles (and guidelines	2.4.6.6.6 Chose notations and tools that facilitate achieving security and its assurance	2.4.6.6.7 Have expertise in technologies being used and application domain	2.4.6.6.8 Avoid Known Pitfalls	2.4.6.6.8.1 Avoid common errors and vulnerabilities	2.4.6.6.8.2 Avoid and workaround tools' security endangering weaknesses	2.4.6.6.8.3 Avoid non-malicious pitfalls	Continuous Risk Management	2.4.6.7.1 Consider security or assurance risks together with other risks	3 The Environment	3.1 Nature of Environment
5.3.3 Tradeoff Analysis87																								
5.4 SPECIFICATION 88																								
5.5 REQUIREMENTS VALIDATION 90																								
5.6 ASSURANCE CASE91																								
6 SECURE SOFTWARE DESIGN 95																								
6.3 PRINCIPLES AND GUIDELINES FOR DESIGNING SECURE SOFTWARE.97																								
6.3.2 Damage Confinement and System Resilience 99																								
6.3.3 Vulnerability Reduction100																								
6.4 DOCUMENTATION OF DESIGN ASSUMPTIONS101																								
6.4.1 Environmental Assumptions102																								■
6.7 ARCHITECTURES FOR SECURITY 103																								
6.8 SECURITY FUNCTIONALITY104																								

	2.4.6.5.1 Create and Maintain an Assurance Case	2.4.6.5.2 Ensure security preserving composition at all levels of detail	2.4.6.5.3 Secure Distributed Composition	2.4.6.5.4 Ease production of an accompanying assurance case for the security preserving correctness of compositions	2.4.6.5.5 Design to ease traceability, verification, validation, and evaluation	2.4.6.5.6 Analyzability	2.4.6.5.7 Chain of Trust	62.4..6 Use Production Process and Means that Ease and Increase Assurance	2.4.6.6.1 Ease creation and maintenance of an assurance case	2.4.6.6.2 Use Repeatable, Documented Procedures	2.4.6.6.3 Procedural Rigor	2.4.6.6.4 Engineering Rigor	2.4.6.6.5 Open Design	2.4.6.6.5.1 Review for use of design principles (and guidelines	2.4.6.6.6 Chose notations and tools that facilitate achieving security and its assurance	2.4.6.6.7 Have expertise in technologies being used and application domain	2.4.6.6.8 Avoid Known Pitfalls	2.4.6.6.8.1 Avoid common errors and vulnerabilities	2.4.6.6.8.2 Avoid and workaround tools' security endangering weaknesses	2.4.6.6.8.3 Avoid non-malicious pitfalls	Continuous Risk Management	2.4.6.7.1 Consider security or assurance risks together with other risks	3 The Environment	3.1 Nature of Environment
6.8.1 Identity Management 105																								
6.8.2 Access Control Mechanisms 105																								
6.9 PROPER USE OF ENCRYPTION AND ENCRYPTION PROTOCOLS106																								
6.14 METHODS FOR TOLERANCE AND RECOVERY108																								
6.15 DECEPTION AND DIVERSION108																								
6.16 SOFTWARE PROTECTION109																								
6.17 FORENSIC SUPPORT 110																								
6.18 USER INTERFACE DESIGN110																								
6.19 ASSURANCE CASE FOR DESIGN111																								
7 SECURE SOFTWARE CONSTRUCTION115																								
7.4 CONSTRUCTION OF USER AIDS.124																								

	2.4.6.5.1 Create and Maintain an Assurance Case	2.4.6.5.2 Ensure security preserving composition at all levels of detail	2.4.6.5.3 Secure Distributed Composition	2.4.6.5.4 Ease production of an accompanying assurance case for the security preserving correctness of compositions	2.4.6.5.5 Design to ease traceability, verification, validation, and evaluation	2.4.6.5.6 Analyzability	2.4.6.5.7 Chain of Trust	62.4..6 Use Production Process and Means that Ease and Increase Assurance	2.4.6.6.1 Ease creation and maintenance of an assurance case	2.4.6.6.2 Use Repeatable, Documented Procedures	2.4.6.6.3 Procedural Rigor	2.4.6.6.4 Engineering Rigor	2.4.6.6.5 Open Design	2.4.6.6.5.1 Review for use of design principles (and guidelines	2.4.6.6.6 Chose notations and tools that facilitate achieving security and its assurance	2.4.6.6.7 Have expertise in technologies being used and application domain	2.4.6.6.8 Avoid Known Pitfalls	2.4.6.6.8.1 Avoid common errors and vulnerabilities	2.4.6.6.8.2 Avoid and workaround tools' security endangering weaknesses	2.4.6.6.8.3 Avoid non-malicious pitfalls	Continuous Risk Management	2.4.6.7.1 Consider security or assurance risks together with other risks	3 The Environment	3.1 Nature of Environment
8 SECURE SOFTWARE VERIFICATION, VALIDATION, AND EVALUATION135																								
8.2 ASSURANCE CASE 135																								
8.10 THIRD-PARTY VERIFICATION AND VALIDATION AND EVALUATION147																								
9 SECURE SOFTWARE TOOLS AND METHODS151																								
10 SECURE SOFTWARE PROCESSES.157								■	■	■	■	■	■	■	■	■	■	■	■	■				

	3.1.1 Security is a system, organizational, and societal problem	3.2 Benefits to Environment	3.2.1 Do not cause security problems for systems in the environment	3.2.2 Learn, Adapt, and Improve Organizational Policy	3.3 Limit, Reduce, or Manage Environment-Related Losses	3.3.1 Avoid assumptions about environment	Make only weak non-critical assumptions about environment	3.3.2 Trust only services or components in environment known to be trustworthy	3.3.3 More trustworthy components do not depend on less trustworthy services or entities in environment	Do not invoke untrusted services from within system.	3.3.4 Avoid dependence on protection by environment	3.4 Avoid Environment-Related Uncertainties	3.4.1 Do not rely only on obfuscation or hiding for protection from entities in environment	3.4.2 Need adequate assurance for dependences
2 DANGERS AND DAMAGE 15														
2.3 ATTACKERS18														
2.4 METHODS FOR ATTACKS20														
2.5 NON-MALICIOUS DANGERS TO SOFTWARE 23														
2.6 ATTACKS ACROSS LIFECYCLE24														
2.7 INFORMATION ABOUT KNOWN VULNERABILITIES AND EXPLOITS29														
3 FUNDAMENTAL CONCEPTS AND PRINCIPLES 33														
3.3 BASIC CONCEPTS 35														
3.3.1 Dependability 35														
3.3.2 Security 36														
3.3.3 Software and other Security-related Concerns 37														
3.3.4 Assets 37														
3.3.5 Security-Violation-related Concepts38														
3.3.6 Assurance 38														

	3.1.1 Security is a system, organizational, and societal problem	3.2 Benefits to Environment	3.2.1 Do not cause security problems for systems in the environment	3.2.2 Learn, Adapt, and Improve Organizational Policy	3.3 Limit, Reduce, or Manage Environment-Related Losses	3.3.1 Avoid assumptions about environment	Make only weak non-critical assumptions about environment	3.3.2 Trust only services or components in environment known to be trustworthy	3.3.3 More trustworthy components do not depend on less trustworthy services or entities in environment	Do not invoke untrusted services from within system.	3.3.4 Avoid dependence on protection by environment	3.4 Avoid Environment-Related Uncertainties	3.4.1 Do not rely only on obfuscation or hiding for protection from entities in environment	3.4.2 Need adequate assurance for dependences
3.4 BASIC SOFTWARE SYSTEM SECURITY PRINCIPLES45														
3.4.1 Least Privilege46														
3.4.2 Complete Mediation 46														
3.4.3 Fail-Safe Defaults.46														
3.4.4 Least Common Mechanism46														
3.4.5 Separation of Privilege46														
3.4.6 Psychological Acceptability 47														
3.4.7 Work Factor.47														
3.4.8 Economy of Mechanism 47														
3.4.9 Open Design47														
3.4.10 Analyzability 47														
3.4.11 Recording of Compromises.47														
3.4.12 Defense in Depth 47														
3.4.13 Treat as Conflict 48														
3.4.14 Tradeoffs49														
3.5 SAFETY AND SECURITY 49														
3.5.1 Probability versus Possibility 49														
3.6 SECURE SOFTWARE ENGINEERING50														

	3.1.1 Security is a system, organizational, and societal problem	3.2 Benefits to Environment	3.2.1 Do not cause security problems for systems in the environment	3.2.2 Learn, Adapt, and Improve Organizational Policy	3.3 Limit, Reduce, or Manage Environment-Related Losses	3.3.1 Avoid assumptions about environment	Make only weak non-critical assumptions about environment	3.3.2 Trust only services or components in environment known to be trustworthy	3.3.3 More trustworthy components do not depend on less trustworthy services or entities in environment	Do not invoke untrusted services from within system.	3.3.4 Avoid dependence on protection by environment	3.4 Avoid Environment-Related Uncertainties	3.4.1 Do not rely only on obfuscation or hiding for protection from entities in environment	3.4.2 Need adequate assurance for dependences
3.6.4 Security-Related Architectural Concepts 53														
3.6.5 Secure Software Development Activities57														
3.6.6 Security Functionality 60														
3.6.8 Security Risk Management for Software60														
3.7 SECURITY PROPERTIES ELABORATED63														
3.7.1 Confidentiality63														
3.7.2 Integrity 65														
3.7.3 Availability65														
3.7.4 Accountability 66														
4 ETHICS, LAW, AND GOVERNANCE 71														
4.2 ETHICS71														
4.3 LAW.71														
5 SECURE SOFTWARE REQUIREMENTS 77														
5.2 REQUIREMENTS FOR A SOLUTION 77														

	3.1.1 Security is a system, organizational, and societal problem	3.2 Benefits to Environment	3.2.1 Do not cause security problems for systems in the environment	3.2.2 Learn, Adapt, and Improve Organizational Policy	3.3 Limit, Reduce, or Manage Environment-Related Losses	3.3.1 Avoid assumptions about environment	Make only weak non-critical assumptions about environment	3.3.2 Trust only services or components in environment known to be trustworthy	3.3.3 More trustworthy components do not depend on less trustworthy services or entities in environment	Do not invoke untrusted services from within system.	3.3.4 Avoid dependence on protection by environment	3.4 Avoid Environment-Related Uncertainties	3.4.1 Do not rely only on obfuscation or hiding for protection from entities in environment	3.4.2 Need adequate assurance for dependences
5.2.3 Asset Protection Needs.78														
5.2.4 Threat Analysis80	■	■	■	■	■	■	■	■	■	■	■	■	■	■
5.2.5 Interface and Environment Requirements82	■	■	■	■	■	■	■	■	■	■	■	■	■	■
5.2.13 System Accreditation and Auditing Needs 85														
5.3 REQUIREMENTS ANALYSES 86														
5.3.1 Risk Analysis86														
5.3.2 Feasibility Analysis87														
5.3.3 Tradeoff Analysis87														
5.4 SPECIFICATION 88														
5.5 REQUIREMENTS VALIDATION 90														
5.6 ASSURANCE CASE91														
6 SECURE SOFTWARE DESIGN 95														
6.3 PRINCIPLES AND GUIDELINES FOR DESIGNING SECURE SOFTWARE.97														
6.3.2 Damage Confinement and System Resilience 99			■		■	■	■	■	■	■	■			
6.3.3 Vulnerability Reduction100														

	3.1.1 Security is a system, organizational, and societal problem	3.2 Benefits to Environment	3.2.1 Do not cause security problems for systems in the environment	3.2.2 Learn, Adapt, and Improve Organizational Policy	3.3 Limit, Reduce, or Manage Environment-Related Losses	3.3.1 Avoid assumptions about environment	Make only weak non-critical assumptions about environment	3.3.2 Trust only services or components in environment known to be trustworthy	3.3.3 More trustworthy components do not depend on less trustworthy services or entities in environment	Do not invoke untrusted services from within system.	3.3.4 Avoid dependence on protection by environment	3.4 Avoid Environment-Related Uncertainties	3.4.1 Do not rely only on obfuscation or hiding for protection from entities in environment	3.4.2 Need adequate assurance for dependences
6.4 DOCUMENTATION OF DESIGN ASSUMPTIONS101														
6.4.1 Environmental Assumptions102	■					■	■							
6.7 ARCHITECTURES FOR SECURITY 103														
6.8 SECURITY FUNCTIONALITY104														
6.8.1 Identity Management 105														
6.8.2 Access Control Mechanisms 105														
6.9 PROPER USE OF ENCRYPTION AND ENCRYPTION PROTOCOLS106														
6.14 METHODS FOR TOLERANCE AND RECOVERY108					■	■	■	■	■	■	■			
6.15 DECEPTION AND DIVERSION108														
6.16 SOFTWARE PROTECTION109														
6.17 FORENSIC SUPPORT 110														
6.18 USER INTERFACE DESIGN110														
6.19 ASSURANCE CASE FOR DESIGN111														
7 SECURE SOFTWARE CONSTRUCTION115														
7.4 CONSTRUCTION OF USER AIDS.124														

	3.1.1 Security is a system, organizational, and societal problem	3.2 Benefits to Environment	3.2.1 Do not cause security problems for systems in the environment	3.2.2 Learn, Adapt, and Improve Organizational Policy	3.3 Limit, Reduce, or Manage Environment-Related Losses	3.3.1 Avoid assumptions about environment	Make only weak non-critical assumptions about environment	3.3.2 Trust only services or components in environment known to be trustworthy	3.3.3 More trustworthy components do not depend on less trustworthy services or entities in environment	Do not invoke untrusted services from within system.	3.3.4 Avoid dependence on protection by environment	3.4 Avoid Environment-Related Uncertainties	3.4.1 Do not rely only on obfuscation or hiding for protection from entities in environment	3.4.2 Need adequate assurance for dependences
8 SECURE SOFTWARE VERIFICATION, VALIDATION, AND EVALUATION135														
8.2 ASSURANCE CASE 135														
8.10 THIRD-PARTY VERIFICATION AND VALIDATION AND EVALUATION147														
9 SECURE SOFTWARE TOOLS AND METHODS151														
10 SECURE SOFTWARE PROCESSES.157														

16Bibliography

[Abbott 1976] Abbott, Robert P., Chin, Janet S., Donnelley, James E., Konigsford, William L., Tukubo, Shigeru, and Webb, Douglas A., "Security analysis and enhancements of computer operating systems," *NBSIR 76-1041*, The RISOS Project, Lawrence Livermore Laboratory, Livermore, CA, USA. Published by the Institute for Computer Sciences and Technology, National Bureau of Standards, Washington, DC, USA. Apr. 1976.

[Abrams 1998] Abrams, M. D, “Security Engineering in an Evolutionary Acquisition Environment,” *New Security Paradigms Workshop*, 1998.

[Abran 2004] Abran, Alain, James W. Moore (Executive editors); Pierre Bourque, Robert Dupuis, Leonard Tripp (Editors). *Guide to the Software Engineering Body of Knowledge*, 2004 Version. IEEE Computer Society, 2004.

[Abrial 1996] Abrial, J-R. *The B-Book: Assigning programs to meanings*. Cambridge Press, 1996.

[ACM] ACM Transactions on Information and System Security, Association for Computing Machinery.

[Alexander 1995] Alexander, Perry, “Best of Both Worlds: Combining Formal and Semi-formal Methods in Software Engineering,” *IEEE Potentials*, December/January, 1995.

[Alexander 2001] Alexander, Ian. *Systems Engineering Isn't Just Software*. 2001. Available at http://easyweb.easynet.co.uk/~iany/consultancy/systems_engineering/se_isnt_just_sw.htm.

[Alexander 2005] Alexander, Steven. "Why Teenagers Hack: A Personal Memoir", in *Login*, Vol. 10, No. 1, pp. 14-16, February 2005. Available at http://www.usenix.org/publications/login/2005-02/pdfs/teenagers.pdf.

[Anderson 2001] Anderson, Ross J., Security Engineering: A Guide to Building Dependable Distributed Systems. John Wiley and Sons, 2001.

[Anderson 2004] Anderson, E. A., C. E. Irvine, and R. R. Schell. “Subversion as a threat in information warfare.” *Journal of Information Warfare*, 3:51 -- 64, 2004.

[Andrews 2004] Andrews, Mike, and James A. Whittaker, “Computer Security,” *IEEE Security and Privacy*, pp. 68-71, September/October 2004.

[Andrews et al 1990] Andrews, Peter B., Sunil Issar, Dan Nesmith & Frank Pfenning, *The TPS Theorem Proving System*, 10th International Conference on Automated Deduction, edited by Mark E. Stickel, Lecture Notes in Artificial Intelligence 449, Springer-Verlag, 1990, 641-642.

[ANSI/EIA 748-1998] ANSI/EIA 748-1998. *Earned Value Management System Guidelines*. EIA, 1998.

[ANSI/PMI 99-001-2004] No Author.. *A Guide to the Project Management Body of Knowledge*. Third Edition. Newton Square, PA.: Project Management Institute, Inc. 2004.

[Apvrille 2005] Apvrille, Axelle, and Makan Pourzandi, “Secure Software Development by Example,” *IEEE Security and Privacy*, p. 10-17, July/August 200.5

[Arthon 1990] Arthan, R. D. A formal specification of HOL. Technical Report DS/FMU/IED/SPC001, ICL Defence Systems, April 1990.

[Ashton 2001] Ashton, Gerry. “Cleaning up your Security Act for Inspection”, *Computer Weekly*, Jan 18, 2001.

[Aslam 1995] Aslam, Taimur, “A taxonomy of security faults in the UNIX operating system,” Master's thesis, Purdue University, Aug. 1995.

[Aslam 1996] Aslam, Taimur, Krsul, Ivan, and Spafford, Eugene, “Use of a taxonomy of security faults,” In Proc. 19th NIST-NCSC National Information Systems Security Conference, pages 551-560, 1996.

[Atallah, Bryant and Sytz 2004] Atallah, Mikhail, Bryant, Eric, and Styz, Martin, “A Survey of Anti-Tamper Technologies,” *Crosstalk – The Journal of Defense Software Engineering*, Nov 2004.

[Avizienis 2004] Avizienis, Algirdas, Jean-Claude Laprie, Brian Randell, and Carl Landwehr, “Basic Concepts and Taxonomy of Dependable and Secure Computing,” *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11-33, Jan.-Mar. 2004. Available at http://csdl.computer.org/dl/trans/tq/2004/01/q0011.pdf

[Babich 1986] Babich, W., *Software Configuration Management*, Addison-Wesley, 1986.

[Backes 2002] Backes, M. and B. Pfitzmann. Computational probabilistic non-interference. In Proc. 7th European Symposium on Research in Computer Security (ESORICS), volume 2502 of Lecture Notes in Computer Science, pp. 1–23. Springer, 2002.

[Bahill 1998] Bahill, A.T. and B. Gissing, "Re-evaluating Systems Engineering Concepts Using Systems Thinking". IEEE Transaction on Systems, Man and Cybernetics, Part C: Applications and Reviews, Vol. 28 No. 4 pp. 516-527, November 1998.

[Barden 1995] Barden, Rosalind, Susan Stepney, and David Cooper, *Z in Practice*, Prentice Hall, 1995.

[Barnes 2003] Barnes, John. High Integrity Software: The SPARK Approach to Safety and Security, Addison Wesley, 2003.

[Barnum 2005] Sean Barnum and Gary McGraw. “Knowledge for Software Security,” *IEEE Security and Privacy*, vol.03, no.2, pp. 74-78, March/April 2005.

[Baskerville 2003] Baskerville, R., and Portougal, V. “A Possibility Theory Framework for Security Evaluation in National Infrastructure Protection.” *Journal of Database Management*, 14(2), 1-13, 2003.
[Baskerville 2005a] Baskerville, R., and Portougal, V. “Possibility Theory in Protecting National Information Infrastructure.” In K. Siau (Ed.), *Advanced Topics in Database Research* (Vol. 4). Idea Group, 2005.

[Baskerville 2005b] Baskerville, R., and Sainsbury, R. “Securing Against the Possibility of an Improbable Event: Concepts for Managing Predictable Threats and Normal Compromises.” *European Conference on Information Warfare and Security*, Glamorgan University, UK, 11-12 July 2005.

[Bass 1998] Bass, L., Clements, P., and R. Kazman, *Software Architecture in Practice*, SEI Series in Software Engineering, Reading, MA: Addison Wesley Longman, Inc, 1998.

[Bass 2001] Bass, L., M. Klein, and G. Moreno, *Applicability of General Scenarios to the Architecture Tradeoff Analysis Method*, CMU/SEI-2001-TR-014, ADA396098, Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2001.
Available at: http://www.sei.cmu.edu/publications/documents/01.reports/01tr014.html.

[Bazaz 2005] Bazaz, Anil and Arthur, James D., “On Vulnerabilities, Constraints and Assumptions,” 2005

[Beitler 2003] Beitler, Michael A., *Strategic Organizational Change*, Practitioner Press International; January 17, 2003.

[Bejtlich 2005] Bejtlich, Richard. *Extrusion Detection: Security Monitoring for Internal Intrusions*. Addison-Wesley Professional, 2005

[Bell 2005] Bell, David Elliot. “Looking Back at the Bell-La Padula Model,” *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC ’05).* pp 337-351, December 2005.

[Berg 2005] Berg, Clifford J, High-Assurance Design: Architecting Secure and Reliable Enterprise Applications, Addison Wesley, 2005.

[Berry 2001] Berry, John, “IT ROI Metrics Fall Into Four Groups,” *Internet Week*, July 16, 2001.

[Bernstein 2005] Bernstein, Lawrence and C. M. Yuhas. *Trustworthy Systems through Quantitative Software Engineering.* Wiley-IEEE Computer Society Press, 2005. About reliability not security.

[Bersoff 1980] Bersoff, E., V. Henderson, and S. Siegel., *Software Configuration Management*, Prentice-Hall, 1980.

[Bertino 2005] Bertino,Elisa, and Ravi Sandhu, Database Security-Concepts, Approaches, and Challenges, IEEE Transactions on Dependable and Secure Systems, Vol. 2, No. 1, pp. 2-19, January-March 2005

[Besnard 2001] Besnard, Denis, "Attacks in IT Systems: a Human Factors-Centred Approach". University of Newcastle upon Tyne, 2001. Available at http://homepages.cs.ncl.ac.uk/denis.besnard/home.formal/Publications/Besnard-2001.pdf.

[Beznosov 2004] Beznosov, K. and Kruchten, P. “Towards agile security assurance.” *Proceedings of the 2004 Workshop on New Security Paradigms*. NSPW '04. ACM Press, New York, NY, p. 47-54, September 20 - 23, 2004.

[Birman 1996] Birman, Kenneth, *Building Secure and Reliable Network Applications*, Manning Publications, Inc., 1996.

[Bishop 1995] Bishop, Matt, “A Taxonomy of UNIX System and Network Vulnerabilities,” Technical Report CSE-95-10, Department of Computer Science at the University of California at Davis, May 1995.

[Bishop 1996] Bishop, Matt, and Bailey, David, “A critical analysis of vulnerability taxonomies,” Technical Report CSE-96-11, Department of Computer Science at the University of California at Davis, Sept. 1996.

[Bishop 1999] Bishop, Matt, “Vulnerabilities analysis,” In Proceedings of Recent Advances in Intrusion Detection, pages 125–136, 1999.

[Bishop 2003] Bishop, Matt. *Computer Security: Art and Practice*, Addison-Wesley, 2003.

[Bishop 2006] Bishop, Matt, and Sophie Engle. “The Software Assurance CBK and University Curricula.” *Proceedings of the 10th Colloquium for Information Systems Security Education*, 2006.

[Blackburn 2001] Blackburn, Mark, Robert Busser, Aaron Nauman, and Ramaswamy Chandramouli, *Model-based Approach to Security Test Automation,* National Institute of Standards and Technology, 2001.

[Boehm 2003], Boehm, Barry, and Richard Turner, *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison-Wesley, 2003.

[Bosworth and Kabay 2002] Bosworth, Seymour and Kabay, M. eds. *Computer Security Handbook.* 4th Edition, John Wiley and Sons, 2002.

[Boudra 1993] Boudra, P., Jr. *Report on rules of system composition: Principles of secure system design*. Technical Report, National Security Agency, Information Security Systems Organization, Office of Infosec Systems Engineering, I9 Technical Report 1-93, Library No. S-240, 330, March 1993.

[Breu 2003] Breu, R., K. Burger, M. Hafner, J. Jürjens, G. Popp, G. Wimmel, and V. Lotz, "Key Issues of a Formally Based Process Model for Security Engineering," *Proceedings of the 16th International Conference on Software & Systems Engineering and their Applications (ICSSEA03),* 2003.

[Broadfoot and Broadfoot 2003] Broadfoot, G. and P. Broadfoot, "Academia and Industry Meet: Some Experiences of Formal Methods in Practice," *Proceedings of the Tenth Asia-Pacific Software Engineering Conference*, Chiang Mai, Thailand, IEEE Computer Society, December 2003.

[BS 15000-1] BS 15000-1: 2000, Specification for Service Management. BSI, 2000.

[BS 15000-2] BS 15000-2, 2000, *Code of Practice for Service Management*. BSI, 2000.

[BS ISO/IEC 17799] BS ISO/IEC 17799:2000, Information Technology - Code of practice for information security management.

[BSI 100-2 2005] Bundesamt für Sicherheit in der Informationstechnik version 1. *IT-Grundschutz Methodology*. Bundesamt für Sicherheit in der Informationstechnik, 2005.

[DHS BSI] Department of Homeland Security National Cyber Security Division's "Build Security In" (BSI) web site, (http://buildsecurityin.us-cert.gov).

[Burrows and Martin 1989] Burrows, Michael, Abadi, Martin, and Needham, Roger, "A Logic of Authentication", *Proceedings of the Royal Society*, Volume 426, Number 1871, 1989.

[Bush 2000] Bush, W.R., J.D. Pincus, and D.J. Sielaff, "A Static Analyzer for Finding Dynamic Programming Errors," *Software Practice and Experience*, vol. 30, June 2000.

[Bynum 2001] Bynum, Terrell. "Computer Ethics: Basic Concepts and Historical Overview," ***The Stanford Encyclopedia of Philosophy*, 2001 ed.**, Edward N. Zalta (ed.).
Available at: http://plato.stanford.edu/archives/win2001/entries/ethics-computer/

[Bynum and Rogerson 2004] Bynum, Terrell (ed.) and Simon Rogerson. *Computer Ethics and Professional Responsibility: Introductory Text and Readings*, Blackwell Publishing, 2004.

[Campara 2005] Campara, Djenana. *Secure Software: A Manager's Checklist*. Klocwork Whitepaper, June 20, 2005. Available at www.klocwork.com/company/downloads/SecureSoftwareManagerChecklist.pdf.

[Cannon 2005] Cannon, J. C. *Privacy*, Addison Wesley, 2005.

[Carnegie Mellon University] Carnegie Mellon University Fox Project: Proof-Carrying Code. Available at http://www.cs.cmu.edu/~fox/pcc.html.

[Carter 2004] Carter, Earl, Cisco Systems Inc., *CCSP Self-Study: Cisco Secure Intrusion Detection System,* Cisco Press, 2004.

[CASIS3 2004] *Third Annual Conference on the Acquisition of Software-Intensive Systems*, sponsored by the Software Engineering Institute (SEI) and the Office of the Under Secretary of Defense (Acquisition, Technology, and Logistics), Defense Systems, Software-Intensive Systems, January 26-28, 2004. Available at http://www.sei.cmu.edu/products/events/acquisition/2004-presentations/

[CC 1999] No Author (1999, October*). Common Criteria User Guide*. (downloaded on September 11, 2005)
Available at: http://www.commoncriteriaportal.org/public/consumer/index.php?menu=1

[CC 2005] No Author. *Common Criteria v. 3.0*, The National Institute of Standards and Technology, July, 2005.

[CC 2005, Part 1] No Author. Common Criteria for Information Technology Security Evaluation. Part 1: Introduction and General Model. V. 3.0, Rev. 2. 2005.

[CC 2005, Part 2] No Author. Common Criteria for Information Technology Security Evaluation. Part 2: Security Functional Components. V. 3.0, Rev. 2. 2005.

[CC 2005, Part 3] No Author. Common Criteria for Information Technology Security Evaluation. Part 3: Security Assurance Components. V. 3.0, Rev. 2. 2005.

[CCEVS 2005] *The Common Criteria Evaluation and Validation Scheme website* (August 2005) Available at: http://niap.nist.gov/cc-scheme/vpl/vpl_vendor.html.

[CCIMB-2004-01-001] CCIMB-2004-01-001, Common Criteria for Information Technology Security Evaluation, 2004.

[CERIAS TR 2000-01] Center for Education and Research in Information Assurance and Security and Andersen Consulting, *Policy Framework for Interpreting Risk in Ecommerce Security,* CERIAS Tech Report 2000-01, Center for Education and Research in Information Assurance and Security, Purdue University, 2000.
Available at: https://www.cerias.purdue.edu/tools_and_resources/bibtex_archive/archive/2000-01.pdf.

[Cerven 2002] Cerven, Pavol. Crackproof Your Software: Protect Your Software Against Crackers, First Edition. No Starch Press, 2002.

[Chaves et al, 2005] Chaves, C., H. P. C., L.H Franco, and A. Montest. "Honetnet Maintenance Procedures and Tools", *Proceedings 6th IEEE Systems, Man and Cybernetics Information Assurance Workshop* (IAW 05), pp.252-257, 2005.

[Chen 2004] Chen, Peter, Marjon Dean, Don Ojoko-Adams, Hassan Osman, Lilian Lopez, Nick Xie. *System Quality Requirements Engineering (SQUARE) Methodology: Case Study on Asset Management System.* CMU/SEI-2004-SR-015, Software Engineering Institute, December 2004. Available at http://www.sei.cmu.edu/publications/documents/04.reports/04sr015.html

[Chirillo 2002] Chirillo, John. Hack Attacks Revealed: A Complete Reference for UNIX, Windows, and Linux with Custom Security Toolset. Wiley Publishing, Inc., 2002.

[Christensen 2003] Christensen, Clayton M., *The Innovator's Dilemma*. HarperBusiness, January 7, 2003.

[Chung 1999] Chung, Lawrence, et al. *Non-Functional Requirements in Software Engineering*, Kluwer, 1999.

[CJCSI 3401.03A] Chairman of the Joint Chiefs of Staff Instruction (CJCSI) 3401.03A (current as of August 2005). *Information Assurance (IA) and Computer Network Defense (CND) Joint Quarterly Readiness Review (JQRR) Metrics*. Washington, DC.: Chairman of the Joint Chiefs of Staff.

[CJCSM 6510.01 2004] CJCSM 6510.01, *Defense-In-Depth: Information Assurance (IA) and Computer Network Defense (CND),* DoD Joint Staff, 2004. (Official Use Only)

[Clark and Wilson 1987] Clark, David D. and David R. Wilson, "A Comparison of Commercial and Military Computer Security Policies,*" Proc. of the 1987 IEEE Symposium on Security and Privacy,* IEEE, pp. 184-196, 1987.

[Clarke 2003] Clarke, Patrick E. "Project Targets Software Security," *Military Information Technology, Volume: 7 Issue: 1,* Jan 17, 2003.
Available at: http://www.military-information-technology.com/article.cfm?DocID=38

[Clarke and Grumberg 1999] Clarke, Edmund, Orna Grumberg, and Doron Peled, *Model Checking*, MIT Press, December 1999.

[Clarke and Wing 1996] Clarke, Edmund and Jeannette Wing, “Formal Methods: State of the Art and Future Directions”, *ACM Computing Surveys*, Vol. 28, No. 4, December 1996.

[CNSSI 4009] CNSS, National Information Assurance Glossary, May 2003. Available at http://www.cnss.gov/full-index.html

[COBIT] COBIT (Control Objectives for Information and Related Technology) framework for IT governance at http://www.isaca.org/Template.cfm?Section=COBIT_Online&Template=/ContentManagement/ContentDisplay.cfm&ContentID=15633.

[Cohen 2001] Cohen, Fred, Dave Lambert, Charles Preston, Nina Berry, Corbin Stewart, and Eric Thomas, “A Framework for Deception", Final Report IFIP-TC11, 2001.

[Cohen 2004] Cohen, Lazaro Issi, and Joseph Issi Cohen, *The Web Programmer's Desk Reference*, No Starch Press, 2004.

[Common Criteria Part 1] Common Criteria Project, Common Criteria for Information Technology Security Evaluation Part 1: Introduction and general model, Version 2.1, CCIMB-99-031, August 1999.

[Common Criteria Part 2] Common Criteria Project, Common Criteria for Information Technology Security Evaluation Part 2: Security Functional Requirements, Version 2.1. CCIMB-99-031, August 1999.

[Cranor 2005] Cranor, Lorrie, and Simson Garfinkel. Security and Usability: **Designing Secure Systems that People Can Use**. O’Reilly, 2005.

[Croll 2004] Croll, Paul R. “Best Practice Approach for Software Maintenance - Sustaining Critical Capability,” *Proceedings, 16th Annual Systems and Software Technology Conference*, IEEE, April 2004, pp. 1355-1440.

[CSTB 2004] Committee on Certifiably Dependable Software Systems. *Summary of a Workshop on Software Certification and Dependability*. National Academies Computer Science and Telecommunications Board, National Academies Press, 2004.

[Cusumano 2005] Cusumano, Michael A., “Who is Liable for Bugs and Security Flaws in Software? Attempting to determine fault and responsibility based on available evidence”, *Communications of the ACM*, March 2004/Vol. 47, No. 3 pp. 25-27.

[DACS API] DACS Gold Practice, *Acquisition Process Improvement*. Available at http://www.goldpractices.com/practices/api/

[Dart 1996] Dart, Susan A., “Achieving the Best Possible Configuration Management Solution,” *Crosstalk*, September 1996.

[DAU (SEF) 2001] No Author (January 2001). *Systems Engineering Fundamentals*. Fort Belvoir, VA: Defense Acquisition University, 2001.

[Davis 1993] Davis, A.M. Software Requirements: Objects, Functions and States, Prentice Hall, 1993.

[Davis and Mullaney 2003] Davis, Noopur, and Mullaney, Julia, “The Team Software Process in Practice: A Summary of Recent Results,” Technical Report CMU/SEI-2003-TR-014, September 2003.

[DCID 6/3 2000] Director Central Intelligence. *Protecting Sensitive Compartmented Information Within Information Systems*, (DCID 6/3) Manual. 24 May 2000.

[Dean 1996] Dean, C. Neville. *Teaching and Learning Formal Methods*. Morgan Kaufmann,1996.

[Dean 2004] Dean, C. Neville and Raymond T. Boute, (Eds.). *Teaching Formal Methods: CoLogNET/FME Symposium*, TFM 2004, Ghent, Belgium, November 18-19, 2004. Lecture Notes in Computer Science, Vol. 3294, Springer, 2004
[Despotou 2004] Despotou, Georgios, and Tim Kelly. "Extending the Safety Case Concept to Address Dependability," *Proceedings of the 22nd International System Safety Conference*, 2004.

[Delahaye 1998] Delahaye, Brian L. and Barry J. Smith. *How to Be an Effective Trainer: Skills for Managers and New Trainers*. Wiley; 3rd edition 1998.

[DeLooze 2004] DeLooze, L. "Classification of computer attacks using a self-organizing map," *Proceedings from the Fifth Annual IEEE SMC*, 10-11 June 2004, Pages: 365-369, 2004.

[Deming 1986] Deming, W. Edward. *Out of the Crisis*. Cambridge, MA: MIT Center for Advanced Engineering, 1986.

[Denning 1999] Denning, Dorothy E. *Information Warfare and Security*, pp 46-50. Reading MA: Addison-Wesley, 1999.

[Despotou 2004] Despotou, Georgios, and Tim Kelly, "Extending the Safety Case Concept to Address Dependability," *Proceedings of the 22nd International System Safety Conference*, p. 645-654, 2004.

[Devanbu et al. 1999] Devanbu, P., M. Gertz and Stuart Stubblebine. Security for Automated, Distributed Configuration Management. Proceedings, ICSE 99 Workshop on Software Engineering over the Internet, 1999.
http://www.stubblebine.com/99icse-workshop-stubblebine.pdf

[DIAM 50-4 1997] *DIAM 50-4 Security of Compartmented Computer Operations*. Department of Defense (DoD) Intelligence Information System (DODIIS) Information Systems Security (INFOSEC) Program, 30 April 1997.

[DITSCAP 1997] DoD Instruction 5200.40*, DoD Information Technology Security Certification and Accreditation Process (DITSCAP)*, December 30, 1997.

[DoD 5200.28-STD 1985] DOD 5200.28-STD, Department of Defense Trusted Computer System Evaluation Criteria, 1985.

[DoD 8510.1-M] DoD 8510.1-M DoD *Information Technology Security Certification and Accreditation Process (DITSCAP) Application Manual*, July 31, 2000.

[DoD 8510.1-M] DoD 8510.1-M, *DoD Information Technology Security Certification and Accreditation Process (DITSCAP) Application Manual*, July 31, 2000.

[DoD EVMS 2005] Department of Defense (7 April 2005). *Earned Value Management Implementation Guide*. Washington, DC: US Department of Defense, 2005.

[DoD PMI 2003] No Author (June 2003). US Department of Defense Extension to: A Guide to the Project Management Body of Knowledge (PMBOK® Guide). V. 1.0. Fort Belvoir, VA: Defense Acquisition University, 2003.

[DoDD 5200.39] Department of Defense Directive 5200.39, Security, Intelligence and Counterintelligence Support to Acquisition Program Protection. 10 September 1997.

[DoDD 8500.1] Department of Defense Directive 8500.1 (24 October 2002-certified current as of 21 November 2003). *Information Assurance (IA)*. Washington, DC: US Department of Defense, 2002.

[DoDI 5000.2] Department of Defense Instruction (12 May 2003). *Operation of the Defense Acquisition System*. Washington, DC: US Department of Defense, 2003.

[DoDI 8500.2] Department of Defense Instruction 8500.2 (6 February 2003). *Information Assurance (IA) Implementation*. Washington, DC: US Department of Defense, 2003.

[DoDI S-3600.2] DOD Instruction S-3600.2, Information Operations (IO) Security Classification Guidance, 6 August 1998.

[DoD PMI 2003] No Author (June 2003). US Department of Defense Extension to: A Guide to the Project Management Body of Knowledge (PMBOK® Guide). V. 1.0. Fort Belvoir, VA: Defense Acquisition Univeristy, 2003.

[Dorofee 1997] Dorofee A.J., JA Walker, and RC Williams. “Risk Management in Practice”, *Crosstalk*, Volume 10 #4, April 1997.

[DTI UK 2004] Price, Waterhouse and Coopers, *Information Security Breaches 2003*, Department of Trade and Industry (DTI), U.K., 2004.

[Du 1998] Du, Wenliang, and Aditya P. Mathur. “Vulnerability Testing of Software System Using Fault Injection,” *COAST*, Purdue University, 1998.

[Duce 2003] Duce, David, et al. *Teaching Formal Methods: Practice and Experience*. Workshop at Oxford Brookes University, December 12, 2003. Available at http://cms.brookes.ac.uk/tfm2003/ [Dustin et al. 2001] Dustin, Elfriede, Jeff Rashka and Douglas McDiarmid. Quality Web Systems: Performance, Security, and Usability, First Edition. Boston, MA: Addison-Wesley Professional, 2001.

[EC-Council 2003] International Council of Electronic Commerce Consultants. *Ethical Hacking*. Osb Publisher Pte Ltd, 2003.

[EC-Council CEH 312-50] International Council of Electronic Commerce Consultants. *Certified Ethical Hacker Exam Study Guide*. International Council of Electronic Commerce Consultants, n.d. Available through http://www.itcertkeys.com/shop/product_info.php/products_id/307

[Eeles 2004] Eeles, Peter, *Appendix C: Sample Architectural Requirements Questionnaire*, IBM 30 Apr 2004.
Available at: http://www-128.ibm.com/developerworks/rational/library/4710.html

[Ellison 2003] Ellison, Robert J., and Andrew P. Moore. *Trustworthy Refinement Through Intrusion-Aware Design (TRIAD)*. Technical Report CMU/SEI-2003-TR-002. Software Engineering Institute, October 2002 Revised March 2003.

[Endicott-Popovsky 2003] Endicott-Popovsky, Barbara, “Ethics and Teaching Information Assurance,” *IEEE Security and Privacy* Vol. 1, No. 4, July-August 2003, pp. 65-67

[Epstein 2005] Epstein, Jeremy, Scott Matsumoto, and Gary McGraw. “Software Security and SOA: Danger, Will Robinson!” *IEEE Security & Privacy*. Vol. 4, No.1, pp 80-83, January/February 2006.

[Ernst 2003] Ernst, Michael D. “Static and dynamic analysis: Synergy and duality,” WODA 2003: *ICSE Workshop on Dynamic Analysis*, (Portland, OR), May 9, 2003, pp. 24-27.

[Escamilla 1996] Escamilla, T. “Intrusion Detection: Network Security Beyond the Firewall,”, Wiley, 1998, Chap. 5.Jones, Capers, Software Defect Removal Efficiency, *Computer*, April 1996, Vol.29, #4.

[EU 1995] “Directive 95/46/EC of the European Parliament and of the Council of 24 October 1995 on the protection of individuals with regard to the processing of personal data and on the free movement of such data,” *Official Journal of the European Communities of* 23 November 1995 No L. 281 p. 31. Available at: http://www.cdt.org/privacy/eudirective/EU_Directive_.html

[Evans 2005] Evans, S. and J. Wallner. “Risk-Based Security Engineering Through the Eyes of the Adversary,” *Proc. 6th Ann. IEEE Systems, Man and Cybernetics Information Assurance Workshop* (IAW 05), IEEE CS Press, 2005, pp.158-165.

[Evans and Larochelle 2002] “Improving Security Using Extensible Lightweight Static Analysis,” *IEEE Software*, Jan. 2002, pp. 42-51.

[FAR, 2005] Federal Acquisition Regulation.

[Feathers 2005] Feathers, M.C., *Working Effectively with Legacy Code*, Prentice Hall, 2005.

[Feiler 1990] Feiler, P., “Software Process Support in Software Development Environments,” *Fifth International Software Process Workshop*, ACM Press, October 1990.

[Feiler 1991] Feiler, P., *Configuration Management Models in Commercial Environments*, Tech. report CMU/SEI-91-TR-7, ADA235782, Software Engineering Institute, Carnegie-Mellon University, April 1991.

[Fench 1999] French, Wendell L. *Organization Development and Transformation: Managing Effective Change.* McGraw-Hill/Irwin; 5th edition July 13, 1999.

[Fernandez 2005]E.B.Fernandez and Maria M. Larrondo-Petrie, “Using UML and security patterns to teach secure systems design.” *Proceedings of the American Society for Engineering Education (ASEE 2005) Annual Conference.*

[Fichman 1993] Fichman and Kemerer, “Adoption of Software Engineering Process Innovations: The Case of Object Orientation,” *Sloan Management Review*, Winter 1993, pp. 7-22.

[FIPS 188] FIPS 188, Standard Security Labels for Information Transfer, September 1994.

[FIPS Pub 199] Federal Information Processing Standard (FIPS) Publication 199 (February 2004). *Standards for Security Categorization of Federal Information and Information Systems*. Gaithersburg, MD.: National Institute of Standards and Technology (NIST), U.S. Department of Commerce.

[FIPS Pub 200] Federal Information Processing Standard (FIPS) Publication 200 (July 2005). *Minimum Security Standard for Federal Information Systems*. Gaithersburg, MD.: National Institute of Standards and Technology (NIST), U.S. Department of Commerce, 2005.

[Firesmith 2005] Firesmith, Donald G., "A Taxonomy of Security-Related Requirements," Software Engineering Institue, Carnegie Mellon University, 2005.

[FISMA 2002] Federal Information Security Management Act of 2002, 44 U.S.C. § 3541 et seq.

[Fitzgerald et al 2005] Fitzgerald, John, Peter Gorm Larsen, Nic Plat and Marcel Verhoef. *Validated Designs for Object-oriented Systems*. Springer Verlag, NewYork. 2005.

[Fitzgerald 2002] Fitzgerald, Kevin J., “U.S. Defense Department Requirements for Information Security”, *Crosstalk*, May 2002.

[Flechais 2003] Flechais, I., Sasse, M. A., and Hailes, S. M., “Bringing security home: a process for developing secure and usable systems,” In *Proceedings of the 2003 Workshop on New Security Paradigms* (Ascona, Switzerland, August 18 - 21, 2003). C. F. Hempelmann and V. Raskin, Eds. NSPW '03. ACM Press, New York, NY, 49-57.

[Flickenger 2003] Flickenger, Rob. *Wireless Hacks*. O’Reilly and Associates, Inc., 2003.

[Foster and Foster 2005] Foster, James and Steven Foster, *Programmers Ultimate Security Desk Reference*, Syngress Press, 2005.

[Foster and Osipov 2005] Foster, James and Osipov, Vitaly, et al., *Buffer Overflow Attacks: Detect, Exploit, Prevent*, Syngress Press, 2005.

[Fötinger and Ziegler 2004] Fötinger, Christian S. and Wolfgang Ziegler, "Understanding a Hacker's Mind - A Psychological Insight into the Hijacking of Identities". Krems, Austria: Donau-Universität Krems, 2004. Available at http://www.donau-uni.ac.at/de/studium/fachabteilungen/tim/zentren/zpi/DanubeUniversityHackersStudy.pdf.

[Fowler 1995] Fowler, C. A., and R. F. Nesbit. "Tactical deception in air-land warfare," *Journal of Electronic Defense*, vol. 18, no. 6. June, 1995, pp. 37-44 & 76-79.

[FTC 2000] US Federal Trade Commission, Fair Information Practices Report to Congress, US Federal Trade Commission, 2000
Available at: www.ftc.gov/reports/privacy2000/privacy2000.pdf

[Fugini 2004] Fugini, Mariagrazia, and Carlos Bellettini (Editors). *Information Security Policies and Actions in Modern Integrated Systems*. Idea Group Publishing, 2004.

[Futrell et al. 2002] Futrell, Robert T., Donald F. Shafer and Linda I. Shafer. Quality Software Project Management, First Edition. Upper Saddle River, NJ: Prentice Hall Professional Technical Reference, 2002.

[Gaines & Michael, 2005] Gaines, L. and J. Michael (2005, March). "Service Level Agreements as Vehicles for Managing Acquisition of Software-Intensive Systems". *Defense Acquisition Review Journal*, 37, 284-303, 2005.

[Gallagher et al. 2006] Gallagher, Tom, Bryan Jeffries and Lawrence Landauer. *Hunting Security Bugs*. Microsoft Press, 2006]

[GAO 1999] General Accounting Office, *GAO Internal Control Standard*, 1999.

[GAO 2004] GAO, *Defense Acquisitions: Stronger Management Practices Are Needed to Improve DoD's Software-Intensive Weapon Acquisitions*, GAO Report GAO-04-393, March 2004. Available at http://www.gao.gov/new.items/d04393.pdf.

[Garfinkel 2003] Simson L. Garfinkel, Abhi Shelat, "Remembrance of Data Passed: A Study of Disk Sanitization Practices," *IEEE Security and Privacy*, vol. 01, no. 1, pp. 17-27, 2003.

[Garfinkel 2005a] Garfinkel, Simson L. *Design Principles and Patterns for Computer Systems that are Simultaneously Secure and Usable*, PhD Thesis MIT, 2005
Available at: http://www.simson.net/thesis/

[Garfinkel 2005b] Garfinkel, Simson L., "CSCI E-170 Lecture 09: Attacker Motivations, Computer Crime and Secure Coding". Cambridge, MA: Harvard University Center for Research on Computation and Society, 21 November 2005. Available at http://www.simson.net/ref/2005/csci_e-170/slides/L09.pdf.

[Gasser 1988] Gasser, M. *Building a Secure Computer System.* Van Nostrand Reinhold, 1988.
Available: http://nucia.ist.unomaha.edu/library/gasser.php

[GASSP 1999] GASSP, "Generally Accepted System Security Principles," *International Information Security Forum*, June 1999.

[Gilb 1988] Gilb, Tom. Principles of Software Engineering Management. Boston: Addison-Wesley, 1988.

[Giorgini 2004] P. Giorgini, F. Massacci, J. Mylopoulous, and N. Zannone. "Requirements Engineering meets Trust Management: Model, Methodology, and Reasoning." *Proc. of the 2nd Int. Conf. on Trust Management (iTrust)* 2004.

[Giorgini 2005] Giorgini, Paolo, Fabio Massacci, John Mylopoulos, Nicola Zannone. "Modeling Security Requirements Through Ownership, Permission and Delegation," *13th IEEE International Conference on Requirements Engineering* (RE'05). pp. 167-176, 2005.

[Godbole 2004] Godbole, Nina S. *Software Quality Assurance: Principles And Practice*. Oxford, UK: Alpha Science International, Ltd., 2004.

[Goertzel 2005] Goertzel, K. *Application Developer's Guide to Secure Assembly of Software Components*. Washington, D.C: Booze Allen Hamilton, 2005.

[Goertzel 2006] Goertzel, Karen Mercedes, et al: *Security in the Software Lifecycle: Making Application Development Processes – and Software Produced by Them – More Secure*, Version 1.0 DRAFT. Washington, DC: Department of Homeland Security, 2006. Available at https://buildsecurityin.us-cert.gov/daisy/bsi/89.html.

[Goertzel and Goguen 2005] Goertzel, Karen and Alice Goguen, et al., *Application Developer's Guide to Security-Enhancing the Software Development Lifecycle*. DRAFT, June 2005.

[Goguen and Linde, 1993] J. Goguen and C. Linde, "Techniques for Requirements Elicitation," International Symposium on Requirements Engineering, 1993.

[Goguen and Meseguer 1982] Goguen, J. A. and Meseguer, J., "Security Policies and Security Models," *1982 Symposium on Security and Privacy*, pp.11-20, IEEE, April 1982.

[Goldenson and Gibson 2003] Goldenson, Dennis R. and Gibson, Diane L. *Demonstrating the Impact and Benefits of CMMI*. Special Report CMU/SEI-2003-SR-009, The Software Engineering Institute, Carnegie Mellon University, 2003.

[Gotterbarn 1991] Gotterbarn, Donald. "Computer Ethics: Responsibility Regained," *National Forum*, vol. 71, pp. 26-31, 1991.

[Graff and van Wyk 2003] Graff, Mark G. and Kenneth R. Van Wyk. *Secure Coding: Principles and Practices*. O'Reilly & Associates, 2003.

[Gray 1990] Gray, J. W. "Probabilistic Interference." *Proceedings of the IEEE Symposium on Research in Security and Privacy*. IEEE, pp. 170-179, 1990.

[Guimaraes 2004] Guimaraes, Mario, Herb Mattord, and Richard Austin, "Incorporating security components into database courses," Proceedings of the 1st annual conference on Information security curriculum development, ACM, October 2004.

[Gutmann 2004] Gutmann, P. *Cryptographic Security Architecture: Design and Verification*. Springer-Verlag, 2004.

[Haddad 2004] Haddad, M. and A. Salle. "Identifying Risks in Outsourcing Software-Intensive Projects," *3rd Annual Conference, Acquisition of Software Intensive Systems*, January 2004.

[Hafiz 2004] Hafiz, M., R. Johnson, and R. Afandi, "The Security Architecture of qmail," *Conference on Pattern Languages of Programs, Conference on Pattern Languages of Programs* (PLoP 2004), ACM, 2004.

[Hall 2002a] Hall, Anthony and Rodrick Chapman. "Correctness by Construction: Developing a Commercial Secure System," *IEEE Software*, vol. 19, no. 1, pp.18-25, Jan/Feb 2002.

[Hall 2002b] Hall, Anthony, "Z Styles for Security Properties and Modern User Interfaces," *Formal Aspects of Security, LNCS 2629*, Springer, pp152 – 166, 2002.
Available at: http://www.anthonyhall.org/zstyle.pdf

[Hall 2004] Hall, Anthony, and Rod Chapman. "Correctness-by-Construction". *Cyber Security Summit Taskforce SubgIbraroup on Software Process*. January 2004.

[Han 1997] Han, Jun. "Designing for Increased Software Maintainability," *International Conference on Software Maintenance (ICSM, 97)*, January 1, 1997.

[Hansman 2003] Hansman, Simon, "A Taxonomy of Network and Computer Attack Methodologies," Department of Computer Science and Software Engineering, University of Caterbury, Christchurch, New Zealand, Nov. 2003.

[Harris 2003] Harris, Shon. *All-in-One CISSP Certification.* McGraw-Hill Inc., 2003.

[Harris et al. 2005] Harris, Shon, Allen Harper, Chris Eagle, Jonathan Ness, and Michael Lester. *Gray Hat Hacking: The Ethical Hacker's Handbook*. McGraw-Hill/Osborne, 2005.

[Hatton 1999] Hatton, L. (1999) "Repetitive failure, feedback and the lost art of diagnosis," *Journal of Systems and Software,* 1999.

[Hatton 2001] Hatton, L. "Exploring the role of Diagnosis in Software Failure", *IEEE Software,* July 2001.

[Hatton 2002] Hatton, L. "Safer Language Subsets: an overview and a case history," MISRA C, *Information and Software Technology,* June 2002.

[Havana, 2003] Havana Tiina, Juha Rön̈ing, Communication in the Software Vulnerability Reporting Process, Oulu University Secure Programming Group (OUSPG), Computer Engineering Laboratory, PL 4500, FIN-90014 University of Oulu, Finland.

[Hayes and Over 1997] Hayes, W. and J. W. Over, The Personal Software Process (PSP): An Empirical Study of the Impact of PSP on Individual Engineer,. CMU/SEI-97-TR-001, ADA335543. Pittsburgh, PA: The Software Engineering Institute, Carnegie Mellon University, 1997.

[Heitmeyer 1998] Heitmeyer, Constance. "Using the SCR* Toolset to Specify Software Requirements," *Second IEEE Workshop on Industrial Strength Formal Specification Techniques,* p. 12, 1998.

[Heitmeyer 2005] Heitmeyer, Constance, Myla Archer, Ramesh Bharadwaj and Ralph Jeffords, "Tools for constructing requirements specifications: The SCR toolset at the age of ten," *International Journal of Computer Systems Science and Engineering*, 20(1): 19-35, January 2005.

[Herbert and Chase 2005] Herbert Thompson, and Chase, Scott. *The Software Vulnerability Guide*. Charles River Media, 2005.

[Herbsleb et al, 1994] Herbsleb, J. et al. *Benefits of CMM-Based Software Process Improvement: Initial Results*, CMU/SEI-94-TR-013, Software Engineering Institute, Carnegie Mellon University, 1994.

[Herrmann 2001] Herrmann, Debra S. A Practical Guide to Security Engineering and Information Assurance. Auerbach, 2001.

[HMAC 2002] "The Keyed-Hash Message Authentication Code (HMAC)", FIPS 198, March 2002.

[Hofmeister 2000] Hofmeister, C., R. Nord., D. Soni, *Applied Software Architecture. Reading*, MA: Addison Wesley Longman, Inc, 2000.

[Hoglund 2004] Hoglund, Greg, and Gary McGraw. *Exploiting Software: How to break code*. Addison-Wesley, 2004.

[Hoglund 2005] Hoglund, Greg and Jamie Butler, *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, 2005.

[Holz and Ravnal 2005] Holz, T. and F. Ravnal, "Detecting Honeypots and Other Suspicious Environments," *Proc. 6th Ann. IEEE Systems, Man and Cybernetics Information Assurance Workshop* (IAW 05), IEEE CS Press, pp.29-36, 2005.

[Honeynet 2002] "The Honeynet Project". *Know Your Enemy*. Addison-Wesley, 2002.

[Hope 2004] Hope, Paco, Gary McGraw, and Annie I. Anton, "Misuse and Abuse Cases: Getting Past the Positive," *IEEE Security and Privacy*, pp. 90-92, May 2004.

[Houston and King 1991] Houston, I., and S. King, "CICS Project Report: Experiences and Results from the Use of Z," *Proc. VDM 1991: Formal Development Methods*, Springer-Verlag, New York, 1991.

[Howard 2002] Howard, Michael, and David C. LeBlanc. *Writing Secure Code, 2nd ed.*, Microsoft Press, 2002.

[Howard 2003a] Howard, M., and S. Lipner, "Inside the Windows Security Push," *IEEE Security & Privacy*, vol.1, no. 1, pp. 57-61, 2003.

[Howard 2003b] Howard, M., J. Pincus and J. Wing. "Measuring relative attack surfaces," ***Proceedings of the Workshop on Advanced Developments in Software and Systems Security***, Available as CMU-TR-03-169, August 2003.

[Howard 2005] Howard, Michael, David LeBlanc, and John Viega, *19 Deadly Sins of Software Security*, McGraw-Hill Osborne Media, 1st edition, 2005.

[Howard 2006] Howard, Michael, and Steve Lipner. *The Security Development Lifecycle*. Microsoft Press, 2006.

[Howard and LeBlanc 2003] Howard, Michael and LeBlanc, David, *Writing Secure Code, 2nd Edition*, Microsoft Press, 2003.

[Howell 2005] Howell, C. "Assurance Cases for Security Workshop," (follow-on workshop of the 2004 Symposium on Dependable Systems and Networks), Arlington, Virginia, June 13-15, 2005.

[Huang 2004] Huang, Y., F. Yu, C. Hang, C. Tsai, D. Lee, and S. Kuo, "Securing Web Application Code by Static Analysis and Runtime Protection," *Proceedings of the 13th International Conference on World Wide Web (WWW '04)*. ACM Press, pp. 40-52. 2004.

[Hughes n. d.] Hughes, Jeff, and Martin R. Stytz, *Advancing Software Security–The Software Protection Initiative,* n.d.
Available at: http://www.preemptive.com/documentation/SPI_software_Protection_Initative.pdf

[Humphrey 2000] Humphrey, Watts S. *Introduction to the Team Software Process*, Reading, MA: Addison Wesley, 2000.

[Humphrey 2002] Humphrey, Watts S. *Winning with Software: An Executive Strategy*. Reading, MA: Addison-Wesley, 2002.

[Hunt et al, 2005] Hunt, C., J. R. Bowes, and D. Gardner, "Net Force Maneuver," *Proc. 6th Ann. IEEE Systems, Man and Cybernetics Information Assurance Workshop* (IAW 05), IEEE CS Press, pp.419-423, 2005.

[Huseby 2004] Sverre H. Huseby. Innocent Code: A Security Wake-up Call for Web Programmers. John Wiley & Sons, 2004).

[IBM 2005] IBM. *The IBM Risk and Compliance Framework: addressing the challenges of compliance*, 2005 www.ibm.com/software/info/openenvironment/rcf/pdfs/rcf-white-paper-01-25-05.pdf

[Ibrahim et al, 2004] Ibrahim, Linda, et al, *Safety and Security Extensions for Integrated Capability Maturity Models*. Washington D.C.: United States Federal Aviation Administration, Sept. 2004. Available at http://www.faa.gov/ipg/pif/evol/index.cfm

[IEE 1999] IEE, *Safety, Competency and Commitment*, IEE, ISBN 0 85296 787 X, 1999.

[IEEE 730] IEEE 730 1998 Software Quality Assurance Plans.

[IEEE 730.1] IEEE 730.1 1995 Software Quality Assurance Planning.

[IEEE 828] IEEE 828 1998 Software Configuration Management Plan.

[IEEE 829] IEEE 829 1998 Software Test Documentation.

[IEEE830-98] IEEE Std 830-1998, IEEE Recommended Practice for Software Requirements Specifications, IEEE, 1998.

[IEEE 1008] IEEE 1008 Standard for Software Unit Testing.

[IEEE 1012] IEEE 1012 1986 Software Validation and Verification Plan.

[IEEE 1012a] IEEE 1012a-1998 Content Map to IEEE/EIA 12207.1-1997.

[IEEE 1028] IEEE 1028-1997 Standard for Software Reviews.

[IEEE 1045] IEEE 1045-1992 IEEE Standard for Software Productivity Metrics.

[IEEE 1059] IEEE 1059 1993 Guideline for SVV Planning.

[IEEE 1062] IEEE Std. 1062:1993, *IEEE recommended practice for software acquisition,* Institute for Electrical and Electronics Engineers, 1993.

[IEEE 12207] IEEE/EIA Std. 12207.0:1996, Industry Implementation of International Standard ISO/IEC 12207:1995 – Standard for Information Technology – Software Lifecycle Processes, Institute of Electrical and Electronics Engineers, March, 1998.

[IEEE/ANSI 1042] *IEEE Guide to Software Configuration Management*, IEEE/ANSI Standard 1042-1987,1987.

[IEEE/ANSI 828] *IEEE Standard for Software Configuration Management Plans*, IEEE/ANSI Standard 828-1998, 1998.

[IEEE/EIA 12207.1] IEEE/EIA 12207.1-1997, IEEE/EIA Guide: Industry Implementation of International Standard ISO/IEC 12207:1995, 1998.

[IEEE/EIA 12207.2] IEEE/EIA 12207.2-1997, IEEE/EIA Guide: Industry Implementation of International Standard ISO/IEC 12207:1995, 1998.

[IEEE] *IEEE Security and Privacy* magazine and *IEEE Transactions on Dependable and Secure Computing*. Institute for Electrical and Electronics Engineers Computer Society.

[IEEE] *IEEE Security and Privacy* magazine and *IEEE Transactions on Dependable and Secure Computing.* Institute for Electrical and Electronics Engineers Computer Society.

[IEEE830-98] IEEE Std 830-1998, IEEE Recommended Practice for Software Requirements Specifications, IEEE, 1998.

[INCOSE] International Council on Systems Engineering (INCOSE). *Guide to Systems Engineering Body of Knowledge (G2SEBoK)*. Available at http://g2sebok.incose.org/.

[Ingalsbe 2004] Ingalsbe, Jeffery A. "Supporting the Building and Analysis of an Infrastructure Portfolio of software Using UML Deployment Diagrams," *UML Satellite Activities 2004*, pp 105-117, 2004.

[Intek] http://www.intek.net/Secure/PIS/PIS.htm

[ISACA (AS) 1999] ISACA, "Audit Sampling", *IS Auditing Guideline*, 1999.

[ISACA (CM) 2004] IT Governance Institute, *CobiT Mapping*, ISACA, 2004.

[ISACA (CRSA) 2003] ISACA, "Control Risk Self Assessment", *IS Auditing Guideline*, 2003.

[ISACA (DPC) 1999] ISACA, "Due Professional Care", *IS Auditing Guideline*, 1999.

[ISACA (ICO) 1999] IT Governance Institute, *IT Control Objectives for Enterprise Governance*, ISACA, 1999.

[ISACA (ID) 2003] ISACA, "Intrusion Detection", *IS Auditing Guideline*, 2003.

[ISACA (ISC) 2003] IT Governance Institute, *IT Strategy Committee*, ISACA, 2003.

[ISACA (SDLC) 2003] ISACA, "SDLC Reviews," *IS Auditing Guideline*, 2003.

[ISACA (SO) 2004] IT Governance Institute, *IT Control Objectives for Sarbanes-Oxley*, ISACA, 2004.

[ISACA (URA) 2000] ISACA, "Use of Risk Assessment," *IS Auditing Guideline*, 2000.

[ISACA 2004-CobiT] "CobiT in Academ," *IT Governance Institute*, 2004, available from Information Systems Audit and Control Association: http://www.isaca.org, Accessed: July 20, 2005.

[ISACA 2005-COBIT] "COBIT 3rd Edition Executive Summary," *IT Governance Institute*, Available from Information Systems Audit and Control Association: http://www.isaca.org, Accessed: July 2005.

[ISC 2005] International Information Systems Security Certification Consortium, *Code of Ethics*, 2005. Available at: http://www.isc2.org

[ISO/IEC 9126:1991] ISO/IEC 9126:1991 Information technology - Software product evaluation - quality characteristics and guidelines for their use. ISO, 1991.

[ISO TR 15443 - 1]. ISO JTC 1/SC 27. *Information technology – Security techniques – A framework for IT security assurance – Part 1: Overview and framework*. International Organization for Standardization, 2005.

[ISO/IEC 12207] ISO/IEC Std. 12207:1995, *Information Technology - Software Life Cycle Processes,* International Standards Organization, 1995.

[ISO/IEC 12207b] ISO/IEC Std. 12207:1995/2002*, Information Technology - Software Life Cycle Processes (Amendment 1),* International Standards Organization, 1995.

[ISO/IEC 13526:2002] ISO/IEC 13526:2002. Information Technology – Z formal specifications notation – Syntax, type system and semantics. International Standards Organization, 2002.

[ISO/IEC 13817-1:1996] ISO/IEC 13817-1:1996. Information Technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language. International Organization for Standardization, 1996.

[ISO/IEC 13888] ISO/IEC 13888, Information technology – Security techniques – Non-repudiation

[ISO/IEC 14764] ISO/IEC Std. 14764:1999*, Information Technology – Software Maintenance,* International Standards Organization, 1999.

[ISO/IEC 15026] ISO/IEC Std. 15026:1998, *Information Technology - System and Software Integrity Levels*, International Standards Organization, 1998.

[ISO/IEC 15288] ISO/IEC Std. 15288:2002, *E, Systems Engineering – System Lifecycle Processes*, International Standards Organization, 2002.

[ISO/IEC 15408-3] International Standards Organization, International Standard ISO/IEC 15408-3:1999, *Information technology – Security techniques – Evaluation criteria for IT security*, 1999.

[ISO/IEC 15443] ISO/IEC 4th WD 15443-3, IT Security Techniques – A Framework for IT Security Assurance – Part 3: Analysis of Assurance Methods, International Standards Organization, 2004.

[ISO/IEC-15448] ISO/IEC TR 15446:2004, *Information technology - Security techniques - Guide for the production of Protection Profiles and Security Targets*, JTC1/SC27 Technical Report, International Standards Organization, 2004.

[ISO/IEC 15846] ISO/IEC 15846: 1998, Information technology - Software life cycle processes - Configuration Management, May 5, 1998.

[ISO/IEC 17799] ISO/IEC Std. 17799:2000*, Information Technology - Code of Practice for Information Security Management*, International Standards Organization, 2000.

[ISO/IEC 27001] ISO/IEC 27001: 2005, Information Security Management - Specification With Guidance for Use, 2005.

[ISO/IEC 27003] ISO/IEC 27003: 2005, Information Security Management – Implementation of ISO 27001, 2005.

[ISO/IEC 27004] ISO/IEC 27004: 2005, Information Security Management – Information Security Metrics and Measurement, 2005.

[ISO/IEC-15448] ISO/IEC TR 15446:2004, *Information technology - Security techniques - Guide for the production of Protection Profiles and Security Targets*, JTC1/SC27 Technical Report, International Standards Organization, 2004.

[ISO/IEC PRF TR 19791] ISO/IEC PRF TR 19791 Information technology -- Security techniques -- Security assessment for operational systems. International Organization for Standards, February 6, 2006.

[ISO/TR 13569] ISO/TR 13569, Banking and related financial services – Information security guidelines.

[ISO/TR 17944] ISO/TR 17944:2002 (E) Banking – Security and other financial services – Framework for security in financial systems, ISO, 2002.

[ITIL 1999] IT Infrastructure Library – "ITIL v2: 1999 Best Practice in IT Service,*" Management*, 1999.

[Jackson 2005a] Jackson, David, *CESG EAL4 Study: Study Report*, S.P1273.40.1 Issue: 1.4 (Abridged), Praxis Critical Systems, 22 September 2004.

[Jackson 2005b] Jackson, David and David Cooper, "Where Do Software Security Assurance Tools Add Value?", *NIST Workshop on Software Security Assurance Tools, Techniques, and Metrics*, November, 2005.

[Jackson 2006] Jackson, Daniel. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[Jacky 1996] Jacky, Jonathan, *The Way of Z: Practical Programming with Formal Methods*, Cambridge University Press, 1996.

[Jacquith 2002] Jacquith, Andrew, "The Security of Applications: Not All Are Created Equal," At Stake Research, February 2002.
Available at: http://www.atstake.com/research/reports/acrobat/atstake_app_unequal.pdf

[Jalote 1994] Jalote, Pankaj, *Fault Tolerance in Distributed Systems,* Prentice Hall PTR, 1994.

[JDCSISSS 2001] Joint DoDIIS/Cryptologic SCI Information Systems Security Standards. *DIA DoDIIS Information Assurance (IA) Program*, Official Use Only 31 March 2001.

[Jones 2000] Jones, Capers. *Software Assessments, Benchmarks, and Best Practices*, Reading, MA: Addison-Wesley, 2000.

[Jones 2004] Jones, Capers, Software "Project Management Practices: Failures versus Success," *Crosstalk*, pp, 5-9, October 1, 2004.

[Jones 1990] Jones, Cliff. *Systematic Software Development using VDM*. Prentice Hall: 1990.

[Jordan 2001] Jordan, Tim, "Mapping Hacktivism: Mass Virtual Direct Action (MVDA), Individual Virtual Direct Action (IVDA) and Cyberwars", in *Computer Fraud & Security*. Issue 4, 2001.

[Jürjens 2004] Jürjens, Jan, *Secure Systems Development with UML*, Springer-Verlag, 2004.

[Jürjens 2004] Jürjens, Jan, *Secure Systems Development with UML*, Springer-Verlag, 2004.

[Jürjens 2005] Jürjens, Jan, "Sound Methods and Effective Tools for Model-based Security Engineering with UML," *27th International Conference on Software Engineering*, St.Louis, Missouri, USA; 15 - 21 May 2005.

[Kairab 2005] Kairab, S. *A Practical Guide to Security Assessments*, Auerbach Publications, 2005.

[Karger et al, 1990] Karger, Paul A., Mary Ellen Zurko, Douglas W. Benin, Andrew H. Mason, and Clifford E. Kahn. "A VMM Security Kernel for the VAX Architecture," *1990 IEEE Symposium on Security and Privacy*, IEEE, 1990.

[Kaspersky 2003] Kaspersky, Kris. *Hacker Disassembling Uncovered*. A-List Publishing, 2003.

[Kazman 2000] Kazman R., M. Klein, and P. Clements, *ATAM: Method for Architecture Evaluating the Quality Attributes of a Software Architecture*. Technical Report CMU/SEI-200-TR004. Software Engineering Institute, Carnegie Mellon University, 2000.

[Kazman 2002] Kazman R., J. Asundi, and M. Klein, *Making Architecture Design Decisions: An Economic Approach*, SEI-2002-TR-035. Software Engineering Institute, Carnegie Mellon University, 2002.

[Kelly 1994] Kelly, Leslie. *The ASTD Technical and Skills Training Handbook*. McGraw-Hill Professional, 1994.

[Kelly 2003] Kelly, T. P., *Managing Complex Safety Cases*, Department of Computer Science University of York, 2003.
Available at: http://www.cs.york.ac.uk/~tpk/sss03.pdf

[Keeney 2005] Keeney, Michelle, Eileen Kowalski, Dawn Cappelli, Andrew Moore, and Timothy Shimeall. *Insider Threat Study: Computer System Sabotage in Critical Infrastructure Sectors*. CERT Coordination Center/SEI, May 2005.

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

[King 2001] King, C., C. Dalton and E. Osmanoglu, *Security Architecture: Design Deployment and Operations*, McGraw-Hill/Osborne, 2001.

[King et al. 2000] King, Steve, Jonathan Hammond, Rod Chapman, and Andy Pryor, "Is Proof More Cost-Effective Than Testing?", *IEEE Transactions of Software Engineering*, VOL. 26, No. 8, August 2000.

[Kleen 2001] Kleen, Laura J. *Malicious Hackers: A Framework for Analysis and Case Study*. Master's Thesis, AFIT/GOR/ENS/01M-09. Wright-Patterson Air Force Base, OH: Air Force Institute of Technology, March 2001. Available at http://www.iwar.org.uk/iwar/resources/usaf/maxwell/students/2001/afit-gor-ens-01m-09.pdf.

[Knight 2000] Knight, Eric, "Computer vulnerabilities," Technical report, Draft release 4, Security Paradigm, http://www.securityparadigm.com/compvuln_draft.pdf, 2000.

[Kolawa 2005] Kolawa, Adam, "Hold the Line against App Attacks," *Software Test and Performance*, November 2005.

[Kornecki 2005] Kornecki, A., and J. Zalewski**,** "Experimental Evaluation of Software Development Tools for Safety Critical Real-Time Systems" *NASA Journal Innovations in Systems and Software Engineering*, July, 2005.

[Kotonya 2000] Kotonya, G. and I. Sommerville, *Requirements Engineering: Processes and Techniques*, John Wiley & Sons, 2000.

[Kotter 1996] Kotter, John P., *Leading Change*. Harvard Business School Press; 1st edition January 15, 1996.

[Koziol et al. 2004] Koziol, Jack et al. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. Wiley Publishing, Inc., 2004.

[Krone 2005] Krone, Tony, "Hacking Motives", in *High Tech Crime Brief*. Australian High Tech Crime Centre, June 2005. Available at: http://www.aic.gov.au/publications/htcb/htcb006.pdf.

[Krsul 1997] Krsul, Ivan, "Computer Vulnerability Analysis - Thesis Proposal," The COAST Laboratory, Department of Comuter Sciences, Purdue University, Technical Report CSD-TR-97-026, Apr., 1997.

[Krsul 1998] Krsul, Ivan, Spafford, Eugene, and Tripunitara, Mahesh, "An Analysis of Some Software Vulnerabilities," in *Proceedings of the 21st NIST-NCSC National Information Systems Security Conference*, pp. 111–125, 1998.

[Krutz 2004] Krutz, R., R. Vines, *The CISSP Prep Guide: Mastering the CISSP and ISSEP Exams*, Second Edition, John Wiley & Sons, Chap. 5, 6, 10, 2004.

[Krutz and Vines 2003] Krutz, Ronald and Russell Vines. *The CISSP Prep Guide*. John Wiley and Sons, 2003.

[Laitenberger n. d.] Laitenberger, Oliver, *The Perspective-based Inspection Approach*, Fraunhofer Istitut Experimentelles Software Engineering, Kaiserslautern, n.d.
Available at: http://www.tol.oulu.fi/projects/tarjous/perspective.pdf

[Lakhani and Wolf 2005] Lakhani, Karim R. and Robert G Wolf, "Why Hackers Do What They Do: Understanding Motivation and Effort in Free/Open Source Software Projects", in *Perspectives on Free and Open Source Software*. Cambridge, MA: MIT Press, 2005. Available at http://ocw.mit.edu/NR/rdonlyres/Sloan-School-of-Management/15-352Spring-2005/D2C127A9-B712-4ACD-AA82-C57DE2844B8B/0/lakhaniwolf.pdf.

[Landwehr 1993] Landwehr, Carl E., Bull, Alan R., McDermott, John P., and Choi, William S., "A Taxonomy of Computer Program Security Flaws, with Examples," Naval Research Laboratory, Center for Computer High Assurance Systems Information Technology Division, NRL/FR/5542--93-9591, Nov. 1993.

[Landwehr 2001] Landwehr, Carl, "Computer Security," *IJIS* vol. 1, pp. 3-13, 2001.

[Langley 2005] Langley, *Formal Methods*, 2005. Available at: shemesh.larc.nasa.gov/fm/fm-what.html

[Larus 2004] Larus, James R., Ball, Thomas, Das, Manuvir, DeLine, Robert, Fahndrich, Manuel, Pincus, Jon, Rajamani, Sriram K., Venkatapathy, Ramanathan, "Righting Software," *IEEE Software*, May/June 2004.

[Leavens 2005] Leavens, Gary T., Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok, "How the design of JML accommodates both runtime assertion checking and formal verification," *Science of Computer Programming*, Volume 55, pp. 185-205, Elsevier, 2005.

[Le Grand 2005] Le Grand, Charles H. Software Security Assurance: A Framework for Software Vulnerability Management and Audit. Longwood, FL: CHL Global Associates, 2005.

[Lee 1997] Lee, E. "Software Inspections: How to Diagnose Problems and Improve the Odds of Organizational Acceptance", *Crosstalk*, Vol.10 #8 1997.

[Leeson and Coyne 2006] Leeson, Peter T. and Christopher J. Coyne, "The Economics of Computer Hacking", in *Journal of Law, Economics and Policy*, Vol. 1, No. 2, pp. 473-495, 2006. Available at http://www.ccoyne.com/Economics_of_Computer_Hacking.pdf.

[Leon 2004] Leon, Alexis. Software Configuration Management Handbook, Second Edition. Norwood, MA: Artech House Publishers, 2004.

[Leveson 1986] Leveson, N. G. 1986. "Software safety: why, what, and how." *ACM Comput. Surv.* 18, 2 (Jun. 1986), 125-163. http://doi.acm.org/10.1145/7474.7528

[Leveson 1995] Leveson, Nancy G. *Safeware: System Safety and Computers,* Addison-Wesley, 1995.

[Leveson 2004] Leveson, Nancy. "A Systems-Theoretic Approach to Safety in Software-Intensive Systems," *IEEE Transactions on Dependable and Secure Computing* 1, 1 (January-March 2004): 66-86, 2004.

[Lin 2004] Lin, Yuwei. *Hacking Practices and Software Development.* Doctoral Thesis. York, UK: University of York, September 2004. Available at http://opensource.mit.edu/papers/lin2.pdf.

[Linger 1994] Linger, Richard. "Cleanroom Process Model," *IEEE Software*, IEEE Computer Society, March 1994.

[Linger 2004] Linger, Richard, and Stacy Powell, "Developing Secure Software with Cleanroom Software Engineering." Paper prepared for the Cyber Security Summit Task Force Subgroup on Software Process, February 2004.

[Lipner 2005a] Lipner, Steve and Michael Howard, *The Trustworthy Computing Security Development Lifecycle*, Microsoft, 2005.
Available at : http://msdn.microsoft.com/security/default.aspx?pull=/library/en-us/dnsecure/html/sdl.asp#sdl2_topic8?_r=1

[Lipner 2005b] Personal conversation with Samuel Redwine, July 13, 2005.

[Lipson 2002] Lipson, H.F., N.R. Mead, A.P. Moore. "Assessing the Risk of COTS Usage in Survivable Systems," *Cutter IT Journal* 15:5. May 2002.

[Lough 2001] Lough, Daniel L., "A Taxonomy of Computer Attacks With Applications to Wireless Networks," Virginia Polytechnic Institute and State University, 2001.

[Lukatsky 2003] Lukatsky, *Protect Your Information with Intrusion Detectio*, A-LIST Publishing, Chap. 1, 4, 6, 2003.

[Magnusson and Nordström 1994]Magnusson, L. and Nordström, B., "The ALF proof editor and its proof engine" In Types for Proofs and Programs, LNCS Vol 806, Nijmegen, 1994, pp 213-237.

[Manadhata and Wing 2004] Manadhata, P. and J. M. Wing. “Measuring A System's Attack Surface,” CMU-TR-04-102, January 2004.
Available at: http://www-2.cs.cmu.edu/afs/cs/project/calder/www/tr04-102.pdf

[Mann 1999] Mann, D and D. Christey, “Towards a Common Enumeration of Vulnerabilities,” The MITRE Corporation, Bedford MA, 1999.

[Mantel 2002] Mantel, Heiko, “On the Composition of Secure Systems,” *IEEE Symposium on Security and Privacy*, p. 88, 2002.

[Martin 2003] Martin, R. “Integrating your information security vulnerability management capabilities through industry standards (CVE&OVAL.” *Systems, Man and Cybernetics*, IEEE International Conference, Volume 2, 5-8 Oct. 2003 Page(s):1528-1533, 2003.

[Martin 2005] Martin, Robert A., Christey, Steven M., Jarzombek, Joe, “The Case for Common Flaw Enumeration,” *NIST Workshop on Software Security Assurance Tools, Techniques, and Metrics*, Long Beach, CA., Nov., 2005.

[McDermott 1999] McDermott, J. and Fox, C. “Using Abuse Case Models for Security Requirements Analysis.” In *Proceedings of the 15th Annual Computer Security Applications Conference*). IEEE Computer Society, p. 55, 1999.

[McDermott 2001] McDermott, J., “Abuse-Case-Based Assurance Arguments,” *Proc. Annual Computer Security Applications Conference*, December 2001.

[McGraw 2003] McGraw, Gary E., “On the Horizon: The DIMACS Workshop on Software Security”, *IEEE Security and Privacy*, March/April 2003.

[McGraw 2004a] McGraw, Gary, “Software Security,” *IEEE Security and Privacy*, March 2004.

[McGraw 2004b] McGraw, Gary, and Bruce Potter, “Software Security Testing.” *IEEE Security and Privacy*, pp. 81-85, September/October 2004.

[McGraw 2005] McGraw, Gary, “The 7 Touchpoints of Secure Software,” *Software Development*, September 2005.

[McGraw 2006] McGraw, Gary. *Software Security: Building Security In*. Addison Wesley, 2006.

[McGraw and Morrisett] Gary McGraw and Greg Morrisett, “Attacking Malicious Code: A report to the Infosec Research Council.” submitted to *IEEE Software* and presented to the Infosec Research Council.
Available at: http://www.cigital.com/~gem/malcode.pdf

[McLean 1994] McLean, J. “Security Models.” Encyclopedia of Software Engineering (J. Marciniak editor). Wiley 1994.
[Mead 2003] Mead, Nancy R. “Lifecycle Models for High Assurance Systems,” *Proc. of Software Engineering for High Assurance Systems: Synergies between Process, Product, and Profiling (SEHAS 2003)*, Software Engineering Institute, p. 33, 2003.
Available at: http://www.sei.cmu.edu/community/sehas-workshop/

[Mead 2005] Mead, Nancy R., and Ted Stehney. “Security Quality Requirements Engineering (SQUARE) Methodology**”.** *Software Engineering for Secure Systems* (SESS05), 2005.

[Meadows 1996] Meadows, C. The NRL Protocol Analyzer: An overview. Journal of Logic Programming, 26(2):113-131, 1996.

[Meier 2004] Meier, J.D., Alex Mackman, Srinath Vasireddy, Michael Dunner, Ray Escamilla, and Anandha Murukan, *Improving Web Application Security: Threats and Countermeasures*, Microsoft,

2004.
Available at: http://download.microsoft.com/download/d/8/c/d8c02f31-64af-438c-a9f4-e31acb8e3333/Threats_Countermeasures.pdf

[Meier 2005a] Meier, J.D., Alex Mackman, and Blaine Wastell, *Threat Modeling Web Applications*, Microsoft Corporation, May 2005.
Available at: http://msdn.microsoft.com/security/default.aspx?pull=/library/en-us/dnpag2/html/tmwa.asp

[Meier 2005b] Meier, J.D., Alex Mackman, and Blaine Wastell, *Cheat Sheet: Web Application Security Frame*, Microsoft Corporation, May 2005.
Available at: http://msdn.microsoft.com/security/default.aspx?pull=/library/en-us/dnpag2/html/tmwacheatsheet.asp?_r=1

[Meier 2005c] Meier, J.D., Alex Mackman, and Blaine Wastell, *Template Sample: Web Application Threat Model*, Microsoft Corporation, May 2005.
Available at: http://msdn.microsoft.com/security/default. aspx?pull=/library/en-us/dnpag2/html/tmwatemplatesample.asp?_r=1

[Mell et al. 2005] Mell, Peter, Tiffany Bergeron and David Henning, “NIST Special Publication 800-40, Creating a Patch and Vulnerability Management Program, Version 2.0”. NIST, November 2005.

[Meredith 2000] Meredith, J. and S. Mantel, *Project Management – A Managerial Approach*. Fourth Edition. New York, NY: John Wiley & Sons, 2000.

[Merkow 2005] Merkow, Mark S. and Jim Breithaupt, *Computer Security Assurance Using the Common Criteria*, Thompson Delamr Learning, 2005.

[Merkow and Breithaupt 2004] Merkow, Mark S. and Jim Breithaupt, *Computer Security Assurance*, Thomson Delmar Learning, 2004.

[Meunier 2004] Meunier, Pascal, “Secure Programming Educational Material,” Purdue University, 2004. Available at: http://www.cerias.purdue.edu/homes/pmeunier/secprog/

[Meunier 2004] Meunier, Pascal, “Secure Programming Educational Material,” Purdue University, 2004. Available at: http://www.cerias.purdue.edu/homes/pmeunier/secprog/

[Meyer 1991] Meyer, B. “Design by Contract,” Advances in Object-Oriented Software Engineering, D. Mandrioli and B. Meyer. eds. Prentice Hall. Englewood Cliffs, N.J. pp. I-50, 1991.

[Microsoft 2006] Microsoft. *Regulatory Compliance Planning Guide Version: 1.0*. Microsoft, 7/7/2006 http://www.microsoft.com/downloads/details.aspx?FamilyID=BD930882-0D39-4900-9A79-B91F213ED15D&displaylang=en

[Microsoft Security Regulatory Compliance Site]
http://www.microsoft.com/technet/security/learning/compliance/all/default.mspx

[MIL-HDBK-245D] No Author, Department of Defense Handbook for Preparation of Statement of Work, April 3rd, 1996.

[Miller 1990] Miller, B. P., Fredriksen, L., and So, B. “An empirical study of the reliability of UNIX utilities,” Commun. ACM 33, 12 (Dec. 1990), pp. 32-44.

[Mills and Linger 2002] H. Mills and R. Linger, “Cleanroom Software Engineering,” *Encyclopedia of Software Engineering*, 2nd ed., (J. Marciniak, ed.), John Wiley & Sons, New York, 2002.

[Ministry of Defence 2003b] Ministry of Defence. Defence Standard 00-42 Issue 2, *Reliability and Maintainability (R&M) Assurance Guidance Part 3 R&M Case,* 6 June 2003.

[Ministry of Defence 2004a] Ministry of Defence. Interim Defence Standard 00-56, *Safety Management Requirements for Defence Systems Part 1: Requirements*, 17 December 2004.

[Ministry of Defence 2004b] Ministry of Defence. Interim Defence Standard 00-56, *Safety Management Requirements for Defence Systems Part 2: Guidance on Establishing a Means of Complying with Part 1*, 17 December 2004.

[MOD Def Std 00-42, Part 3, 2003] Ministry of Defence Standard 00-42 Issue 2, *Reliability and Maintainability (R&M) Assurance Guidance. Part 3, R&M Case*, 6 June 2003.

[Moffett 2004] Moffett, Jonathan D. Charles B. Haley, and Bashar Nuseibeh, *Core Security Requirements Artefacts,* Security Requirements Group, The Open University, UK, 2004.

[Moffett and Nuseibeh 2003] Moffett, Jonathan D. and Bashar A. Nuseibeh, *A Framework for Security Requirements Engineering*, Report YCS 368, Department of Computer Science, University of York, 2003.

[Moore 1999] Moore, Geoffrey A., *Inside the Tornado : Marketing Strategies from Silicon Valley's Cutting Edge.* Harper Business; Reprint edition July 1, 1999.

[Moore 2002] Moore, Geoffrey A. *Crossing the Chasm*. Harper Business, 2002.

[Moteff 2004] Moteff, John, Computer Security: *A Summary of Selected Federal Laws, Executive Orders, and Presidential Directives* (Order Code RL32357), Congressional Research Services, April 16, 2004.

[Murdock 2005] Murdock, J (editor). *Security Measurement White Paper V.2.0.* – Prepared on behalf of the PSM Safety & Security TWG. (Practical Software and Systems Measurement) York, UK, 12 July 2005.

[NASA 1995] Formal Methods Specification and Verification Guidebook for Software and Computer Systems: Volume 1: Planning and Technology Insertion , July 1995.
Available at http://www.fing.edu.uy/inco/grupos/mf/TPPSF/Bibliografia/fmguide1.pdf

[NASA Guidebook] National Aeronautics and Space Administration (NASA) *Software Assurance Guidebook* (NASA-GB-A201). Available at http://satc.gsfc.nasa.gov/assure/agb.txt.

[Naur 1993] Naur, P. “Understanding Turing's Universal Machine - Personal Style in Program Description,” *The Computer Journal*, Vol 36, Number 4, 1993.

[NCSC 1993] National Computer Security Center. *A Guide to Understanding Covert Channel Analysis of Trusted Systems*, NCSC-TG-030, NCSC, November 1993.
Available at: http://www.radium.ncsc.mil/tpep/process/overview.html

[NCSC-TG-006-88] NCSC-TG-006-88. *A Guide to Understanding Configuration Management in Trusted Systems*. National Computer Security Center, 1988.

[Neumann 1986] Neumann, Peter G. “On Hierarchical Design of Computer Systems for Critical Applications.” *IEEE Transactions on Software Engineering*, Volume 12, Number 9, September pp. 905-920, 1986

[Neumann 1995] Neumann, Peter G, *Architectures and Formal Representations for Secure Systems*, Final Report SRI Project 6401, October 2, 1995.

[Neumann 2000] Neumann, P. G., *Practical architectures for survivable systems and networks*, Technical report, Final Report, Phase Two, Project 1688, SRI International, Menlo Park, California, 2000.

[Neumann 2003] Neumann, P.G. Principled Assuredly Trustworthy Composable Architectures (Draft), Dec., 2003.

[NIPP 2006] Department of Homeland Security. *National Infrastructure Protection Plan*. Department of Homeland Security, 2006. Available at www.dhs.gov/nipp.

[NIST 800-26] National Institute of Standards and Technology (NIST 800-26), *Security Self-Assessment Guide for Information Technology Systems*, November 2001.

[NIST SAMATE 2005] National Institute of Standards and Technology, Software Diagnostics Conformance and Testing Division, Software Diagnostics and Conformance Testing Division, web site: samate.nist.gov, Accessed 8 Aug 2005.

[NIST Special Pub 800-23] National Institute of Standards and Technology (NIST) Special Publication 800-23, *Guidelines to Federal Organizations on Security Assurance and Acquisition/Use of Tested/Evaluated Products*. Gaithersburg, MD: US Department of Commerce, August 2000.

[NIST 800-26] National Institute of Standards and Technology (NIST 800-26), *Security Self-Assessment Guide for Information Technology Systems*, November 2001.

[NIST Special Pub 800-27] Stoneburner, Gary, Clark Hayden, and Alexis Feringa, *Engineering Principles for Information Technology Security (A Baseline for Achieving Security),* NIST Special Publication 800-27 Rev A, June 2004.

[NIST Special Pub 800-30] National Institute of Standards and Technology (NIST) Special Publication 800-30, *Risk Management Guide for Information Technology Systems*. Gaithersburg, MD: US Department of Commerce, July 2002.

[NIST Special Pub 800-33 2001] NIST SP 800-33, *Underlying Technical Models for Information Technology Security*, December 2001.

[NIST Special Pub 800-37] Ross, Ron, Marianne Swanson, Gary Stoneburner, Stu Katzke, and Arnold Johnson, NIST Special Publication 800-37, *Guide for the Security Certification and Accreditation of Federal Information Systems*, May 2004.

[NIST Special Pub 800-40] NIST SP 800-40 *Procedures for Handling Security Patches*, NIST, September 2002.

[NIST Special Pub 800-53] Ross, Ron et al. *Recommended Security Controls for Federal Information Systems,* NIST Special Publication 800-53, Feb. 2005.

[NIST Special Pub 800-55] National Institute of Standards and Technology (NIST) Special Publication 800-55, *Security Metrics Guide for Information Technology Systems*. Gaithersburg, MD: US Department of Commerce, July 2005.

[NIST Special Pub 800-60] Barker, William C. *Guide for Mapping Types of Information and Information Systems to Security Categories*, NIST Special Publication 800-60, June 2004.

[NIST Special Pub 800-64] Grance, Tim, Joan Hash, and Marc Stevens, *Security Considerations in the Information System Development Life Cycle, Revision 1*, National Institute of Standards and Technology, 2004.

[NIST Special Pub 800-67] National Institute of Standards of Technology (NIST) Special Publication 800-67, Rev 1, *Security Considerations in the Information System Development Life Cycle*, June 2004.

[NIST FIPS 200] NIST: Federal Information Processing Standards Publication (FIPS PUB) 200: *Minimum Security Requirements for Federal Information and Information Systems.* March 2006. Available at http://csrc.nist.gov/publications/fips/fips200/FIPS-200-final-march.pdf.

[Northcutt 2003] Northcutt, S, "Computer Security Incident Handling: An Action Plan for Dealing with Intrusions, Cyber-Theft, and Other Security-Related Events", SANS Institute, 2003.

[NRC 1999] Committee on Information Systems Trustworthiness, *Trust in Cyberspace*, Computer Science and Telecommunications Board, National Research Council, 1999.

[NRC 2001] National Research Council (NRC) Computer Science and Telecommunications Board (CSTB). *Cybersecurity Today and Tomorrow: Pay Now or Pay Later.* National Academies Press, 2002. Available at http://darwin.nap.edu/books/0309083125/html.

[NRL Handbook 1995] Naval Research Laboratory, *Handbook for the Computer Security Certification of Trusted Systems*, US Naval Research Laboratory, 1995

[NSA 1990] National Security Agency, NSA/CSS Manual 130-1, *Operational Computer Security*, October 1990.

[NSA 2002] National Security Agency, The Information Systems Security Engineering Process (IATF) v3.1. 2002.

[NSA 2004] Information Assurance Directorate, National Security Agency, *U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness*, Version 0.621. National Security Agency, 1 July 2004.

[NSTISSAM INFOSEC/2-00] National Security Telecommunications and Information Systems Security Advisory Memorandum (NSTISSAM)/2-00, *Advisory Memorandum on the Strategy for Using the National Information Assurance Partnership (NIAP) for the Evaluation of Commercial Off-The-Shelf (COTS) Security Enabled Information Technology Products*. Fort Mead, MD: US National Security Agency, 8 February 2000.

[NSTISSP No. 11] National Security Telecommunications and Information Systems Security Policy (NSTISSP) No. 11, *National Policy Governing the Acquisition of Information Assurance (IA) and IA-Enabled Information Technology Products*, Fort Mead, MD: US National Security Agency, July 2003.

[OCEG 2006] Open Compliance & Ethics Group. Foundation Guidelines v1. Open Compliance & Ethics Group, 7/3/2006 http://www.oceg.org/ItemView.aspx?Id=13892

[OCL 2.0] Object Management Group, *UML 2.0 OCL Specification*. Object Management Group, 2005. Available at http://www.omg.org/docs/ptc/03-10-14.pdf

[OIS, 2004] Guidelines for Security Vulnerability Reporting and Response, Organization for Internet Safety, Version 1.5, 19 April 2004.

[OMB 1999] Office of Management and Budget, *Evaluating Information Technology Investments*, 1999. Available at: at http://www.itmweb.com

[OMB A11, Part 7] Office of Management and Budget, Circular No.A-11, Part 7, *Planning, Budgeting, Acquisition, and Management of Capital Assets*. Washington, DC: OMB, June 2005.

[Open Group 2004] Open Group, Security Design Patterns (SDP) Technical Guide v.1, April 2004.

[OWASP 2005] OWASP. *A Guide to Building Secure Web Applications and Web Services 2.0.* The Open Web Application Security Project, Black Hat Edition, July 27, 2005.

[OWASP 2006] Open Web Application Security Project. “Software Quality Assurance” chapter in A Guide to Building Secure Web Applications and Web Services, 2.1 (Draft 3). OWASP Foundation, February 2006.
http://www.owasp.org/index.php/Software_Quality_Assurance

[Owre et al 1999] Owre, S., N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

[Parnas 1985] Parnas, D. L. and D. M.Weiss, “Active design reviews: principles and practices,” In *Proceedings of the 8th international Conference on Software Engineering* (London, England, August 28 - 30, 1985), International Conference on Software Engineering, IEEE Computer Society Press, p. 132-136. 1985.

[Patterson and Conner 1983] Patterson, Robert W. and Darryl R. Conner, “Building Commitment to Organizational Change,” *Training and Development Journal*, pp. 18-30, April 1983.

[Payne 2004] Payne, Jeffery E., “Regulation and Information Security: Can Y2K Lessons Help Us?”, *IEEE Security and Privacy*, March/April 2004.

[Peltier 2003] Peltier, T., J. Peltier, and J. Blackley, *Managing a Network Vulnerability Assessment*, Auerbach Publications, 2003.

[Peterson 2006] Pederson, Allan, Navi Partner, and Anders Hedegaard. “Designing a Secure Point-of-Sale System”, *Proceedings of the Fourth IEEE International Workshop on Information Assurance (IWIA ’06).* pp 51-65, April 2006.

[Pfleeger 1997a] Pfleeger, Charles, *Security in Computing*, Prentice Hall PTR, 1997.

[Pfleeger 1997b] Pfleeger, S. and L. Hatton, "Do formal methods really work", *IEEE Computer*, Jan 1997.

[Pfleeger 1997c] Pfleeger, Shari Lawrence, and Les Hatton, "Investigating the Influence of Formal Method", *IEEE Computer*, vol. 30, no. 2, Feb 1997.

[Pfleeger 2003] Pfleeger, Charles P. and Shari Lawrence Pfleeger. *Security in Computing* Third Edition. Prentice Hall PTR, 2003.

[Piessens 2002] Piessens, Frank, "A taxonomy of causes of software vulnerabilities in internet software," *Supplementary Proceedings of the 13th International Symposium on Software Reliability Engineering* (Vouk, M., ed.), pp. 47-52, 2002.

[PLOVER 2005] “The Preliminary List Of Vulnerability Examples for Researchers (PLOVER),” MITRE Corporation, 2005 Available at http://cve.mitre.org/docs/plover/.

[PMBOK 2004] Bolles, D., and Fahrenkrog, S. *A Guide to the Project Management Body of Knowledge (PMBOK*— ANSI/PMI 99-001-2004. Third Edition. Newton Square, PA.: Project Management Institute, Inc. 2004

[Pomeroy-Huff 2005] Marsha Pomeroy-Huff, Julia Mullaney, Robert Cannon, and Mark Sebern. *The Personal Software Process (PSP) Body of Knowledge*, Version 1.0. Special Report CMU/SEI-2005-SR-003, Software Engineering Institute, 2005. Available at http://www.sei.cmu.edu/publications/documents/05.reports/05sr003.html

[Pooya 2005] Pooya Jaferian et. al., "RUPSec: Extending Business Modeling and Requirements Disciplines of RUP for Developing Secure Systems," 31st EUROMICRO Conference on Software Engineering and Advanced Applications, August 2005, pp. 232-239.

[Powell 1999] Powell, S., C. Trammell, R. Linger, and J. Poore, *Cleanroom Software Engineering: Technology and Process*, Addison Wesley, Reading, MA, 1999.

[Prasad 1998] Prasad, D., *Dependable Systems Integration using Measurement Theory and Decision Analysis*, PhD Thesis, Department of Computer Science, University of York, UK, 1998.

[Praxis 2004] Praxis Critical Systems Ltd, *EAL4 Common Criteria Evaluations Study*, September 2004. Available at: http://www.cesg.gov.uk/site/iacs/itsec/media/techniques_tools/eval4_study.pdf

[Prieto -Diaz 2003] Prieto -Diaz, Ruben, *Common Criteria Evaluation Process*, Commonwealth Information Security Center Technical Report CISC-TR-2002-003, 2003.
Available at: http://www.jmu.edu/cisc/research/publications/CCevaluationProcesTR03-5.pdf

[Princeton University] Princeton University Secure Internet Programming: Proof-Carrying Code. Available at http://www.cs.princeton.edu/sip/projects/pcc/.

[Pullum 2001] Pullum, L. L. *Software Fault Tolerance*, Artech House, 2001.

[Purser 2004] Purser, S., *A Practical Guide to Managing Information Security*, Artech House, Chap. 1, 8, 2004.

[Radack 2005] Radack, Shirley, editor, *Standards for Security Categorization of Federal Information and Information Systems,* Federal Information Processing Standard (FIPS) 199, July 10, 2005.

[Ramachandran 2002] Ramachandran, J., *Designing Security Architecture Solutions*, New York: John Wiley & Sons, 2002.

[Ramfos 2004] Ramfos, Antonis, and Ross Velentzas, "An eye opener on open source Internet security," *IST Features*, 22 July 2004.
Available at:
http://istresults.cordis.lu/index.cfm/section/news/Tpl/article/BrowsingType/Long%20Feature/ID/6916
6

[Rasmussen 2004] Rasmussen, Michael and Natalie Lambert, *Security Assurance Needed In Software Development Contracts*, Research Report, Forrester, May 24, 2004.
Available at: http://www.forrester.com/ER/Research/List/Analyst/Personal/0,2237,830,00.html

[Rattray 2001] Rattray, Gregory J., "The Cyberterrorism Threat", Chapter 5 in Smith, James M. and William C. Thomas (editors), The Terrorism Threat and U.S. Government Response: Operational and Organizational Factors. U.S. Air Force Academy, Colorado: March 2001. Available at http://www.usafa.af.mil/df/inss/Ch%205.pdf.

[Rechtin 2000] Rechtin, E. *Systems Architecting of Organizations: Why Eagles Can't Swim*. Boca Raton, FL: CRC Press, 2000.

[Redmill 2005] Redmill, Felix, "Theory and practice of risk-based testing," *Software Testing, Verification and Reliability*, Volume 15, Issue 1, p. 3-20, March 2005.

[Redwine 2004] Redwine, Samuel T., Jr., and Noopur Davis (Editors). *Processes for Producing Secure Software: Towards Secure Software*. vols. I and II. Washington, D.C.: National Cyber Security Partnership, 2004.
Available at http://www.cigital.com/papers/download/secure_software_process.pdf

[Redwine 2005a] Redwine, Samuel T., Jr., *Dependability Properties: Enumeration of Primitives and Combinations with Observations on their Measurement.* Commonwealth Information Security Center, Technical Report CISC-TR-2004-001, June 2005.

[Redwine 2005b] Samuel T. Redwine, Jr., *Principles for Secure Software: A Compilation of Lists,* Commonwealth Information Security Center, Technical Report CISC-TR-2005-002, 2005.

[Redwine 2006] Redwine, Samuel T. (editor), *Software Assurance: A Guide to the Common Body of Knowledge*, US Department of Homeland Security, 2006 (available on *Build Security In* website)

[Reed 2004] Reed, Thomas C., *At the Abyss: An Insider's History of the Cold War*, Presidio Press, 2004.

[Rescorla 2001] Rescorla, Eric, *SSL and TLS: Designing and Building Secure Systems*, Addison-Wesley, 2001.

[Riggs 2003] Riggs, S., *Network Perimeter Security: Building Defense In-Depth*, Auerbach Publications, 2003.

[Riguidel 2004] Riguidel, Michel, Gwendal Le Grand, Cyril Chiollaz, Sved Naqvi, Mikael Formanek, "D1.2 Assessment of Threats and Vulnerabilities in Networks", Version 1.0. European Union Security Expert Initiative (SEINIT), 31 August 2004. Available at http://www.seinit.org/documents/Deliverables/SEINIT_D1.2_PU.pdf

[Robertson 1999] Robertson S. and J. Robertson, *Mastering the Requirements Process*, Addison-Wesley, 1999.

[Rogers 1995] Rogers, Everett, *Diffusion of Innovations*, Free Press, 1995.

[Rowe 2004a] Rowe, Neil C., "Designing Good Deceptions in Defense of Information Systems," ACSAC, 2004.
Available at: http://www.acsac.org/2004/abstracts/36.html

[Rowe 2004b] Rowe, N., and H. Rothstein, "Two Taxonomies of Deception for Attacks on Information Systems," *Journal of Information Warfare*, Vol. 3, No. 2, pp. 27-39, July 2004.

[Ryan 2002] Ryan, Julie J. C. H. and Daniel J. Ryan, *Institutional and Professional Liability in Information Assurance Education*. 2002. Available at http://www.danjryan.com/Institutional and Professional Liability in Information Assurance Education.doc

[S4EC] System Security, Survivability, and Safety Engineering Criteria (S4EC) Project, www.s4ec.org

[Safe Harbor 2000] U.S. Department of Commerce, *Safe Harbor Privacy Principles*, U.S. Department of Commerce, July 21, 2000.
Available at: http://www.export.gov/safeharbor/SHPRINCIPLESFINAL.htm

[Safire 2004] Safire, William, *The Farewell Dossier*, New York Times, February 2, 2004.

[SafSec Guidance] "SafSec Methodology: Guidance Material", *SafSec: Integration of Safety and Security*. Available at: http://www.safsec.com/safsec_files/resources/50_3_SafSec_Method_ Guidance_Material_3.0.pdf.

[SafSec Introduction] "Why SafSec?", *SafSec: Integration of Safety and Security*.
Available at: http://www.safsec.com/safsec_files/resources/50.6_Why_SafSec.pdf

[SafSec Standard] "SafSec Methodology: Standard." *SafSec: Integration of Safety and Security*. Available at: http://www.safsec.com/safsec_files/resources/50_2_SafSec_Method_Standard_3.0.pdf

[Saitta 2005] Saitta, Paul, Brenda Larcom and Michael Eddington, "Trike v.1 Methodology Document," [Draft], 20 June 2005.
Available at: http://www.hhhh.org/trike/papers/Trike_v1_Methodology_Document-draft.pdf

[Saltzer and Schroeder 1975] Saltzer, J. H. and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278-1308, 1975.
Available online at http://cap-lore.com/CapTheory/ProtInf/

[SAMATE 2005] "The Software Assurance Metrics and Tool Evaluation (SAMATE) project," National Institute of Science and Technology (NIST), (http://samate.nist.gov).

[SCC Part 1 2002] Sherwood, J. E., *The Security Certification Criteria Project*, Information Assurance Engineering Division SPAWAR System Center Charleston, 1August 2002.
Available at: http://www.s4ec.org/scc_overview.pdf

[SCC Part 2 2002] Information Assurance Engineering Division SPAWAR System Center Charleston, Security Certification Criteria for Information Assurance Enabled Systems and Infrastructures Part 2:

Functional Certification Criteria, Version 0.2, Information Assurance Engineering Division SPAWAR System Center Charleston, 1 August 2002.
Available at: http://www.s4ec.org/scc_part2_ver0-21.pdf

[Schell 2005] Schell, Roger. “Creating High Assurance for a Product: Lessons Learned from GEMSOS.” (Keynote Talk) *Third IEEE International Workshop on Information Assurance*, College Park, MD, USA March 23-24, 2005. Available at http://www.iwia.org/2005/Schell2005.PDF

[Schlesinger 2004] Schlesinger, Rich (ed.). *Proceedings of the 1st annual conference on Information security curriculum development*. Kennesaw, Georgia, ACM, October 08 - 08, 2004

[Schneier 1999] Schneier, Bruce, “Attack Trees: Modeling security threats,” *Dr. Dobb's Journal*, December 1999.

[Schneier 2000] Schneier, Bruce, *Secrets and Lies: Digital Security in a Networked World*, John Wiley & Sons, 2000.

[Schoonover 2005] Schoonover, Glenn, Presentation, *Software Assurance Summit*, National Defense Industries Association, September 7-8, 2005.

[Schumacher 2006] Schumacher, Markus, Eduardo Fernandez-Buglioni, Duane Hybertson, Frank Buschmann, Peter Sommerlad. *Security Patterns: Integrating Security and Systems Engineering***.** John Wiley & Sons, 2006. (forthcoming)

[Schwalbe 2006] Schwalbe, K., *Information Technology Project Management*. Fourth Edition. Boston, MA: Thompson Course Technology, 2006.

[SDI 1992] Department of Defense Strategic Defense Initiative Organization. *Trusted Software Development Methodology*, SDI-S-SD-91-000007, vol. 1, 17 June 1992.

[Seacord 2005] Seacord, R., *Secure Coding in C and C++*, Boston, MA: Addison Wesley Professional, 2005.

[Seacord and Householder 2005] Seacord, R. and A. Householder, *A Structured Approach to Classifying Security Vulnerabilities*, Technical Note: CMU/SEI-2005-TN-003, Carnegie Mellon University: Software Engineering Institute, January 2005.

[SEI 1990] "Configuration Management: State of the Art", *SEI Bridge*, Software Engineering Institute, Carnegie-Mellon University, March 1990.

[SEI TT] SEI, *Technology Transition Practices*.
Available at: http://www.sei.cmu.edu/ttp/value-networks.html

[Seminal Papers] Seminal Papers - *History of Computer Security Project*, University of California Davis Computer Security Laboratory
Available at: http://seclab.cs.ucdavis.edu/projects/history/seminal.html

[Senge 1994] Senge, Peter M., “The Fifth Discipline,*”* *Currency*, 1st edition, October 1, 1994.

[Sheyner 2002] Sheyner, Oleg, Somesh Jha, and Jeannette M. Wing, “Automated Generation and Analysis of Attack Graphs,” *Proceedings of the IEEE Symposium on Security and Privacy*, 2002.

[SHS 2002] Secure Hash Standard (SHS), FIPS 180-2, August 2002.

[Sinclair 2005] Sinclair, David, “Introduction to Formal Methods”, Course Notes, 2005.
Available at: www.computing.dcu.ie/~davids/courses/CA548

[Sindre 2000] Sindre, G. and A.L. Opdahl, "Eliciting Security Requirements by Misuse Cases," *Proc. 37th Int'l Conf. Technology of Object-Oriented Languages and Systems (TOOLS-37 '00)*, IEEE Press, pp. 120-131, 2000.

[Skoudis 2002] Skoudis, Ed, Counter Hack: A Step-by-step Guide to Computer Attacks and Effective Defenses, Prentice Hall, 2002.

[Smedinghoff 2003] Smedinghoff, Thomas J., "The Developing U.S. Legal Standard for Cybersecurity," Baker and McKenzie, Chicago, May 2003.
Available at: www.bakernet.com/ecommerce/us%20cybersecurity%20standards.pdf

[Smedinghoff 2004] Smedinghoff, Thomas J. "Trends in the Law of Information Security," *World Data Protection Report*, The Bureau of National Affairs, Inc. vol. 4, no. 2, August 2004.

[Snow 2005] Snow, Brian. "We need Assurance!" *Proceedings of the 21st Annual Computer Security Applications Conference* (ACSAC '05). pp 3-10, December 2005.

[Sobel 2000] Sobel, Ann E. Kelley. "Empirical Results of a Software Engineering Curriculum Incorporating Formal Methods," *ACM Inroads*, March 2000

[Software Tech News 2005] "Secure Software Engineering", *DoD Software Tech News*, Data Analysis Center for Software, July 2005.
Available at: http://www.softwaretechnews.com

[Sommerville 1997] Sommerville, Ian, and P. Sawyer, *Requirements Engineering: A Good Practice Guide*, John Wiley & Sons, 1997.

[Sommerville 2004] Sommerville, I., *Software Engineering*, 7th ed., Pearson Education, 2004.

[Sommerville 2006] Sommerville, Ian, *Software Engineering*, 8th ed., Pearson Education, 2006.

[SOUPS 2005] *Symposium on Usable Privacy and Security (SOUPS)*, ACM, July 6-8, 2005 Available at: http://cups.cs.cmu.edu/soups/2005/program.html

[Spivey 1992] Spivey, J.M., *The Z Notation: A Reference Manual*, 2nd Edition, Prentice-Hall, 1992.

[SREIS 2005] *Symposium on Requirements Engineering for Information Security* (SREIS 2005) Paris, France, August 29, 2005. See http://www.sreis.org/

[Srivatanakul 2003] Srivatanakul, Thitima, John A. Clark, Susan Stepney, Fiona Polack. "Challenging Formal Specifications by Mutation: a CSP security example," *apsec*, , *10th Asia-Pacific Software Engineering Conference* (APSEC'03), 2003, p. 340.

[SSE-CMM 3.0] *Systems Security Engineering Capability Maturity Model (SSE-CMM) version 3.0*, International Systems Security Engineering Association (ISSEA), 2004.
Available at: http://www.sse-cmm.org/model/model.asp

[Stavridou 2001] Stavridou, Victoria, Bruno Dutertre, R. A. Riemenschneider, and Hassen Sa¨ıdi, *Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEXII'01)*, 2001.

[Stoneburner 2001] NIST SP 800-33, Underlying Technical Models for Information Technology Security, December 2001.

[Stroud 2004] Stroud, R., I. Welch, J. Warne, and P. Ryan, "A qualitative analysis of the intrusion-tolerance capabilities of the MAFTIA architecture," *International Conference on Dependable Systems and Networks*, IEEE, 2004.

[Sturm, Morris, Jander, 2000] Sturm, R., W. Morris, and M. Jander, *Foundations of Service Level Management*, Indianapolis, IN: Sams, 2000.

[Swanson ND] Swanson, Marianne, Security Self Assessment Guide for Information Technology Systems, ND.

[SWEBOK] Abran, Alain, James W. Moore (Executive editors); Pierre Bourque, Robert Dupuis, Leonard Tripp (Editors). *Guide to the Software Engineering Body of Knowledge*. 2004 Edition. Los Alamitos, California: IEEE Computer Society, Feb. 16, 2004. Available at http://www.swebok.org

[Swiderski 2004] Swiderski, F. and W. Snyder, *Threat Modeling*, Microsoft Press, 2004.

[Szor 2005] Szor, Peter, *The Art of Computer Virus Research and Defense*, Addison-Wesley Professional, 2005.

[Thayer and Yourdon 2000] Thayer, Richard H., editor, and Edward Yourdon. Software Engineering Project Management, Second Edition (Paperback) Hoboken, NJ: Wiley-IEEE Computer Society Press, 2000.

[Thomas 2002] Thomas, Douglas. *Hacker Culture.* Minneapolis, MO: University of Minnesota Press, 2002.

[Thompson 1984] Thompson, Ken, “Reflections on Trusting Trust”, *Communication of the ACM*, Vol. 27, No. 8, pp. 761-763, August 1984.

[Thompson 2005] Thompson, H. H. and S. G. Chase, *The Software Vulnerability Guide*, Charles River Media, 2005.

[Thuraisingham 2005] Thuraisingham, Bhavani, Database and Applications Security: Integrating Information Security and Data Management, CRC Press, 2005.

[Thornburgh 2005] Thornburgh, Nathan. “The Invasion of the Chinese Cyberspies (And the Man Who Tried to Stop Them) -
An exclusive look at how the hackers called TITAN RAIN are stealing U.S. secrets”, Time, 9/5/2005]

[Tsipenyuk 2005] Tsipenyuk, Katrina, Brian Chess, and Gary McGraw. Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors. IEEE Security & Privacy. Vol. 4, No.2, pp 81-84, November/December 2005.

[Turner 2002] Turner, R.G., *Implementation of Best Practices in U.S. Department of Defense Software-Intensive System Acquisitions*, Ph.D. Dissertation, George Washington University, 31 January 2002. Available at http://www.goldpractices.com/survey/turner/index.php.

[University of California-Berkeley] University of California-Berkeley: Proof Carrying Code. Available at http://raw.cs.berkeley.edu/pcc.html.

[US Army 2003] US Army, Field Manual (FM) 3-13: *Information Operations: Doctrine, Tactics, Techniques, and Procedures*, 28th Nov., 2003. (Particularly Chapter 4 on Deception)

[DOS 5220.22-M Matrix 1995] US Department of Defense, “Cleaning and Sanitization Matrix,” Section 8, *DOS 5220.22-M*, Washington, D.C., 1995; www.dss.mil/isec/nispom_0195.htm.

[US DoD 1996] Joint Chiefs of Staff, DoD JP 3-58, *Joint Doctrine for Military Deception*, 31 May 1996.

[Van Grembergen 2004] Van Grembergen, W., *Strategies for Information Technology Governance*, Idea Group Publishing, Chap.11, 2004.

[Vaughn 2003] Vaughn, Steven J., “Building Better Software with Better Tools”, *IEEE Computer*, Vol 36, No 9, September 2003.

[Vaughn and George 2003] Vaughn, Rayford and Vinu George, “Application of Lightweight Formal Methods in Requirement Engineering,” *Crosstalk: The Journal of Defense Software Engineering*, January 2003.

[Verdon 2006] Verdon, Denis, "Security Policies and the Software Developer," *IEEE Security and Privacy*, vol. 4, no. 4, pp. 42-49, Jul/Aug, 2006.

[Vernon 2006] Richard C. Vernon, Cynthia E. Irvine and Timothy E. Levin. “Toward a Boot Odometer,” *7th Annual IEEE Information Assurance Workshop*. West Point, New York, June 21 June 21-23, 2006

[Verton 2005] Verton, Dan. *The Insider: A True Story*. Llumina Press, 2005.
[Viega 2000] Viega, John, et al, “Statically Scanning Java Code: Finding Security Vulnerabilities,” *IEEE Software*, vol. 17, no. 5, pp. 68-74, Sept/Oct 2000.

[Viega 2005] Viega, J., *The CLASP Application Security Process*, Secure Software, 2005. Available at http://www.securesoftware.com

[Viega and McGraw 2001] Viega, John, and Gary McGraw, *Building Secure Software: How to Avoid Security Problems the Right Way*, Reading, MA: Addison Wesley, 2001.

[Viega and McGraw 2002] Viega, John and Gary McGraw, *Building Secure Software*, Addison-Wesley, 2002.

[Viega and Messier 2003] Viega, John and Matt Messier, *Secure Programming Cookbook*, O’Reilly and Associates, Inc., 2003.

[Violino 1997] Violino R, “Measuring Value: Return on Investment”, *Information Week*, No. 637, pp. 36-44, June 30, 1997.

[Vizedom 1976] Vizedom, Monika, *Rites and Relationships: Rites of Passage and Contemporary Anthropology*, Beverly Hills, CA: Sage Publications, 1976 [Abrams 1998] Abrams, M. D, “Security Engineering in an Evolutionary Acquisition Environment,” *New Security Paradigms Workshop*, 1998.

[Walsh 2003] Walsh, L., "Trustworthy Yet?", *Information Security Magazine*, Feb. 2003.
Available at: http://infosecuritymag.techtarget.com/2003/feb/cover.shtml

[Warkentin and Vaughn 2006] Warkentin, Merrill and Rayford Vaughn. *Enterprise Information Systems Assurance and System Security: Managerial and Technical Issues*. Idea Group Publishing, 2006.

[Weber 2004] Weber, Sam, Karger, Paul A., Paradkar, Amit, “A Software Flaw Taxonomy: Aiming Tools at Security,” ACM Software Engineering for Secure Systems - Building Trustworthy Applications (SESS'05) St. Louis, Missouri, USA., June 2004.

[Weinberg] The Virginia Satir change model, adapted from G. Weinberg, *Quality Software Management, Vol. 4: Anticipating Change*, Ch 3.

[Weiss ND] Weiss, Gus W. *The Farewell Dossier – Duping the Soviets*
Available at: http://www.cia.gov/csi/studies/96unclass/farewell.htm.

[Wheeler 2003] Wheeler, David, *Secure Programming for Linux and Unix HOWTO* v3.010, 3 March, 2003.
Available at: www.dwheeler.com/secure-programs/Secure-Programs-HOWTO.pdf.

[Wheeler 2005] Wheeler, David A., “Software Configuration Management (SCM) Security”. May 2005. http://www.dwheeler.com/essays/scm-security.html

[Whitgift 1991] Whitgift, D., *Methods and Tools for Software Configuration Management*, John Wiley and Sons, England, 1991.

[Whitman and Mattord 2005] Whitman, Michael and Herbert Mattord, *Principles of Information Security*, 2nd Edition, Thomson Course Technology, 2005.

[Whittaker and Thompson 2003] Whittaker, James, and Herbert Thompson. *How to Break Software Security.* Addison-Wesley, 2003.

[Whittaker and Thompson 2004] Whittaker, J. A. and H. H. Thompson. *How to Break Software Security: Effective Techniques for Security Testing.* Pearson Education, 2004.

[Whitten 1999] Whitten, A. and J.D. Tygar, "Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0," *Proc. Ninth USENIX Security Symposium*, 1999.

[Williams 1982] Williams, T. W., and K. P. Parker, "Design for Testability - A Survey," *IEEE Trans. Computers*, Vol. C-31, No. 1, pp. 2-15, January 1982.

[Williams 1998] Williams, Jeffrey R. and George F. Jelen, *A Framework for Reasoning about Assurance*, Document Number ATR 97043, Arca Systems, Inc., 23 April 1998.

[Williams and Fagan 1992] Williams, S. and P. Fagan, "Secure Software: Management Control Is a Key Issue". London, UK: Institute o Electrical Engineers (IEE) Colloquium on Designing Secure Systems, June 1992.

[Wimmel 2002] Wimmel, Guido, Jan Jürjens, Specification-Based Test Generation for Security-Critical Systems Using Mutations, Proceedings of the 4th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering, p.471-482, October 21-25, 2002.

[Wong 2001] Wong, Kelvin, "Friend Or Foe: Understanding The Underground Culture and the Hacking Consciousness". Melbourne, Australia: RMIT University, 20 March 2001. Available at http://www.security.iia.net.au/downloads/friend_or_foe.pdf and http://www.security.iia.net.au/downloads/new-lec.pdf.

[Woodcock and Davies 1996] Woodcock, Jim and Jim Davies. *Using Z: Specifications, Refinement, and Proof*. Prentice Hall. 1996.

[Wyk and McGraw 2005] van Wyk, Kenneth and Gary McGraw, *After the Launch: Security for App Deployment and Operations*, Presentation at Software Security Summit, April 2005.

[Wyk and Graff 2003] Van Wyk, Kenneth R. and Mark G. Graff: *Secure Coding: Principles and Practices. Sebastopol, CA:* O'Reilly Media Inc., 2003.

[YCC 2003] Your Computer Center, "Viruses and Software Vulnerabilities," 2003. Available at www.ycc.com/security/details/virus.htm

[Yee 2002] Yee, Ka-Ping , "User interaction design for secure systems," In *Proceedings of the 4th International Conference on Information and Communications Security*, Springer-Verlag, LNCS 2513, 2002.

[Yee 2003] Yee, Ka-Pin, "Secure interaction design and the principle of least authority," *Workshop on Human-Computer Interaction and Security Systems*, part of CHI2003, ACM SIGCHI, 2003. Available at: http://sims.berkeley.edu/˜ping/sid/yee-sidchi2003-workshop.pdf

[Yee 2004] Yee, Ka-Pin, "Aligning security and usability," *Security & Privacy Magazine*, 2: 48–55, Sept/Oct 2004

[Yee 2005] Yee, Ka-Pin, "Guidelines and strategies for secure interaction design," Lorrie Cranor and Simson Garfinkel, editors, *Security and Usability*. O'Reilly, 2005.

[Younan 2003] Younan, Yves, “An Overview of Common Programming Security Vulnerabilities and Possible Solutions,” Vrije Universiteit Brussel Faculteit Wetenschappen Departement Informatica en Toegepaste Informatica, Aug., 2003.

[Younan 2004] Younan, Yves, Joosen, Wouter, and Piessens, Frank, “Code Injection in C and C++ - A Survey of Vulnerabilities and Countermeasures,” Katholieke Universiteit Leuven Department of Computer Science, Jul., 2004.

[Zimmerman 1997] Zimmerman, Michael, “Configuration Management, Just a Fashion or a Profession.” White Paper, usb GmbH, 1997.

[Zitser 2005] Zitser, Misha, Leek, Tim, Lippmann, Richard, “Testing Static Analysis Tools Using Exploitable Buffer Overflows From Open Source Code,” Foundations of Software Engineering, Newport Beach, CA, December, 2005.

[Zwicky et al, 2000] Zwicky, Elizabeth D., Simon Cooper, and D. Brent Chapman. *Building Internet Firewalls* (2nd ed.), O'Reilly, 2000.

17Index

Software Assurance

A Guide to the Common Body of Knowledge to Produce, Acquire, and Sustain Secure Software

Workforce Education and Training Working Group

DHS/DoD Software Assurance

Edited by Samuel T. Redwine, Jr.