Department of Homeland Security

SECURITY IN THE SOFTWARE LIFECYCLE

Making Software Development Processes—
and Software Produced by Them—More Secure

DRAFT Version 1.2 - August 2006

FOREWARD

Dependence on information technology makes software assurance a key element of business continuity, 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. The integrity of key assets depends upon the reliability and security of the software that enables and controls those assets. 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 software with exploitable weaknesses, security-enhanced processes and practices—and the skilled people to perform them—are required to build software that can be trusted not to increase risk exposure.

In an era riddled with asymmetric cyber attacks, claims about system reliability, integrity and safety must also include provisions for built-in security of the enabling 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: (1) system interdependence and software dependence has software as the weakest link; (2) software size and complexity obscures intent and precludes exhaustive test; (3) outsourcing and use of unvetted software supply chain increases risk exposure; (4) attack sophistication eases exploitation; and (5) reuse of legacy software interfaced with other applications in new environments introduces other unintended consequences increasing the number of vulnerable targets. The growing extent of the resulting risk exposure is rarely understood.

The number of threats specifically targeting software is increasing, and the majority of network- and system-level attacks exploit vulnerabilities in application level software. 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 justifiable confidence that the software that 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, process, technology, and acquisition throughout the software life cycle. Collaborative efforts seek to shift the paradigm away from patch management and to achieve 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 more resilient operations.

In their Report to the President entitled *Cyber Security: A Crisis of Prioritization* (February 2005), the President's Information Technology Advisory Committee (PITAC) summed up the problem of non-secure software:

> *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:

1. **Trustworthiness:** No exploitable vulnerabilities exist, either maliciously or unintentionally inserted;

2. **Predictable Execution:** Justifiable confidence that software, when executed, functions as intended;

3. **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 at https://buildsecurityin.us-cert.gov and the *Software Assurance Common Body of Knowledge* (CBK) 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 developers' guide, entitled *Security in the Software Life Cycle: Making Software Development Processes—and Software Produced by Them—More Secure*, is also being published as part of the DHS Software Assurance program, and is freely downloadable from the DHS BuildSecurityIn portal.

The main goal of *Security in the Software Life Cycle* is to arm developers, project managers, and testers with the information they need to start improving the security of the practices and processes they use to produce software. A growing number of practitioners in the software engineering and security communities have identified tools and practices that they have found in "real world" to improve the likelihood of producing resulting software with fewer faults and other weaknesses that can be exploited as vulnerabilities. In addition, while it is not always their explicit objective, the practices and technologies described in *Security in the Software Life Cycle* should also help increase software's overall dependability and quality.

Unlike other works published on secure software engineering, secure programming, secure coding, application security, and similar topics, *Security in the Software Life Cycle* does not set out to recommend

a specific approach to the software security problem. Instead, it offers alternative approaches for addressing specific security issues. Where it does resemble such works is in the more detailed discussion of software security principles and practices in Appendix G. However, the scope of the information provided in Appendix G is broader than that found in many other published works with similar content. Also unlike other such works, the other sections of *Security in the Software Life Cycle* discuss a number of life cycle process models, development methodologies, and supporting tools that have been shown in "real world" software development projects, across government, industry, and academia in the United States (U.S.) and abroad, to reduce the number of exploitable software faults that can be targeted as vulnerabilities to compromise the software, the data it processes, or the computing and networking resources on which the software depends.

No single practice, process, or methodology offers a universal "silver bullet" for software security. With this in mind, this document has been compiled as a reference intended to present an overview of software security and inform software practitioners about a number of practices and methodologies which they can evaluate and selectively adopt to help reshape their own development processes to increase the security and dependability of the software produced by those processes, both during its development and its operation.

Security in the Software Life Cycle is a part of the DHS Software Assurance Series, and it is expected to contribute to the growing Software Assurance community of practice. This freely-downloadable document is intended solely as a source of information and guidance, and is not a proposed standard, directive, or policy from DHS. Indeed, the document has evolved based on input from a variety of contributors and reviewers in industry, academia, and government, and both derives from and reflects the growing body of knowledge on reducing exploitable software faults. This document will continue to evolve with usage and changes in practice; therefore, comments on its utility and recommendations for improvement will always be welcome.

Joe Jarzombek, PMP
Director for Software Assurance
National Cyber Security Division
Department of Homeland Security
Joe.Jarzombek@dhs.gov
https://buildsecurityin.us-cert.gov/

CREDITS AND ACKNOWLEDGEMENTS

Security in the Software Life Cycle, Version 1.2 (Draft), August 2006

Lead Author and Editor: Karen Mercedes Goertzel

Co-Authors: Theodore Winograd, Holly Lynne McKinley, and Patrick Holley, Booz Allen Hamilton

We would like to thank the following people for their excellent contributions to the content of this document: Michael Howard and Steven Lipner (Microsoft Corporation), Mark S. Kadrich (Symantec), William Bradley Martin (National Security Agency), and Dr. Bill Scherlis (Carnegie Mellon University). Your efforts are greatly appreciated.

Thanks also to the following people who took time to review and comment on the various drafts of this document: Daniel Cross (Department of Homeland Security), Bob Ellison (Software Engineering Institute), Jeremy Epstein (WebMethods), Lucinda Gagliano (Booz Allen Hamilton), Joe Jarzombek (Department of Homeland Security), Shelah Johnson (Booz Allen Hamilton), Mandee Khera (Cenzic Inc.), Morana Marco (Foundstone, Inc.), Robert Martin (MITRE Corporation), C. McLean (Ounce Labs, Inc.), Nancy R. Mead (Software Engineering Institute), Don O'Neill (Center for National Software Studies), Samuel T. Redwine, Jr. (James Madison University), Robin Roberts (Cisco Systems), Robert Seacord (Software Engineering Institute), Michael J. Sherman (Digital Sandbox, Inc.), Michael Smith (Symantec), Frank Stomp (Wayne State University), Richard Struse (Voxem), Edward Tracy (Booz Allen Hamilton), and Stan Wisseman (Booz Allen Hamilton).

Thanks also to the members of the Processes and Practices Working Group and the other DHS-sponsored Software Assurance Working Groups for their insights and suggestions throughout the development of this document.

TABLE OF CONTENTS

List of Tables

List of Figures

1. INTRODUCTION

Software is ubiquitous. Many functions within the public and private sectors are highly dependent on software to handle the sensitive and high-value data on which people's privacy, livelihoods, and very lives depend. National security—and by extension citizens' personal safety—relies on increasingly complex, interconnected, software-intensive information systems—systems that in many cases use the uncontrolled Internet or Internet-exposed private networks as their data bus.

Software-intensive systems run the nation's critical infrastructure (see *Note* below)—electrical power grids, water treatment and distribution systems, air traffic control and transportation signaling systems, nuclear, biological, and chemical laboratories and manufacturing plants, etc. Those systems, too, are increasingly being interconnected via the Internet. Security systems for banks and prisons are similarly software-intensive and networked via the Internet.

> *NOTE: The* Critical Infrastructure Glossary of Terms and Acronyms *published by the U.S. Federal Government's White House Critical Infrastructure Assurance Office (see Appendix A:A.1.1) states that a critical infrastructure constitutes "those systems and assets—both physical and cyber—so vital to the Nation that their incapacity or destruction would have a debilitating impact on national security, national economic security, and/or national public health and safety."*

Businesses often store and process their most sensitive information using software-intensive systems that are directly connected to the Internet. These systems are increasingly being exposed as Web services that enable this sensitive information to be accessed and manipulated by other Web services, which are themselves software systems—all without human intervention. This increased exposure has made sensitive business information and the software systems that handle it visible to people in "the outside world" who were never aware of their existence before. Not all of those people are well-intentioned.

Private citizens' financial transactions are exposed to the Internet, by Web applications and Web services used to shop online, bank online, pay taxes online, buy insurance online, invest online, register children for school online, and join various organizations online. In short, software-intensive systems and other software-enabled capabilities have become the "keys to the kingdom".

The era of information warfare, cyber terrorism, and computer crime is well underway. Nation-state adversaries, terrorists, and criminals are all targeting this multiplicity of software-intensive systems. And by and large, those systems are not attack-resistant or attack-resilient enough to prevent the more determined efforts from succeeding. NIST Special Publication 800-42, *Guideline on Network Security Testing,* states the problem simply and concisely: "Many successful attacks exploit errors ('bugs') in the software code used on computers and networks." Section 2 of this document addresses the threats to software in greater detail.

Software-intensive systems, and particularly information systems, are depended upon to perform user authentication, authorization, and access control—that is, to determine who is authorized to use them, and make sure those authorized users are able to access the system's functionality and the information they store and process, while keeping all unauthorized entities out. To accomplish these security control functions, many large distributed software-intensive systems are protected by isolating them behind bastions of collective authentication portals, filtering routers, firewalls, encryption systems, and intrusion detection systems which are depended on by those systems to keep unauthorized entities out. These protection systems, too, are implemented in full or in part by software.

This increasing dependence on software to get critical jobs done, including protecting other software, means that software's value no longer lies solely (if it ever did) in its ability to enhance or sustain productivity and efficiency, but also in its ability to continue operating dependably regardless of events that threaten to subvert or compromise its dependable operation. The ability to trust, with a justified level of confidence, that software will remain dependable under all circumstances is the objective of Software Assurance. By "all circumstances", we mean:

1. The presence in the software and its environment of unintentional faults,
2. Exposure of the operational software to accidental events that threaten its dependability,
3. Exposure of the software to intentional threats to its dependability threats, both in development and in operation.

Software Assurance, accordingly, includes the disciplines of software reliability (also known as "software fault tolerance"), software safety, and software security. The focus of this document is on the third of these: software security, which is the ability of software to resist, tolerate, and recover from events that intentionally threaten its dependability. Section 2.1 provides a more extensive discussion of Software Assurance to provide a context in which *software security* can be understood.

NOTE: Appendix A includes a listing, with definitions, of terms used in this document, as well as amplifications of all abbreviations and acronyms used throughout this document.

The objective of software security is to design, implement, configure, and support software systems in ways that enable them to:

1. Continue operating correctly in the presence of most attacks by either *resisting* the exploitation of faults or other weaknesses in the software by the attacker, or *tolerating* the errors and failures that result from such exploits;

2. Isolate, contain, and limit the damage resulting from any failures caused by attack-triggered faults that the software was unable to resist or tolerate, and recover as quickly as possible from those failures.

A number of factors can influence how likely software is to be secure. These factors include the choice of programming languages and development tools used to implement the software, and the configuration and behavior of components of its development and execution environments. But it is increasingly believed that the most critical difference between secure software and insecure software lies in the nature of the development process used to conceive (specify, architect, and design), implement, deploy, support, and update that software.

A development organization that adopts a "security-enhanced" software development process will be adopting a set of practices that should initially reduce the number of exploitable faults and other weaknesses that could manifest as vulnerabilities in the operational software. Over time, as these practices become more codified, they should decrease the likelihood that such vulnerabilities are introduced into software the first place. More and more, research results and "real world" experiences indicate that correcting potential vulnerabilities as early as possible in the software development life cycle, mainly through the adoption of security-enhanced processes, is far more cost-effective than the currently-pervasive approach of developing and releasing frequent patches.

According to Soo Hoo *et al* in "Return on Security Investments" (see Section 1.5 for reference), the return on investment for secure software engineering can be *as high as 21 percent.*

1.1. Purpose of this Document

This document describes a number methodologies, life cycle process models, sound practices (often referred to as "best practices"), and supporting technologies that have been reported to increase the likelihood that the resulting software will be free of exploitable faults, and also the likelihood that artifacts of the conceptualization phases of the software development life cycle will incorporate elements that are conducive to good security. Coincidentally, while it is not always their explicit objective, many of the security-oriented methodologies and practices described herein may help in the production of software that is higher in quality and reliability as well as security.

Due to the current state of security measurement capabilities, it is difficult to directly assess the degree to which a particular methodology, process, practice, or technology affects a software-intensive system's security, but many

organizations promoting and/or adopting these techniques have noticed improvements in their own software's security.

By gathering descriptive overviews of a number of existing processes and practices in a single document, we hope to provide a convenient basis from which development organizations that recognize the need to change their development processes and practices in order to produce more secure software can become aware of, and begin to compare and contrast, different development methods and techniques that other organizations have found effective. This information should similarly be of interest to development organizations that have already undertaken such security-enhancements, and wish to learn about other approaches that may help their efforts progress even further.

In summary, the reporting of practices and processes in this document should help:

1. Increase the reader's awareness and understanding of the security issues involved in the conception, implementation, and production of software;

2. Help the reader recognize how the software development process and its supporting practices can either contribute to or detract from the security of software;

3. Provide an informational context in which the reader can begin to recognize and understand the security deficiencies in his/her organization's current life cycle processes and practices;

4. Provide enough information about existing "security-enhanced" practices and security-minded approaches to general practices to enable the reader to begin identifying and comparing potential new practices that can be adopted and/or adapted to augment, adjust, eliminate, or replace his/her organization's current non-secure practices.

The techniques and practices described in this document should improve the likelihood that software in development should be protected from tampering by malicious developers, and that operational software will be able to resist or tolerate and recover from attacks. The insider threat to operational software presents a more difficult challenge—one which improvements to the security of software will be inadequate to address on its own.

1.2. Intended Audience

This document is primarily intended to be read by people involved in the creation of software-intensive systems, either through coding "from scratch" or through the integration or assembly of acquired or reused software packages. This document should help these readers increase their understanding of the security issues associated with the engineering of software, including the reuse of software of unknown pedigree (SOUP), and should help them identify ways in which they can improve their processes and practices so they result in software that is better able to withstand the threats to which it is increasingly subject. Specifically, the intended audience for this document includes: requirements analysts, software architects and designers, programmers, integrators, testers, and software project managers.

Although it is not primarily intended for their use, this document should also be of interest to the following groups, who are increasingly important in encouraging the development and delivery of secure software:

1. **Acquisition Personnel**: This document should help them recognize the importance of security as a consideration in the specification and evaluation of software products and outsourced engineering services, and should help them define appropriate security requirements and evaluation criteria when soliciting and procuring such products and services.

2. **Information and Network Security Practitioners:** This document can help familiarize security engineers, risk analysts and managers, certifiers, accreditors, and other security practitioners with the threats, attacks, vulnerabilities, and engineering challenges that are unique to the software elements of the

information systems and networks they are responsible for securing. This knowledge should help them specify and implement more effective system-level security architectures, risk mitigations, and countermeasures.

This document presumes that the reader is already familiar with good general systems and software engineering and testing processes, methodologies, practices, and technologies.

NOTE: To benefit fully from the content of this document, the reader should also be familiar with some key information security and cyber security concepts, such as "privilege", "integrity", "availability". Some recommended sources for information on these topics are identified in Section 3.

1.3. Document Structure and Scope

This document is divided into five (5) sections and seven (7) appendices. A list of references is provided at the end of each section or appendix. Included in each list are source materials used in compiling this document. In addition, some reference lists also include additional informative materials on specific topics discussed in that particular section or appendix. Table 1-1 summarizes the structure and content of this document.

Table 1-1. Document Structure and Content

Section No. and Title	Content of Section
Section 1. Introduction	Describes the purpose, intended audience, structure, and scope of the document.
Section 2. Why Security is a Software Issue	Introduces the threats to which much software is subject, the motivations and mechanisms of the attackers who target software. This section also introduces various factors in the software's development process that contribute to its vulnerability to those threats.
Section 3. What Makes Software Secure?	Examines the properties of software that make it secure, and discusses how other desirable traits of software can also contribute to its security.
Section 4. Security in the Software Development Life Cycle	Presents a survey of efforts to security-enhance leading life cycle processes, process improvement models, and development methodologies. Also discusses how generic methodologies might be security-enhanced. Many organizations have found that such improvements increase the likelihood of producing secure software. The section then discusses security education and training for developers. Finally, the section closes with a checklist of practices that can be said to constitute a "minimal responsible level of practice" for secure software development.
Appendix A. Terms, Definitions, Abbreviations, Acronyms	Provides a glossary of terms with their definitions, as those terms are used in this document. Also lists all abbreviations and acronyms used throughout the document, along with their amplifications.
Appendix B. For Further Reading	Provides a suggested reading list that is intended to augment the references listed at the end of each section/appendix.
Appendix C. Reconciling Agile Methods with a Secure Development Life Cycle	Discusses ways in which use of agile methods might possibly be reconciled with the need to security-enhance the development life cycle.
Appendix D. Standards for Secure Life Cycle Processes and Process Improvement Models	Introduces national and international standards efforts to define security-enhanced process improvement models and life cycle processes.
Appendix E. Software and Application Security Checklists	Lists some software and application security checklists available in the public domain that may be useful to security reviewers and testers.

Table 1-1. Document Structure and Content (cont'd)

Section No. and Title	Content of Section
Appendix F. Security Concerns Associated with Outsourced and Offshore Software Development	Presents excerpts of key publicly-available government documents that discuss the security concerns associated with outsourcing of software development services and acquisition of offshore-developed software products.
Appendix G. Secure Development Principles and Practices	Suggests specific ways in which developers can begin to security-enhance their own approaches to conceptualization, implementation, distribution, and deployment to help "raise the floor", in terms of defining a minimum set of desirable principles and practices that can be inserted throughout the software life cycle even in organizations that have not established general software engineering practices. For organizations that are already committed to improving the quality and security of their development processes and practices, this discussion may provide further ideas to help move even closer to a fully disciplined, repeatable, secure software life cycle process based on a consistently-applied, coherent set of development practices and supporting methodologies that will not only improve the security of the implemented software being secure, but also its dependability and quality. Ends with a discussion of how development tools and technologies influence the security of software and addresses other considerations that affect software's security, including security of the execution environment and the use of "secure in deployment" systems engineering techniques.
Appendix H. Software Security Awareness Campaigns	Discusses some significant "software security awareness" campaigns that may provide ideas for implementing similar initiatives in other organizations or communities.
Appendix I. Secure Microkernels	Provides a more in-depth discussion of this technology than that found in Appendix G:G.4.3.

1.4. Exclusions

The information in this document should be helpful to developers of application-level and system-level software that run on general purpose computers. Developers of embedded software, firmware, and real time software should also find this document useful, to the extent that the properties and development processes for these types of software are the same as those for general purpose computer system software. This said, there are some considerations that are outside the scope of this document. These are:

- **Security issues directly caused or affected by the unique aspects of embedded software, firmware, and real time software.** Such characteristics include the unique computational models, operational constraints, development languages, tools, and practices, etc., used in the creation and operation of such software. These unique aspects may make it difficult for the creators of these types of software to apply some of the methodologies, principles, and practices described here. Embedded software in particular may not be subject to the same threats as other software. Moreover, it may be subjected to its own unique set of threats.

  This document presumes that the software under consideration will be exposed to a network, and that the threats to that software in deployment will be enabled by that network exposure. Beyond a discussion of the use of tamperproof hardware platforms to host software, this document does not address physical threats to the processors on which software operates; in particular, it does not address physical threats that are intended to physically interfere with the signaling mechanisms through which embedded software systems often interact with their physical environment.

- **Threats, attacks, and vulnerabilities that arise due to the purpose or mission the software is intended to fulfill.** For example, the fact that a software application is part of an intrusion detection

system, a Supervisory Control and Data Acquisition (SCADA) system, a military command and control system, a cross-domain solution, a financial funds transfer system, an online shopping system, or an Internet gambling system, is relevant only in so far as the purpose of the software helps indicate (1) criticality or consequence; (2) the priority and importance of its various dependability properties.

- **Information security, including information flow security, and the software security functionality that is often used to achieve them.** Beyond background information in Section 3.1.1, this document does not address the security of the information (including information flows) that software-intensive information systems are intended to enable, control, and/or protect. This document is about producing software that is itself robust against attack, regardless of whether that software handles information (data) or not. Section 3.1.1 also clarifies the distinction between "software security" and "information security".

1.5. References for this Section

National Institute of Standards and Technology (NIST) Special Publication (SP) 800-42, "Guideline on Network Security Testing" (October 2003)-3.11, Post-Testing Actions
http://csrc.nist.gov/publications/nistpubs/800-42/NIST-SP800-42.pdf

Kevin Soo Hoo, A.W. Sadbury, and Andrew R. Jaquith: "Return on Security Investments" (Secure Business Quarterly, Vol. 1 No. 2, 2001)

J. McDermott, Center for High Assurance Computer Systems, Naval Research Laboratory: "Attack-Potential-Based Survivability Modeling for High-Consequence Systems" (*Proceedings of the IEEE 3rd International Information Assurance Workshop*, March 2005)
http://chacs.nrl.navy.mil/publications/CHACS/2005/2005mcdermott-IWIA05preprint.pdf - *or* -
http://www.iwia.org/2005/McDermott2005.pdf

Edward A. Lee: "Embedded Software", chapter in M. Zelkowitz (editor): *Advances in Computers* (Vol. 56; London: Academic Press, 2002)

2. WHY SECURITY IS A SOFTWARE ISSUE

Secure software cannot be intentionally subverted or forced to fail. It is, in short, software that remains dependable (i.e., correct and predictable) in spite of intentional efforts to compromise that dependability. (Read more on the relationship between security and dependability in Section 3).

Before proceeding, the reader is strongly encouraged to review Appendix A:A.1, "Terms and Definitions", which presents list of terms and their definitions as they are used in this document.

Software security matters because so many critical functions have come to be completely dependent on software. This makes software a very high-value target for attackers, whose motives may be malicious (a desire to cause inconvenience), criminal, adversarial, or terrorist. What makes it so easy for attackers to target software is the virtually guaranteed presence of vulnerabilities, which can be exploited to violate one or more of the software's security properties, or to force the software into an insecure state. According to the Computer Emergency Response Team Coordination Center (CERT/CC) at Carnegie Mellon University (CMU), most successful attacks on software result from successful targeting and exploitation of known but non-patched vulnerabilities or unintentional misconfigurations.

Successful attacks on software systems result from human ingenuity. A subtle design flaw may be exploited, or a previously undiscovered development fault or other weakness may be located through the attacker's engineering efforts. This section discusses the threats that target most software, and the shortcomings of the software development process that can render software vulnerable to these threats.

2.1. Software Assurance: Providing Context for Software Security

The term "Software Assurance" refers to the justifiable confidence that software consistently exhibits all of its required properties and functionalities, including dependability. The current emphases of most Software Assurance (by contrast with Software Quality Assurance) practitioners and initiatives are focused predominantly on software safety, security, and reliability.

According to Committee on National Security Systems (CNSS) Instruction No. 4009, "National Information Assurance (IA) Glossary", Software Assurance is:

> *the level of confidence that software is free from vulnerabilities, either intentionally designed into the software or accidentally inserted at anytime during its life cycle, and that the software functions in the intended manner.*

The Department of Defense (DoD) in "DoD Software Assurance Initiative" states further that Software Assurance relates to:

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

The National Aeronautics and Space Administration (NASA) takes a more process-oriented view in the agency's Software Assurance Standard, which states that Software Assurance comprises the:

> *…planned and systematic set of activities that ensures that software processes and products conform to requirements, standards, and procedures. It includes the disciplines of Quality Assurance, Quality Engineering, Verification and Validation, Nonconformance Reporting and Corrective Action, Safety Assurance, and Security Assurance and their application during a software life cycle.*

The National Institute of Standards and Technologies' (NIST) Software Assurance Metrics and Tools Evaluation (SAMATE) initiative and the DHS both agree with NASA's process-oriented view of Software Assurance, but state in addition that the objective of Software Assurance activities is to achieve software that exhibits:

- *Trustworthiness, whereby no exploitable vulnerabilities or weaknesses exist, either of malicious or unintentional origin;*
- *Predictable Execution, whereby there is justifiable confidence that the software, when executed, functions as intended;*
- *Conformance, whereby a planned and systematic set of multi-disciplinary activities ensure that software processes and products conform to their requirements, standards, and procedures."*

These objectives coincidentally correspond to three key properties of secure software.

Finally, the Object Management Group (OMG) Special Interest Group on Software Assurance provides a definition of Software Assurance that combines the "level of confidence" aspect of the CNSS and DoD definitions with the "trustworthiness" aspect of the NIST and DHS definitions:

> *Software Assurance is the software's demonstration of "justifiable trustworthiness in meeting its established business and security objectives."*

What all of these definitions imply, but do not explicitly state, is the relationship between the desirable software properties that they have embraced as crucial to establishing justifiable confidence in software, and the overarching desirable property of all software: *dependability*—which because of the ever-increasing quantity and sophistication of cyber attacks now targeting and exploiting software, requires *security*.

2.2. Threats to the Security (and Dependability) of Software

Software is subject to two general categories of threats:

1. **Threats during development (mainly insider threats):** A "rogue" developer can sabotage the software at any point in its development life cycle, through intentional exclusions from, inclusions in, or modifications of the requirements specification, the threat models, the design documents, the source code, the assembly/integration framework, the test cases and test results, or the installation/configuration instructions and tools. The secure development processes, methodologies, and practices described in this document are, in part, designed to help reduce the exposure of software to insider threats during its development process.
2. **Threats during operation (both insider and external threats):** Any software system that runs on a network-connected platform is likely to have its vulnerabilities exposed to attackers during its operation. The level of exposure will vary depending on whether the network is public or private, Internet-connected or not. The larger, more "open" and uncontrolled the network, the more exposed the software will be to external threats. But even if the network is private, small, and closely managed, there may still be a threat from untrustworthy elements in the software's authorized user community (i.e., "malicious insiders").

2.2.1. Why Attackers Target Software

Attackers search for exploitable faults and other vulnerabilities in software to either:

- Compromise one of the security properties of the software (see Section 3 for the discussion of software security properties). For example, by compromising the integrity of a software process, the attacker can append malicious logic to that process; or

- Compromise the dependability of either or both of a pair of interacting components, most often by exploiting vulnerability in their interaction mechanism. For example, an attacker might hijack the message sent by one component to another, replace it with a spurious message in which the header points back to a third, malicious component instead of the valid sender. The second component (recipient) will then be tricked into sharing data with the malicious component rather than with the valid sender of the original message. The result will be the compromise of both the confidentiality of the data and integrity of the system containing the two components.

The security properties of software can be intentionally compromised through several mechanisms:

1. Direct attacks (by humans or rogue processes) on the software; such attacks either attempt to exploit known or suspected vulnerabilities, or to insert malicious code;

2. Intentional triggering of an external fault (e.g., in an interface mechanism) at the software boundary (or surface), leaving the software more vulnerable to direct attack;

3. Intentional changes of execution environment state from correct/expected to incorrect/unexpected, potentially resulting in exploitable misbehavior in the software;

4. Triggering of the execution of malicious logic embedded within the software;

5. Attacks on the external services the software relies on to perform functions on its behalf, or defense in depth measure relied upon to protect the software from attack.

Many compromises are achieved through a particular sequencing of exploits that target a combination of vulnerabilities in one or more components, especially if the components are assembled in a way that emphasizes one component's reliance on a vulnerable function in another. The vulnerabilities that are most likely to be targeted are those found in the components' external interfaces, because those interfaces provide the attacker with a direct communication path to the vulnerabilities.

A number of well-known attacks, particularly those originating from the Internet, target software that is known to incorporate interfaces, protocols, design features, or development faults that are both well understood and widely publicized to harbor inherent weaknesses; such software includes Web applications (including browser and server components), Web services, database management systems, and operating systems.

Attackers who target software generally have one or more of the objectives summarized in Table 2-1. In some cases these objectives are ends in themselves, but in most cases they are simply means to a broader end. For example, an attacker may simply wish to read a non-public document or crash a Web server. More often, however, the attacker's goal will be to gain unauthorized privileges to take control of the targeted system to accomplish other objectives, such as downloading the content of customer databases containing credit card numbers which can then be used in fraudulent transactions.

Until recently, the main or only response to the increasing number of Internet-based attacks was to rely on a veritable arsenal of "defense in depth" countermeasures at the operating system, network, and database or Web server levels, while failing to directly address the insecurity of the application-level software. There are two critical shortcomings with this approach:

1. The security of the application depends completely on the robustness of the "fortress wall" of protections that surround it;

2. The defense in depth protections themselves, with few exceptions, are as likely to harbor exploitable development faults and other weaknesses as the application software they are (ostensibly) protecting.

Table 2-1. Why Attackers Target Software

Objective	Property Compromised	Life Cycle Phase	Type of Software Susceptible
Modify/subvert the functionality of the software (often to more easily accomplish another compromise)	integrity	development or deployment	all systems
Prevent authorized users from accessing the software, or preventing the software from operating according to its specified performance requirements (known as "denial of service")	availability	deployment	all systems
Read data controlled, stored, or protected by the software that he/she is not authorized to read	confidentiality	deployment	information systems
Access to data, functions, or resources he/she is not authorized to access, or perform functions he/she is not authorized to perform	access control	deployment	all systems
Obtain privileges above those authorized to him/her (to more easily accomplish another compromise) (known as "escalation of privilege")	authorization	deployment	all systems

Attackers have discovered that they are able to either compromise the defense in depth measures themselves in order to "break through" the fortress wall, or to simply bypass them, often by using (misusing) valid interfaces to application-level software in unintended ways. These exploitable interfaces originate in deficiencies in the operating models of many applications—especially Web applications and Web services—in combination with deficiencies in the lower level technologies in the applications' execution environments.

A prime example is the use of Transmission Control Protocol (TCP) port 80 for transmitting numerous protocols used in Web applications and Web services, making it impossible to configure firewalls to selectively block different application-level protocols. This is because network firewalls accomplish such blocking by closing off TCP ports, which in the case of port 80 is not possible because its closure would block the desirable traffic along with the undesirable. Another example is the lack of even basic security measures being built into the protocol themselves: for example, though the standard for Secure Socket Layer (SSL) requires servers to be authenticated to clients, it leaves authentication of clients to servers as optional, while the standard for Simple Object Access Protocol (SOAP) used in Web service interactions does not include any requirement for service-service authentication.

The wide publicity about the literally thousands of successful attacks on software hosted on or accessible from the Internet has only made the attacker's job easier. Attackers can study the numerous reports of security vulnerabilities in a wide range of commercial and open source software programs, and as a result are able to very quickly craft sophisticated, "surgical" attacks that exploit those specific vulnerabilities.

Add to this the growing "insider threat" problem: an increasing number of attacks against software originate from its own authorized users, and even more worrying, its developers. "Insiders" already have some level of authorized access to the software—in the case of end users, to the software in deployment; in the case of developers, to the software in development. Insiders often have the same goals as outsiders when attacking software, but their advantage of greater access and privilege gives their attacks much higher likelihood of success and potential for damage, while reducing likelihood of detection. Neither the access controls that protect the software from unauthorized access (by developers, when stored on the development system, and by end users when deployed on its operational host), nor operational defense in depth protections designed to detect and block externally-sourced attacks, will be effective at preventing or detecting insider attacks.

The potential for damage by malicious developers is even greater for software based on modern application processing models—Service Oriented Architectures (SOA), peer-to-peer computing, agent-based computing, and grid computing. In all of these models, "users" include software processes that operate independent of human intervention. This means malicious logic embedded by malicious developers in such independent software entities has an even greater likelihood of going undetected once that software is operational.

The only hope of preventing such insertions during software's development process is to include a rigorous set of reviews of the design and implemented software that include looking specifically for unexpected logic, exploitable faults, and other weaknesses. Moreover, careful configuration management of source code and executables is needed to ensure that every action by developers throughout the software's development and post-release support phases irrevocably logged, including every check in and check out of configuration controlled artifacts, and every modification to the code base. To aid in the detection of tampering, all unauthorized and unnecessary access to the configuration controlled artifacts should prevented. Just because developers are authorized to modify code does not mean they need to do so, particularly not just before distribution.

Security controls in the software's life cycle process should not be limited to the design, implementation, and test phases. It is important to continue performing code reviews, security tests, and strict configuration control during the post-release support phase, to ensure that updates and patches do not add security weaknesses or malicious logic to existing software products.

It may be possible to observe trends that indicate subversive developer behavior by configuring access controls in the execution environment to allow *only* execute-access to deployed binaries, bytecode, and runtime-interpreted source code by anyone except the administrator (i.e., deny all read- and write-access). In addition, the same access controls should be configured to deny write-access to any software configuration files by anyone except the administrator. All actions performed by the administrator should be audited, and those audit logs analyzed regularly by a third party (not the administrator) to detect any evidence of intentionally or unintentionally compromising activity by the administrator. These measures should help reduce the vulnerability of the deployed software to malicious code insertions, reverse engineering, or incorrect configuration.

2.2.2. Attacks and the Vulnerabilities They Target

When it comes to software, there remains some confusion in the categorization of threats, attacks, and vulnerabilities. This confusion arises, at least in part, because many vulnerabilities are named for the attack that exploits them (possibly as a kind of "shorthand" for "the outcome of this particular type of attack"), while others are named for the event that results from such exploits (again, likely a shorthand for "the event that results when this vulnerability is successfully exploited").

For example, "buffer overflow" is used to designate both the vulnerability that is exploited and that attack pattern that exploits it, when in fact it most accurately pertains to the *result* of that exploitation. The real vulnerability is faulty buffer handling by the software, i.e., a memory buffer of a fixed size is allocated to receive input data, but the size of the input data is not checked to ensure it does not exceed the size of the allocated buffer, nor does the software prevent too-large data from being written into the buffer.

The attack pattern, then, is to input an amount of data that the attacker suspects will be larger than the typical buffer allocated to receive the input (clues as to buffer size include the nature of the expected input, for example a Social Security No. has a fixed length, so an attacker would be justified in expecting that the buffer allocated to receive Social Security Numbers as input would have a size that directly relates to the size of a Social Security Number, so that inputting a paragraph of text into a form field designed to receive only 9 numerical digits might have a good chance of overflowing a badly managed buffer). The outcome of the exploit, if it succeeds, is the actual buffer overflow: the portion of data that exceeds the allocated buffer size has to be written to adjacent memory space outside the bounds of the too-small buffer. Details on the objectives and variants of buffer overflows are widely documented (some of these documents are listed in Section 2.4).

Rather than attempt a taxonomy of attack patterns or vulnerabilities in this document, the resources in Table 2-2 should provide the reader with a substantial background that will help them understand the exact nature of the most common attacks to which software is subject, and the vulnerabilities that are typically targeted by those attacks.

2.3. Sources of Software Insecurity

Most commercial and open source applications, middleware systems, and operating systems are extremely large and complex. In normal execution, these software-intensive systems can transition through vast number of different states. These characteristics of software make it particularly difficult to even make consistently correct, let alone consistently secure.

This said, a large percentage of security weaknesses in software could be avoided were developers only to consciously think about avoiding them. Unfortunately, most people involved in software development are not taught how to recognize the security implications of certain software requirements or omission of requirements. Nor do they learn the security implications of how software is modeled, architected, designed, implemented, tested, and prepared for distribution and deployment. Without this knowledge, they way software is designed and implemented may not only deviate from its specified security requirements, those requirements may have been inadequate in the first place. Finally, this lack of knowledge will prevent the developer from recognizing and understanding how the mistakes he/she makes during development can manifest as exploitable weaknesses and vulnerabilities in the software when it becomes operational.

Developers and designers are also seldom taught why measures that are intended to reduce the quantity of unintentional faults and other weaknesses in software are not always effective when it comes to recognizing and removing intentional faults, or to handling the failures that result when unintentional faults and other vulnerabilities that remain in deployed software are intentionally exploited by attackers.

Reducing exploitable weaknesses begins with the specification of software requirements. Software that includes requirements for security constraints on process behaviors and handling of input, resistance and/or tolerance of intentional failures, etc., is far more likely to be engineered up front to remain dependable in the face of attack. A security-enhanced life cycle process goes even further: it acknowledges that dependability—which can only be said to have been achieved if it is exhibited at all times, including when the software is subjected to hostile conditions and malicious inputs—is *always* a requirement for software. There is no point to software even existing unless it can always be trusted to fulfill its purpose correctly whenever it is called upon to do so.

Moreover, threats to software exist whether requirements analysts recognize them or not. The unavoidable presence of security risk means that at least a minimal amount of software security will be needed even if explicit requirements for it have not been captured in the software's specification. The reality is that security risk exists even when documented security requirements do not. A security-enhanced life cycle process should balance documented requirements against that reality. It should (at least to some extent) compensate for security inadequacies in the software's requirements by adding risk-driven practices and checks to the requirements-driven practices and checks at all of the phases of the life cycle.

Developing software from the beginning with security in mind will be more effective by orders of magnitude than trying to prove after it has been implemented, through testing and verification, that it is secure. For example, attempting to prove that an implemented system will never accept an unsafe input (i.e., proving a negative) will be impossible. By contrast, it is not only easier to design and implement the system so that input validation routines check *every* input that the software receives against a set of pre-defined constraints, testing the input validation function to prove that it is consistently invoked and correctly performed every time input enters the system will be included in the system's functional testing, and thus will not be seen as "extra work".

2.3.1. Supply and Demand, and Software Security

In modern business operations, users are extremely dependent on software-intensive information systems. This dependency makes it extremely difficult and often impossible for them to avoid or reject those systems, even when their software elements operate unexpectedly or incorrectly, fail often, or are frequently compromised by viruses and other malicious code. Every day more and more sensitive, critical, and personal information is exposed to the Internet by software systems that cannot be trusted to handle that information safely—software systems that are based on insecure technologies, and riddled with avoidable faults and other weaknesses, all of which make the software vulnerable to attack.

Moreover, an ever increasing number of critical activities are being trusted to high-consequence software systems. *High-consequence* software systems are those in which a failure could result in serious harm to a human being in the form of loss of life, physical injury or damage to health, loss of political freedom, loss of financial well-being, or disastrous damage to the human's environment. Large-scale software systems that support a very large number of software users are also considered high-consequence because they are not only difficult to recover after a failure, but because it would be extremely difficult and/or expensive to make reparations to the affected humans for the damages that would result from such a failure. Examples of high-consequence software systems include the software elements of national security systems, medical control systems, banking systems, Supervisory Control And Data Acquisition (SCADA) systems, and electronic voting systems. Distressingly, more and more high-consequence software systems are being exposed to the Internet.

Years of past experience have unfortunately led the users of software to believe that no improvement in its quality or security can be expected. Ironically, users' resignation to the inadequacy of software is then misinterpreted by software suppliers as clear evidence that the users are perfectly happy with the bad software they have had no alternative but to accept and even expect. Thus a vicious cycle perpetuates itself.

However the circumstances in which this vicious cycle has been allowed to spin out of control are changing rapidly. California introduced a law in 2003 requiring companies to disclose when personal data is lost or stolen. Since then, many states have followed suit with similar laws. As a result, the public has been made aware of numerous high-profile incidents where personal data has been compromised. Individual users are beginning to understand the connection between identity theft and computer fraud and the software-intensive systems that handled their compromised personal data. Similarly, governments and other large institutional user organizations are recognizing the comparable link between the growing threat of cyber terrorism and information warfare, and the high-consequence software systems on which millions of human lives and livelihoods depend.

While many of these breaches are a direct result of faulty software, many are not. For example, in 2006 a Department of Veterans Affairs (VA) laptop containing personal data on 26.5 million current and former members of the armed services was stolen from an employee's home. Many of these breaches result from inadequate security requirements. For example, the VA's software requirements could have specified that the personal data be encrypted and/or stored on a remote server so that the theft of the laptop would not have affected the sensitive information.

In the face of these threats, some leading software suppliers have started to recognize that they cannot afford to wait for a truly catastrophic cyber attack to occur that is made possible by an avoidable vulnerability in software they produced. They have started to institute new development processes and practices with the express objective of producing more secure software. This is an important start, and it is hoped that it represents the beginning of a trend that will eventually take hold among all software suppliers and integrators.

By remembering that they too are software users, and as such also victims of faulty software, software developers and their managers can begin to clarify their understanding of the threats to software, and to prioritize their responses to those threats (in the form of improvements to their development practices).

Developers should also take advantage of resources that have long been familiar to Information Technology (IT) security practitioners in order to establish and solidify their awareness and understanding of the security vulnerabilities observed in commercial and open source software products. Not only are such products likely to be used as components of larger software systems, but even custom-developed software is likely to exhibit similar vulnerabilities. An understanding of such vulnerabilities should make them easier for custom developers to avoid in the software they create. Moreover, a similar understanding should make it easier for security testers, both of custom-developed and acquired and reused components, to recognize those vulnerabilities in implemented software. Finally, developers of commercial and open source software should make a routine of frequently consulting the leading vulnerability reporting databases to identify and prioritize critical vulnerabilities for which they need to develop and distribute patches. These vulnerability databases include those listed in Table 2-2 below:

Table 2-2. Vulnerability Reporting Databases

Database Name	URL
NIST National Vulnerability Database	http://nvd.nist.gov/
US-CERT Vulnerability Notes Database	http://www.kb.cert.org/vuls/
MITRE Corporation Common Vulnerabilities and Exposures (CVE)	http://cve.mitre.org/
Open Source Vulnerability Database	http://www.osvdb.org/
Internet Security Systems (ISS) X-Force Database	http://xforce.iss.net/xforce/search.php
SecurityFocus Vulnerabilities	http://www.securityfocus.com/vulnerabilities
Purdue CERIAS Cooperative Vulnerability Database	https://cirdb.cerias.purdue.edu/coopvdb/public/
Web Application Security Consortium Web Hacking Incidents Database	http://www.webappsec.org/projects/whid/

2.3.2. Exploitable Development Faults and Other Weaknesses

Software programs—and especially networked application-level software—are most often compromised because of the intentional exploitation of faults and other weaknesses that manifest from:

- Complexities, inadequacies, and/or changes in the software's processing model (e.g., Web or service-oriented architecture model);
- Incorrect developer assumptions, including assumptions about the capabilities, outputs, and behavioral states of the software's execution environment, or about inputs to be expected from external entities (users, software processes);
- Flawed specification or design or defective implementation of:
    - the software's interfaces with external entities. Development mistakes of this type include inadequate (or non-existent) input validation, error handling, and exception handling;
    - the components of the software's execution environment (from middleware level and operating system level to firmware and hardware level components).

The ability to determine the exploitability of a software fault/weakness is an inexact science at best. Even if extreme caution is applied when implementing software from scratch, most software systems are composed from a combination of acquired, reused, and from-scratch components. While it may be possible, through careful, extensive code review, to identify all faults and weaknesses in from-scratch and open source code, and even to determine the potential exploitability of those faults and weaknesses, identifying comparable faults/weaknesses and their exploitability in acquired or SOUP binary components is for all intents and purposes impracticable. There are not adequate tools, resources, time, or expertise available.

Rather than attempt to judge whether a particular fault/weakness (or category of fault/weakness) is exploitable, developers should simply consider *all* software faults and weaknesses to be potentially exploitable, and focus on strengthening the security of the software's interfaces with external entities, on increasing its tolerance of all faults (including intentional faults), and on modeling and protecting the software against the more subtle, complex attack patterns involving externally-forced sequences of interactions among combinations of components or processes that were never intended to interact during normal software execution.

Most faults and weaknesses can be prevented by applying the necessary level of care and consistency in security-enhancing the activities and practices associated with the conceptualization phases of the development life cycle (requirements, architecture, and design). However, no matter how faithfully a security-enhanced life cycle is adhered to, as long as software continues to grow in size and complexity, some number of exploitable faults and other weaknesses are sure to exist.

Developer mistakes are unavoidable. Even if they are avoided during conceptualization (e.g., due to use of formal methods) and implementation (e.g., due to exhaustive, accurate code reviews and careful, extensive testing), vulnerabilities may still be introduced into software during its distribution and deployment.

The nature of the threat is changing more rapidly than the software's artifacts can be adapted to counteract those threats, even when an agile development process is used. It is a truism that to be 100% effective, defenders must anticipate *all* possible vulnerabilities, while attackers need find only *one*.

If the development organization's time and resource constraints prevent secure development techniques from being applied to the entire software system, those portions of software that should be given highest priority in terms of the need for careful, secure engineering are:

1. Components in which will be placed a high degree of trust, such as components that implement security functions and other high-consequence functions. Federal Information Processing Standard (FIPS) 199 describes a process for categorizing a software system's impact level; this process can be adapted to prioritize the consequence of software components within a software system);

2. Interfaces between components (modules, processes, etc.) within the software system;

3. Interfaces between the software system and (1) its environment, and (2) its users.

2.3.3. Inadequate Development Principles and Practices

Some of the inadequacies in the software's development process that may contribute to the presence of exploitable faults and weaknesses in the software include:

1. **Non-security-enhanced life cycle control processes:** Software development processes should contain security checkpoints (peer reviews, design reviews, code reviews, security testing, etc.) throughout the life cycle. Such checkpoints help prevent rogue developers from subverting software by including exploitable faults, weaknesses, or backdoors, or by embedding malicious code, that goes undetected. Security checkpoints also increase the likelihood of finding any exploitable faults/weaknesses/backdoors that may have been included by mistake. Security checkpoints should ensure that all exit criteria (including security criteria) are satisfied at the end of one phase before the software is able to enter the next phase.

2. **Undisciplined development practices:** Undisciplined development processes can lead to the inclusion of unintended requirements in or omission of intended requirements from the software's specification, security flaws in the software's design, defects in the software's implementation, or definition of insecure parameters for its configuration. On large development projects, development teams within the larger organization need to be sure to communicate frequently:

- To ensure that their assumptions about each other's portions of the software remain valid throughout the development life cycle;

- To address emergent security properties that can only be assured by considering the interactions among components within the system as a whole.

Secure configuration management should be instituted to ensure that the version of software that checked in after passing each security checkpoint cannot be modified; changes made to address checkpoint findings should always result in a code version with a unique version number.

3. **Incorrect developer assumptions or misunderstanding of assumptions:** The assumptions developers make about the anticipated state changes in the execution environment, and in the software itself in response to environment state changes, often do not anticipate the intentional state changes in the software or its environment triggered by attacks. A key source of exploitable weaknesses in software are incorrect and incomplete assumptions about the behavior of the software under all possible conditions, not just conditions associated with "normal use". While the complexity and size of modern software systems makes it impossible to model and analyze all possible conditions, it should be possible to selectively model and analyze a representative subset of "abnormal use" conditions—specifically those associated with misuse and abuse.

   Another source of vulnerabilities is the developer's lack of understanding of how to always design and build software appropriately given correct assumptions, even when those assumptions are included in the requirements. Incorrect or misunderstood developer assumptions can result the following:

   - Incomplete, omitted, overly-general, or poorly-stated functionality-constraining and non-functional requirements;

   - Failure to translate an actionable requirement, particularly when the requirement is negative or non-functional (e.g., a requirement that the software exhibit a particular property, or that a particular behavior or function be constrained);

   - Failure to architect and design software that satisfies its actionable non-functional and negative requirements;

   - Failure to recognize the security implications of different languages, tools, or techniques and how they are used when implementing the software;

   - Not knowing where to begin a security evaluation of an acquired or reused software component in order to anticipate how that software will behave in combination with other components and how that behavior affects system security;

   - Failure to recognize the need for risk-driven security testing (testing under abnormal conditions based on attack patterns) and stress testing (testing under abnormal activity, inputs, etc…), even when the software's requirements specification is considered thorough and accurate;

   - Failure to use a disciplined, iterative development methodology in which security assessments are embedded in each development phase, and that ensures that each time a change is made to the architecture, design, or implementation, the new risks that may be introduced by that change are assessed;

   - Limitation of test cases to normal operating conditions, without "stress testing" to add abnormal conditions and other conditions that stress the software's design assumptions;

- Failure to exercise the necessary degree of caution required when preparing the software for distribution, to ensure that it does 'not contain any residual features (debugger commands, comments containing sensitive information [such as information about residual uncorrected faults], etc.) that could make the software vulnerable in deployment;

- Failure to use appropriate tools or techniques to assure that security standards have been met prior to deployment;

- Failure to design the software for its intended operational environment, in terms of administration and usage features.

To develop software that is secure, developers must overcome these inadequacies in their own approach to software specification, design, and implementation.

4. **Insufficient capture of requirements for security properties:** After functional security requirements are captured, software requirements analysts need to address non-functional security requirements for the software's security properties. Developers also need to learn how to translate negative and non-functional requirements into functional behaviors or constraints in the software's design and implementation.

5. **Design mistakes:** The design of the software needs to anticipate all of the different state changes that could result from failures triggered by attack patterns (which often manifest as anomalies in input streams). The design also needs to address security aspects associated with acquired or reused software, such as the need to remove or isolate and contain "dormant" functions not used in normal operation. Such isolation and constraint will reduce the risk that such unused functions could be unexpectedly executed as the result of an unanticipated state change in the software were it to come under attack.

6. **Inadequate documentation:** Documented requirements, threat models, use and abuse/misuse cases, etc., provide the basis upon which testers can formulate effective security test plans. The design specification needs to document thoroughly all inputs, outputs, and their possible data format characteristics so that effective input validation can be implemented, and also all anticipated state changes, so that effective exception handling can be implemented. The security test plan needs to include tests that exercise misuse/abuse cases, not just use cases.

7. **Insufficient tools and resources for developers:** The number and quality of tools available to assist developers in performing software security reviews and tests is increasing all the time. For example, tools that specifically support security code audits will help ensure that developers discover and address problems as early as possible, thus increasing the speed in which *secure* software can be released. It is the developer's responsibility to identify the appropriate tools and resources, and the project manager's to make sure they are provided to the developers.

8. **Project management decisions that undermine secure software production:** Project management that places little or no emphasis on security, or worse that is driven by considerations that are diametrically opposed to security, can totally undermine the efforts of the software's creators to security-enhance their processes and produce secure software. For example, unrealistic development schedules that do not allow enough time to accommodate security checks and validations, including those associated with evaluation and impact analysis of components to be acquired, reused, or replaced, given that such components are likely to be released according to a variety of different schedules.

   Another major project management failure is lack of support and encouragement for developers to become security-knowledgeable, through education and training, so that they can recognize the aspects of their development choices that result in software that is vulnerable to exploit and compromise. Without such knowledge, developers may not even recognize the security implications of certain design and

implementation choices. Moreover, were project managers to obtain an appropriate level of security education/training, they would be less likely to take decisions that undermine software's security.

2.4. Other Relevant Considerations

2.4.1. Social Engineering Attacks

A class of attacks, known as "social engineering attacks" (examples: phishing, pharming) have become increasingly prevalent on the Internet. The objective of most of these attacks is to deceive authorized users into disclosing sensitive information to which the attacker, as an unauthorized party, is not entitled. For example, through a deceptive email, the attacker may persuade a user to reveal his/her user ID and password. The attacker might then use that information to masquerade as the authorized user, and get access to a software-intensive system. In this way, the attacker is able to violate the intent of the system's security policy, though not actually the policy itself: the attacker used a valid ID and password, which is likely all the policy requires-they just happened to be obtained through illicit means.

Violation of the intent of the system's security policy, while definitely a major security problem, does not represent either a violation of the system's security measures nor its security properties. The security measures operated correctly and predictably: they were given a valid user ID and password and they allowed access by the person who submitted that valid ID/password. The fact that the ID and password were obtained through social engineering was a personnel security issue: the valid user should have been trained not to be so trusting of the deceptive email.

In any case, no security properties of the software that implemented the system's identification and authentication, authorization, or access control functions were violated.

There are potential cases of social engineering that could lead to the compromise of software security properties. The objective of such attacks would most likely be to deceive or subvert a developer or administrator into performing the compromising acts (rather than the attacker performing those acts). For example, through social engineering a attacker might persuade a developer to embed malicious logic in the software base to which the developer has authorized write-access. Or the attacker might persuade an administrator to reconfigure the software in an insecure manner. In both examples, the attacker, through social engineering, transforms a trusted user into an insider threat.

By contrast, a social engineering attack wherein the attacker obtains the developer's password then uses it to access the configuration management system himself/herself would be comparable to the attack involving deceptive email, described earlier. And like the email attack, this attack would violate the intent of the system's security policy, but not the policy itself. User ID/password-based authentication systems have no way of recognizing whether the person entering the ID and password is in fact the person to whom the ID/password belong. The fact that the attacker goes on to exploit his/her system access to plant malicious logic in the source code base-with the result being a compromise the software's security properties-does not change the fact that the social engineering part of the attack was a personnel security issue, not a software security issue. The attack that compromises the software security would be the act of surreptitiously planting malicious code, not the masquerading by the attacker as an authorized developer. The use of biometrics for authentication is currently the only effective countermeasure against masquerading attacks like those described here.

It is important for developers to avoid the common mistake of confusing violations of the security policy that governs the use of a software-intensive system (or, in the case of social engineering, the violation of the intent of that policy) with compromises of the security/dependability properties of that system. As stated earlier, social engineering (phishing, pharming, etc.) is indeed a large and growing problem in cyberspace. But social engineering attacks do not directly threaten the dependability of software. Social engineering is a personnel security issue. All users, including developers, should be trained to recognize possible social engineering attack

patterns, and to be highly suspicious of any emails, phone calls, web pages, or other communications that contain one or more of those attack patterns.

2.4.2. Software Security vs. Information Security

The most significant difference between software security and information security (which, for our purposes, includes cyber security), lies in the inherent difference between software and information (which, in IT, is manifested as data). Information (which is, in its electronic form, data) is passive by nature: it cannot perform actions; it can only have actions performed upon it. (For data that has "active content" embedded within it, the reader should remember the data itself is passive even though it is being used as a delivery mechanism for software, i.e., the data and the software embedded in it remain distinct entities, each of which retains its own inherent properties.)

Moreover, how the data is created, and whether it is correct or not, has no effect on the ability to secure it. It will be possible to encrypt a document before sending it over network regardless of whether that document was created by a word processor, a desktop publishing program, or a text editor. The fact that information in a database is inaccurate does not make it any harder to protect that information from unauthorized access. A crucial difference between software and data that makes achieving security such a different proposition for each is that data does not become vulnerable because of how it was created. It becomes vulnerable only because of how it is stored or how it is transmitted. And though information security does extend to protecting the information systems that store, transmit, and process the information, the reason those systems must be protected is to ensure that the information cannot be compromised if the system is violated. Information system security measures focus exclusively on counteracting those threats to the system that if successful would compromise the information within the system.

Unlike information, software can exist simultaneously in two states: it exists as a passive file (or series of files) that contain the binary executable code stored on a file system (or on transportable media). Like other digital files, binary software files can be transmitted over networks. Theoretically, software in its passive file state is subject to the same attack patterns as other files, including data files, and may be protected against them by the same security protections. However, in actuality the value of software as a target is different than the value of data, and its accessibility requirements (and the security policy governing that accessibility) are different as well. Therefore, it possible and even likely that software in its passive file state will be subject to different attack patterns and require different protections (or at least different levels of protection) than data.

The second state of software is its active, or executing, state. An executing software component or software-intensive system can be seen as an "actor" to which operating system-level privileges must be authorized, in the same way such privileges must be authorized to human users. However this is where the analogy ends. Human users are not subject to the same threats as executing software.

Finally, there is a third aspect of software security that has no analogy in information security: the need to protect the ability of the software to perform its computational and memory management functions. These are functions that the software performs as software, rather than as a proxy for a human user.

Because of its multifaceted nature, software not only can be vulnerable because of the way in which it, in its passive file state, is stored or transmitted, but more significantly, because of the way in which it was created (developed). The vulnerabilities in executing software originate in the process used to create that software: the decisions made by its developers, the flaws they inadvertently or intentionally include in its specification and design, and the faults and other defects they intentionally or unintentionally include in its implemented code.

In summary, information needs to be secure because of what it is, and how it is acted upon by other entities. Software needs to be secure because of what it does, including how it acts upon other entities.

Because the *raison d'etre* of an information system is to process data, to an IA practitioner, if the data is not secure, the system (which is composed in large part of software) is also insecure. Not only that, the majority of countermeasures used to protect data are implemented by software: by the security controls and functions in operating systems, application software, and the software components of security systems such as firewalls and cryptosystems. For these reasons, it can be difficult for some people to understand the difference between the objectives of software security, information security, and information system security:

1. **Main objective of information security:** to protect information from unauthorized disclosure, modification, or deletion. The means by which this is achieved are the security functions and controls of the information system, many of which are implemented by software;

2. **Main objective of information system security:** to produce and deploy information systems that include all of the security controls and functions necessary to ensure that the information they handle will be protected (consistent with the objective of information security). This will entail protecting the security controls and functions against unauthorized modification or denial of service. Secure information systems engineering is considered at the architectural level. How the software components of the information system are developed is usually not a consideration of the IA/cyber security practitioners concerned with information systems security. This said, using secure software development practices will clearly improve the security of information systems by improving the security of their software components at a more integral level than information systems engineering addresses;

3. **Main objective of software security:** to produce software that will not be vulnerable to unauthorized modification or denial of service during its execution. Part of this will entail preventing the subversion of the software's artifacts during their development. If the software happens to be a component of an information system, using development practices that increase the likelihood that it the software will be secure should contribute to the ability of the information system to achieve its security objectives. Note that not all software systems handle information. Embedded software systems communicate not by passing information (in the form of data), but by exchanging electrical signals with their physical environment. Information security and information systems security are irrelevant for embedded systems. But software security is just as relevant for an embedded system as it is for an information system.

In summary, the focus of information and information system security is to protect information. The rationale behind protecting an information system is inextricably linked to the function that an information system is intended to perform, i.e., the processing of information. The reason an information system needs to be secure is to ensure that the information it processes will be secure.

By contrast with information and information system security, the focus of software security is to produce software that will not be vulnerable to subversion or denial of service during its execution, regardless of the function that it performs.

Obviously, there are significant overlaps among the key objectives of these three entities, and dependencies among the means by which those objectives are achieved. However, these overlaps and dependencies should not be misinterpreted to mean that, or that the security requirements all three entities will be identical, or that their key security objectives can all be achieved by the same means.

For software practitioners to be able to work effectively with IA and cyber security practitioners, they need to:

1. Recognize that there are a number of terms that are shared by both disciplines.

2. Understand that the meanings of these shared terms when used in an IA/cyber security context can often vary, sometimes only subtly, from their meanings in a software security context. These variances are mainly due to the issues discussed in Section 3 and above.

To help software practitioners gain this understanding, the "primers" in section 2.5.1 are suggested as good introductory material to information and cyber security terminology and concepts. These resources are categorized according to the general business sector for which they were authored (e.g., government, industry, academia, etc.). This list is meant to be representative, not comprehensive (and certainly not exhaustive).

2.5. References for this Section

CNSSI 4009, National IA Glossary (revised 2006)
http://www.cnss.gov/Assets/pdf/cnssi_4009.pdf

DoD Software Assurance Initiative (13 September 2005)
https://acc.dau.mil/CommunityBrowser.aspx?id=25749

NASA-STD-2201-93, Software Assurance Standard
http://satc.gsfc.nasa.gov/assure/astd.txt

NIST SAMATE portal
http://samate.nist.gov/index.php/Main_Page

DHS BuildSecurityIn portal
https://buildsecurityin.us-cert.gov/portal

OMG: A White Paper on Software Assurance (revised 28 November 2005)
http://adm.omg.org/SoftwareAssurance.pdf - or -
http://swa.omg.org/docs/softwareassurance.v3.pdf

Andrew Jaquith, @stake, Inc.: "The Security of Applications: Not All Are Created Equal" (February 2002)
http://www.atstake.com/research/reports/acrobat/atstake_app_unequal.pdf

Andrew Jaquith, @stake, Inc.: "The Security of Applications, Reloaded"
http://www.atstake.com/research/reports/acrobat/atstake_app_reloaded.pdf

CERT/CC, Carnegie Mellon University: "Overview of Incident and Vulnerability Trends"
http://www.cert.org/present/cert-overview-trends/

DHS U.S. Computer Emergency Response Team (US-CERT)
http://us-cert.gov

2.5.1. Introductory IA and Cyber Security Information

2.5.1.1. U.S. Government Resources

NSA: National Information Security (INFOSEC) Education and Training Program: Introduction to Information Assurance (IA) (1998)
http://security.isu.edu/ppt/shows/information_assurance.htm - *or* -
http://security.isu.edu/ppt/pdfppt/information_assurance.pdf

Committee on National Security Systems Instruction (CNSSI) 4009: National IA Glossary (revised 2006)
http://www.cnss.gov/Assets/pdf/cnssi_4009.pdf

U.S. Marine Corps Systems Command (MARCORSYSCOM) Command, Control, Communications, Computers, Intelligence Integration (C4II) Information Assurance Division: An IA Tutorial
http://www.marcorsyscom.usmc.mil/Sites/PMIA%20Documents/How%20Do%20I/2.%20What%20is%20IA%20How%20do%20I%20start/An%20IA%20Tutorial.ppt

MARCORSYSCOM C4II Information Assurance Division: Introduction to IA (March 2003)
http://www.marcorsyscom.usmc.mil/Sites/PMIA%20Documents/How%20Do%20I/2.%20What%20is%20IA%20How%20do%20I%20start/Introduction%20to%20IA%20dated%2011%20Mar%2003.ppt

NIST: Special Publication (SP) 800-12: *An Introduction to Computer Security: The NIST Handbook* (October 1995)
http://csrc.nist.gov/publications/nistpubs/800-12/800-12-html/index.html - *or* -
http://csrc.nist.gov/publications/nistpubs/800-12/handbook.pdf

Congressional Research Service: Creating a National Framework for Cybersecurity: An Analysis of Issues and Options (RL32777; 22 February 2005)
http://fpc.state.gov/documents/organization/43393.pdf

2.5.1.2. European Government Resources

Alain Esterle, Hanno Ranck and Burkard Schmitt, European Union Institute for Security Studies: Information Security: A New Challenge for the EU (Chaillot Paper No. 76; March 2005)
http://www.iss-eu.org/chaillot/chai76.pdf

Finnish Communications Regulatory Authority: Information Security Basics
http://www.ficora.fi/englanti/tietoturva/ttperusteet.htm

2.5.1.3. Research and Academic Sector Resources

National Academy of Sciences Computer Science and Telecommunications Board: *Cybersecurity Today and Tomorrow: Pay Now or Pay Later* (2002) *[excerpts]*
http://books.nap.edu/openbook/0309083125/html/index.html - or - http://books.nap.edu/html/cybersecurity/

Jacques S. Gansler and Hans Binnendijk, editors, National Defense University: Information Assurance: Trends in Vulnerabilities, Threats and Technologies-Working Paper (May 2003)
http://www.ndu.edu/ctnsp/IAverMay03.pdf

Computer Security (LV 142 A): Professor Mark Burgess' weekly lecture summaries
http://www.iwar.org.uk/comsec/resources/security-lecture/index.html

2.5.1.4. Industry (Including Professional Association) Resources

Information Systems Security Association (ISSA): Generally Accepted Information Security Principles (GAISP)
http://www.issa.org/gaisp/gaisp.html

Abe Usher's Information Assurance Browser *[runs only on Windows-based systems running Netscape or Internet Explorer]*
http://sharp-ideas.net/research/information_assurance_browser.html - *or* -
http://sharp-ideas.net/research/information_assurance_6.xml

Mike Chapple, Debra Shinder, Ed Tittel: TICSA Certification Information Security Basics *[excerpts from* TruSecure International Computer Security Association Certified Security Associate Training Guide *(Que Publishing, 2002)]*
http://www.informit.com/articles/article.asp?p=30077&rl=1

3. WHAT MAKES SOFTWARE SECURE?

A key objective of software assurance is to provide justifiable confidence that software is free of vulnerabilities, and that it not only functions in the intended manner, but that this "intended manner" will not compromise the security and other required properties of the software, its environment, or the information it handles.

3.1. Attack Resistance, Tolerance, and Resilience

In addition to trustworthiness, predictable execution, and conformance to requirements, to be considered secure, at the individual component and whole system levels, software must be attack-resistant or attack-tolerant, and at the whole system level it must also be attack-resilient. To achieve attack-resistance or attack-tolerance, both software components and whole software systems should:

1. **Be able to recognize attack patterns** in the input data or signals they receive from external entities (humans or processes);

2. **Be able to either:**

   - **Resist attack-patterned input.** Resisting attack patterns and external faults can be achieved by blocking input data streams that contain attack patterns and by terminating the software's connection with the hostile or faulty external entity; *or*

   - **Tolerate the failures that result from a successful attack or intentional external fault.** Tolerating failures that result from attack patterns or external faults means the software continues to operate in spite of those failures. Often such continued operation can only be achieved for critical processes: the software enters a soft failure mode in which non-critical processes are gracefully terminated and only critical processes are sustained (or, in some cases, required performance levels are ignored, so the software continues to operate but with lower throughput). Depending on the exception handling capabilities of the software, continued operation in soft failure mode may only be possible up to a certain threshold of degraded operation, after which the software may have to terminate operation completely (hard failure).

     What secure software must never do is simply crash. No hard failure should be allowed without first purging all data from temporary memory, if at all possible signaling users of the imminent failure, and only then safely releasing all resources and terminating the software process. This is what is meant by "failing safe", which will reduce the possibility of an attacker gaining read-access to sensitive data in temporary memory, or hijacking resources that were released in an unsafe way.

Attack-resilience, which is a form of *survivability,* has proved possible to achieve only at the whole system level. To achieve attack-resilience, software systems must:

3. **Be able to recover** from any failures that result from successful attacks on the software (including intentional external faults) by resuming operation at or above some predefined minimum acceptable level of service in the short term, then eventually recovering full service at the specified level of performance. The software system's recovery should occur as soon as the source of the attack has been isolated and blocked, and the resulting damage has been contained.

NOTE: The term "attack-resilient" is intended to be consistent with the definition of "computer system resilience" in American National Standard T1.523-2001, "Telecom Glossary 2000". Indeed, attack-resilience should be seen as a critical aspect of computer system resilience.

Note that, while the ability to tolerate and respond to an attack is highly desirable, there are often many difficulties in identifying a specific fault as the weakness that was targeted as a vulnerability. However, without such fault identification, it may not be possible to enact an appropriate response.

3.2. Properties of Secure Software

3.2.1. Attributes of Security as a Property of Software

Several lower-level properties may be seen as attributes of security as a software property:

1. **Availability:** The software must be operational and accessible to its intended, authorized users (humans and processes) whenever it is needed.

2. **Integrity:** The software must be protected from subversion. Subversion is achieved through unauthorized modifications by authorized entities, or any modifications by unauthorized entities. Such modifications may include overwriting, corruption, tampering, destruction, insertion of unintended (including malicious) logic, or deletion. Integrity must be preserved both during the software's development and during its execution;

3. **Confidentiality:** In the context of software security, confidentiality applies to the software itself rather than to the data it handles. Confidentiality for software means either its existence, its characteristics (including its relationships with its execution environment and its users), and/or its content are obscured or hidden from unauthorized entities, most often to prevent them from learning enough about it, e.g., through reverse engineering, to craft effective attacks against it;

Two additional properties commonly associated with human users are also required in software entities that act as users (e.g., proxy agents, Web services, peer processes). These properties are:

4. **Accountability:** All security-relevant actions of the software-as-user must be recorded and tracked, with attribution of responsibility. This tracking must be possible both while and after the recorded actions occur. The audit-related language in the security policy for the software system should indicate which actions are considered "security relevant";

5. **Non-repudiation:** Pertains to the ability to prevent the software-as-user from disproving or denying responsibility for actions it has performed. Non-repudiation measures were originally intended to ensure that users could not deny having sent or received email messages. However, the value of non-repudiation for activities other than message transmission is being increasingly recognized.

NOTE: These definitions security properties are based on the standard definitions of these terms found in Federal Information Processing Standard (FIPS) Publication 200, Minimum Security Requirements for Federal Information Systems*, but have been reflect the sometimes subtle distinctions between these properties when required in software, rather than in information or human users.*

The main goal of the software security practices described in this document is to achieve a software development process whereby:

- Exploitable faults and other weaknesses are avoided by well-intentioned developers;

- The likelihood is greatly reduced or eliminated that malicious developers can intentionally implant exploitable faults and weaknesses or malicious logic into the software;

- The software will be attack-resistant or attack-tolerant, and attack-resilient;

- The interactions among components within the software-intensive system, and between the system and external entities, do not contain exploitable weaknesses.

3.2.2. Security Properties of Software Interactions and Behavioral States

Four key factors affect how securely software interacts, both internally (e.g., one module or process receiving input from or providing output to another) and externally (e.g., application-level software receiving data from or providing data to its execution environment, or receiving data from/or providing data to another application-level component, e.g., a Web service, a client, a database). These factors are:

1. The interface mechanisms used to enable the software's internal and external interactions;

2. The processing model implemented for the whole software system will determine the way in which the system's components are structured, as well as what each component is responsible for doing in terms of its interaction with the system's other components. The nature of a component's responsibilities has implications for the security properties/attributes desired in the component (as well as the security functions in may need to perform).

   For example, in a Web-based system with extremely decentralized control of clients (including their execution environments) but at least somewhat centralized control of servers (and their environments), the overall processing model is characterized by having separate security requirements for clients and servers. By contrast, in a peer-to-peer model, all components would be expected to exhibit the same security properties.

   The different ways in which the processing model influences what security properties its components need to display is often due in large part to the nature of the technologies that are associated with that model. For example, in a Web processing model, there are a number of vulnerabilities in the communications protocols the model dictates be used for inter-component interactions. The components, then, must exhibit certain security properties to avoid being compromised via the intentional exploitation of vulnerabilities in the interface mechanisms they use;

3. How the software (including all of its components) was configured during installation, which will strongly influence the software's external interactions;

4. How vulnerabilities discovered during the software's operation were addressed (e.g., security patches).

In summary, the software's processing model imposes certain requirements for, and constraints on, how each component is able to ensure or contribute to ensuring its own secure behavior. The processing model influences whether a given component needs to demonstrate additional security properties to compensate for the lack of those properties in the interface mechanisms that the processing model requires it to use, and for the lack of security properties in the components with which it interacts because of the inherent nature of those components resulting from their role in the processing model.

The way in which a given component is configured within its environment will determine whether the need for security properties must be satisfied by the component itself, or whether the component can "delegate" the responsibility for providing those properties because of the component's ability to rely consistently and continuously on protections afforded to it by its environment.

3.3. Other Desirable Properties of Software and Their Impact on Security

There are properties of software which, while they do not directly make software secure, make it possible to *assure* that software is secure. These are:

- Dependability, which incorporates correctness, predictability, reliability, and safety, as well as security;
- Smallness, simplicity, and traceability.

3.3.1. Dependability and Security

In simplest terms, dependability is the property of software that ensures that the software always operates correctly.

It is not surprising that security as a property of software and dependability as a property of software both share a number of subordinate properties (or attributes). The most obvious, to security practitioners, are availability and integrity. However, according to Algirdas Avizienis *et al* in "Basic Concepts and Taxonomy of Dependable and Secure Computing", there are a number of other properties that are shared by dependability and security, including reliability, safety, survivability, maintainability, and fault tolerance. Further discussion of the properties of secure software appeared in Section 3.1.

To better understand the relationship between security, dependability, and their properties consider the nature of threats to security-and, by extension, dependability-and the outcomes if those threats are successful. Any successful threat to the dependability software results from a fault (i.e., the adjudged or hypothesized cause of an error or failure). According to Avizienis et al, software faults fall into three general categories:

1. **Development fault:** a type of weakness that originates during the software's development process, i.e., they are "built in" to the software.

2. **Physical fault:** a type of weakness that originates in a defect or anomaly in the hardware on which the software runs.

3. **External fault:** a type of weakness that originates outside of the software and its hardware platform. External faults may enter the software as part of user input, variables passed by the environment, messages received from other software, etc.

All faults, regardless of category, are either human-made or natural. Human-made faults may be intentional or unintentional. For purposes of this (and indeed any non-metaphysical) discussion, all natural faults are considered unintentional.

Intentional human-made faults may or may not be malicious. Non-malicious intentional faults often result from bad judgment. For example, a developer makes a tradeoff between performance and usability on the one hand and security on the other that results in a design decision that includes fault (a development fault).

Faults in software become a threat to dependability when they are active. Dormant faults (those that exist but do not result in errors or failures) are of concern because of their potential for causing problems if they become active. Active faults represent security threats under either (or both) of the following conditions:

1. The active fault is human-made, intentional, and malicious;
2. The active fault results in an error or failure that:
    - leaves the software vulnerable to the compromise of any of its required properties;
    - compromises a security property in the data the software processes, protects, or makes accessible.
    - compromises a security property in any of components of the software's execution environment, or in any external entity with which the software interacts.

Even natural faults can manifest as security threats. For example, consider a system in which the software runs on a separate physical platform from the application firewall appliance that is relied on to filter out potentially

dangerous input before it reaches the software. Now, imagine that a huge storm causes water seepage in the ceiling of the computer room that drips down on top of the firewall appliance. Due to faulty wiring, when the water hits it, the appliance short circuits and fails. The software's platform, by contrast, is positioned so that the dripping water doesn't touch it. In this scenario, it would be possible for the software to continue operating without the protection of its firewall: thus, a natural fault would have left the software vulnerable.

There is a desirable property of security that does not necessarily hold for dependability: confidentiality. In the context of security, confidentiality intended to help counteract "reconnaissance" attacks that attempt to discover as much as they can about the software in deployment and its environment so that the attacker can craft more effective attack patterns against the software. One form of reconnaissance attack is reverse engineering of binaries or byte code.

3.3.1.1. Correctness and Security

From the standpoint of quality, correctness is a critical attribute of software that should be consistently demonstrated under all anticipated operating conditions.

Several advocates for secure software engineering have suggested that good software engineering is all that is needed to ensure that the software produced will be free of exploitable faults and other weaknesses. There is a flaw in this thinking. Correctness under anticipated conditions is not enough to ensure that the software is secure because the conditions that surround the software when it comes under attack are very likely to be *un*anticipated. Most software specifications do not include explicit requirements for the software's functions to continue operating correctly under unanticipated conditions. Software engineering that focuses only on achieving correctness under anticipated conditions does not ensure that the software will remain correct under unanticipated conditions.

If explicit requirements for secure behavior are not specified, then requirements-driven engineering, which is used increasingly to increase the correctness of software, will do nothing to ensure that correct software is also secure. In requirements-driven engineering, correctness is assured by verifying that the software operates in strict accordance with its specified requirements. If the requirements are deficient, the software still may be deemed correct as long as it satisfies the requirements that do exist.

The requirements specified for the majority of software are limited to functional, interoperability, and performance requirements. Determining that such requirements have been satisfied will do nothing to ensure that the software will also behave securely even though it operates correctly. Unless a requirement exists for the software to contain a particular security property or attribute, verifying correctness will indicate nothing about security. A property or attribute that is not captured as a requirement will not be tested for: no effort will be made to discover whether the software contains that function or property.

It is much easier to specify and satisfy functional requirements stated in positive terms ("the software will perform such-and-such a function"). Security properties and attributes, however, are often non-functional ("this process must be non-bypassable"). Even "positively" stated requirements may reflect inherently negative concerns. For example, the requirement "if the software cannot handle a fault, the software must release all of its resources, then terminate execution" is in fact just a more positive way of stating the requirement that "a crash must not leave the software in an insecure state".

Moreover, it is possible to specify requirements for functions, interactions, and performance attributes that result in insecure software behavior. By the same token, it is possible to implement software that deviates from its functional, interoperability, and performance requirements (i.e., software that is, from a requirements engineering perspective, incorrect) without that software behaving insecurely.

Software that executes correctly under anticipated conditions, can not be considered secure when used in an operating environment with unanticipated conditions that lead to unpredictable behavior. However, it may be

possible to consider software that is *in*correct but completely predictable to be secure *if* the incorrect portions of the software do not manifest as vulnerabilities. Thus, it does not follow that correctness will necessarily help assure security, nor that incorrectness will necessarily manifest as insecurity. However, correctness in software is just as important a property as security. Neither property should ever have to be achieved at the expense of the other.

A number of development faults in software that can be exploited by attackers can be avoided by engineering for correctness. By reducing the total number of faults in software, the subset of those faults that are exploitable (i.e., are vulnerabilities) will be coincidentally reduced. However, complex vulnerabilities—those caused through series of interactions among components, for example—may not arise from component-level incorrectness. Each interaction may, in fact, be perfectly correct: it is only the sequence and combination of interactions that creates the vulnerability. Engineering for correctness will not eliminate such complex vulnerabilities.

For purposes of requirements-driven engineering, no requirement for a software function, interface, performance attribute, or any other attribute of the software should ever be deemed "correct" if that requirement can only be satisfied in a way that allows the software to behave insecurely, or which makes it impossible to determine or predict whether the software will behave securely or not. Better yet, every requirement should be specified in a way that ensures that the software will always and only behave securely when the requirement is satisfied.

3.3.1.1.1. "Small" Faults, Big Consequences

There is a conventional wisdom among many software developers that says faults that fall within a specified range of speculated impact ("size") can be tolerated, and allowed to remain in the software. This belief is based on the underlying assumption that "small faults have small consequences". In terms of faults with security implications, however, this conventional wisdom is wrong. Dr. Nancy Leveson suggests that failures in large software-intensive systems with significant human interaction will increasingly result from multiple minor faults, each insignificant by itself, but collectively placing the system into a vulnerable state.

Consider the impact of a "small" fault: An input function writes data to a buffer without first performing a bounds check on the data. This occurs in a program that runs with "root" privilege. If an attacker submits a very long string of input data that includes both malicious code and a return address pointer to that code, because the program does' not do bounds checking, the input will be accepted by the program and will overflow the stack buffer that receives it, allowing the malicious code to be loaded onto the program's execution stack, and overwriting the subroutine return address so that it points to that malicious code. When the subroutine terminates, the program will jump to the malicious code, which will be executed, operating with "root" privilege. This particular malicious code is written to call the system shell, thus enabling the attacker to take control of the system. (Even if the original program had' not operated at root privilege, the malicious code may have contained a privilege escalation exploit to gain that privilege).

"Small" fault, big consequence.

Obviously, when considering software security, the *perceived* size of a fault is not a reliable predictor of the magnitude of the *impact* of that fault. For this reason, the risks of every known fault—regardless of whether detected during design review, implementation, or testing—should be analyzed and mitigated or accepted by authoritative persons in the development organization.

For high assurance systems, there is no justification for tolerance of known faults. True software security is achievable only when all known aspects of the software are understood, and verified to be predictably correct. This includes verifying the correctness of the software's behavior under a wide variety of conditions, including hostile conditions. This means that software testing needs to include observing its behavior when:

- Attacks are launched against the software itself;
- The software's inputs or outputs (data files, arguments, signals, etc.) are compromised;

- The software's interfaces to other entities are compromised;
- The software's execution environment is attacked.

See Appendix G:G.7 for information on software security testing techniques and tools that will enable the observation of software behaviors and state changes in response to a wide variety of insecure and anomalous interactions.

3.3.1.2. Predictability and Security

Predictability means that the software's functionality, properties, and behaviors will always be what they are expected to be as long as the conditions under which the software operates (its environment, the inputs it receives) are also predictable. For dependable software, this means the software will never deviate from correct operation under anticipated conditions.

Software security extends predictability to the software's operation under unanticipated conditions, and specifically conditions in which attackers attempt to exploit faults in the software or its environment. The best way to ensure predictability of software under unanticipated conditions is to minimize the presence of faults and other weaknesses, to prevent the insertion of malicious logic, and to isolate the software to the greatest extent possible from unanticipated environmental conditions.

3.3.1.3. Reliability, Safety, and Security

> *NOTE: While this document's use of the term "reliability" is consistent with the definition of the term found in IEEE Standard 610.12-1990,* Standard Glossary of Software Engineering Terminology *(see Appendix A:A1.1)—which defines reliability as "the ability of a system or component to perform its required functions under stated conditions for a specified period of time"—it is more closely aligned with the definition of the term in the National Research Council's study* Trust in Cyberspace*—which defines reliability as "the capability of a computer, or information or telecommunications system, to perform consistently and precisely according to its specifications and design requirements, and to do so with high confidence". Software safety can be seen as the most "extreme" manifestation of reliability.*

The focus of reliability for software is on preserving predictable, correct execution despite the presence of unintentional faults and other weaknesses and unpredictable environment state changes. Software that is highly-reliable is often referred to as high-confidence software (implying that a high level of assurance of that reliability exists), or fault-tolerant software (implying that fault tolerance techniques were used to achieve the high level of reliability).

Software safety can be seen as an extreme case of reliability. The consequences, if reliability is not preserved in a safety-critical system, can be catastrophic: human life may be lost, or the sustainability of the environment may be compromised.

Software security extends the requirements of reliability and safety to the need to preserve predictable, correct execution even in the face of *intentional* faults/weaknesses and environmental state changes. It is this *intentionality* that makes the requirements of software security somewhat different than those of safety and reliability. Failures in a reliability or safety context are expected to be random and unpredictable. Failures in a security context, by contrast, result from human effort (direct, or through the agency of malicious code). Attackers tend to be persistent, and once they successfully exploit a vulnerability, they tend to repeat that exploit over and over—against the same system or against another system that is suspected or known to contain the same vulnerability—as long as that vulnerability is present and the outcome of the attack remains satisfactory.

Until recently, many software reliability and safety practitioners have not concerned themselves with software security issues. This is in part because many of the systems they have worked on have been embedded systems that, heretofore at least, did not interact directly with human users, that communicated not via messages and data

passing but through signaling, and were not exposed to networks. Unfortunately, the increasing sophistication of embedded software means that none of these assumptions are necessarily true any longer. Embedded systems are increasingly resembling other software systems: they have user interfaces (e.g., cell phones with browser-like interfaces), they receive and transmit data (e.g., automobile controllers that are addressable by "OnStar" type monitoring and maintenance systems; human ID implants that transmit location and physiological status via Web-enabled wireless network connections), and they are exposed to networks (e.g., SCADA systems that can be remotely reset via dialup and even Internet connections; robotic surgical equipment that can be controlled by via a satellite network link by a doctor hundreds or even thousands of miles away).

Safety-critical software, including software traditionally used in closed systems such as automobiles and weapons systems were originally specified and designed with only tightly constrained and controlled human interfaces or no direct human interfaces at all. Security considerations were seldom if ever included in their specification and design. Because security faults are seldom modeled during safety engineering, they are likely to be overlooked during code safety inspections and fault injections.

And yet, such software is increasingly being made network-addressable. Cost-reduction imperatives are even driving some organizations to expose their safety-critical systems to highly exposed public networks, as is the case with an increasing number of SCADA systems that are being remotely administered via the Internet. The physical constraints of SCADA systems (which are typically embedded systems with simple operating systems and very little memory) would make it very costly to upgrade their numerous components. Developers of these and other embedded systems have some very difficult security trade-offs to consider as their systems become increasingly exposed to networks and the new threats that go along with that exposure.

Even if such software is warranted free of traditional safety faults, it may still contain *security* weaknesses that can be exploited through network-accessible interfaces that were never considered when the software was designed. Thus, though individual components may be free of faults, the software will no longer behave appropriately when exposed to external faults originating from the new networked interfaces, faults such as repeated changes to the environmental conditions surrounding the software—changes typical of those associated with attacks on environment-level components.

Another concern for safety engineers is whether software can be deemed safe when its engineering overlooks dormant faults that may not occur accidentally, but which may be "planted" by malicious developers. Were the exploitation of such a fault in the software during its execution turn it from dormant to active, resulting in a series of errors that caused the safety-critical software to fail, would the damage in terms of lives lost be any less catastrophic because the failure was not accidental?

Software that has contains even a single exploitable dormant fault should not be considered safe.

Fortunately, security is increasingly being seen as a requirement for safety-critical software, and software security practices are being added to safety engineering. In fact, several of the methodologies and tools introduced in Section 4 originated in the software safety community.

3.3.2. Smallness, Simplicity, Traceability, and Security

Software that satisfies its requirements through simple functions that are implemented in the smallest amount of code that is practical, with process flows and data flows that are easily followed, will be easier to comprehend and maintain. The fewer the dependencies in the software, particularly dependencies on environment-level entities, the easier it will be to implement effective failure detection and to reduce the exposure of vulnerabilities (i.e., what is termed the "attack surface" in Microsoft Threat Modeling).

Smallness and simplicity should be properties not only of the software's implementation but of its design, as they will make it easier for reviewers to discover design flaws that could manifest as exploitable weaknesses in the implementation.

Traceability will enable the same reviewers to ensure that the design satisfies the specified security requirements, and that the implementation does not deviate from the secure design. Moreover, traceability provides a firm basis on which to define security test cases.

3.4. Secure Architecture *versus* Security Architecture

Note that we refer to a *secure* architecture rather than a *security* architecture. The traditional security architecture deals with system-level issues such as security perimeters, security zones, access control, and authorization. The original purpose of the separate security architecture was to compartmentalize the system to simplify making assumptions about threats, networks, hardware, users, etc.

With the increased connectivity and integration of systems, this purpose is becoming increasingly difficult to accomplish. The ideal situation would be abandonment of the notion of a separate "security architecture", and instead recognizing that security should be deeply embedded within the overall software and system architectures.

The main architectural security issue for software is how well its architecture satisfies its specified security requirements, both functional and non-functional. The goal is not to create a separate security architecture, but to verify that the overall software architecture displays all necessary security properties and attributes and includes all necessary security functionalities.

Modeling the secure software architecture should encompass the software's own components and all security constraints and protections afforded to the software by its execution environment (e.g., by trusted processor modules [TPMs], security microkernels, virtual machines, sandboxes, access controls, "code security", etc., all of which are discussed in Appendix G:G.3.3.1, with further information on security microkernels in Appendix I).

The hardest part of developing a secure architecture for software is the trade-off analysis that must take place. An architectural countermeasure may reduce the risk from one vulnerability, but may increase (or even introduce) that of another. Moreover, because the threat environment that surrounds software in deployment is constantly changing, what appears to be secure software today may become vulnerable in the future due to the appearance of a new attack technique. Costs alone force software engineers to make decisions about which vulnerabilities have highest priority when it comes to risk mitigation.

In large systems assembled from multiple components, it is always possible that an unanticipated emergent behavior can arise even when the behaviors of all of the individual components are correct. The software engineer is likely to find it difficult to predict emergent system behavior during normal operations, let alone under hostile or otherwise adverse operating conditions.

3.4.1. System Security *versus* Software Security

In both industry and government, system security is generally defined at the architectural level, and uses a "secure the perimeter" approach to prevent malicious actors and input from crossing the boundary and entering the system from outside. The premise of the "secure the perimeter" approach is that most of the system components within the boundary are themselves incapable of resisting, withstanding, or recovering from attacks.

Traditionally, system security has been achieved almost exclusively through use of a combination of network and operating system layer mechanisms, controls, and protections. More recently, application security measures have been added that extend to the application layer the same types of mechanisms, controls, and protections found at the network and operating system layers. The resulting combination of security measures at the system architecture, network protocol, and application levels results in layered protection referred to as "defense in depth".

System security measures are both preventive and reactive. The preventive measures include firewalls and filters, intrusion detection systems, virus scanners, trending and monitoring of network traffic. The objective of all of

these mechanisms is to block input that is suspected to contain attack patterns or signatures, and to prevent access or connection to the system by unknown or suspicious actors. The reactive measures include mobile code containment, malicious code containment and eradication, patching (location and correction of known security vulnerabilities usually after they already have been exploited), vulnerability scanning, and penetration testing.

By contrast with system security, which focuses on protecting the system's already-codified operational capabilities and assets, the basic premise of *software* security is that it is not a feature that can simply be "added on" after all the of the software's other features have been codified. This is because many security exploits against software target defects in the software's design or code. Moreover, software's emergent security properties must be identified as security requirements from the software's conception. Security requirements for software will, of course, include requirements for security functions (i.e., the use applied cryptography). But they must also include the software's security properties and behavioral/functional constraints.

3.4.2. "Application Security" and Software Security

A growing industry has emerged to address the need for *application security*. Currently, the application security measures being produced by this industry are mostly reactive (post implementation) system security techniques that focus on:

1. Establishing a protective boundary around the application that can recognize and either block or filter input that contains recognized attack patterns;

2. Constraining the extent and impact of damage that might result from the exploitation of a vulnerability in the application;

3. Discovering points of vulnerability in the implemented application so as to help developers and administrators identify necessary countermeasures.

Reactive application security measures are often similar to techniques and tools used for securing networks, operating systems, and middleware services (e.g., database management systems, Web servers). They include such things as vulnerability scanners, intrusion detection tools, and firewalls or security gateways. Often, these measures are intended to strengthen the boundaries *around* the application in order to protect against exploitation of for the vulnerabilities in the application.

In some cases, application security measures are applied as stopgaps for from-scratch application software until a security patch or new version is released. In other cases, application security measures provide ongoing defense in depth to counter vulnerabilities in the application. In software systems that contain acquired or reused (commercial, government off the shelf, open source, shareware, freeware, or legacy) binary components, application security techniques and tools may be only cost effective countermeasure for vulnerabilities in those components.

Application security as typically practiced today provides few if any techniques and tools that aid in the developer in producing software that has very few vulnerabilities in the first place. A software security perspective, by contrast, not only incorporates the protective, post-implementation techniques, but addresses the need to specify, design, and implement the application so that its attack surface (i.e., extent of exposure of its vulnerabilities) is minimized.

The focus of this document is on software security. The main message of this document is that a disciplined, repeatable security-enhanced development process should be instituted so that application security measures are used because they are determined in the design process to be the best approach to solving a software security problem, not because they are the only possible approach after the software is deployed.

This said, using systems engineering approaches can be helpful to further protect securely-engineered software in deployment by reducing the software's exposure to threats in various operational environments. These measures

may be particularly useful for reducing risk for software, such as commercial and open source software, that is intended to be deployable in a wide variety of threat environments and operational contexts. APPENDIX G:G.3.3 discusses some systems engineering approaches to defense in depth protection of software in deployment.

3.5. Security of Software Systems Built from Components

Because security is an emergent property, it is impossible to predict accurately the security of a software system integrated/assembled from a set of software components (or modules, or programs) by observing the individual behaviors of those components in isolation, though static analysis techniques can contribute to the understanding of inter-component interactions and resultant component behaviors.

The security of a component assembly or integrated system emerges from the behaviors of all of its components as they interact during the system's execution. Even if it can be determined that an individual component is inherently secure, that determination may reveal very little about the security of that component as it operates in combination with the other components.

Predicting the properties of a given system component may not be possible, even when it is fully understood in isolation. This is due in part to the difficulty of identifying architectural mismatches, and associated property mismatches, among the different components of the system. It is harder still to determine how these mismatches might affect the security of the system as a whole. Most developers do not have the skills or techniques needed to determine which properties need to be disclosed by each component in a particular combination of components. So, while a given software component may be considered secure in a specific context, it does not necessarily hold that the same software component will still be secure in a different context, e.g., when interacting with a different set of components. Further compounding the problem is the difficulty of predicting inter-component interactions in a dynamic environment, or even in a static environment when a larger software system is under consideration. Such predictions are seldom accurate.

Determining whether the system that contains a given component, module, or program is secure requires an analysis of how that component/module/program is used in the system, and how the system as a whole will mitigate the impact of any compromise of its individual components that may arise from a successful attack on those components or on the interfaces between them. Risks of insecurity can be reduced through:

1. Vetting all acquired or reused and from-scratch components prior to acceptance and integration into the whole system;

2. Examining interfaces, observation of instances of trust relationships, and implementing wrappers when needed;

3. Security testing of the system as a whole. If source code is unavailable, the tester should execute as wide a variety of binary object ("black box") security tests as possible.

Some specific security concerns associated with extensive use of acquired or reused software are:

- Establishing the security-relevant attributes and assurance levels of candidate software products;

- Reliable disclosure of each candidate product's properties, attributes, and functionalities to other components of the system;

- Resolving mismatches between the properties and functionalities of different components;

- Establishing aggregate assurance for software systems integrated from components that have different properties and security assurance levels;

- Difficulty predicting behaviors of individual components, interactions among components, and interactions between application-level and environment-level components in dynamic environments, such as virtual machines, in which many environmental details are not defined until runtime;

- Difficulty determining assurance levels of acquired or reused software that can be reconfigured after the software goes into operational production;

- In the case of SOUP components, inability to infer anything about the likely trustworthiness of the component based on knowledge of its development process or supplier reputation.

NOTE: NIST SP 800-33, Underlying Technical Models for Information Technology Security *defines "assurance" as "grounds for confidence that the security objectives are met and encompasses both correct and sufficient security capabilities." For software, this definition must necessarily be amended to read "…both correct and sufficient security capabilities* and properties*".*

The most common rationale for favoring use of acquired or reused software over from-scratch development of software is that from-scratch software is more expensive to develop and maintain. However, when security, fault tolerance, safety, etc., are critical properties, this rationale does not necessarily hold. The cost of from-scratch development may, in fact, be lower over the software's lifetime than the costs associated with the numerous ongoing risk assessments, safeguards and countermeasures that need to be implemented and maintained to mitigate the security vulnerabilities that are prevalent in most acquired or reused software.

In some instances, the problems associated with acquired or reused software can be addressed by techniques such as security wrappers or an application proxy. These techniques can add to the cost and complexity of the system, even though the use of acquired or reused components was meant to reduce cost. Security wrappers are a prime example. In some cases, it is not be possible within the constraints of a development project's schedule and budget to craft effective technical countermeasures against all of the security vulnerabilities in acquired or reused software. The other options are to accept the risk of using insecure software components—an option that is unacceptable if that risk is too high—or to forego using acquired or reused software, and develop a secure alternative from scratch instead. Before committing to use any acquired or reused software, a risk analysis to the overall system's security properties should be performed that considers all of these different options and their associated full-life cycle costs.

When open source software is used, the developer is able to review, and if necessary (and possible within the constraints of the component's open source license) rewrite any problematic sections of the open source code. However, as with all "white box" components, a security code review is limited in its ability to help the developer predict how the white box components will behave when integrated or assembled with black box (binary) components for which code review is not possible (except through the difficult process of reverse engineering). Note also that the modification of open source software reduces as the benefits of its being acquired because the modifications have to be treated as "from scratch" code unless those modifications are fed back and incorporated into the publicly-released open source code base. This is also true of proprietary or legacy source code obtained under non-disclosure agreement or code escrow agreement.

3.5.1. Development Challenges When Using Acquired or Reused Components

NOTE: See also APPENDIX G:G.5 on secure component assembly.

The shift of software, and particularly application, development away from from-scratch coding to integration or assembly of acquired or reused software programs, modules, or components requires concomitant shifts in the emphases and scheduling of several phases of the development life cycle. For secure integration/assembly of acquired or reused components to be possible, the system must have a robustly secure architecture, and the components selected must be thoroughly vetted for their security properties, secure behaviors, and vulnerabilities.

Extensive use of an acquired and reused components approach does not remove the requirement for sound engineering practices. The traditional view of the software development life cycle beginning with requirements specification and proceeding, in linear (though not necessarily "waterfall") progression through architecture and high-level design, detailed design, implementation, testing, deployment, and maintenance, is likely to be unrealistic.

A disciplined spiral or iterative development methodology will better accommodate the necessary evolutionary exploration of the highest risk components and interfaces, and the continuous visibility of the software system's critical security properties. Using executable representations of the software system, regenerated frequently throughout its iterative life cycle, can help developers continuously refine, mature, and make more secure the architecture of the software system.

3.5.1.1. Security Issues Associated with Technological Precommitments

Increasingly, organizations make project-wide and even organization-wide commitments to specific technologies, suppliers, or individual products without considering the security impact of such blanket technological precommitments on the ability to accurately specify and satisfy requirements in the systems that are then constrained to using those precommitted technologies and products.

Technological precommitments, while they may achieve better interoperability and economy of scale across the organization, must be examined thoroughly (using data from a risk analysis) to ensure that the system satisfies its security and other requirements. The requirements specification process for a software system that must submit to technological precommitments should be iterative. The first iteration should ignore such precommitments and capture the requirements, including the security requirements, based on the user's needs statement, organizational mission, the system's threat model, and any governing policy and standards mandates (including guidelines and practices).

For example, the mobile code technology policy originally stated in the Assistant Secretary of Defense for Command, Control, Communications and Intelligence (C3I)'s 7 November 2000 Memorandum, "Policy Guidance for Use of Mobile Code Technologies in Department of Defense (DoD) Information Systems", and reiterated within DoD Instruction 8500.2, *Information Assurance (IA) Implementation* (2 June 2003) will necessarily limit the types of mobile code technologies that can be used in developing DoD Web applications. A comparable policy is included in Director of Central Intelligence Directive 6/3, *Protecting Sensitive Compartmented Information within Information Systems—Manual* (24 May 2000).

In some cases, business requirements (e.g., the need to provide users with remote access to required information) force technology choices. This is already happening in the specification of Web services and SOAs.

Regardless of the rationale behind technological precommitments, they should be reviewed to determine whether:

1. Any particular requirements should be rewritten to ensure they can be satisfied within the constraints imposed by those precommitments;

2. Any additional requirements need to be added to mitigate any known vulnerabilities that use of the precommitted technologies/products may introduce.

The architecture and design of the system, then, needs to incorporate any constraints necessary to ensure that vulnerabilities in a precommitted technology or product are not able to negatively affect the security of the system in which that technology/product must be used.

If a precommitted technology or product proves to be too insecure to enable its vulnerabilities to be adequately mitigated when it is used in the system under development, the risk analyses performed through the system's development life cycle can be presented as evidence in making a persuasive case for waiving precommitment in the case of this particular system.

3.5.1.2. Security Concerns with Outsourced and Offshore-Developed Software

A major security challenge associated with acquisition of commercial software, open source software, and other software of unknown pedigree (SOUP) is how to determine that such software has not been developed by criminal elements or entities hostile to the U.S. government. A similar security concern arises with the acquisition of contracted development services: how to establish and sustain tight security controls on outsourced software development projects. In both cases, the objective is inadvertently accepting "acquired"-software that contains embedded malicious logic, intentionally-planted vulnerabilities, or subversive features that could later be exploited by U.S. adversaries. Appendix F includes excerpts from U.S. government reports that discuss the main perceived security issues for government acquirers of offshore-developed commercial software and outsourced software development services.

An equally significant problem arises when contracted work is done by geographically dispersed developers. The problem is not so much the likelihood of malicious code insertion but the difficulty of defining and enforcing a secure architecture consistently across all parts of the development team. These issues often do not become apparent until the various components and subsystems developed at different locations are collected and put together during the software's integration phase.

Faced with increasing outsourcing by U.S. commercial software firms to "offshore" (non-U.S.) developers, including developers in countries whose governments may have neutral or somewhat hostile relations with the U.S. government, DHS and DoD (assisted by the Intelligence Community) have undertaken software assurance-related activities that focus on the problem of software pedigree, i.e., the ability to determine the true originator (and nature of that originator) of software. These activities also include efforts to mitigate the potential risks posed by hostile foreign originators of software. The pedigree concern is not limited to "offshoring" however. Increased hiring by firms operating in the U.S. of non-U.S. citizens (i.e., "green card holders") from hostile foreign countries represents a potential threat to the security of U.S.-sourced software.

In their FY2006 Research, Development, Technology, and Engineering (RDT&E) Budget Item Justification, dated February 2005, the ASD(NII) Information Systems Security Program (ISSP) listed among its accomplishments in Fiscal Year 2004 the following:

> *Initiated a major Software Assurance study to develop processes and structures to mitigate the risks of malicious code and other threats introduced into information technology products (from foreign intelligence sources, other adversaries and competitors influencing, infiltrating and becoming technology vendors of information technology products and services (both foreign or domestic) or from intentional or unintentional changes to software by individuals).*

This study led to the establishment of a Software Assurance "Tiger Team", co-sponsored by the Under Secretary of Defense for Acquisition, Technology and Logistics (USD[AT&L]) and the Assistant Secretary of Defense for Network and Information Integration (ASD[NII]). Recognizing that it would be impossible to eliminate foreign elements from the DoD's software supply chain, the Tiger Team initiated a program of interrelated acquisition, intelligence, and systems engineering activities designed to address the security and assurance concerns associated with DoD acquisition and use of commercial (and to some extent open source) software and outsourced development services, particularly for software destined to become part of mission-critical and other high-consequence DoD systems. A particular focus of the Software Assurance Tiger Team's efforts is minimizing the risk posed by offshore software development and (to a lesser extent) potentially-hostile foreign developers within U.S.-based software firms.

In early 2006, the Defense Science Board initiated a Task Force on Software Assurance, chaired by Dr. Bob Lucky and co-sponsored by the USD(AT&L), the Assistant Secretary of Defense for Network and Information Integration ASD(NII), and the Commander of the U.S. Strategic Command (USSTRATCOM). The purpose of the Task Force is specifically to assess the risk the DoD runs as a result of foreign influence on its software, and to suggest technology and other measures to mitigate that risk.

3.5.2. Issues Associated with Fault Analysis of Component-Based Systems

Dr. Nancy Leveson in "A Systems-Theoretic Approach to Safety in Software-Intensive Systems" (IEEE Transactions on Dependable and Secure Computing (Vol. 1 No. 1, January-March 2004) has studied issues of fault analysis for safety-critical systems, and has made some observations about the limitations of such analyses for systems assembled from multiple components. These observations are directly relevant for security-critical systems, and thus are summarized here.

There is some evidence that fault analysis techniques must change if they are to be effective in analyzing the security of complex software-intensive systems. The analysis of a physical system often exploits the fact that such a system can be decomposed into parts, the parts examined separately, and the system behavior predicted from the behavior of its individual components. The validity of such a physical fault analysis assumes that:

1. The components are not subject to feedback loops or non-linear interactions;

2. The behaviors of the components are the same when each is examined alone as when they are all playing their role in the whole system;

3. The principles governing the assembly of the components into the whole system are straightforward. The interactions among the individual components are simple enough that they can be considered separately from the behavior of the system that contains them.

These assumptions fail quickly for complex software-intensive systems. The assumption for almost all causal analysis for engineered systems today is a model of accidents (the safety corollary of security compromises) that assumes they result from a chain of failures and human errors. From an observed error, the analysis backward through the chain eventually stops at an event that is designated as the cause. A root cause selected from the chain of events usually has one or more of the following characteristics:

1. It represents a type of event that is familiar and thus easily acceptable as an explanation for the accident;
2. It is a deviation from a standard;
3. It is the first event in the backward chain for which a "cure" is known;
4. It is politically acceptable as the identified cause.

Event-based models of accidents, with their relatively simple cause-effect links, were created in an era of mechanical systems and then adapted for electro-mechanical systems. The use of software in engineered systems has removed many of the physical constraints that limit complexity and has allowed engineers to incorporate greatly increased complexity and coupling in systems containing large numbers of dynamically interacting components. In the simpler systems of the past, where all the interactions between components could be predicted and handled, component failure was the primary cause of accidents. In today's complex systems, made possible by the use of software, this is no longer the case.

The same applies to security and other system properties: While some vulnerabilities may be related to a single component only, a more interesting class of vulnerability emerges in the interactions among multiple system components. Vulnerabilities of this type are system vulnerabilities and are much more difficult to locate and predict. Leveson has analyzed a number of safety failures from a systems perspective. Her analysis suggests that multiple errors involving the interdependencies among computing components and among those components and human operations contributed to the eventual failures. Simple fault analysis will be of very limited use in detecting these types of complex errors.

3.6. References for this Section

Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr in their article "Basic Concepts and Taxonomy of Dependable and Secure Computing" (*IEEE Transactions on Dependable and Secure Computing,* Vol. 1, No. 1, January-March 2004)

Algirdas Avizienis, Jean-Claude Laprie, and Brian Randell: "Dependability and its Threats: A Taxonomy" (*Building the Information Society: Proceedings of the IFIP 18th World Computer Congress,* August 2004)
http://rodin.cs.ncl.ac.uk/Publications/avizienis.pdf

Technical Cooperation Programme Joint Systems and Analysis Group Technical Panel 4 (JSA-TP4): "Systems Engineering for Defence Modernisation—Systems of Systems, Simulation Based Acquisition and Integrated Digital Environments" (TR-JSA-TP4-1-2001; Version 1, January 2001); specifically see section on "Military SoS"
http://www.dtic.mil/ttcp/jsatp4syseng4defmod.doc

American National Standard T1.523-2001, "Telecom Glossary 2000"
http://www.atis.org/tg2k/t1g2k.html

International Organisation for Standardisation/International Electrotechnical Committee (ISO/IEC) Standard 11179, "Information Technology—Metadata Registries (MDR)"
http://isotc.iso.org/livelink/livelink/fetch/2000/2489/Ittf_Home/PubliclyAvailableStandards.htm

NIST: "Appendix A, Terms and Definitions" in Federal Information Processing Standard (FIPS) 200, *Minimum Security Requirements for Federal Information Systems* (March 2006)
http://csrc.nist.gov/publications/fips/fips200/FIPS-200-final-march.pdf

Fabio Boschetti, *et al*: "Defining and Detecting Emergence in Complex Networks" in *Lecture Notes in Computer Science,* Vol. 3684 (Heildeberg: Springer Berlin, 2005). *Search at:*
http://www.springerlink.com/app/home/main.asp

National Research Council Committee on Information Systems Trustworthiness: *Trust In Cyberspace* (1999)
http://newton.nap.edu/html/trust/

NIST SP 500-209, "Software Error Analysis" (March 1993)
http://hissa.nist.gov/HHRFdata/Artifacts/ITLdoc/209/error.htm

Bala Subramaniam: "Effective Software Defect Tracking" (*CrossTalk: The Journal of Defense Software Engineering*, April 1999)
http://www.stsc.hill.af.mil/crosstalk/1999/04/subramaniam.asp - *or* -
http://www.stsc.hill.af.mil/crosstalk/1999/04/subramaniam.pdf

IEEE Std. 1228-1994, Standard for Software Safety Plans (August 1994)
http://ieeexplore.ieee.org/xpl/standardstoc.jsp?isnumber=9808&isYear=1994

NIST: "Section 3.2, Achieving Security Objectives" in SP 800-33, *Underlying Technical Models for Information Technology Security* (December 2001)
http://csrc.nist.gov/publications/nistpubs/800-33/sp800-33.pdf

Assistant Secretary of Defense for Command, Control, Communications and Intelligence (ASD C3I): "Policy Guidance for Use of Mobile Code Technologies in Department of Defense (DoD) Information Systems" (7 November 2000)
http://www.defenselink.mil/nii/org/cio/doc/mobile-code11-7-00.html

Department of Defense Instruction (DODI) 8500.2, "Information Assurance (IA) Implementation" (2 June 2003)
http://www.dtic.mil/whs/directives/corres/html/85002.htm

Director of Central Intelligence Directive (DCID) 6/3, "Protecting Sensitive Compartmented Information within Information Systems—Policy and Manual"
http://www.fas.org/irp/offdocs/DCID_6-3_20Policy.htm - *and* -
http://www.fas.org/irp/offdocs/DCID_6-3_20Manual.htm

4. SECURITY IN THE SOFTWARE DEVELOPMENT LIFE CYCLE

In their article "Security Guidelines: .NET Framework 2.0", J.D. Meier *et al* assert

> *"To design, build, and deploy secure applications, you must integrate security into your application development life cycle and adapt your current software engineering practices and methodologies to include specific security-related activities."*

As the experiences of software development organizations like Microsoft have begun to demonstrate, instituting sound security practices throughout the development life cycle does produce demonstrably less vulnerable software (see Section 4.3.2.1 for information on Microsoft's experiences with its Security Development Lifecycle).

Ideally, the development organization will establish security standards that developers must conform to during development, as well as security criteria that must be satisfied during the acquisition of commercial software or the selection of reusable or open source software. Section 5 provides a sample of the many methods, tools, and techniques used by various organizations to improve security within the development process.

4.1. Identifying, Analyzing, and Managing Security Risk Throughout the Software Life Cycle

First, some definitions:

- Threat: an actor or event;
- Actor: An entity that may violate one or more security properties of an asset. Actors may be external or internal to the organization that owns the asset;
- Asset: something of value to the organization (e.g., information in electronic or physical form, software systems, employees);
- Risk: an expectation of loss expressed in terms of the likelihood that a particular threat will exploit a particular vulnerability with a (harmful) result in terms of impact on the business or mission. Impact may be expressed in terms of cost, loss of business/mission effectiveness, loss of life, etc.;
- Outcome: the immediate result of violating the security property of an asset (e.g., modification, deletion, denial of service).

It will be easier to produce software that is secure if risk management activities and checkpoints are integrated throughout the development life cycle, from prior to its inception through the software's decommissioning.

An initial system-level risk assessment is done before the software development process begins. This initial assessment focuses on the business assets, threats, likelihood of risks, and their potential business impacts. The system risk assessment provides input to the requirements specification and defines the context for the security aspects of the software's architecture and design.

The software's architectural risk assessment then refines the system risk assessment by analyzing how well the software address the system risks, suggesting mitigation strategies, and identifying additional risks (technological, usage, etc) that are added by the software architecture. Threats in this context are captured by a profile of the most likely attackers, including information on their likely resources, skills, motivations, etc.

Attacks on software can be viewed as "threat vectors" that can lead to compromise of the software itself, or to the exploitation of one of the software's weaknesses to compromise either the data the software handles, or one or more of the external entities with which it interacts. The main objectives of security-oriented risk analysis are to:

1. Identify, or adjust the baseline list, all potential threats to the software, and rank them according to likelihood of exploitation, and severity and magnitude of impact on the business or mission. The potential of each identified vulnerability to be exploited to compromise the software or any entity with which the software interacts should be captured in the risk assessment;

2. Identify, or adjust the baseline list, any residual vulnerabilities in the software's security properties and attributes, and associated functions, and identify the changes to the software requirements, design, or implementation that are needed to eliminate those vulnerabilities;

3. Estimate the cost of implementing each identified change compared to the cost of the impact to the business or mission user if the vulnerability were to be exploited.

The results of the risk analysis 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 of the software. The software's specified security properties/attributes and associated functionalities should be directly aimed at either eliminating the vulnerabilities identified in the risk assessment or minimizing their exploitability.

4.1.1. Risk Management Considerations

Risk analysis should be repeated iteratively throughout the software's life cycle to maintain the expectation level of security from the initial risk analysis. Additional security requirements discovered during the design, implementation, and testing phases should be incorporated back into the system's requirements specification. When found during testing, weaknesses that manifest as vulnerabilities should be analyzed to determine whether they originated with the system's requirements, design, or implementation. The root causes should be corrected to remove or mitigate the risk associated with that vulnerability. Risk analysis can also help prioritize security requirements to focus resources on those functions that introduce the greatest vulnerabilities to the system as a whole. Section 4.1.1 provides information on some significant software security risk analysis methodologies and supporting toolsets.

Some risks are avoidable and can be eliminated, for example by changing the software system's design or the components or configuration of its operating environment. However, there will likely be some significant risks that cannot be eliminated, but which must instead be anticipated so that the resulting failures can either tolerated by the system, or the system must be engineered to be able to contain and isolate and quickly recover from the resulting damage. The system's security requirements must be specified to include the need to address all such unavoidable risks.

A combination of risk analysis methods can be applied to software throughout the development life cycle. After initial risk analysis, further analysis can determine which components of the software may 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 life cycle 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 analysis:

1. **Mission or business purpose,** as captured in its needs statement;
2. **Objectives,** as captured in its requirements specification;
3. **Structure and interfaces,** as depicted in its architecture and design specifications;

4. **Behaviors,** as revealed through its security testing. A key is to look at the behaviors generated during testing and see if those behaviors could enable or cause a security compromise.

After the initial risk analysis is performed, all subsequent software development life cycle activities will have the objective of minimizing and managing those risks, specifically by iteratively re-verifying that the risks have, in fact, been correctly understood, and their required eliminations or mitigations have been adequately stated. The outcome of these re-verifications will be refining the system's security requirements describing the specific security properties and mechanisms that must be incorporated into the system's design, and implemented in the system itself, to mitigate the assessed security risks to an acceptable level.

Even after it has gone into production, the system will undergo periodic risk analyses to 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 accumulated over time or to address new threats.

4.1.2. Software Risk Assessment and Threat Modeling Methodologies

This section examines a part of the risk-driven development process: software risk assessment, which has also come to be known as "threat modeling", as this is the term chosen by Microsoft to name their software risk assessment process. To know what mitigation steps must be taken, the architecture team must understand the threats and the risks associated with those threats.

Establishing the risk assessment, or threat model, for a software system is a good first step. However further steps must be taken to avoid complacency. Every risk assessment/threat model is a "living document" which should be revisited regularly to ensure it continues to reflect both changes to the software artifact to which it pertains (especially new vulnerabilities that may be introduced), as well as to the evolving threats to which the system is subject, and the resulting changes to perceived risks. It is in the interest of security that all parts of the management chain agree to take action when vulnerabilities are expressed before the release of a product.

4.1.2.1. ACE Threat Analysis and Modeling

Version 1 of the Microsoft Threat Modeling Process represented the combination of earlier threat modeling methods used by Microsoft and by @stake (later purchased by Symantec). In their book *Threat Modeling*, Frank Swiderski and Window Snyder of Microsoft explained that the core artifact of Threat Modeling is the threat model—a detailed textual description and graphical depiction of significant threats to the software system being modeled. The threat model captures the ways in which the software's functions and architecture may be targeted, and identifies the potential threat agents, i.e., vectors for delivering threat-associated attacks. The goals of these attacks form the basis for building a hierarchical tree of the security-related preconditions that would render those attacks possible.

In Threat Modeling V.1, analysis was performed from the perspective of the attacker. The STRIDE model was used to help define the threat scenarios to be modeled. STRIDE is an acronym that categorizes different types of threats around which scenarios could be based—*S*poofing, *T*ampering, *R*epudiation, *I*nformation disclosure, *D*enial of service, *E*levation of privilege. Complementing STRIDE was DREAD, a risk calculation methodology introduced in David LeBlanc's *Writing Secure Code* and used in Step 6 of Threat Modeling V.1 to rate the threats and prioritize the importance of their countermeasures/mitigations. As with STRIDE, DREAD is an acronym in which each letter represents the first letter of a question about the threat the answer to which helps determine its associated risk level—*D*amage potential, *R*eproducibility, *E*xploitability, *A*ffected users, *D*iscoverability. Once a threat's attributes are ranked, a mean of the five attribute ratings is taken; the resulting value represents the perceived overall risk associated with the threat. This process is repeated for all identified threats, which are then prioritized by descending order of overall risk value (highest risk first, lowest risk last).

In March 2006, Microsoft announced to the imminent release of Version 2.0 of what they now call ACE (Application Consulting and Engineering) Threat Analysis and Modeling. This new version provides:

- A new threat modeling methodology and process intended to be easier for software developers, architects and other stakeholders who are not security experts to understand and execute;
- Completely reengineered Threat Modeling application (tool).

To make ACE Threat Analysis and Modeling easier to use, Microsoft have eliminated STRIDE and DREAD and shifted the perspective of the analyst from that of an attacker to that of a defender. The focus of the Threat Modeling analysis is now identification and analysis of *threats* rather than of attacks (by contrast, Version 1 started by identifying possible attacks, then extrapolating their associated threats). The perspective shift reflects Microsoft's new belief that the defender is in a better position than the attacker to understand the threats to the system. The software's stakeholders understand the value of different portions of the software, and how those portions are accessed while the attacker (unless he is a "malicious insider") can only speculate on the software's internal portions based on his/her observation of the portions that are externally exposed or discovered through reconnaissance attacks.

As with Version 1, ACE Threat Analysis and Modeling V.2 remains an iterative process. It begins during the software's architecture/high-level design phase then, as the design progresses into subsequent phases of the life cycle, is repeated to add layers of detail to the initial high-level threat model.

A core idea behind ACE Threat Analysis and Modeling is that an event can only be considered a threat when it results in a negative business or mission impact. For instance, "compromising the confidentiality of a classified database" would be deemed a threat because an identifiable negative mission impact would result. This more specific definition of what constitutes a threat is intended to make ACE Threat Analysis and Modeling V.2 more accurate than Version 1, in which events modeled as threats did not have to result in a perceivable business or mission impact. For example, in Version 1, an overlong input string that caused a method to raise an exception and reject the input would have been modeled as a threat.

In ACE Threat Analysis and Modeling V.2, by contrast, such an event might be identified as an exploitable software weakness, but because it does not directly result in a negative business or mission impact, it would not be modeled as a threat. In this way, ACE Threat Analysis and Modeling V.2 attempts to clarify the distinction between *threats, attacks,* and *vulnerabilities*.

> *NOTE: It is incumbent upon the analyst who uses ACE Threat Analysis and Modeling to clearly understand how terminology is defined by the ACE methodology, because most risk assessment methodologies construe the same terminology differently.*

The ACE Threat Analysis and Modeling methodology incorporates a set of predefined attack libraries describing the effective mitigations to each attack type associated with each threat. The new Threat Modeling Application supports the iterative analyses described above. It also automatically generates threat models based on a defined application context, then maps those threat models to relevant countermeasures identified in the attack library.

4.1.2.2. PTA Calculative Threat Modeling Methodology

PTA (Practical Threat Analysis) Technologies has developed what they call a Calculative Threat Modeling Methodology (CTMM). CTMM attempts to refine and expand on Version 1 of the Microsoft Threat Modeling Process to overcome what PTA considers to be that version's limitations (note that the emergence of ACE Threat Analysis and Modeling may render moot some of the concerns that led PTA to develop CTMM):

- No support for relating threats to financial losses caused by attacks;
- No ranking/prioritization of countermeasures according to their effectiveness in reducing risk;

- Reliance on "predefined" cases, making the tool difficult to adapt for modeling other threat scenarios;
- No support for a complete system view for threat analysis or risk management;
- Limited reporting and collaboration capabilities.

PTA developed CTMM to complement or augment standards-based risk assessment procedures by providing a methodology and automated tools specifically geared to defining threats, vulnerabilities, and countermeasures. The PTA toolset includes a database of relevant security entities from which documentation can be automatically generated in support of the evaluation procedures required by a number of standards-based risk assessment and risk management methodologies including:

- International Standards Organization/International Electrotechnical Committee (ISO/IEC) 17799,
- British Standard (BS) 7799,
- System Security Engineering Capability Maturity Model (SSE-CMM),
- Operationally Critical Threat, Asset, and Vulnerability Evaluation (OCTAVE),
- Federal Information Technology Security Assessment Framework (FITSAF),
- U.S. Federal Information Processing Standard (FIPS) 199,
- Generally-Accepted Information Security Principles (GAISP),
- Control Objectives for Information and related Technology (COBIT),
- Information Technology Infrastructure Library (ITIL),
- NIST Special Publication 800-30,
- Information Security Foundation (ISF) Fundamental Information Risk Management (FIRM), Information Risk Analysis Methodologies (IRAM), and Simplified Process for Risk Identification (SPRINT),
- Certicom Security Auditor's Research Assistant (SARA),
- Business Impact Analysis (BIA).

In addition, the enterprise-level version of the tool provides entity libraries for ISO 17799 and BS 7799 2002.

4.1.2.3. CORAS and SECURIS

The European Union (EU)-funded Consultative Objective Risk Analysis System (CORAS) project (IST-2000-25031) was intended to develop a base framework applicable to security critical systems to supply customizable, component-based roadmaps to aid the early discovery of security vulnerabilities, inconsistencies, and redundancies, and to provide methods to achieve the assurance that security policy has been implemented in the resulting system. The main objectives of the CORAS Project were to:

1. Develop a tool-supported methodology for model-based risk analysis of security-critical systems. This methodology is supported by a framework created by synthesizing risk analysis methods, object oriented modeling techniques, semiformal methods, and supporting tools. The objective of this synthesis is the improvement of security analysis and security policy implementation for security-critical systems. The framework is intended to be used both when developing new systems and when maintaining and improving legacy systems;

2. Assess the applicability, usability, and efficiency of the CORAS framework through extensive experimentation in e-commerce, telemedicine, and telecommunications;

3. Investigate the commercial viability of the CORAS framework, and pursue its exploitation within relevant market segments. Also use CORAS as the basis for influencing relevant European and international standards efforts.

The CORAS framework comprises:

- Standards for precise and unambiguous evaluation, description, and definition of analyzed objects and the risks to which they are exposed;
- Standards for accurate specification of security requirements that form the basis for security policy;
- Adaptation or extension of a Reference Model for Open Distributed Processing (RM-ODP)-based reference model for modeling security-critical systems;
- Unified Modeling Language (UML)-based specification language with extensions (Object Constraint Language [OCL] and other techniques) for security risk analysis;
- Libraries of standard modeling elements for analyzed object models (includes a library of reusable experience packages);
- Methods for consistency checks of the analysis results;
- Methods for the comprehensible presentation and communication of object analysis results and security requirements. Includes a standard vulnerability assessment report format and an eXtensible Markup Language (XML)-based markup language for exchanging risk assessment data, in support of qualitative modeling, management, and documentation of risks;
- Automated tool to support the methodology, including an assessment repository and a repository to hold the reusable experience packages.

The Research Council of Norway's model-driven development and analysis of Secure Information Systems (SECURIS) project builds on the results of several research projects, most notably CORAS and a comparable tool-supported model-driven system development methodology developed by the EU-funded Component-Based Interoperable Enterprise (COMBINE) project. The intended products of the SECURIS project will be four prototype tools and an accompanying methodology to support:

1. Capture and formalization of security requirements;
2. Model-driven specification and implementation of security policies;
3. Model-driven specification and development of security architectures;
4. Model-driven security assessment.

4.1.2.4. Trike

Trike is an open source conceptual framework, methodology, and supporting toolset that supports system and software security risk auditing through repeatable automated generation of threat models. The Trike methodology is designed to enable the risk analyst to describe completely and accurately the security characteristics of the system, from high-level architecture to low-level implementation details. Trike's consistent conceptual framework provides a standard language that enables communication among members of the security analysis team, and between the security team and other system stakeholders.

The current version of the Trike methodology is under significant ongoing refinement by its developers, but they should provide sufficient detail to allow its practical use. Trike generates the threat model using the Trike toolset

that supports automatic generation of threat and attack graphs. The input to the threat model includes two additional Trike-generated models also generated by Trike, a requirements model and an implementation model, along with notes on system risk and work flows.

Trike provides high levels of automation. It is predicated on a defensive perspective that differs from those of other threat modeling methodologies, which are based on an offensive attacker perspective. The methodology also imposes a greater degree of formalism to the threat modeling process. However, to date much of the Trike methodology is still in the experimental stage, and has not been fully exercised against real systems.

4.1.2.5. Emerging Software Risk Assessment Methods

Other methods for assessing security risk in software systems are emerging. Among these, one of the most promising appears to be Threat Modeling based on Attacking Path Analysis (T-MAP), a risk assessment methodology with supporting automated tools. T-MAP is geared towards modeling of threats to COTS software using data gleaned from thousands of software vulnerability reports mined from the NIST National Vulnerability Database and several other sources. The University of Southern California researchers performed a "proof of concept" case study using T-MAP to analyze the cost-effectiveness of patching and upgrades for improving security of COTS-based systems, and are performing further research to experiment with various Web search and AI text-reading technologies to improve the accuracy of the automated data collecting engine used to expand the T-MAP database of vulnerability information on which the accuracy of T-MAP relies.

4.1.3. System Risk Assessment Methodologies

Some general IT/system security risk assessment methodologies, while not specifically defined or adapted for examination of software systems, have been reported by software practitioners to provide excellent support in their software security risk analysis and threat modeling activities. A subset of these methodologies and tools are available under commercial or open source licenses, and are listed in Table 4-1, below. Comparable proprietary methodologies either used only within a single firm on its own software (e.g., Sun Microsystems' Adaptive Countermeasure Selection Mechanism/Security Adequacy Review [ACSM/SAR]) or used in a firm's fee-for-service offerings but not made licensed to other users (e.g., Cigital's Software Quality Management [SQM] and Aspect Security's proprietary methodology and tools), are not referenced here.

Table 4-1. System Risk Assessment Methodologies and Tools

Supplier	Methodology	Supporting Tool
NIST	Draft SP 800-26 Revision 1, "Guide for Information Security Program Assessments and System Reporting Form" (August 2005). *This update to the November 2001 NIST SP 800-26, "Security Self-Assessment Guide for Information Technology Systems" is intended to be used in conjunction with*: SP 800-30 "Risk Management Guide for Information Technology Systems" (July 2002)	Automated Security Self-Evaluation Tool (ASSET)
CMU SEI	OCTAVE and OCTAVE-Secure (OCTAVE-S)	same as methodology
Siemens/Insight Consulting	Central Computer and Telecommunications Agency (CCTA) Risk Analysis and Management Method (CRAMM)	same as methodology
Amenaza	n/a	SecurITree
Isograph	n/a	AttackTree+

4.2. Requirements-Driven Engineering and Software Security

The accurate specification of nonfunctional security requirements depends to a great extent on the ability to predict all of the states and behaviors that will ever arise throughout the lifetime of the software and its operating environment'. These not only include the states and behaviors that occur during normal usage, but also those that occur as the result of the dormant intentional faults that only become active when the software is attacked.

Current threat modeling techniques enable the risk analyst to predict the environmental state changes that are likely to result from the manifestation of known threats (i.e., known attack patterns). Such threat modeling is useful for specifying the change in the software's state that will be needed in response to these attack-induced environment state changes if the software is to keep operating correctly.

What threat modeling cannot do is foresee the future. It cannot with meaningful accuracy predict truly novel threats and attack patterns. And because it cannot, any software engineering process that is entirely requirements-driven, even with threat modeling contributing to the requirements process, will fail to adequately ensure the software's ability to continue operating correctly throughout its lifetime. Even if the software can be demonstrated to satisfy all of its requirements under all anticipated operating conditions, it is not possible to predict whether that software will continue to satisfy those requirements when subjected to unanticipated conditions, such as exposure to new threats that did not exist at the time the software's threat models were most recently revised. To meet these challenges in an ideal manner, software should be built to be easily modified when such novel attacks occur.

A software requirements specification necessarily reflects a fixed set of assumptions about the state of the software's operating environment and the resulting state of the software itself. The specification is necessarily written under the assumption that the software can continue to satisfy its requirements in the face of each of these anticipated state changes. Verifying correctness and predictability of software behaviors only under anticipated operating conditions is of limited value when it comes to verifying software security. For software to be considered secure, its secure behavior must be consistently demonstrated and verifiable even when the software is subjected to unanticipated conditions.

Realistically, the software environment's state and the software's state in response to its environmental state, while it may remain constant for some percentage of the time, will be in flux the rest of the time, and not always according to predictable patterns of change. Given the volatile threat environment confronting the system and its environment, as well as the constant cycle of patching, reconfiguration, and updating of software (at the application and environment levels), the security of the software's state at a given point should be considered an indicator of the software's expected behavior over time. This expected behavior can and should be evaluated to capture security requirements related to the expected behavior.

The success of requirements-based engineering necessarily depends on the specifier's ability to anticipate and model with a high degree of accuracy and thoroughness the full range of different environment states with which the software will be confronted, and to describe each of the software's own state changes in response to those anticipated changes in environmental state (again, with the objective of enabling the software to continue operating as required throughout those changes). While requirements may be incomplete due to the specifier's limited ability, they are more often incomplete due to future changes in those requirements or because stakeholders do not see a pressing need for a particular aspect of the software, such as security.

Even if such a comprehensive specification could be written, there remains a category of state changes that it may not be possible to model with the required high degree of accuracy and thoroughness. These are the state changes that result from a changing "threat environment" surrounding the software.

Current threat modeling techniques (see Section 4.1.1) enable the risk analyst to predict the environmental state changes that are likely to result from the manifestation of known threats (i.e., known attack patterns). Such threat modeling is useful for specifying the change in the software's state that will be needed in response to these attack-induced environment state changes if the software is to keep operating correctly.

What threat modeling cannot do is foresee the future. It cannot with meaningful accuracy predict truly novel threats and attack patterns. And because it cannot, any software engineering process that is entirely requirements-driven, even with threat modeling contributing to the requirements process, will fail to adequately ensure the software's ability to continue operating correctly throughout its lifetime. Even if the software can be demonstrated to satisfy all of its requirements under all anticipated operating conditions, it is not possible to predict whether that software will *continue* to satisfy those requirements when subjected to unanticipated conditions, such as exposure to new threats that did not exist at the time the software's threat models were most recently revised. To meet these challenges in an ideal manner, software should be built to be easily modified when such novel attacks occur.

4.2.1. Risk-Driven Software Engineering

To develop secure software, a combination of requirements-driven and risk-driven software engineering techniques is needed. Risk-driven engineering acknowledges the impossibility of predicting all future environmental state changes with a high enough degree of accuracy to ensure that software can remain correct over its lifetime.

Risk-driven engineering institutes a constant "pulse check" of the environment and of the software's response to it. This "pulse check" takes the form of ongoing iterative threat modeling and risk analysis. New threats (and associated attack patterns) that emerge are analyzed, and the results are fed back into the requirements specification process.

But risk-driven engineering does not stop there: there is recognition that the software's potential vulnerabilities to new threats must be determined and mitigated as soon as those threats arise, regardless of the software life cycle phase. In a risk-driven approach, correction of software does not await the update of the requirements specification; nor does the search for vulnerabilities await a new requirements-driven test plan.

Software's testing includes security reviews and tests that are not driven by requirements compliance verification objectives, but by the need to identify and remediate immediately any vulnerabilities in the software that the new threats can successfully target.

The remediations possible for such vulnerabilities will also depend on the life cycle phase. If early enough in the life cycle, it is possible that the requirements specification can be adjusted, and the software redesigned accordingly so that the vulnerability is eliminated before coding/integration. But if the vulnerability is identified after implementation/ integration, it is more likely to need a remediation that involves a "secure in deployment" measure, such as a security patch, a reconfiguration, a wrapper, a filter/firewall, etc. Such "in deployment" remediations will ideally be temporary stopgaps until a new release of the software, reflecting an updated specification and design from which the vulnerability is eliminated. And so, just as requirements-driven engineering forms the basis for quality assurance, risk-driven engineering forms the basis for risk management.

One vendor that has implemented a risk-driven engineering development life cycle as a response to its high profile to attackers is Microsoft. Microsoft has delineated three security principles for software that emphasize the importance of considering security throughout the software's life cycle:

1. **Software should be secure by design:** The basic design of the software should be secure. We would reinterpret this principle to say: "Software should be demonstrably secure in its conception (i.e., requirements specification), its architecture and design, and its implementation. Relevant information is provided in Section 5 to address the implications of this principle.

2. **Software should be secure by default:** The supplier's "default" configuration for their software product should be as restrictive as possible. It should be the purchaser who assumes the risk for any choice to deploy the software in a configuration that is less restrictive. This is, in essence, the approach taken by suppliers of firewalls. All services/ports are "off" in the firewall as shipped. It is the purchaser who then assumes any risk posed by turning certain services/ports on, not the firewall supplier. Section 5 discusses

the this and other security aspects of preparing software for distribution and deployment, as well as of the security of the actual distribution/shipment process.

3. **Software should be secure in deployment:** The software's security can be maintained after the software has gone into production. Specific activities associated with the developer's role in helping ensure the ongoing security of production software appear in Appendix G:G.8.

Security-enhancing the activities that comprise the software development life cycle generally entails shifting the emphasis and expanding the scope of existing life cycle activities so that security becomes as important as the other objectives to be achieved in the developed software. For this to happen, security must be explicitly stated among the list of all required properties of that software. In practical terms, security-enhancing will have the following general effects on the phases of the life cycle:

1. **Requirements, design, and implementation:** The initial functional requirements for the software will undergo a baseline security vulnerability and risk assessment. This will form the basis for specifying security requirements for the software which, when addressed through the system's design and selected assembly alternative, should be able to be iteratively adjusted to reflect mitigations/reductions in risk achieved through good security engineering.

2. **Reviews, evaluations, and testing:** Security criteria will be considered in all specification, design, and code reviews throughout the software development life cycle, and in the system engineering evaluations for acquired or reused software selection. Security testing will be included at all appropriate points throughout the life cycle (using life cycle phase-appropriate techniques), and not just "saved up" for the Security Test and Evaluation (ST&E) at the end.

The software development project manager should implement a software development plan that includes contingency plans for the worst-case scenarios. The development schedule should provide adequate time for the necessary security checkpoints and validations, and should also build in time to accommodate the inevitable delays caused by unanticipated problems.

The software development life cycle process flow may be based on any of a variety of software development models, such as Linear Sequential (also known as "Waterfall"), Spiral, or Staged Delivery. Regardless of which process model is used, it usually includes a comparable core set of phases, though with slightly different names.

Regardless of the specific methodology and model used, the development life cycle represents a phased approach to software development, from conception through maintenance. John Steven of Cigital has suggested a good core set of factors that should be addressed regardless of the development methodology and life cycle model being followed:

1. How are security stakeholders identified? How are their interests understood?
2. How are threats that concern stakeholders identified, assessed, prioritized for the software?
3. How is the design verified to satisfy its security objectives and requirements?
4. How is the design proven to be "good enough", i.e., free of exploitable faults and other weaknesses?
5. How is the code proven faithful to the design?
6. How is the code proven to be "good enough", i.e., free of exploitable faults and other weaknesses?
7. How are threats and vulnerabilities weighed against risks to the business/mission and stakeholders?
8. How are attacks identified and handled?

These questions form a good basis for shaping the risk management activities and validations that will be associated with each phase of the software life cycle.

4.2.2. Post-Implementation Security Validations

When it comes to security-enhancing the development life cycle process for the software elements of systems that will eventually undergo system security certification and accreditation (C&A), FIPS 140-2 certification, or Common Criteria (CC) evaluation, the main consideration is to ensure that the different phases of the C&A or evaluation processes are mapped to the appropriate software development process phases. Such a mapping will simplify coordination of timing of production of the artifacts required for the accreditation or evaluation.

One benefit of such a mapping will be to minimize the duplication of effort required to satisfy the needs of both processes. This can be accomplished by designing the software development process' artifacts so that they also satisfy the requirements of accreditation or evaluation artifacts. The objective is to avoid having to generate a whole separate set of evaluation artifacts, but rather to be able to submit the artifacts produced during the normal course of software development, possibly with some minimal level of adaptation, to the certifier or evaluation lab.

4.2.2.1. C&A in the Development Life Cycle

As for the life cycle itself, the major federal government C&A methodologies mandated to date—including the U.S. federal National Information Assurance Certification and Accreditation Process (NIACAP) and now-superseded DoD Information Technology Security Certification and Accreditation Process (DITSCAP) provide little guidance on the processes used to develop the systems that will be accredited, while the U.S. Federal Information Security Management Act (FISMA) and the U.S. DCID 6/3 do include defined guidelines for the testing phase of the development life cycle, although less-well-defined guidelines for the rest of the life cycle activities.

The new U.S. DoD Information Assurance Certification and Accreditation Process (DIACAP), which has replaced the DITSCAP, requires C&A related documentation, review, and test activities to be included earlier in the life cycle. The DIACAP is slanted towards the spiral systems engineering life cycle, as mandated in DoD Acquisition policy (defined in DoD Directive 5000.2, DoD Instruction 5000.2, and the DoD Acquisition Guidebook), *Operation of the Defense Acquisition System*.

NIST SP 800-64, *Security Considerations in the Information System Development Life Cycle*, identifies the positioning throughout the system development life cycle of the C&A activities detailed in NIST SP 800-37, *Guide for the Security Certification and Accreditation of Federal Information Systems*. The newer C&A methodologies, and their supporting guidance, can be seen as expanding the validations from what were originally solely post-development activities towards a holistic set of analyses and tests beginning much earlier in the development life cycle.

4.2.2.2. Late Life Cycle Validations and Software Security Testing

Some amount of penetration testing is often performed in connection with C&A. However, the extent and focus of that testing will vary at discretion of each certifier. In all cases, whatever penetration testing is done focuses on identifying whole-system versus individual component vulnerabilities. The objective of this penetration testing is to ensure the security of the system as a whole, under expected usage and environmental conditions. More granular testing is not performed that would reveal how the components of the system would behave in response to intentional faults and weaknesses.

While the CC does include specific criteria for evaluating the development process for the Target of Evaluation (TOE), the bulk of the evaluation activity for the highest Evaluation Assurance Levels (EALs), and all of the evaluation activity for the lower EALs, is limited to review and analysis of the written assurance argument for the TOE, submitted by the TOE's developer/supplier. CC evaluation at the higher EALs includes a vulnerability assessment of the evaluation target to discover exploitable weaknesses in the software components that make up the TOE being evaluated. However, these vulnerability findings are not published in the evaluation report, nor maintained and tracked through later releases of the same product. Also CC evaluation's benefits are limited,

because the evaluations consider only systems in which security functionality is an integral or exclusive part. This said, DoD and DHS have undertaken an effort to analyze and suggest enhancements to both the CC and the National Information Assurance Partnership (NIAP) evaluation process, to better address software security issues in CC evaluations.

However, it is currently unlikely that the methodologies for government C&A and for CC evaluation, will provide adequate evidence for the certifier or evaluator to determine whether the software components of the system under consideration will behave securely under all possible conditions. Nor will the evidence provided enable the certifier/evaluator to zero in on those components that are likely the cause of any insecure behavior observed in the system as a whole. For this reason, these late life cycle security validations must be augmented by earlier-in-the-life cycle tests that "drill down" into the individual components to ensure that each component is attack-resistant and attack-resilient, so that any deficiencies in the security of the system as a whole can be more easily pinpointed to the interfaces among components or between the system and its environment.

4.3. Using Process Improvement Models and Life Cycle Methodologies to Increase Software Security

The following are definitions of key terms used throughout Section 4 and the remainder of this document:

- **Process Improvement Models:** A process improvement model defines an integrated set of processes of the entire software or system life cycle, including the processes for planning, project management, software development, systems engineering, risk management, quality assurance, configuration management, etc. Process improvement models are intended to define a system for analyzing, quantifying, and enhancing the efficiency and quality of the generic "processes" (business processes, engineering processes, or other types of processes) involved in product production and delivery. The main objective is to improve the efficiency and adaptability of each process to make it repeatable and to improve its quality and, as a result, minimize the number and magnitude of errors in the process. A third objective is to provide a framework for consistently applying the process across multiple instances of its use (e.g., different development projects).

  Process improvement models are sometimes referred to as process assessment models or capability maturity models. The most familiar process improvement models currently in use are the SEI's Capability Maturity Model (CMM) and its variants (most notably SEI's CMM-Integration [CMMI], the Federal Aviation Administration's [FAA] integrated CMM [iCMM], the Secure Systems Engineering-Capability Maturity Model (SSE-CMM, originated by the National Security Agency, but now an international standard [ISO/IEC 21827]), and General Electric's SixSigma. All process improvement models are predicated on the notion that product quality cannot be ensured without first guaranteeing the quality of the process by which the product is developed. A "product" can be anything from an airplane engine to a hospital's medical services to software.

- **System Engineering or Software Life Cycle Standards:** There are a number of standards that identify the activities that should be included in the system engineering or software development life cycle, such as ISO/IEC 12207 and 15288 and IEEE P1074. I addition to identifying and providing high level descriptions of life cycle activities, these standards usually define the artifacts that are expected to be produced during each life cycle phase, with high level descriptions of what those artifacts should contain and how they should be evaluated.

- **Software (or System) Development Life Cycle (SDLC) Methodologies:** An SDLC methodology identifies the set of activities to be performed at each phase of the SDLC, and in some cases also what artifacts should be produced as a result. Some SDLC Process models/methodologies go further to describe how the activities should be performed. Examples include Rational Unified Process (RUP), the "waterfall" software development process specified in the now-obsolete Department of Defense Standard

No. 2167A (DOD-STD-2167A), the SEI's Team Software Process (TSP) and Cleanroom Software Engineering process, and the various agile methods.

- **Phase-Specific Software Development Methodologies:** Phase specific methodologies tend to be more detailed than SDLC methodologies in that they describe not only what activities should occur during a particular life cycle phase, but also how those activities should be performed. They are often used in conjunction with SDLC methodologies, especially those less descriptive of how lifecycle activities should be performed. Some software development methodologies cover the all or many phases of the SDLC (e.g., agile methods, Aspect Oriented Software Development [AOSD]), while others focus on only one or two phases, usually those at the beginning of the Life Cycle (e.g., Model-Driven Architecture [MDA], formal methods, object-oriented modeling using Unified Modeling Language [UML]).

In March 2004, the National Cyber Security Taskforce (NCST) produced a report entitled *Processes to Produce Secure Software: Towards More Secure Software* (see Appendix B:B.2). This report was intended to provide a set of recommendations to the DHS for activities to be undertaken in the context of their Software Assurance initiative. However, much of the information in the report is widely applicable. Of most relevance for our purposes are three areas of the report:

1. A set of generic requirements for any development process to be able to produce secure software;

2. A set of recommended technical and management practices, intended to aid in security-enhancing of software development-related engineering and management processes;

3. A methodology for evaluating and qualifying existing software engineering and development life cycle processes and practices to determine the extent to which they encourage or assist in producing secure software.

Adopting a process improvement model, an SDLC methodology, and supporting phase-specific development methodologies can play an important role in increasing the likelihood that the software produced by that process will be more secure by increasing the likelihood that it will exhibit several of the subordinate properties of security, such as correctness, predictability, etc., and thus will harbor fewer exploitable faults and weaknesses.

4.3.1. Security-Enhanced Process Improvement Models

It is a truism that just because a process is efficient, repeatable, and applied in a consistent, disciplined way, there is no guarantee that the process is actually good, or for our purposes, "security-enhancing". For any process improvement or capability maturity model to be useful in improving the security of software produced under that model, the first step must be to define processes that are inherently security-enhancing. Equally important, these security-enhancing processes must be simple and straightforward enough to be adopted without excessively increasing project costs, impacting schedule, or requiring for extensive specialist training. Only when these things can be accomplished, will the process improvement model be able to help enforce the consistent, disciplined, and quality-assured use of processes at all, let alone those geared towards enhancing the security of software produced by all development teams across the enterprise over time. Fortunately, a number of efforts have been undertaken to adapt or extend existing CMMs or to define new secure process models.

The selection and implementation of process improvement models is seldom if ever the responsibility of the developer or even the project manager. Instead, commitment to such a model is made at the enterprise or organization level, which implies an audience that is outside that addressed by this document. Because process improvement models are beyond the scope of this document, they are discussed in Appendix D, which describes several noteworthy efforts to add security activities and/or checkpoints to existing process improvement models, CMMs, as well as standards that define high-level SDLC processes.

4.3.2. Security-Enhanced SDLC Methodologies

There is nothing inherently "security-enhancing" about most development methodologies. For example, object-oriented modeling using UML will not improve software security…unless it includes modeling of threats and attacks, security-relevant behaviors and interfaces, and misuse and abuse cases. Unless the security properties of software are explicitly considered by a development methodology, that methodology should not be expected to enhance the security of that software. A security-enhanced methodology should also provide for repeated, iterative risk assessments, impact analyses and security modeling, security evaluation of components, integration/assembly option prototyping and security evaluation, and integration security testing along with support for secure coding activities and security review of source code.

In selecting and applying any SDLC methodology (and supporting phase-specific methodologies), developers and integrators need to consider whether the methodology focuses primarily or exclusively on from-scratch coding, or whether it also incorporates adequate security evaluations of acquired and reused components, modeling of assembly options and configurations, and engineering of countermeasures to minimize the exposure and maximize control and protection of the interface boundaries between components. Otherwise, the benefits of the software-enhanced SDLC methodology will be limited to software developed from scratch, and will leave systems integrated/assembled from components at the mercy of the likely-to-be-security-deficient development processes used by the components' suppliers.

Because market pressures tend to dominate the production of commercial software, and much open source software at least originates from academic prototypes and other less-disciplined development efforts, security verification and testing activities are often not incorporated into the development processes used to produce those components. It is very difficult for the integrator to get visibility into the development processes that were used to produce acquired or reused components as the majority of the commercial software in a component assembly or integrated system will not have been subjected remotely the same degree of rigor applied to by developers working under the governance of a life cycle process model and structured development methodology.

With Microsoft's Security Development Lifecycle, and Oracle Corporation's Software Security Assurance Process, two of the largest commercial software suppliers in the world have made public commitments to security-enhancement of their software development processes, and to training their developers in the importance and meaning of security for their firms' software products. It is hoped that many other commercial software suppliers and open source developers will be inspired to follow the lead of these two companies. It is also hoped that this document can help them take the first steps towards that goal.

The remainder of Section 4.3.2 describes security-enhanced SDLC methodologies that have been published by commercial and academic entities, in some cases with supporting tools and training, for adoption by other development teams and organizations.

4.3.2.1. Comprehensive, Lightweight Application Security Process

John Viega, founder of Secure Software, Inc., has published a structured process for introducing security in the early stages of the software development life cycle. The Comprehensive, Lightweight Application Security Process (CLASP) represents a collection of methods and practices that can be used collectively to start identifying and addressing appropriate application security concerns well before any code is written for the system. CLASP is arguably the first defined life cycle process with the specific objective of enhancing the security (versus safety, correctness, or high-quality) of the early stages of the software development life cycle. As a formal process emphasizing accuracy and quality assurance, CLASP shares objective traits native to more industry-driven CMM-based life cycle process models.

CLASP includes instructions, guidance, and checklists, for activities that comprise its structured process. Thirty (30) specific activities are expressed in CLASP that can be used and adapted to increase security awareness across the development team. These activities are assigned to the following typical roles found throughout the life cycle,

designating both owners and participants: (1) project managers, (2) requirements specifiers, (3) software architects; (4) designers (including user interface designers and database designers), (5) implementers, (6) integrator and assemblers, (7) testers and test analysts, and (8) security auditors.

For each role-based activity cited, CLASP describes:

1. The implementation of the activity (e.g., when and how it should be performed);
2. The level of risk associated with omitting the activity;
3. The estimated cost for implementing the activity.

Together, these three factors form the basis for the cost/benefit analysis of applying CLASP to a specific application development effort, and a rationale for adopting the methodology.

CLASP assigns responsibility and suggests accountability for each activity, and delineates two different "roadmaps": one that supports new system development using an iterative, or "spiral", methodology, and one that supports maintenance/enhancement of legacy systems with the emphasis on management of the current development effort. Both roadmaps incorporate consistent testing and analysis of the application's security posture through any upgrades or enhancements.

Although CLASP is designed to insert security methodologies into each life cycle phase, the suggested activities are clearly self-contained. CLASP is intended for adaptation to any software development process. However, in organizations that do not assign developers permissions based on roles, CLASP would have to be adapted somewhat—or another security-enhanced life cycle model may be appropriate. This allows flexibility within the CLASP model, and also provides security measures that can be summarily integrated within many other models.

As mentioned earlier, CLASP is available as a plug-in to the Rational Unified Process (RUP) development methodology, or as a reference guide to a standalone development process. The CLASP plug-in to RUP is available free-of-charge but requires a RUP license to implement. Because of its relatively recent release, however, little information has been made available about how the CLASP plug-in works or how it is currently being accepted into the secure systems development community.

CLASP provides notation for diagramming system architectures, but suggests a collection of UML extensions to provide for the explanation of security elements. Via the plug-in to RUP, CLASP supports development based on the standards set by UML. Security is taking hold of software modeling, and is evidenced through the corroboration of these methods.

4.3.2.2. Team Software Process for Secure Software Development

The Carnegie Mellon University's Software Engineering Institute (SEI) and CERT/CC jointly developed the Team Software Process for Secure Software Development (TSP-Secure—previously referred to as TSP for Secure Systems). TSP-Secure defines a set of processes and methods for producing high-quality, secure software. The objectives of TSP-Secure are to reduce or eliminate software vulnerabilities that result from software design and implementation mistakes, and to provide the capability to predict the likelihood of vulnerabilities in delivered software.

As its name suggests, TSP-Secure builds on SEI's Team Software Process. The philosophy at the core of TSP is based on two premises:

1. Engineers and managers need to establish and maintain an effective teamworking environment. TSP's operational processes help them form engineering teams and establish an effective team environment, then guide those teams in doing the work.

2. Engineers are most efficient when they have specific guidance on how to implement complex, multi-phase engineering processes. TSP guides engineers through the engineering process, reducing the likelihood that they will inadvertently skip steps, organize steps in an unproductive order, or spend unnecessary time figuring out what to do next.

The TSP includes a systematic way to train software developers and managers, to introduce the methods into an organization, and to involve management at all levels. TSP is well-established and in use by several organizations, with observed metrics for quality improvement published in SEI reports.

TSP-Secure augments TSP by inserting security practices throughout the software development lifecycle. In addition to providing an operational process for secure software production, TSP-Secure provides techniques and practices for:

- Vulnerability analysis by defect type,
- Establishing predictive process metrics and checkpoints,
- Quality management for secure programming,
- Design patterns for common vulnerabilities,
- Security verification,
- Removal of vulnerabilities from legacy software.

A team that commits to using TSP-Secure is expected to already be knowledgeable about the fundamentals of good software engineering practices. TSP-Secure adopters attend an SEI workshop in which they are introduced to the common causes of vulnerabilities, and to practices that will enable them avoid or mitigate those vulnerabilities.

After training, the team is ready to plan its software development work. Along with business and feature goals, the team defines the security goals for the software system, then measures and tracks those security goals throughout the development life cycle. One (or more) team member assumes the role of Security Manager, who is responsible for ensuring that the team addresses security requirements and issues through all of its development activities.

Through a series of proofs-of-concept and pilot projects, TSP-Secure was shown to produce near defect-free software with no security defects found either during security audits or after several months of use.

4.3.2.3. Microsoft Trustworthy Computing SDL

NOTE: The material in this section and its subsections was originally authored and copyrighted © 2005 by Microsoft Corporation excluding certain portions, which are copyrighted © 2004 by the Institute of Electrical and Electronics Engineers, Inc. This copyrighted material is reprinted here with permission, with all rights reserved by the original copyright holders.

Beginning with Bill Gates' Trustworthy Computing memo in January 2002, Microsoft embarked on an ambitious set of process improvements to create more secure software. The process security and privacy element of these improvements is called the Security Development Lifecycle (SDL). According to Microsoft's security engineers, to date the software developed under the SDL has exhibited more than a 50 percent reduction in vulnerabilities when compared with versions of the same software developed prior to the adoption of the SDL. (Note that according to the same security engineering, SDL has been applied in the development of Windows XP and SQL Server only; other Microsoft products have yet to benefit from the security-enhancements of the process by which they are created and maintained.)

Microsoft feels that security is a core requirement for all software vendors, driven by market forces, the need to protect critical infrastructures, and the need to build and preserve widespread trust in computing. A major challenge for all software vendors is to create more secure software that requires less updating through patches and less burdensome security management.

According to Microsoft, the key to the software industry's ability to meet today's demand for improved security is to implement repeatable processes that reliably deliver measurably improved security. Therefore, software vendors must transition to a more stringent software development process that focuses, to a greater extent, on security. Such a process is intended to minimize the number of security vulnerabilities resulting from the design, implementation, and documentation and to detect and remove those vulnerabilities as early in the development life cycle as possible. The need for such a process is greatest for enterprise and consumer software that is likely to be used to process inputs received from the Internet, to control critical systems likely to be attacked, or to process personally identifiable information. There are three facets to building more secure software:

1. Repeatable process;
2. Engineer education;
3. Metrics and accountability.

If Microsoft's experience is a guide, adoption of the SDL by other organizations should not add unreasonable costs to software development and the benefits of providing more secure software (e.g., fewer patches, more satisfied customers) far outweigh the costs.

The SDL involves modifying a software development organization's processes by integrating tasks and checkpoints that lead to improved software security. This text summarizes those tasks and checkpoints and describes the way that they are integrated into a typical software development life cycle. The intention of these modifications is not to totally overhaul the process, but rather to add well-defined security reviews and security deliverables.

A critical cornerstone of the success of SDL at Microsoft is the ongoing education and training of the firm's developers in the techniques and technologies required for secure development. Microsoft's educational philosophy and curricula are described in Section 4.4.

The SDL maps security-relevant tasks and deliverables into the existing software development life cycle. The SDL is not meant to replace the current process; it is, in fact, process-agnostic. The following sections outline the major phases of the SDL and their required tasks and deliverables. Figure 4-1 depicts, at a very high level, the phases of the SDL.

Figure 4-1. SDL Improvements to the Software Development Process.

4.3.2.3.1. Requirements Phase

The need to consider security "from the ground up" is a fundamental tenet of secure system development. During the requirements phase, the product team makes contact with the central security team to request the assignment of a security advisor who serves as point of contact, resource, and guide as planning proceeds. The security advisor assists the product team by reviewing plans, making recommendations, and ensuring that the security team plans appropriate resources to support the product team's schedule. The security advisor remains the product

team's point of contact with the security team from project inception through completion of the Final Security Review (FSR) and software release.

The requirements phase is the opportunity for the product team to consider how security will be integrated into the development process, identify key security objectives, and otherwise maximize software security while minimizing disruption to plans and schedules. As part of this process, the team needs to consider how the security features and assurance measures of its software will integrate with other software likely to be used together with its software.

4.3.2.3.2. Design Phase

The design phase identifies the overall requirements and structure for the software. From a security perspective, the key elements of the design phase are:

1. **Define security architecture and design guidelines.** Define the overall structure of the software from a security perspective, and identify those portions of the software whose correct functioning is essential to system security.

2. **Document the elements of the software attack surface.** Given that software will not achieve perfect security, it is important that only features that will be used by the vast majority of users be exposed to all users by default, and that those features be installed with the minimum feasible level of privilege.

3. **Conduct threat modeling.** Using a structured methodology, the threat modeling process identifies threats that can do harm to each asset and the likelihood of harm being done (an estimate of risk), and helps identify countermeasures that mitigate the risk. The process also helps drive code review and security testing, as the highest-risk components require the deepest code review and the most thorough testing. Microsoft's Threat Modeling methodology was discussed in Section 4.1.1.1.

4.3.2.3.3. Implementation Phase

During the implementation phase, the product team codes, tests, and integrates the software. Steps taken to remove or, better yet, avoid including security faults and weaknesses significantly reduce the likelihood of their remaining in the final version of the software. The elements of the SDL that apply are:

1. From the threat modeling task, understand which components are highest risk.
2. Apply coding and testing standards and sound practice at peer review and prior to code check-in.
3. Apply security testing tools, especially fuzzing tools.
4. Apply static analysis code scanning tools and binary inspection tools.
5. Conduct security code reviews.

Document sound security practice for the users of the product, and if necessary build tools to help the users ascertain their level of security.

4.3.2.3.4. Verification Phase

The verification phase is the point at which the software is functionally complete and enters beta testing. Microsoft introduced their "security push" during the verification phase of Windows Server 2003 and several other software versions in early 2002. The main purpose of the security push is to review both code that was developed or updated during the implementation phase and especially "legacy code" that was not modified. The product team may conduct penetration testing at this stage to provide additional assurance that the software is capable of resisting the threats it will encounter after release. Expert third party penetration testers can also be engaged at this point.

4.3.2.3.5. Release Phase

During the release phase, the software is subject to a FSR. The FSR is an independent review of the software conducted by the central security team for the organization. Tasks include reviewing bugs that were initially identified as security bugs, but on further analysis were determined not to have impact on security, to ensure that the analysis was done correctly. An FSR also includes a review of the software's ability to withstand newly reported vulnerabilities affecting similar software. An FSR for a major software version will require penetration testing, additional code review and, potentially, security review and penetration testing by independent outside contractors who supplement the central security team.

4.3.2.3.6. Support and Servicing Phase

Despite the use of the SDL during development, state of the art development practices do not yet support shipping software that is completely free from vulnerabilities—and there are good reasons to believe that they will never do so. Product teams must prepare to respond to newly-discovered vulnerabilities in shipping software. Part of the response process involves preparing to evaluate reports of vulnerabilities and release security advisories and updates when appropriate. The other element of the response process is conducting the post-mortem of each reported vulnerability and taking action as necessary, such as updating the SDL process, education and tools usage.

4.3.2.3.7. Results of Implementing the SDL at Microsoft

Microsoft feels that it is premature for them to make conclusive claims that the SDL improves the security of software, but the results to date have been encouraging with an across the board reduction of approximately 50-60% in post-release vulnerabilities in Microsoft software compared with earlier software versions released prior to the introduction of the SDL. Microsoft's experience thus indicates that the SDL can be effective at reducing the incidence of security vulnerabilities. Implementation of the SDL during the development of Windows Server 2003, Windows XP SP2, SQL Server 2000 Service Pack 3, and Exchange 2000 Server Service Pack 3 in particular resulted in significant improvements in the software security of those releases and of subsequent versions. These continuing improvements reflect the effectiveness of Microsoft's ongoing enhancements to SDL, which appear to be resulting in further improvements in the security of their software . Microsoft has observed that the incremental implementation of the elements that comprise SDL yields incremental improvements. They view this as an important sign that the process is effective, though not yet perfect; SDL is expected to continue evolving and improving into the foreseeable future.

The development and implementation of the SDL represents a major investment for Microsoft, and a major change in the way that their software is designed, developed, and tested. The increasing importance of software to society emphasizes the need for Microsoft and the industry as whole to continue to improve software security. Microsoft SDL practitioners have published papers on SDL and books on specific techniques used through the different SDL phases in an effort to share Microsoft's experience across the software industry.

4.3.3. Secure Use of Non-Security-Enhanced Methodologies

Software development methodologies are not typically driven by the same organizational objectives as process capability models. Whereas capability models are designed to improve the application of virtually any generic process at an organizational level to satisfy specific business objectives, software development methodologies are intended to define lower-level, functional and technical constraints to software specification, design and implementation practices specifically. The intent of using a structured development methodology is to increase the likelihood that the software produced will be correct, of high quality, and in some cases more reliable or reusable. In common with process capability models, however, with very few exceptions software development methodologies are geared toward improving software quality rather than software security.

Some practitioners have reported that with the general reduction in overall faults in their software they have observed a coincidental reduction in the subset of those faults that were exploitable as security vulnerabilities. We use the word "coincidental" here because the reduction of vulnerabilities is usually not the objective of adopting a structured methodology. If a security-enhanced methodology were used, with the specific objective of systematically eliminating and, better yet, preventing exploitable faults in software, it is expected that the reductions in exploitable faults would not only be more significant, they would not merely be the coincidental byproduct of overall fault reduction.

It is particularly important that this kind of coincidental reduction in vulnerabilities not be allowed to lull developers into complacency, or be interpreted incorrectly to mean that good software engineering and quality assurance are adequate to improve the security of software. The intentionality of threats to software security necessarily renders inadequate any quality-only focused approaches to "fixing" the security of software. Without a risk assessment, it will be impossible to determine whether the vulnerability reductions that coincidentally accrue from quality engineering (or reliability engineering or safety engineering) address any of the most critical security problems in the system.

The following sections describe new "secure" structured methodologies that have recently appeared and ways in which popular development methodologies are being enhanced to specifically address security concerns.

4.3.3.1. Using Model Driven Architecture to Achieve Secure Software

Model Driven Architecture (MDA) was developed by the Object Management Group (OMG) as a way to transform a UML model into a Platform Specific Model (PSM) that provides the developer with all the necessary information for implementing the modeled system on a particular platform. By automating the transformation from UML model to a PSM, OMG intends for most source code to be automatically generated, thus reducing the amount and complexity of the remaining code that must still be designed and written by human developers. This approach is expected to coincidentally reduce the number of faults in the resulting software.

The eventual goal of MDA is to enable an entire software program to be automatically generated from models. MDA modeling languages have precisely-defined semantics that allow for fully automated formal verification through tool-driven model checking; by contrast, traditional programming languages have fairly loosely defined semantics and must be, in large part, verified manually.

While one of the Pervasive Services extensions to MDA originally conceived by the OMG was Security Services, the current Pervasive Services specification does not contain Security Services; OMG does, however, suggest that additional Pervasive Services may be defined eventually, either derived from the list of standard Common Object Request Broker Architecture (CORBA) services (which include security services), or added at the suggestion of OMG members.

A significant amount of research is being done outside OMG itself to extend MDA, for example by combining it with elements of Aspect Oriented Design (AOD), or by defining new UML-based security modeling languages and MDA-based secure design models. Tools such as ArcStyler from Interactive Objects and IBM/Rational Software Architect, offer support of some form of "model driven security" for security modeling and model checking and automatic code generation from security models.

However, as the majority of initiatives for security-enhancing UML or Aspect Oriented Software Development (AOSD), the various MDA-based security-enhancements undertaken to date have focused on modeling access control and authorization or security policy, but have not undertaken the challenge of modeling the nonfunctional properties that would make software in and of itself attack-resistant or attack-tolerant and attack-resilient.

Even without security extensions, use of MDA has been noted to reduce the number of quality defects in the modeled, auto-generated source code. Among the overall number of flaws and weaknesses avoided through use of MDA, some subset is likely to have security implications. But while MDA, like other quality engineering and

reliability engineering techniques, may coincidentally reduce software security flaws as it reduces overall flaws, on its own it can do little to help reduce the number of valid features or benign anomalies in the software that have no impact on the software's quality/reliability under normal usage by non-malicious users, but which manifest as vulnerabilities when targeted by attackers. Until MDA and its supporting tools enable modeling of non-functional software security properties, they will be of only limited use in improving the security of software.

4.3.3.2. Secure Object-Oriented Modeling with Unified Modeling Language

Most structured development methodologies, including methodologies supporting object-oriented development, do not focus on security. Rather, they treat security as a generic nonfunctional requirement for which no specific security-focused artifacts are needed. Recognizing this deficiency, some efforts have been undertaken to integrate security concerns into object-oriented development that includes creating models in Unified Modeling Language (UML). Most notable among those are UMLSec and Secure UML, which are described later in this section.

UML is the industry-standard language for specifying, visualizing, constructing, and documenting the artifacts of software. Though it is often used incorrectly to designate object-oriented modeling in general, UML is not a methodology per se. It provides only a standard notation and suggested set of modeling artifacts. It does not standardize or even explicitly describe development practices. While UML is inherently deficient in its lack of explicit syntax that supports security modeling (e.g., misuse and abuse case modeling), according to Gunnar Peterson in his article "Top Ten Information Security Considerations in Use Case Modeling", it is possible to apply a security-informed mindset to use case modeling. Unfortunately, Peterson's main focus is on modeling of security functions of software systems, versus modeling of security properties of software. Some of the points made in Peterson's article are worth noting by developers for whom use cases are a key artifact of their object-oriented modeling efforts.

1. **Negative requirements:** Where functional requirements usually stipulate that the system must do something, security requirements are often expressed in terms of ensuring that the system must *not* do something; when translated into the design, negative requirements manifest as constraints on functionality, e.g., upward ranges on acceptable input lengths.

2. **Non-functional requirements:** Functional requirements indicate what high-level business features are required. Nonfunctional requirements for properties tend to be more nebulous and lack of precise metrics for determining whether they have been satisfied in the implemented software. In practical terms, non-functional requirements must ultimately translate into functionality: for example, a requirement that the software must remain reliable in the face of external faults and failures may translate into a design that includes rigorous input validation, fault tolerance mechanisms, and particular modes of exception handling.

3. **Security tradeoff analyses:** Use case models provide a format in which to express the findings of the architectural tradeoff analysis of security mechanisms at different points in the system, and establish a basis for making the necessary tradeoff decisions.

4. **Security stakeholders:** The use case model documents all stakeholders that have some interest in the outcome of the development project. These stakeholders should include those who have an interest in the security of the system being developed, such as the certifier and accreditor.

5. **Security preconditions and post-conditions:** The use case should capture all security preconditions, such as required authentications and authorizations that must be completed before a user is able to access the functionality described in the use case. These preconditions may include enforcement of security policy that defines the acceptable states the software is allowed to enter. Post-conditions should document the set of security-relevant states possible at the end of the use case. For example, the post-conditions should capture what must be done by the software at the completion of the use case, for example

disabling the user's session, locking open accounts, deleting temp files and cache, releasing accesses to resources, and relinquishing privileges.

6. **Security implications of exceptional and alternate data and control flows:** A fundamental principle in security design is to plan for failure. From a security standpoint, exceptional and alternate flows in the use case highlight paths that often become attack vectors once the software is deployed. These flows should be reviewed to ensure that the system is not likely to enter an insecure state, and to identify areas in which to deploy security mechanisms such as audit logs and intrusion detection tools to catch security exceptions as they occur.

7. **Authorization of actors:** An analysis of the actors involved in the use case model forms the basis for defining the authorization structures, such as roles and groups, required to support the system's security policy.

8. **Modeling identity:** A digital identity is the result of a set of processes, such as authentication events that get mapped onto some principal(s) and are evaluated. Identity is a foundational element for security functionality. Use case modeling can help in the precise definition and use of rule sets for building, validating, and exchanging identities in a system.

9. **Security implications of use case relationships:** Use case models feature two types of relationships: includes and extends. Each relationship has direct security implications. For example the outcome of "including" an access control use case can alter the behavior of the related use case depending on whether the outcome of the access control is a "pass" or a "fail". The extends relationship, by contrast, does not alter behavior of the preceding use case. If a use case "extends" to a monitor an event and the monitoring agent is nonfunctional, the flow of the preceding use case may still be allowed to continue without being monitored.

10. **Mapping use cases to threat models:** Threat modeling maps possible threats to the system, and their impacts if they are successfully manifested. This enables the designer to designate the appropriate security countermeasures in the system to counteract those threats. The use case allows the threat model to address both an end-to-end and component-level view of the system and the disposition of countermeasures within it.

11. **Support for architectural security decisions:** The design tradeoffs necessitated by security do not exist in isolation. Use case modeling provides an end-to-end view of the system, enabling easier identification and analysis of the impacts of choosing to deploy or not to deploy certain security protections, etc., at different points in the system.

12. **Security test case development:** Security-centric use case elements should be associated with a set of test cases to demonstrate the system's ability to enforce security policy, and to handle the results of threats that successfully target it (misuse and abuse cases).

The following UML profiles are intended to add expressions for security functions or properties to UML.

- **CORAS UML Profile:** Developed under the umbrella of the CORAS Project (see Section 4.1.2.3), the CORAS UML profile for security assessment introduces a meta model that defines an abstract language for supporting model-based risk assessment. This profile provides a mapping of classes in the meta model to UML modeling elements by defining so-called "stereotypes". It also introduces special symbols ("icons") for representing these stereotypes in UML diagrams.

  The CORAS UML profile for security assessment was submitted to the OMG and adopted as a recommended standard by the OMG technical meeting in London in November 2003; it is now undergoing finalization. The focus of the CORAS UML profile is the modeling of threats, such as buffer

overflow exploits, and associated "treatments" (countermeasures).

Unlike UMLsec and SecureUML (see below), the CORAS UML Profile appears to be directly relevant to modeling software security properties and attributes (versus security functions implemented in software).

- **SecureUML:** Conceived by Torsten Lodderstedt, David Basin, and Jürgen Doser in the Institute for Computer Science at the University of Freiburg (Germany), and applied practically by Foundstone as the basis for their design of secure authorization systems, SecureUML is a UML-based modeling language for expressing Role Based Access Control (RBAC) and authorization constraints in the overall design of software systems. It is not clear whether SecureUML lends itself to further extension to support broader modeling of software security properties, such as integrity, availability, non-compromisability, non-exploitability, and resilience.

- **UMLsec:** The brainchild of Jan Jürens (Munich University of Technology and University of Oxford), UMLsec adds extensions to a formal subset of standard UML to produce a modeling language tailored to the needs of secure systems developers. UMLsec is intended to encourage developers to consider security requirements in a system context from the earliest design phases. The language enables the developer to evaluate UML specifications for security vulnerabilities in the system design based on established rules of secure engineering encapsulated in a checklist. Because it is based on standard UML, UMLsec should be useful to developers who are not specialized in secure systems development.

  UMLSec's key shortfalls are (1) its assumption that all underlying security algorithms used in the system are secure, and (2) the inflexibility of its modeling capabilities and validations that do not account for the unpredictability of attacks.

  As a modeling tool for security as a property, UMLSec, like SecureUML (and standard UML), appears to be inadequate, and indeed, as with SecureUML, its documented uses in the lab have been limited to modeling access control and authorization functions.

It has been suggested that object-oriented development and its focus on components (objects) exacerbates the problem of developing secure software. The discussion of AOSD in Section 4.3.3.3 provides some insight into the problem, though AOSD may fall short as a solution.

4.3.3.3. Using AOSD to Produce Secure Software

It has been observed that one weakness of object-oriented modeling is that it focuses only on security properties as they relate to specific software functionalities, but cannot efficiently capture "cross-cutting" security properties that hold true across the system. For example, UML does not include such a construct as a "misuse case" or an "abuse case", which makes it difficult to use UML to capture such properties as "resilience after a successful attack" or "reliability in the face of an intentional fault or weakness", although it is well-suited to capturing functionality-related properties such as "self-correction capability" or "detection of unintentional faults and weaknesses".

By contrast, Aspect Oriented Modeling (AOM, a phase of AOSD) specifically addresses this deficiency in object oriented modeling regarding depiction of cross-cutting properties. AOM extends the expressions possible using object-oriented modeling methods, such as RUP, MDA, and other methods that UML supports. The specific intent of AOM is to achieve good separation of concerns in the models of complex systems and to increase the expressiveness of object-oriented modeling to capture effectively the complexities of cross-cutting concerns such as security.

An AOM design model consists of a set of aspect models, the situations and in which cross-cutting occurs, and a primary model. The aspect models describe how a single objective is achieved in the design while the primary model is a traditional system architecture, depicting the functionality of the system. By separating the various

aspects from the primary model, it is possible to devise rules for weaving the aspect models with the primary model. Because the aspects and weaving rules are separate from the primary model, they can be reused. In lieu of available Aspect Oriented Programming (AOP) techniques, the aspect models can be combined with the primary model prior to implementation. This results in a stronger implementation in which the cross-cutting concerns themselves become modular portions (objects) within the system.

AOM supports expression of cross-cutting concerns such as security by allowing those concerns to be captured in a separate, modular component of the system model. The cross-cutting security concerns expressed in an aspect-oriented model essentially define the high-level requirements for the security capabilities and security properties within an integrated system or component assembly. These security aspects include the security properties or self-protecting characteristics of the software itself, such as non-bypassability and domain separation, and the security functions (or services) the software provides to its users, such as digital signature and authentication.

AOP enables the developer to write the software's nonfunctional one time at one location, then to have that code automatically inserted into the functional source code at appropriate points. This approach makes it easier for developers to tailor systems to satisfy implementation-specific nonfunctional requirements, such as those for security properties. There is some question about which point in the life cycle is appropriate for providing these nonfunctional options. It may be possible to implement them as configuration options, specified in a configuration file that is bound to the software during linking, at start up, or at runtime. Java EE (Java Platform, Enterprise Edition), for example, has a feature that delays some design decisions until deployment.

The problem with implementing these decisions in source code is that it is difficult to analyze code inserted for multiple nonfunctional requirements to determine how those inserted code modules interact and how a change to the aspect code will impact on the whole system. There is no construct similar to an object class, to encapsulate actions in AOP. Without such a feature, AOP must be used with caution if the resulting software is to be analyzable to verify its security. Indeed, for purposes of security assurance, the ability to analyze the software is more important than the simpler expression of cross-cutting security properties and attributes.

4.3.3.4. Can Agile Development Methods Produce Secure Software?

The collection of software development methodologies that fall under the umbrella of "Agile Methods", with one notable exception (noted in the list of agile methods below), all share in common their authors' commitment to the core principles of the Agile Manifesto. The Agile Manifesto makes it clear that an "agile" software process is more than just processes for producing software. "Agility" requires certain philosophical and cultural commitments from developers, managers, and customers.

According to Matthew Bishop of University of California at Davis in his book *Computer Security: Art & Science*, agile development methodologies are:

> *"…based on rapid prototyping and best practices such as separate testing components, frequent reviewing, frequent integration of components, and simple design. A project is driven by business decisions, not by project stakeholders, and requirements are open until the project is complete. The design evolves as needed to remove complexity and add flexibility. Programmers work in teams or pairs. Component testing procedures and mechanisms are developed before the components are developed. The components are integrated and tested several times a day.*
>
> *One objective of agile development is to put a minimal system into production as quickly as possible and then enhance it as appropriate. Use of this technique for security has several benefits and several drawbacks. The nature of an evolving design leaves the product vulnerable to the problems of an add-on product. Leaving requirements open does not ensure that security requirements will be properly implemented into the system. However, if threats were analyzed and appropriate security requirements developed before the system was designed, a secure or trusted system could result. However, evidence of trustworthiness would need to be adduced after the system was developed and implemented."*

The most noteworthy of the software methodologies that fall under the "agile" umbrella are listed in Table 4-3. With one exception, the developers of these methods are all committed to making and keeping their methods consistent with the Agile Manifesto. In addition to the "pure" agile methods listed in Table 4-3, there are a number of "hybrid" methods that strive to compensate for perceived inadequacies in a particular agile method by augmenting or combining that method with elements of another method. One example of an agile hybrid is XP@Scrum which, as its name suggests, combines practices from both XP and Scrum. In some cases, the two methods combined are not both agile, as is the case with Robert Martin's dxProcess, which is more an attempt to adapt the Rational Unified Process to exhibit some aspects of agility than it is to define a new agile method. Other methods, such as Context-Driven Testing, are highly compatible with and supportive of agile development.

While the differences between agile methods range from superficial (e.g., assigning different names to comparable life cycle practices) to more significant (e.g., some agile methods, such as Adaptive Software Development, Scrum, and Dynamic System Development Method, focus more heavily on project management and collaboration practices while others, such as eXtreme Programming, focus on software development practices), all agile methods promote life cycle practices which, in combination, are intended to mutually support each other by compensating for shortcomings or defects in the other practices. Moreover, with the exception of Lean Development, all agile development methodologies are consistent with the core principles of the Agile Manifesto. However, each methodology prioritizes and achieves those principles somewhat differently.

Table 4-2. Major Agile Methods

Method	Abbreviation	Author(s)/Affiliation
Agile Software Process	ASP	Mikio Aoyama/Nanzan University and Fujitsu (Japan)
eXtreme Programming	XP	Kent Beck, Ward Cunningham/Tektronix; Ron Jeffries/Object Mentor and XProgramming.com
Crystal family of methods	None	Alistair Cockburn/IBM
Adaptive Software Development	ASD	Jim Highsmith, Sam Bayer/Cutter Consortium
Scrum	None	Ken Schwaber/Advanced Development Methods; Jeff Sutherland/PatientKeeper
Feature-Driven Development	FDD	Jeff De Luca/Nebulon
Dynamic System Development Method	DSDM	DSDM Consortium (UK)
Lean Development	LD	*Bob Charette/ITABHI Corp.
Whitewater Interactive System Development with Object Models	Wisdom	Nuno Jardim Nunes/Universidade da Madeira; João Falcão e Cunha/Universidade dao Porto

**not committed to the Agile Manifesto*

The discussion below identifies aspects of the Agile Manifesto core values that may conflict with the need to produce secure software, and those that may actually aid in secure software development. Appendix C presents some approaches proposed by members of the agile community for addressing security within the context of agile development.

Some recent research efforts have been undertaken to combine agile approaches with security engineering in hopes of producing a kind of secure agile engineering method. Tappenden *et al* in their paper *Agile Security Testing of Web-Based Systems via HTTPUnit* suggest that "agile security engineering" can be achieved by applying the same values that drive agile software engineering to the traditional practice of mitigating security risks in software.

4.3.3.4.1. Areas of Potential Conflict Between the Agile Manifesto and Software Security

Below is a restatement of the Agile Manifesto's core principles. Those principles that have negative or ambiguous implications for software security are noted, as are those that are likely to contribute to security.

Table 4-3. Core Principles of the Agile Manifesto

No.	Principle	Implication for Security
1	The highest priority of agile developers is to satisfy the customer. This is to be achieved through early and continuous delivery of valuable software.	Negative, unless customer is highly security-aware. There is a particular risk that security testing will be inadequate or excluded because of "early delivery" imperatives.
2	Agile developers welcome changing requirements, even late in the development process. Indeed, agile processes are designed to leverage change to the customer's competitive advantage.	Negative, unless customer is careful to assess the security impact of all new/changing requirements, and include related requirements for new risk mitigations when necessary.
3	Agile projects produce frequent working software deliveries. Ideally, there will be a new delivery every few weeks or, at most, every few months. Preference is given to the shortest delivery timescale possible.	Negative, unless customer refuses to allow schedule imperatives to take precedence over security.
4	The project will be built around the commitment and participation of motivated individual contributors.	Neutral. Could be Negative when the individual contributors are either unaware of or resistant to security priorities.
5	Customers, managers, and developers must collaborate daily, throughout the development project.	Neutral. Could be Positive when all participants include security stakeholders (e.g., risk managers) and have security as a key objective.
6	Agile developers must have the development environment and support they need.	Neutral. Could be Positive when that environment is expressly intended to enhance security.
7	Developers will be trusted by both management and customers to get the job done.	Negative, unless developers are strongly committed and prepared to ensure security is incorporated into their process and products.
8	The most efficient and effective method of conveying information to and within a development team is through face-to-face communication.	Negative, as the assurance process for software is predicated on documented evidence that can be independently assessed by experts outside of the software project team.
9	The production of working software is the primary measure of success.	Negative, unless "working software" is defined to mean "software that always functions correctly *and* securely."
10	Agile processes promote sustainable development.	Neutral
11	The developers, as well as the project's sponsors and the intended users (either of whom could be the "customer"), should be able to maintain a constant pace of progress indefinitely.	Neutral
12	Agility is enhanced by continuous attention to technical excellence and good design.	Positive, especially when "technical excellence and good design" reflect strong expertise in and commitment to software security.
13	Simplicity, which is defined as the art of maximizing the amount of work not done, is essential to successful projects and good software.	Positive, if simplicity is extended to the design and code of the software as this will make them easier to analyze and their security implications and issues easier to recognize.
14	The best architectures, requirements, and designs emerge from self-organizing teams. At regular intervals, the team must reflect on how to become more effective, then tune and adjust its behavior accordingly.	Neutral

Of most importance in the context of this document are the embedded assumptions in the agile approach that have the potential to conflict with the priorities and needs of secure software engineering. Several aspects of agile development are frequently cited as being in conflict with the needs of secure software engineering:

- Agile requirements modeling and capture does not accommodate non-functional requirements;
- Test case definition and test execution in the context of TDD do not accommodate security tests;
- There is no role for security experts on agile development teams;
- Planning for software security is not easily accommodated on agile development projects.

The following are aspects of the Agile Manifesto's core principles that represent potential sources of difficult when the software to be produced must be secure:

- **Early, frequent, and continuous software deliveries:** One of the basic arguments used against agile development for secure software is that the focus on security adds time to the development process, and the Agile Manifesto's insistence on frequent deliveries within the shortest timescales possible means that agile methods by definition cannot be adapted to add extra time for anything. Security, if it is to be achieved, must be accomplished within the very short timescales of agile delivery schedules. In agile development, development never stops. It continues even as requirements and design change and software is iteratively tested. The highly iterative nature of the agile development life cycle would require any independent assessor to be involved in each iteration. This is impractical. Independent reviews already increase to the time and cost to produce software. The more involved the independent assessor (usually an expensive expert) must be throughout the life cycle, the larger the necessary increases in time and cost.

  For example, it has been estimated that an independent assessment of each release iteration in XP could potentially extend the time to produce that iteration from a few days to a few months or longer, depending on the software's size and complexity. The very Manifesto principles that make agile methods conflict with the need to "freeze" software at certain points to give an independent reviewer/tester time to comprehend fully the software's security and assurance implications. The need to continuously "respond to change" leads to frequent redesign (called *refactoring* in XP). Software components may be assigned new functionality which may no longer work effectively within the software's required security constraints.

- **Changing requirements, even late in the life cycle:** The Agile Manifesto emphasizes the need to respond to change over the need to follow a development plan. When the software being developed is high-consequence, any departure from plan can prove disastrous, as has been demonstrated by the software-originated crashes of satellites and failures of telephone systems. Agile proponents have argued that it was not responsiveness to change but failure to note current conditions when assessing the change's potential impact on the critical system properties that led to those catastrophes.

  To be successful, those who follow agile methods must respond to change *intelligently*, with full consideration given to all of the conditions under which change will take place, and all of the potential impacts of the change given those conditions. In agile development, testing is seen as the natural means of responding to change safely. Also, the value placed on responding to change must not outweigh the value placed on the need to produce working software, which in the case of software security means software that continues working even when exposed to intentional external faults and failures.

- **Daily collaboration between developers and customers:** The reliance of agile development on the ongoing direct communication between developers and their customers conflicts with the need for independent security reviewers and testers to maintain a distance from the development process to remain uninfluenced by close involvement with the development team or intimate familiarity with the software to be reviewed/tested.

- **Implicit trust in developers:** In agile development, the entire development team is held responsible for achieving the requirements of the customer. This includes the responsibility for achieving software security, if that is an explicit customer requirement. Agile development does not adapt easily to the inclusion of experts, including security experts, in the development team. Nor is there any place in the testing scheme for agile development for red teams and other "third parties" who are key to the independent verification of security assurance, as is required during the C&A of software systems. The role of security experts in agile development is discussed in Appendix C.

- **Preference for face-to-face versus written communication:** Agile software is developed very quickly, so the life cycle enters its maintenance/upgrade phase early and remains there for the majority of the project. In agile development, developers alone are responsible for maintenance, as well as for all reviews and tests. The developer has the benefit of intimate knowledge of the software throughout its life cycle, and whether that knowledge is gained through oral communications rather than more formal written documentation is irrelevant. However, independent security verification is based on the premises that the tester has no involvement with the development team to avoid being influenced by their priorities or opinions of the software.

  Lack of direct involvement with the development team and process also means the independent tester must rely in large part on the software's documentation to become familiar enough with the software to test it effectively. Because agile processes emphasize oral over written communications, production of written documentation may not be seen as a high priority by agile teams, particularly when faced with short delivery deadlines. It is very unlikely that such minimal documentation would be adequate to satisfy the needs of independent security testers and system certifiers.

- **"Working" software as the primary measure of success:** The focus of agile development is on producing software that achieves the most "valuable" functionality. This means that agile software testing focuses on verifying requirements compliance and correctness of functionality and classes. Agile security testing does not extend to penetration testing or other non-functional security tests, nor does it include key activities of software *security* testing, such as examining the least exercised parts of the software, simulating pathological user behaviors, violating software boundaries through input of intentionally incorrect values, stressing obscure areas of code to determine their vulnerability to exploit, and deeply analyzing the test scenarios themselves, to understand how thorough the tests are.

A few other aspects of agile development that conflict with the needs of software security arise from certain agile methods' or developer's interpretation of Agile Manifesto principles, rather than from explicit statements in the principles themselves. These aspects are discussed below.

- **Agile approach to design:** Because in agile development design is done "on the fly", the developer may not be able to recognize the broader security implications and impacts of each of his design decisions.

- **Agility as a philosophy, not just a methodology:** Few books on agile methods mention security, and those that do include little more than a superficial discussion. Since agile development is as much if not more a philosophy of software development as a methodology for a software process, agile detractors suggest that it will be difficult if not impossible to get agile developers to embrace the importance of security when achieving it may require them to violate (or at least bend) one or more of the core principles of the Agile Manifesto. Agile proponents argue that nothing in the Agile Manifesto forbids adding activities and practices "that are not explicitly mentioned in the [agile] literature". They suggest that any agile method can be adapted or extended to include the necessary security-related activities, such as threat modeling and independent (third party) security tests.

- **Test-driven development and software security testing needs:** Agile methods do not easily accommodate another core practice for software security (and safety) assurance: the *independent* review and testing of software by a disinterested third party, i.e., a security expert. Because there is never truly a

"frozen" software baseline in agile development (at least not one that lasts beyond a few days), independent security testing is really not practical in an agile context. Also, the volatility of software in agile development would render any Common Criteria evaluation for agile-developed software obsolete even before the Target of Evaluation could be documented. In addition, the preparation, execution, and analysis of results associated with security reviews and tests take more time than most agile methods allow.

On the other hand, agile development can lead to better unit-level security because of the inherent code-to-the-test philosophy. By producing unit tests first, then coding to achieve test success, security can be enhanced *if* intelligent, security-oriented tests are written first. Such tests could be engineered to check that a lot of common vulnerabilities have been avoided. Conventional software development methods have tended to add robust security towards the end of the development cycle. Agile programming, implemented with security as an explicit objective, may change that tendency.

- **Agile development versus secure configuration management:** Pair programming, a practice in XP, requires that all agile developers work in pairs, sharing a single workstation, to ensure that the code is examined continuously by the "observer" developer (the one who is not actually writing the code). Pair programming is claimed by XP proponents to decrease the number of faults—including security faults—discovered early in the implementation phase.

  In development environments in which sharing of a single workstation by two developers is unacceptable, the practical logistics of pair programming would have to be adjusted to ensure that a "second pair of eyes" still has near-continuous access to the software as it is written. Better yet, organizations that wish to implement truly secure development environments while also using an agile development process should choose an agile method that is more supportive of the secure environment objective.

4.3.3.4.2. Aspects of Agile Methods that May Benefit Software Security

As with all development methodologies, agile practices have the objective of encouraging good general software engineering. As noted earlier, one of the core principles of the Agile Manifesto is that "Agility is enhanced by continuous attention to technical excellence and good design". As noted in Section 3, any practice that improves software correctness may coincidentally improve software security by reducing the number of overall faults, among which some percentage are likely to have security implications.

Agile proponents often cite key practices included in many agile methods that are likely to reduce the number of exploitable faults and weaknesses introduced into software. Such practices include enforcing coding standards, striving for simplicity of design, test-driven development, pair programming, and continuous integration.

Test driven development (TDD, or "continuous testing") is a cornerstone of all agile methods. TDD requires every specified requirement to be reflected in the software's acceptance test plan. Developers write all test cases before they begin coding, enabling them to verify continuously that the implemented code has achieved all of its requirements and test objectives as it is written. The objective of TDD is to enable faults and other weaknesses to be detected and corrected as early in the development life cycle as possible. Agile testing is also automated to the greatest extent possible, to make it easier to run the continuous, iterative series of test cases against code as it is developed.

The flexibility and dynamism of agile methods would appear to fit the needs of risk-driven software engineering *if* (and this is an important "if") the requirements-driven mentality that underpins all agile methods can be modified to acknowledge that changing risk profiles and threat models are at least as important, if not more important, in terms of the agile need to "respond to change" as user-driven changes to functional requirements.

Agile development at this point is driven almost exclusively by functional requirements. The biggest challenge, then, may be to persuade agile developers to violate one of the core principles of the Agile Manifesto, the

"customer" as the only driver for requirements, and to embrace requirements-by-mandate (i.e., requirements driven by policy, directive, or law). Possibly the best way to make this change successfully is to redefine the concept of the "customer" as it is used in the Agile Manifesto, so that the "customer" includes all the stakeholders in the software, including the software system's accreditor (at a minimum), risk manager, any other personnel responsible for ensuring that the software begins and remains secure during its operational lifetime.

Equivalent practices are not found in all agile methods. The question of how well a particular agile development process can be adapted to achieve the objectives of software security probably comes down to determining whether the specific agile methodology used can be easily extended to accommodate the necessary security analyses, reviews, and tests throughout the life cycle (with the extra time and cost these represent). A related question of greater importance to agile advocates is whether such a security-enhanced methodology, if too many extra activities are added, can still be considered "agile".

4.3.3.5. Applying Formal Methods to the Development of Secure Software

Formal methods apply mathematical techniques and precise mechanisms for reasoning to the design, production, and evaluation of software. A formal method normally combines:

- **Specification:** Use of a mathematical or a logical model, called formal model, of the system or protocol along with its security requirements specified mathematically. In practice, specifications are partial—addressing portions of a system and specific aspects of requirements. System requirements that can be specified precisely include both functional attributes, which address the results computed by a system, and so-called non-functional attributes, which address the means by which those results are obtained, including security, performance, usability, etc. In addition, specifications can relate to the engineering of the system itself—its component structure, internal information flows, and other aspects of design intent;

- **Verification:** An effective and tractable procedure to determine whether the system or protocol produced satisfies its requirements. In practice, verification is accomplished using a variety of mathematically-based techniques, such as theorem proving (with a diverse range of logical formalisms), model checking, analysis-based verification and abstract interpretation, type inference, and others. In addition, verification can be supported by hybrid techniques, such as a combination of inference and testing.

Use of formal specification and verification methods helps remove ambiguity and clarify understanding of the software being produced. Besides the distinction of *means* noted above, there is also a distinction of *outcomes* between formal techniques and conventional techniques for software quality. Conventional techniques include primarily software testing, inspection, and design evaluation. These techniques are essential to create greater confidence that designs and implementations are consistent with requirements. But they generally can provide only partial assurances—they cannot give confidence that all possible cases are covered, even when assuming correctness of all other components in a system (e.g., the language compiler and runtime system). Testing, for example, makes use of coverage metrics, but these are only heuristic guides regarding how the test cases represent the potentially unbounded set of possible inputs and outputs.

Mathematically-based specification and verification, on the other hand, is intended to create a genuinely positive assurance regarding the consistency of specification and implementation for the attributes that are modeled (again, assuming correctness of other system components). These techniques use mathematical tools to enable comprehensive theorems to be proved for software systems. These are based on the fact that computer programs are, in fact, themselves mathematical objects that can be reasoned about.

4.3.3.5.1. Formal Methods and Software Engineering

Formal methods can be applied both *a priori*, i.e., as a software artifact is being developed, to limit the introduction of flaws (or faults, if the method is applied during implementation) into that artifact, and *a posteriori*, i.e., after the software artifact has been produced, to identify and remove any existing flaws, faults, or other

weaknesses. In practice, the most successful projects employ hybrid approaches, combining *a priori* and *a posteriori* usage. These projects also employ tools that support incremental progress by developers and verifiers. The approaches used at Kestrel and Praxis ("Correctness by construction" that focuses on functional attributes) and Microsoft (Software-specification, Language, Analysis, and Model-checking [SLAM], PRE*f*ast, Standard Annotation Language [SAL], Fugue, etc., that focuses on non-functional attributes), though very different in details, are similar in that they all take a hybrid *a priori/a posteriori* approach.

Different formal methods and languages cover different phases in developing a software system, from requirements definition to system specification, down to low-level design and implementation, as well as acceptance evaluation for outsourced components and Application Programming Interface (API) compliance for users of libraries and frameworks. The main benefit to using formal methods and languages is the ability to exploit tools that automate reasoning about a software system's description at levels of abstraction appropriate to each development phase.

In practice, there can be many such descriptions to enable factoring of a large problem into more tractable smaller problems. Each of the resulting descriptions can then also be manually reviewed and checked against earlier, higher-level descriptions to ensure consistency as details are progressively added. This facilitates *validation*, i.e., the establishment of consistency between the actual intent for the system and the documented specification of that system. In software engineering practice, many errors are validation errors—errors in which requirements are incorrectly captured. In this regard, formal techniques can assist in this aspect of software engineering.

4.3.3.5.2. Formal Methods and Secure Software Engineering

When applied to the problem of secure software engineering, formal methods have been used to specify and mathematically prove the correctness of security functionalities (e.g., authentication, secure input/output, mandatory access control) and security-related trace properties (e.g., secrecy). However, to date it remains a research challenge to develop formal methods for the specification and verification of *non*-trace security properties in software, such as non-subvertability of processes or predictability of software behavior under unexpectedly changing environment conditions associated with malicious input, malicious code insertions, or intentional faults and weaknesses.

Because software security properties must often be expressed in negative terms, i.e., in terms of what the software must *not* do, it is particularly difficult to specify requirements for those properties (formally or informally), and then to mathematically prove that those requirements have been correctly satisfied in the implemented software. These are often called *safety* properties, which are universal statements that "nothing bad will happen".

There are techniques to establish safety properties within software components when there are solid guarantees regarding the environment in which the software components are to be embedded. But the reality is that it is profoundly difficult for engineers who develop logical models to anticipate all potential changes in environment state so they can formally model the resulting changes in software state with precision. And, because the environment state is a physical reality that (in most cases) can be only partially modeled, it is effectively impossible to prove mathematically that, given an *unanticipated* change in environment state, the software will never react by entering a state it is not supposed to enter, perform a function it is not supposed to perform, or demonstrate a behavior it is not supposed to demonstrate. The problem is that, in the context of software embedded in physical environment, no methods, formal or otherwise, can yield guarantees regarding the absence of a certain behavior under all possible circumstances.

The inability to produce guarantees regarding physical systems is fundamental—it is not a limitation of these particular methods. It is, rather, a limitation of our ability to model the completeness of physical reality ranging, for example, from the enormous diversity of faults and failures in sensors and actuators to willful physical interventions on computer hardware. Indeed, this consideration suggests that, for major systems and embedded systems of all kinds, a practicable engineering approach will include a "defense in depth" toolkit including at least four components:

1. Diverse formal methods focused on various functional and non-functional attributes of software;
2. Testing and inspection to complement and validate;
3. Probabilistic models for reasoning about faults and failures in sensors and actuators;
4. Threat and vulnerability analyses to support modeling of exploitable faults and weaknesses.

In general, all reasoning in all sciences is done on models, with models being based on abstraction. More abstract models support easier proofs, but usually prove less. Finding the right level of abstraction depends on what we want to prove, and how much work we are able to spend on it. There are three security models that are widely used: information-theoretic, complexity-theoretic, and symbolic. The symbolic security models were offered to mitigate the sheer complexity of complexity-theoretic analyses, which in some cases seriously hampered proof verification. Symbolic secrecy proofs are considerably simpler, and thus less error prone than computational proofs, but they also prove less. This situation is somewhat analogous to higher-level versus low-level programming languages: the former are easier to use, the latter allow finer control; in all cases, the price to be paid in terms of complexity often results in less assurance, rather than more. Regardless, people keep using both high-level and low-level languages as appropriate, given the task to be performed.

The task of security engineering, it seems, is to define modeling methodologies at a level of abstraction that precludes introduction of vulnerabilities because of attempting to model more than the engineer can comprehend. Initial modeling should be done at an abstract level (usually a symbolic model). As needed, this model can then be refined to capture information-theoretic and computational aspects. These three models of security can then be seen as nesting in each other. Design and analysis should progress from the simplest to the most precise models, making the models more manageable and less error-prone.

Formal methods, like other techniques for improving software quality, have particular benefits. These benefits often complement the benefits of other techniques. There are important areas where formal methods and validations (e.g., theorem proving, software analysis, model checking, design analysis) can yield results that are not feasible with other methods. In other areas, particularly the modeling of physical faults and interventions, formal methods are less appropriate.

With respect to security, formal methods are like many other reliability engineering and quality engineering techniques: they increase the likelihood of overall software *correctness*, and thereby coincidentally decrease the likelihood that security flaws, faults, and weaknesses will appear in the software. In any complex system, assurance is achieved by combining methods that are appropriate to (1) the type of component being engineered (e.g., processor chip, sensor, actuator, software components in various languages) and (2) the requirements and challenges placed on the system by its environment and requirement. At the whole-system level, the end result may be a probabilistic measurement indicating significantly lower probabilities that certain kinds of faults and weaknesses remain in the software components of that system.

4.3.3.4.4. Successful Uses of Formal Methods to Solve Software Security Problems

The following are examples of successful uses of formal methods to the software security problem.

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

1. The integrity of how raw bits are interpreted in software as abstract values;

2. The integrity of the access pathways to those values. Type checking affects all software engineering practices using modern languages.

Model Checking:
Model checking is a formal method that occasionally produces results that contain no false negatives. Model checking usually involves a depth-bounded search for counter-examples with an arbitrary depth bound. With some analysis, it is possible to determine whether a model checking result is trustworthy enough to form the basis for positive assurance; however, such a determination is not intrinsic to the technique.

An example of model checking is implemented by the SLAM tool developed by Microsoft Research, and based on federally-funded formal methods research done in academia. Most "blue screens of death" in Microsoft products of the 1990s were caused by faulty device driver code developed by independent third parties. The SLAM tool uses model checking and binary decision diagrams to directly assess protocol compliance in these device drivers. The assurance evidence produced by the tool has become a prerequisite to Microsoft's decision as to whether to include various device drivers in Windows XP. Moreover, use of SLAM has greatly reduced the frequency of "blue screens of death" since 2001.

In addition to SLAM, Microsoft uses specifications (with the SAL tool), deep program analysis (with the PRE*f*ast tool), and other tool-driven techniques based on formal methods.

4.3.3.5.3. Limitations of Formal Methods

It is important that the benefits of formal methods not be oversold. The techniques are not equally applicable to all problems. For example, while formal methods could be applied to user interface design, they are not the most effective technique for building what is essentially an informal model. The techniques are also sensitive to choices of programming language, design notations, specification languages, functional and non-functional attributes to be evaluated, and so on. Perhaps most importantly, the techniques vary widely in the way that scale is addressed.

In addition, some techniques support *composability*—integrating findings about separate components into findings about the overall system—while other techniques are not composable—forcing evaluation of an entire subsystem or major component. When the complexity of the verification process is high, this kind of scale-up may not be computationally feasible. Composability is thus desirable both as a way to handle scale and as a recognition that components of a system may be developed separately, even by separate organizations, and so would best be verified separately.

In practice, both scalability and composability have proven elusive for many formal techniques. A final consideration is usability. This includes both training—the extent to which the persons who are using verification tools must have special mathematical background and skills—and incentives—the extent to which developers have intrinsic interest in employing the techniques (for example, because the tools have side benefits for productivity or because developers can more effectively warrant quality attributes in their specifications, designs, or code).

To be successful in using formal methods, developers require expertise and significant knowledge in how those methods are most appropriately applied. While not everyone on the development team needs the same level of proficiency in formal methods, all must appreciate the role of those methods in the development life cycle. Formal methods have been most successfully applied in engineering high-consequence software, because the extra developer expertise and knowledge, and the additional effort and cost, are more easily justified for life-critical software. Formal methods have also proven successful in specification and mathematical proof of small, well-structured security logic systems, such as cryptographic algorithms, operating system reference models, and security protocols.

There are other situations in which formal methods, appropriately deployed, may actually improve productivity and reduce costs. This is likely true for the type-checking systems built into modern programming languages

(Java, C#, and Ada95, for example) and some highly attribute-specific tools, such as Microsoft's SLAM used to assist both in development and in acceptance evaluation for Windows device-driver protocol compliance. There is already economic evidence emerging (Barry Boehm) that aggregate life cycle costs may be equal or lower for high-confidence systems even when development costs may be higher.

4.3.4. Emerging Software Development Methodologies

Several new security-enhanced SDLC methodologies are under development, including:

- **Appropriate and Effective Guidance for Information Security (AEGIS):** a secure software engineering method based on the spiral model of software development. Developed through research by the University College London Department of Computer Science, and evaluated through a series of case studies jointly funded by the UK Engineering and Physical Sciences Research Council and British Telecom, the objective of AEGIS is to integrate security and usability concerns into UML modeling of functional requirements. In this way, AEGIS is intended to help developers overcome the complexity of integrating security into software-intensive systems designs while simultaneously ensuring that those designs result in systems that are usable.

- **RUPSec:** Researchers in Tehran have proposed a set of extensions to RUP that they have designated RUPSec. The proposed extensions are intended to add and integrate a number of activities, roles, and artifacts into RUP that will capture, document, and model threats and security requirements for the system under development. The researchers acknowledge that their RUPSec extensions do not yet cover all RUP disciplines. Additional disciplines are being addressed in their ongoing research.

4.3.5. Visa U.S.A. Payment Application Best Practices

As part of its Cardholder Information Security Program (CISP), the credit card company Visa U.S.A. developed a set of Payment Application Best Practices (PABP) to assist the software vendors of payment applications in developing those applications to support their users' (i.e., merchants and service providers, including those who operate online) compliance with the Payment Card Industry (PCI) Data Security Standard.

In addition to promoting their best practices through a vendor education program, Visa has instituted a certification scheme that enables a vendor to contract with a Visa-Qualified Data Security Company (QDSC) to audit the vendor's application to determine whether it does indeed conform to the PABP. Those vendors whose applications pass the QDSC's audit will have their applications added to Visa's list of CISP-Validated Payment Applications; Visa encourages all of their merchant/service provider members to use the validated applications on this list. To maintain their applications on the list, vendors must also undergo an Annual On-Site Security Assessment by a QDSC in accordance with the PABP document.

The PABP themselves are presented in an application security checklist format. The PABP checklist includes descriptions of the best practice requirements, and provides test procedures designed to enable vendors to verify that the PABP practices have been satisfied in their applications. The checklist also acts as a test report, by providing blank fields into which the vendors can enter their test results.

As noted above, the objective of the collective PABP is to ensure that applications are compliant with the PCI Data Security Standard. For this reason, the majority of the thirteen (13) best practices and the supporting sub-practices in the document focus on the security functionality and data protections provided by the application. However, there are three best practices that address software security issues rather than security functionality:

1. Best Practice #5, "Develop secure applications", promotes the use of secure coding guidelines, and recommends that the application be checked against the Open Web Application Security Program (OWASP) Ten Most Critical Web Application Security Vulnerabilities to ensure that the secure coding guidelines used were adequate. In addition, Best Practice #5 recommends that applications be developed

"based on industry best practices" with information security considerations to be included "throughout the software development life cycle".

2. Best Practice #7, "Test applications to address vulnerabilities", encourages the vendor to establish a vulnerability management program that includes subscription to vulnerability alert services, and establishment of a secure, disciplined patch management program for all acquired software components and systems in the application itself and its execution environment.

3. Best Practice #10, "Facilitate secure remote software updates", promotes the practice of trusted distribution of software updates from vendor to customer system.

The PABP document is structured in the form of a checklist, and appears in the list of software and application security checklists in Appendix E.

4.4. Security Awareness, Education and Training

According to Glen Kunene, in his interview-article "Security Training Falling Through the Education Cracks", "Even today, the average developer is insufficiently trained in secure coding practices, and few universities are paying any attention." In the same article, interviewee Brian Cohen, Chief Executive Officer of SPI Dynamics, goes further to observe "Our universities are letting us down." In the vast majority of universities developers are taught that the highest-value principles for good software are functionality and performance; security, if taught at all, is characterized as an "optional" principle that runs a distant third in importance behind functionality and performance, or it is marginalized as applicable only in specialty software written for cryptosystems and network security protocols. One Johns Hopkins professor explained in his interview for Kunene's article, the main obstacle to adding security to computer science curricula: "Most of the tenured faculty view secure coding techniques as an exotic, boutique discipline, not as part of the core curriculum for computer science."

Microsoft's Steve Lipner and Michael Howard bemoan the inadequacy of computer science education. In their description of the SDL for this document, they stated "The majority of software engineers know little about building secure systems, and there is no near-term indication that educational institutions are likely to make the changes needed to introduce their students to the security challenges of today's networked environment." Most developers are not being taught how to recognize and understand the security implications of how they specify and design software, write code, integrate/assemble components, test, package, distribute, and maintain software—and to gain the knowledge they need to change their current development practices to do all those things more securely.

Resigned, at least temporarily to the inadequacy of university computer science programs, an increasing number of firms are undertaking their own in-house training and certification programs to teach their own developers how to write software securely. In their whitepaper on the SDL (published on the Microsoft Developer Network), Lipner and Howard observe "An organization that seeks to develop secure software must take responsibility for ensuring that its engineering population is appropriately educated…. It is critical that software development organizations provide in-house security education to all personnel developing software." Microsoft as well as Oracle Corporation (which has published a whitepaper on its Software Security Assurance Process) can be seen as pioneers and role models in developer security education: not only do the firms require their development personnel to receive training in secure development, they have also adjusted their employees' performance assessment and compensation schemes to reinforce their educational mandates.

Microsoft, for example, has established an extensive internal annual training program as part of its SDL initiative. This program could serve well as a model for practitioner training in all software development organizations. Microsoft's annual classes for its software engineering personnel cover the following topics:

- Security basics (all new hires);
- Threat modeling (designers, program managers, architects);

- Secure design principles (designers, program managers, architects);
- Implementing threat mitigations (developers);
- Fuzz testing (testers);
- Security code reviews (developers);
- Cryptography basics (all personnel involved in software development).

Some Microsoft classes are delivered through classroom lectures; others, to better accommodate employee schedules, are provided through online courseware whenever possible. Some classes include short labs and hands-on exercises. Microsoft plans to introduce more labs into their curriculum over time.

In addition to in-house training programs, an increasing number of specialty training firms are attempting to fill in the gaps in developer education by offering not only single classes but also comprehensive software assurance and secure programming curricula. The Software Security Summit represents another approach to practitioner training: it is a three-day annual "intensive" of tutorials and presentations on secure development topics.

Even on the university front, the picture is changing, albeit very gradually—as slowly, in fact, as small pockets of academics at one university at a time. Matthew Bishop at University of California at Davis, one academic who has embraced the importance of secure software, has observed that "defensive programming" and secure software principles can be taught to student developers by:

1. Emphasizing these principles in standard programming classes;
2. Critiquing programs and documentation;
3. Requiring students to apply them in all their programming assignments in all their classes, while also requiring specially-tailored "defensive programming" exercises to help drive the importance of secure software home.

In his program at UC-Davis, that is exactly how Bishop teaches—with the result that University of California-Davis has received grants for secure software research projects from several agencies, including NASA and the National Security Agency (NSA). Another leading advocate of software security education is Sam Redwine of James Madison University (JMU). Redwine not only spearheaded the establishment of JMU's Master's concentration in secure software, but has led the DHS effort to define a software assurance *Common Body of Knowledge* (described in Section 4.4.1) to form the basis for other universities, colleges, and technical training programs to develop their own secure software engineering curricula.

Although internalizing the importance of secure software principles will not solve all security problems related to software, it will instill in student developers the discipline of thinking through all possible errors, of checking for corrupt inputs and parameters, and of programming defensively to handle unexpected problems. The result will be that they write programs that are better thought out, better structured, and more secure. Defensive programming teaches students how to look for hidden assumptions in their programs, and how to defend against attempts to exploit those assumptions to cause their programs to behave in unexpected or undesirable ways. Questioning assumptions leads the student developer to new insights and a deeper understanding not only of security but of computer science in general—an important goal for any developer's education.

All of the above said, it is neither fair nor accurate to lay the blame for lack of secure software engineering entirely on doorstep of trainers and educators. Even software that has been developed to absolutely secure standards will become vulnerable if it is run in an inherently insecure execution environment (it is the operating system after all, that actually controls execution and memory). The discussion of technological precommitments in Section 3.5.1.1 becomes particularly important in this regard, as does the discussion of secure operating environments in Appendix G:G.3.3. Moreover, simply engaging in good software development practices, even if they are not expressly "secure", should go a long way towards preventing common developer mistakes, such as

poor memory allocation and buffer management practices, lack of input validation, and throwing general exceptions for all failures with no regard for cause of the failure or the impact of a hard failure.

Another major influence on the security of software is the peer review of code. As long as the "time to market" imperative trumps all other considerations, this will remain a critical problem. It won't matter how well educated developers are in secure software engineering practices if their management never allows them the time or resources they need to do their job correctly. This is the reason that security awareness needs is needed not just for developers and project managers, but also for senior managers and customers. See Appendix G for more information on awareness campaigns that have been undertaken in industry for the latter groups.

4.4.1. DHS Guide to the Software Assurance Common Body of Knowledge

To assist those in academia and the technical training industry in teaching secure software principles and associated practices, the DHS has produced a *Software Assurance: A Guide to the Common Body of Knowledge to Produce, Acquire, and Sustain Secure Software* (heretofore referred to as "the CBK") that can be downloaded from the DHS Software Assurance BuildSecurityIn portal. Produced by the DHS and DoD jointly-sponsored Software Assurance Workforce Education and Training Working Group, the CBK is intended to contribute to the adequate education and training of current and future software practitioners in the security of software. The ultimate goal of the CBK is to eliminate skill shortages in government and industry while also satisfying curriculum needs in universities, colleges, trade schools, and technical training programs.

Specifically, the CBK seeks to provide information that can:

- Influence and support the software engineering curricula produced by academia;
- Provide a basis for extending/improving the content of technical training provided to the current workforce;
- Assist personnel responsible for software purchases/acquisition in obtaining (more) secure software;
- Help evaluators and testers recognize software security-related criteria beyond what is covered in their current evaluations, and consider the implications of those criteria for evaluations of persons, organizations, and products;
- Encourage and facilitate standards bodies to include security-related items in current and future standards;
- Provide software practitioners with a resource for self-education in secure software engineering or acquisition of secure software.

In addition, the CBK seeks to influence the content of future editions of the IEEE Computer Society's *Guide to the Software Engineering Body of Knowledge*.

The information in the CBK addresses the following:

1. Identification of "conventional" software development activities and their "conventional" aspects;
2. Additional activities or aspects of activities that are relevant for producing secure software;
3. Knowledge needed to perform these additional activities.

Using this information, a teacher or trainer should be able to better craft software engineering curricula or training classes that incorporate items 2 and 3 into their approach to teaching item 1.

4.4.2. University Programs with a Software Security Focus

A number of schools have added secure programming classes to their curricula, but only a few universities have joined JMU in offering degree concentrations or specialties that reflect a recognition of the importance of security as part of the software engineering discipline.

A larger number of schools have well-established research programs and laboratories devoted to various aspects of software security. These programs and labs provide students (usually at the graduate level) with in-depth exposure to software security principles, reinforced by hands-on activities such as threat modeling, vulnerability assessment, "defensive programming", methodology validation, etc.

Below is a list of universities with degree programs or concentrations in secure software engineering, or as well as those that have active, ongoing research programs/laboratories devoted to secure software engineering. Schools with only individual research projects are not listed here.

4.4.2.1. North America

- Auburn University Samuel Ginn College of Engineering IA Laboratory Research Areas: (1) Software Vulnerability Assurance, (4) Software Process for Secure Software Development, (5) Reverse Engineering, (6) Artificial Intelligence for Vulnerability Assessment
  http://www.eng.auburn.edu/users/hamilton/security/Information_Assurance_Laboratory_Research_Areas_Dec_2003.html

- Carnegie Mellon University: CyLab—Software Assurance Interest Group
  http://www.cylab.cmu.edu/default.aspx?id=177

- Carnegie Mellon University: Software Engineering Institute
  http://www.sei.cmu.edu/

- James Madison University M.S. in Computer Science, Concentration in Secure SW Engineering
  http://www.cs.jmu.edu/sse/

- Northeastern University: Institute for Information Assurance Software and Architecture Security
  http://www.iia.neu.edu/iia_architecture.html

- Northern Kentucky University Graduate Certificate in Secure Software Engineering
  http://mscs.nku.edu/faq.html#q1

- Purdue Center for Education and Research in Information Assurance and Security (CERIAS) Software Vulnerabilities Testing Group
  http://www.cerias.purdue.edu/about/history/coast/projects/vuln_test.html

- Purdue University Secure Software Systems (S3)
  http://www.cs.purdue.edu/s3/

- State University of New York (SUNY) at Stony Brook: Secure Systems Lab
  http://seclab.cs.sunysb.edu/seclab/

- University of California at Berkeley: The Software Security Project
  http://www.cs.berkeley.edu/~daw/research/ss/

- University of Colorado at Colorado Springs: Certificate in Secure Software Systems
  http://eas.uccs.edu/CS/Graduate/certsecuresoftwaresystems.php

- Walden University M.S. in SW Engineering, Specialization in Secure Computing
  http://www.waldenu.edu/c/Schools/Schools_2740.htm

4.4.2.2. Europe and Australia

- Bond University (Queensland, Australia) Centre for Software Assurance
  http://www.sand.bond.edu.au/

- Deutsches Forschungszentrum für Künstliche Intelligenz (German Research Center for Artificial Intelligence): Transfer Center - Sichere Software (SiSo; Secure Software)
  http://www.dfki.de/siso/

- Fraunhofer Institute Experimentelles Software Engineering (IESE) (Kaiserslautern, Germany): Department of Security and Safety
  http://www.iese.fhg.de/Core_Competencies/ITS/

- Munich (Germany), Technical University of, Faculty for Informatics Chair IV: Competence Center in IT Security, Software & Systems Engineering
  http://www4.in.tum.de/research/security/index_en.shtml

- Oldenburg (Germany), University of: TrustSoft Graduate School of Trustworthy Software Systems
  http://trustsoft.uni-oldenburg.de/

- Oulu University (Finland) Secure Programming Group
  http://www.ee.oulu.fi/research/ouspg/

- Radboud University Nijmegen (Netherlands) Institute for Computing and Information Sciences Laboratory for Quality Software (LaQuSo): Security of Systems (SoS) Group
  http://www.sos.cs.ru.nl/

4.4.3. Secure Software Awareness Initiatives

In addition to DHS's Software Assurance program, which includes a number of awareness initiatives including the BuildSecurityIn portal and the Business Case Working Group, some other noteworthy awareness initiatives have been undertaken in industry. As they are more relevant to high-level managers than to those directly involved in software development, two prominent current initiatives are described briefly in Appendix G.

4.4.4. Software Security Professional Certifications

To date, with very few exceptions, none of the professional certification bodies for software developers or security engineers offers certifications in software security or secure software development, or even includes such information in their required bodies of knowledge (BOKs). Two exceptions are described in Sections 4.4.4.1 and 4.4.4.2.

There are also several systems engineering, software engineering, software testing, and IA certifications that include some secure systems engineering, software dependability, or application security information in their BOKs; these certifications are listed in Section 4.4.4.3. Until a software security certification by a well-known certification body is well-established, one or more these other certifications may be desirable for developers or testers who wish to demonstrate their knowledge of software security relevant issues, techniques, and technologies.

4.4.4.1. EC-Council Certified Secure Programmer and Certified Secure Application Developer

The International Council of Electronic Commerce Consultants (EC-Council) has established what appear to be the first and only professional certifications of secure software development expertise. The Certified Secure Programmer (ECSP) and Certified Secure Application Developer (CSAD) certification programs are designed to ensure that programmers and developers are informed about the inherent security drawbacks in various programming languages and software architectures, and to educate and encourage them to use secure programming practices to overcome these inherent drawbacks and avoid vulnerabilities in their software.

The ECSP program is intended for all application developers and development organizations, while the CSAD standardizes the knowledge gained about secure application development by incorporating best practices promoted by experts in various application and vendor domains.

Unlike vendor or domain specific certifications, the ECSP and CSAD exposes the developer to the security issues associated with a variety of programming languages and application architectures and platforms, in hopes of giving developers a chance to learn from the strengths and weaknesses of technologies other than those they use in their own development projects.

To obtain the ECSP certification, the developer must pass EC-Council's Certified Secure Programmer exam. To obtain the CSAD certification, the developer must first obtain an EC-Council-approved vendor application certification (includes a variety of Linux vendors', Microsoft, Sun Microsystems, Oracle, and IBM certifications), then pass EC-Council's Certified Secure Programmer exam.

In addition to its ECSP and CSAD, the EC-Council offer Licensed Penetration Tester (LPT), Certified Security Analyst (CSA) and Certified Ethical Hacker (CEH) certifications.

4.4.4.2. Secure University Software Security Engineer Certification

To obtain the Secure University (SU) Software Security Engineer Certification (SSEC), the application must first complete the following hands-on classes offered by the SU:

- Fundamentals of Secure Software Programming
- How to Break and Fix Web Security
- How to Break and Fix Software Code/Security
- Hacking Software—Attacker Techniques Exposed
- Software Security Testing Best Practices
- Software Testing Bootcamp (hands-on lab-oriented complement to Hacking Software)
- Software Security Penetration Testing
- Introduction to Reverse Engineering

Collectively, these classes cover the following information:

- Introduction to software security, including a discussion of common software coding and design errors and flaws, system-level issues, data and information disclosure issues, network-based issues, and tools.
- Web vulnerabilities;
- Threats and tools, including threat modeling and incorporating threats into software/system design, development, testing and deployment;

- Defensive coding principles;
- Security testing and quality assurance, including penetration testing;
- Reverse engineering.

According to the SU, the classes and SSEC are intended for software testers, software developers, development and test managers, security auditors and anyone involved in software production for resale or internal use will find it valuable. Information Security and IT managers; Information Assurance Programmers; Information Security Analysts and Consultants; Internal Auditors and Audit Consultants; QA Specialists.

4.4.4.3. Other Certifications with Software Security-Relevant Content

Table 4-4 lists several professional certifications intended for software or security practitioners; these certifications include software security-relevant content in their BOKs.

Table 4-4. Software Security-Relevant IA and Software Certifications

Certification Name	Sponsoring Organization	Intended Audience
GIAC (Global Information Assurance Certification): specifically the GIAC Level 5 Web Application Security (GWAS), GIAC Level 6 Reverse Engineering Malware (GREM), and GIAC Security Malware (GSM) certifications	SANS Institute	IA professionals
ISSPCS Practitioner, Professional, Mentor, and Fellow certifications. "Practioner" is similar to CISSP, but with an SSE-CMM viewpoint and emphasis	International Systems Security Professional Certification Scheme (ISSPCS)	IA professionals
CISSP (Certified Information Systems Security Professional) certification with ISSEP (Information Systems Security Engineering Professional) concentration	International Information Systems Security Certification Consortium (ISC^2)	IA professionals who do information systems security engineering for DoD and/or Intelligence Community
Certified Reliability Engineer (CRE)	American Society for Quality (ASQ)	Software practitioners
Certified Software Development Professional (CSDP). The CSDP is the most highly regarded professional certification for software engineers	IEEE Computer Society	Software engineers
Certified Software Test Professional (CSTP)	International Institute for Software Testing (IIST)	Software testers
Multiple professional qualifications for software and systems engineers	Information System Examination Board (ISEB) of the British Computer Society (BCS)	Software practitioners, systems engineers
Microsoft Certified Application Developer (MCAD); Microsoft Certified System Developer (MCSD). Microsoft now offers two elective software security-relevant MCAD/MCSD exams: (1) Implementing Security for Applications with Microsoft Visual Basic .NET and (2) Implementing Security for Applications with Microsoft Visual C# .NET. As yet, these exams are purely optional: they do not count towards the four Core Exams that are required to attain an MCAD or MCSD certification.	Microsoft Corporation	Software developers

4.5. Minimum Set of Acceptable Secure Software Engineering Practices

In *Writing Secure Code*, Michael Howard and David LeBlanc state:

> *"To better focus on security…add process improvements at every step of the software development life cycle, regardless of the life cycle model [in] use." They go on to say that "simply adding some 'good ideas' or a handful of 'best practices' and checklists to a poor development process will result in only marginally more secure [software]."*

This caveat notwithstanding, any strategy for security enhancing the software development life cycle should have as its ultimate goal the establishment of a repeatable life cycle process and supporting software development methodology (or methodologies) that explicitly address security issues. The move towards more secure software should not wait until that ultimate goal is reached. There are "good ideas" and "best practices" (or, more accurately, *sound* practices) that, while they will certainly not achieve completely secure software, when tactically inserted into current life cycle activities, regardless of process model or methodology being used (or in the absence of either), will enable developers to begin seeing at least some modest improvements in the software they produce.

The following checklists (Sections 4.5.1 and 4.5.2) represent a minimum acceptable set of secure software engineering practices that a software engineering organization should adopt sooner rather than later to begin improving the security of the software they produce. These practices should be seen as intermediate steps on the evolutionary path towards the ultimate goal of establishing a repeatable, disciplined security-enhanced life cycle process.

Appendix G examines and elaborates on all of these practices, and suggests additional practices that can be adopted to further enhance the security of the development process and increase its likelihood of producing secure software.

4.5.1. Whole Life Cycle Practices

The following practices span the entire software life cycle.

1. Security entry and exit criteria: For each life cycle phase, establish security entrance and exit criteria for both development processes and development artifacts, and enforce compliance with those criteria throughout the software's lifetime.

2. Secure configuration management: Establish and maintain secure CM practices, with supporting technology, for all software artifacts throughout the software's lifetime.

3. Security Risk Management:

    a. Perform iterative threat and vulnerability assessments and re-assessments throughout the software's lifetime.

    b. Revise development artifacts (e.g., requirements and design specifications, from-scratch code base) and issue interim security patches as needed, to counteract new threats and mitigate discovered vulnerabilities.

4.5.2. Life Cycle Phase-Specific Practices

The following practices are specific to one or more life cycle phases.

1. Requirements Engineering Phase: Specify requirements for security properties and constraints on functionality, not just for security functionality.

2. Architecture and Design Phases:

    a. Isolate all trusted/privileged functions.

    b. Avoid using high-risk services, protocols, and technologies.

    c. Use white-box and black-box techniques to evaluate the trustworthiness and secure behavior of all reusable (commercial, open source, public domain, and legacy) components before committing to use them. Do not use insecure or untrustworthy components unless they can be effectively isolated and constrained to prevent their negative impact on the rest of the system.

    d. Include only necessary functionality in from-scratch components. Remove or isolate unnecessary functions in acquired/reused components.

    e. Perform comparative security assessments of different assembly/integration options using different combinations of candidate components before committing to any component.

3. Implementation Phase:

    a. Minimize code size and complexity and write code to be traceable.

    b. Separate data control and program control.

    c. Enforce a consistent coding style across all from-scratch components.

    d. Whenever possible, choose a programming language(s) with positive security features and low security risk (e.g., Java rather than C; Perl rather than AJAX).

    e. Assume that all input will be hostile, and validate it accordingly.

    f. Implement security-aware exception handling that anticipates and safely reacts to environment-level security incidents and failures.

    g. Use security-aware assembly and integration techniques.

4. Testing Phase:

    a. Use white box techniques (e.g., security code review) to locate and remove vulnerabilities in from-scratch components before assembling/integrating those components into the system.

    b. Use black box techniques to locate vulnerabilities in/assess the security robustness of the whole system before its distribution/deployment.

5. Distribution Phase:

    a. Clean up code before distribution to remove debugger hooks, hard-coded credentials, informative comments in user-viewable code, unused calls, data-collecting trapdoors, default accounts and groups, relative pathnames, pathnames to unreferenced, unused, or hidden files, etc.

    b. Set configuration defaults as restrictively as possible before distribution.

    c. Implement trusted distribution techniques (e.g., code signatures, authentication of download links) to prevent or make evident tampering with distributed code.

6. Sustainment Phase:

    a. Perform impact assessments, and address unacceptable impacts, for all patches before distribution.

    b. Use software rejuvenation and reconfiguration techniques to prevent vulnerabilities that emerge due to software ageing.

4.6. References for this Section

J.D. Meier, Alex Mackman, Blaine Wastell, Prashant Bansode, and Chaitanya Bijwe: Security Guidelines: .NET Framework 2.0 (October 2005)
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag2/html/PAGGuidelines0003.asp

Matthew Bishop: *Computer Security: Art and Science* (Addison-Wesley, 2002)
http://nob.cs.ucdavis.edu/book/book-aands/

Rex Black: Improving Software Security: Seven Steps to Reduce Security Risk in the Software You Build (2006)
http://www.rexblackconsulting.com/publications/Improving%20Software%20Security.pdf

Charles H. LeGrand, CHL Global Associates: Software Security Assurance: A Framework for Software Vulnerability Management and Audit (Ounce Labs, 2005)
http://www.ouncelabs.com/audit/

Frank Swiderski and Window Snyder, Microsoft Corporation: *Threat Modeling* (Microsoft Press, 2004)

CORAS: A Platform for Risk Analysis of Security Critical Systems
http://www2.nr.no/coras/

The CORAS Project
http://coras.sourceforge.net/

CORAS: A Tool-Supported Methodology for Model-Based Risk Analysis of Security Critical Systems
http://heim.ifi.uio.no/~ketils/coras/

SECURIS: Model-Driven Development and Analysis of Secure Information Systems
http://www.sintef.no/content/page1____1824.aspx

The SECURIS Project: Model-Driven Development and Analysis of Secure Information Systems
http://heim.ifi.uio.no/~ketils/securis/index.htm

PTA Technologies: Practical Threat Analysis for Securing Computerized Systems
http://www.ptatechnologies.com/

Trike: A Conceptual Framework for Threat Modeling
http://dymaxion.org/trike/

Tools: Demo Versions of Trike
http://www.octotrike.org/

University of California-Davis: Tools that Check Programs
http://seclab.cs.ucdavis.edu/projects/testing/

NIST Draft SP 800-26 Revision 1, "Guide for Information Security Program Assessments and System Reporting Form" (revision 1, August 2005)
http://csrc.nist.gov/publications/drafts.html#sp800-26rev1

NIST SP 800-30, "Risk Management Guide for Information Technology Systems" (July 2002)
http://csrc.nist.gov/publications/nistpubs/800-30/sp800-30.pdf

NIST: ASSET
http://csrc.nist.gov/asset/

SEI: OCTAVE and OCTAVE-S
http://www.cert.org/octave/

Siemens/Insight Consulting: CRAMM
http://www.insight.co.uk/products/cramm.htm - *and* -
http://www.cramm.com/

Amenaza SecurITree
http://www.amenaza.com/

Isograph AttackTree+
http://www.isograph-software.com/atpover.htm

Emory A. Anderson, et al: Subversion as a Threat in Information Warfare
http://cisr.nps.navy.mil/downloads/04paper_subversion.pdf

Attacks on Web Applications
http://www.kriha.de/krihaorg/docs/lectures/security/Webattacks/

Dan Sellers, Microsoft Canada: Threat Modeling
http://silverstr.ufies.org/threatmodeling.ppt

Microsoft Corp: Threat Modeling Web Applications —
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag2/html/tmwa.asp

Microsoft: Improving Web Application Security: Threats and Countermeasures
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/ThreatCounter.asp

Microsoft Threat Analysis & Modeling v2.0 BETA2
http://www.microsoft.com/downloads/details.aspx?familyid=aa5589bd-fb2c-40cf-aec5-dc4319b491dd&displaylang=en

P. Torr, Microsoft Corporation: Guerrilla Threat Modeling
http://blogs.msdn.com/ptorr/archive/2005/02/22/GuerillaThreatModelling.aspx - *or* -
https://www.threatsandcountermeasures.com/wiki/default.aspx/Original.ThreatsAndCountermeasures.ThreatModeling?diff=y

Threat Modeling
https://www.threatsandcountermeasures.com/wiki/default.aspx/Original.ThreatsAndCountermeasures.ThreatModeling?diff=y

Gary McGraw, Cigital: Risk Analysis in Software Design (IEEE *Security & Privacy*, May/June 2004)
http://www.cigital.com/papers/download/bsi3-risk.pdf

Rajeev Gopalakrishna, Eugene H. Spafford, Jan Vitek, Purdue University Center for Education and Research in Information Assurance and Security (CERIAS): "Vulnerability Likelihood: A Probabilistic Approach to Software Assurance" (CERIAS TR 2005-06)
https://www.cerias.purdue.edu/tools_and_resources/bibtex_archive/view_entry.php?bibtex_id=2781

Rajeev Gopalakrishna, Eugene H. Spafford, Purdue University CERIAS: "A Trend Analysis of Vulnerabilities" (CERIAS TR 2005-05)
https://www.cerias.purdue.edu/tools_and_resources/bibtex_archive/view_entry.php?bibtex_id=2782

Charles H. Le Grand, CHL Global Associates: Software Security Assurance: A Framework for Software Vulnerability Management and Audit (October 2005)
http://www.ouncelabs.com/audit/

Software Risk and Regulatory Compliance: At-a-Glance Guides
http://www.ouncelabs.com/newsletter/current/software-security-compliance.htm

Common Criteria (CC) Evaluation Scheme
http://niap.nist.gov/cc-scheme/index.html

NSTISSI 1000, National IA Certification and Accreditation Process
http://www.cnss.gov/Assets/pdf/nstissi_1000.pdf

DoD 5200.40, DoD Information Technology Security Certification and Accreditation Process (DITSCAP, 30 December 1997)
http://iase.disa.mil/ditscap/ - *and* -
http://www.dtic.mil/whs/directives/corres/html/520040.htm

DoD Acquisition Policy Documents
http://www.dtic.mil/whs/directives/corres/html/50001.htm - *and* -
http://www.dtic.mil/whs/instructions/corres/html/50002.htm - *and* -
http://akss.dau.mil/dapc/index.html

NIST SP 800-64, Security Considerations in the Information System Development Life Cycle (revision dated June 2004)
http://csrc.nist.gov/publications/nistpubs/800-64/NIST-SP800-64.pdf

NIST SP 800-37, Guide for the Security Certification and Accreditation of Federal Information Systems (May 2004)
http://csrc.nist.gov/publications/nistpubs/800-37/SP800-37-final.pdf

Howard, Michael, and Steve Lipner: *The Security Development Lifecycle* (Microsoft Press, May 2006)

Steve Lipner, Michael Howard, Microsoft Corporation: The Trustworthy Computing Security Development Lifecycle
http://msdn.microsoft.com/security/sdl - *or* -
http://msdn.microsoft.com/security/default.aspx?pull=/library/en-us/dnsecure/html/sdl.asp

Michael Howard, Microsoft Corporation: How Do They Do It?: A Look Inside the Security Development Lifecycle at Microsoft (MSDN Magazine, November 2005)
http://msdn.microsoft.com/msdnmag/issues/05/11/SDL/default.aspx

Steve Lipner, Microsoft Corporation: "Practical Assurance: Evolution of a Security Development Lifecycle" (*Proceedings of the 20th Annual Computer Security Applications Conference*, 6-10 December 2004)
http://www.acsa-admin.org/2004/papers/Lipner.pdf

Secure Software: CLASP Information
http://www.securesoftware.com/process/

The Team Software Process and Security
http://www.sei.cmu.edu/tsp/tsp-security.html

Noopur Davis, SEI: Developing Secure Software (*DoD Software Tech News*, Vol. 8, No. 2, July 2005)
http://www.softwaretechnews.com/stn8-2/noopur.html

OMG Model Driven Architecture: "The Architecture of Choice for a Changing World"
http://www.omg.org/mda/

David Basin, ETH Zurich, Martin Buchheit, Bernhard Hollunder, and Torsten Lodderstedt, Interactive Objects Software GmbH, and Jürgen Doser, Universität Freiburg (Germany): ArcSecure—Model Driven Security
http://www.softwarefoerderung.de/projekte/vortrag_ARCSECURE.pdf

Manuel Koch, Free University of Berlin (Germany) and Francesco Parisi-Presicce, George Mason University: Formal Access Control Analysis in the Software Development Process (2002)
http://www.inf.fu-berlin.de/inst/ag-ss/papers/FMSE03.pdf

Gunnar Peterson, CTO, Arctec Group: Top Ten Information Security Considerations in Use Case Modeling (2005)
http://www.arctecgroup.net/briefings.htm

Thitima Srivatanakul, John A. Clark, and Fiona Polack, University of York (UK): Writing Effective Security Abuse Cases (Univ. of York Technical Report YCS-2004-375, 14 May 2004)
http://www.cs.york.ac.uk/ftpdir/reports/YCS-2004-375.pdf

Guttorm Sindre, Norwegian University of Science and Technology, and Andreas L. Opdahl, University of Bergen (Norway): Templates for Misuse Case Description
http://www.ifi.uib.no/conf/refsq2001/papers/p25.pdf

Jill Srivatanakul, John A. Clark, and Fiona Polack, University of York Computer Science Department: "Effective Security Requirements Analysis: HAZOP and Use Cases" (Information Security: 7th International Conference (ISC 2004) September 2004; Springer-Verlag Heidelberg LNCS 3225)
http://www.springerlink.com/index/T2LHC2NABXAF8V2W

Jill Srivatanakul, John A. Clark, and Fiona Polack, University of York Computer Science Department: "Writing Effective Security Abuse Cases" (Technical Report YCS-2004-375, May 2004)
http://www.cs.york.ac.uk/ftpdir/reports/YCS-2004-375.pdf

Torsten Lodderstedt, David Basin, and Jürgen Doser, Institute for Computer Science at the University of Freiburg (Germany): SecureUML: A UML-Based Modeling Language for Model-Driven Security (2002)
http://kisogawa.inf.ethz.ch/WebBIB/publications-softech/papers/2002/0_secuml_uml2002.pdf

Torsten Lodderstedt, Albert-Ludwigs-Universität, Freiburg im Breisgau (Germany): Model Driven Security from UML Models to Access Control Architectures (Doctoral Dissertation, 2003)
http://deposit.ddb.de/cgi-bin/dokserv?idn=971069778&dok_var=d1&dok_ext=pdf&filename=971069778.pdf

Foundstone, Inc. SecureUML tool
http://www.foundstone.com/index.htm?subnav=resources/navigation.htm&subcontent=/resources/proddesc/secureuml.htm

UMLsec Homepage
http://www4.in.tum.de/~umlsec/

Aspect-Oriented Software Development Interests
http://www.cs.wpi.edu/~gpollice/Interests/AOSD.html

Aspect-Oriented Modeling Research Group: Publications
http://www.cs.colostate.edu/~georg/aspectsPub/aspectsIndex.htm

Manifesto for Agile Software Development
http://agilemanifesto.org/

A. Tappenden, P. Beatty, and J. Miller, University of Alberta, and A. Geras and M. Smith, University of Calgary: "Agile Security Testing of Web-Based Systems via HTTPUnit" (*Proceedings of the AGILE2005 Conference*, July 2005)
http://www.agile2005.org/RP4.pdf

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

Jim Highsmith, Cutter Consortium: "What Is Agile Software Development?" (*CrossTalk: Journal of Defense Software Engineering*, October 2002)
http://www.stsc.hill.af.mil/crosstalk/2002/10/highsmith.html

Martin Fowler: The New Methodology
http://www.martinfowler.com/articles/newMethodology.html

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

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

William N. Robinson, Georgia State University: Monitoring Software Security Requirements Using Instrumented Code (2001)
http://www.sreis.org/old/2001/papers/sreis002.pdf

SafSec
http://www.safsec.com/

World Wide Web Virtual Library: Formal Methods
http://vl.fmnet.info/

Ivan Fléchais, M. Angela Sasse, and Stephen M. V. Hailes, University College London: "Bringing Security Home: A Process for Developing Secure and Usable Systems" (ACM 1-58113-000-0/00/000: *Proceedings of the New Security Paradigms Workshop 03,* August 2003)
http://www.softeng.ox.ac.uk/personal/Ivan.Flechais/downloads/nspw2003.pdf

Ivan Fléchais: *Designing Secure and Usable Systems* (Ph.D. thesis submitted to University College London Department of Computer Science, February 2005)
http://www.softeng.ox.ac.uk/personal/Ivan.Flechais/downloads/thesis.pdf

M. Angela Sasse and Ivan Fléchais: "Usable security: "What is it? How do we get it?", chapter in Lorrie Faith and Simson Garfinkel (editors): *Designing Secure Systems* (O'Reilly, 2005)

University College London: GetRealSecurity - AEGIS project page
http://getrealsecurity.cs.ucl.ac.uk/aegis.htm

Mohammad Reza Ayatollahzadeh Shirazi, Pooya Jaferian, Golnaz Elahi, Hamid Baghi, and Babak Sadeghian, Amirkabir University of Technology (Tehran Polytechnic): "RUPSec: An Extension on RUP for Developing Secure Systems" (*World Enformatika Society Transactions on Engineering, Computing, and Technology,* Vol. 4, February 2005)
http://www.enformatika.org/data/v4/v4-51.pdf

Visa USA: Cardholder Information Security Program: Payment Applications
http://usa.visa.com/business/accepting_visa/ops_risk_management/cisp_payment_applications.html

Glen Kunene, Senior Editor: "Security Training Falling through the Education Cracks" (*DevX*, 22 February 2005)
http://www.devx.com/security/Article/27323/

Matthew Bishop, UC-Davis and Deborah Frincke, U.S. Dept. of Energy Pacific Northwest National Lab: Teaching Robust Programming (IEEE *Security & Privacy*, March/April 2004)
http://nob.cs.ucdavis.edu/~bishop/papers/2004-robust/

Rose Shumba, James Walden, Stephanie Ludi, Carol Taylor, and Andy Ju An Wang: Teaching the Secure Development Lifecycle: Challenges and Experiences (*Proceedings of the 10th Colloquium for Information Systems Security Education*, 5-8 June 2006)
http://www.cisse.info/proceedings10/pdfs/papers/S04P02.pdf

Samuel P. Liles and Reza Kamali, Purdue University: An Information Assurance and Security Curriculum Implementation (*Issues in Informing Science and Information Technology,* Volume 3, 2006)
http://informingscience.org/proceedings/InSITE2006/IISITLile135.pdf

Oracle Software Security Assurance Process
http://www.oracle.com/solutions/security/docs/software-security-assurance-process.pdf

EC-Council Certified Secure Programmer and Certified Secure Application Developer certifications
http://www.eccouncil.org/ecsp/index.htm

APPENDIX A. DEFINITIONS, ABBREVIATIONS, AND ACRONYMS

This appendix provides a listing, with definitions, of terms used in this document, as well as amplifications of all abbreviations and acronyms used throughout this document. Attorney Paul Spiegal has a perceptive saying: "Everyone knows what something means until there is a problem." The "problem" Mr. Spiegal refers to arises particularly when the same term is used by people from different cultures, disciplines, or communities, each of which very likely has its own definition of the term. In many cases, there may be general agreement or similarity among definitions, but each community's definition will have its own implications, shading, and connotation.

In a discipline like software assurance, the "community" actually derives from practitioners within several other disciplines that have different, and sometimes even conflicting, objectives: the software engineering community, the systems engineering community, and the information assurance/cyber security community. Each community's practitioners bring their own contexts and understandings to the table, with the result that many software assurance practitioners use the same terms, but mean something slightly (or not so slightly) different by them. Predictably, communication conflicts arise.

Because no standard "software assurance glossary" yet exists, to avoid definitional conflicts when compiling this appendix, the authors reviewed a number of standard and/or widely accepted glossaries that originated in the software engineering, systems engineering, and information assurance communities. The definitions included in this appendix were selected or created according to the following methodology:

- When a single definition was commonly agreed upon across multiple glossaries, or when a term appeared in only one glossary, the authors used that definition.
- When different but harmonious definitions existed across multiple glossaries, the authors developed a definition that was consistent with all of the agreeing definitions.
- When different and conflicting definitions existed, the authors developed a definition that most clearly aligned with those existing definition(s) that were consistent with this document's intended usage of the terms.
- When no definitions existed in published glossaries, the authors created a definition that reflected this document's usage of the term.

Unlike definitions, amplifications of abbreviations and acronyms are not subject to debate, and thus those in this appendix can be taken as universally accepted.

A.1. Definitions

NOTE: Abbreviated designations for the source glossaries from which each definition is derived are noted in brackets following the definitions below; these abbreviations are amplified in section A.3.

ABUSE: intentional or reckless misuse (i.e., use in an unintended way), alteration, disruption, or destruction. *[CNSSI 4009]*

ANOMALY: any condition in the software's operation or documentation that departs from the expected. This expectation can come from the software's documentation (e.g., requirements specification, design documentation, user documentation), its source code or other development artifact(s), or from someone's perception or experience with the software (e.g., an observation of how the software operated previously). *[IEEE Std 1012-1986, IEEE Std 1044-1993]*

ASSURANCE: justifiable grounds for confidence that the security goals (integrity, availability, confidentiality, accountability, etc.) of the system implementation have been adequately satisfied. *[NIST SP 800-27-Rev.A]*

ATTACK: attempt to gain unauthorized access to the system's services or to compromise the system's dependability. In software terms, an attack is a malicious intentional fault, usually an external fault, that has the intent of exploiting a vulnerability in the targeted software or system. *[Avizienis* et al*, CNSSI 4009]*

AVAILABILITY: the degree to which the services of a system or component are operational and accessible when needed by their intended/authorized users. In the context of security, availability pertains to authorized services/actions only, and the need for availability generates the requirement that the system or component is able to resist or withstand attempts at unauthorized deletion or denial of service, regardless of whether those attempts are intentional or accidental. *[Avizienis* et al*, IEEE Std 610.12-1990, NIST SP 800-27-Rev.A]*

COMPONENT: one of the parts or elements that make up a system. A component may be hardware or software and may be divisible into smaller components. Note: With regards to software, the terms module, component, and unit are often used interchangeably, or defined to be sub-elements of one another. For example, a unit can be defined as either:

- a separately testable element specified in the design of a software component, or
- a software component that is not subdivided into other components, or
- a logically separable part of a computer program.

According to definition (1) a unit is a sub-element of a component; according to definitions (2) and (3), a unit is synonymous with a component. Furthermore, a "module" can be defined as either:

- a program unit that is discrete and identifiable with respect to compilation, combination with other units, and loading (thus, a module is a type of unit), or
- a logically separable part of a program (thus a module is the same as a unit).

To avoid confusion, for purposes of this document:

- Component is the only term used;
- "Unit" and "module" are considered synonymous with "component", not sub-elements of a component;
- The definition of component used herein is the synthesis of definitions (2) and (3) of "unit" (above).
- Software components are widely understood to have following characteristics:
    - Contractually specified interfaces
    - Explicit context dependencies
    - Ability to be deployed independently
    - Ability to be assembled/composed by someone other than their developer.

[IEEE Std 610.12-1990, Szyperski, UML]

COMPROMISE: a violation of the security policy of the system, or an incident in which any of the security properties of the system are violated its dependability is intentionally degraded. *[CNSSI 4009]*

CORRECTNESS: the property that ensures that software performs all of its intended functions as specified. Correctness can be seen as:

- the degree to which software is free from faults in its specification, design, and implementation; or

- the degree to which software, documentation and other development artifacts satisfy their specified requirements; or

- the degree to which software, documentation, and other development artifacts meet user needs and expectations, regardless of whether those needs and expectations are specified or not.

In simple terms, software that is correct is (1) error-free and (2) consistent with its specification. *[CNSSI 4009, DoD 500.59-M, IEEE Std 610.12-1990]*

COUNTERMEASURE: an action, device, procedure, technique, or other measure that reduces the vulnerability of a component or system. *[CNSSI 4009]*

CRITICAL SOFTWARE: software whose failure could have an impact on safety, or could cause large financial or social loss. Also referred to as high-consequence software. *[IEEE Std 1012-1986]*

DEFECT: a problem in the implemented software that renders it unfit for use. Defects are often caused by the software's non-conformance to its specification, though security defects may also be caused by the omission from the specification of adequate security requirements. *[NIST SP 500-209]*

DEFENSE IN DEPTH: strategy that combines people, technology, and operations capabilities to establish protective barriers that span different layers and dimensions of a system in its operational environment in order to isolate that system from potential sources of attack. *[CNSSI 4009]*

DENIAL OF SERVICE: an action or series of actions that (1) prevents access to a software system by its intended/authorized users; (2) causes the delay of its time-critical operations; or (3) prevents any part of the system from functioning. *[CNSSI 4009, NIST SP 800-27-Rev.A]*

DEPENDABILITY: the degree to which the software is operable and capable of performing functionality or of delivering a service that can justifiably be relied upon (i.e., trusted) to be correct. To achieve dependability, the software must be able to avoid service failures that are more frequent or severe, or longer in duration, than is acceptable to users. Dependability may be viewed according to different, but complementary, properties (or "instances of dependability") required for a system to be considered dependable:

- Availability
- Integrity
- Reliability
- Safety
- Maintainability
- Security.

Two other properties are closely related to dependability:

1. Survivability
2. Trustworthiness.

In software, availability and reliability emphasize the avoidance of failures, safety emphasizes the avoidance of a specific class of failures (catastrophic failures), and security emphasizes the prevention of exploitable development faults (intentional and unintentional) and malicious external faults. *[Avizienis* et al*, IFIP WG 10.4, TM 5-698-2]*

EXECUTION ENVIRONMENT: the combined aggregate of hardware, software, and networking entities in the software's operational environment that directly affects the execution of the software. *[CNSSI 4009]*

ERROR: (1) a deviation of one of the software's states from correct to incorrect, or the discrepancy between the condition or value actually computed, observed, or measured by the software, and the true, specified, or theoretically correct value or condition—these are errors on the part of the software itself, during its operation; (2) A human action that results in software containing a development fault. Examples include omission or misinterpretation of user requirements in a software specification, or incorrect translation or omission of a requirement in the design specification—these are errors on the part of the developer, and to avoid confusion, the term "mistake" is preferred for this type of developer error. An error is that part of the system state which is liable to lead to subsequent failure: an error affecting the service is an indication that a failure occurs or has occurred. An error escalates into a failure when it propagates beyond the ability of the software to continue operating through (or in spite) of that error. An error is sometimes referred to as a "bug". Just as "mistake" as used to indicate an error committed by a human, an "error" can be used to indicate a mistake committed by executing software. *[IEEE Std 610.12-1990, NASA-GB-8719.13, NIST SP 500-209]*

EVENT: an occurrence or incident that may affect the performance of a component or system. [CNSSI 4009] Exception: an event that causes the suspension of normal program execution. Types of exceptions include:

- addressing exceptions,
- data exceptions,
- operation exceptions,
- overflow exceptions (e.g., buffer, stack, and heap overflows),
- protection exceptions,
- underflow exceptions.

[IEEE Std 610.12-1990]

FAILURE: (1) the inability of a system or component to perform, or the non-performance by the system or component, of an intended function or service; or (2) a deviation of a function or service from its specified, expected performance, resulting in its incorrect performance. Failures fall into two general categories:

- Value failures: the functionality or service no fulfills its specified/expected purpose, i.e., it no longer delivers its specified/expected value;
- Timing failures: the timing of the functionality or service no longer falls within its specified/expected temporal constraints.

A failure results when one or more errors (caused by active faults) have propagated to the extent that the software can no longer perform correctly. "Hard" failures completely shut down the system. "Soft" failures permit the system to continue operating, but with only partial capability. Soft failures are usually achieved through the selective termination of non-essential processes when a failure is determined to be imminent; in this, soft errors contrast with errors which permit the software to continue operating with full capability. A soft failure generally manifests as an incorrect computational result or unexpected software behavior that can be perceived by the user. A hard fault generally manifests as a "crash" of the software program. Some failures may be transient, while others may be intermittent. For example, an error in a calculation may result in the return of an incorrect result or value—which represents a failure in the calculation process.

However, if the same calculation process subsequently returns only correct results/values, the failure was transient. If, however, the calculation process sometimes returns incorrect results/values and at other times returns correct results/values, the failure is intermittent. If all users have the same perception of the failure, it is

considered consistent. If different users have different perceptions of the same failure, it is considered inconsistent or Byzantine; because they are unpredictable Byzantine faults cannot be defined in a fault model of the system. Note that when defined from the user's viewpoint, an event that is considered a failure by one user may be considered merely a tolerable nuisance to another user; this is particularly true when the same software is used in different operational contexts—in critical operational contexts users will have a lower threshold of tolerance for failure than they do in non-critical operational contexts (which is, of course, why some users are willing to put up with unreliability in software that exhibits other desirable properties, while others insist on switching to more reliable software even though it may not exhibit those other desirable properties to the same extent if at all). *[CNSSI 4009, IEEE Std 610.12-1990, IFIP WG 10.4, NASA Glossary, NIST SP 500-209]*

FAULT: the adjudged or hypothesized cause of an error. A fault is considered active when it causes an error or failure; otherwise it is considered dormant. Some dormant faults never become active. In common usage, "bug" or "error" are used to express the same meaning. Software faults include incorrect steps, processes, and data definitions in computer programs. Faults fall into three basic categories:

1. *Development faults:* introduced into the software during some phase of its development; developmental faults include incorrect steps, processes, and data definitions that cause the software to perform in an unintended or unanticipated manner;

2. *Physical faults:* originate from defects in the hardware on which the software runs (hardware faults include such defects as short circuits or broken wires);

3. *External faults:* originate in the interactions between the software and external entities (users, other software).

In most cases a fault can be identified and removed, but in some cases it remains a hypothesis that cannot be adequately verified (e.g., timing faults in distributed systems).

Regardless of its category, a fault may be either human-made or natural. Human-made faults may be intentional or unintentional. Intentional human-made faults may or may not be malicious. Non-malicious intentional faults may simply result from poor decision-making. For example, a tradeoff between performance or usability on the one hand and dependability on the other may cause the software's developer to make a poor design decision that results in a development fault which would be considered intentional (having resulted from a conscious design decision) but non-malicious.

While malicious intentional faults are clearly a security concern, a fault need not be malicious or even intentional to have security implications. An accidental fault that results in errors or failures that leave the software vulnerable are also a security concern. Security faults often differ from other faults. Most non-security faults originate in specification violations: the software does not do something it is supposed to do. Security faults, by contrast, often manifest as additional behavior, i.e., the software does something it is not intended to do. *[IEEE Std 610.12-1990, NASA-GB-8719.13, NASA Glossary, NIST SP 500-209]*

FLAW: a mistake of commission, omission, or oversight in the specification, architecture, or design of software that also results in a defective implementation. Security flaws in software designs often result in weaknesses or vulnerabilities in the software implemented from those designs. Common security flaws include:

- Memory/buffer allocation flaws (e.g., the cause being a mistake of commission: failure to validate input);
- Privilege authorization flaws, which can result in misplaced trust in external entities;
- Timing flaws, which can lead to race conditions.

[CNSSI 4009, Landwehr et al*]*

FORMAL DEVELOPMENT: software development strategy that formally proves the system's design specifications. *[CNSSI 4009]*

FORMAL METHOD: a mathematical argument that verifies that the system satisfies a mathematically-described design specification or security policy. *[CNSSI 4009]*

FORMAL PROOF: the complete and convincing mathematical argument that presents the full logical justification for each proof step and for the truth of the theorem or set of theorems to be proved. *[CNSSI 4009]*

FORMAL VERIFICATION: the process of using formal proofs to demonstrate the consistency between the formal requirements specification or formal security policy of a system and its formal design specification (design verification) or between its formal design specification and its high-level implementation (implementation verification). *[CNSSI 4009]*

INTEGRITY: the quality of a system or component that reflects its logical correctness and reliability, completeness, and consistency. In security terms, integrity generates the requirement for the system or component to be protected against either intentional or accidental attempts to (1) alter, modify, or destroy it in an improper or unauthorized manner, or (2) prevent it from performing its intended function(s) in an unimpaired manner, free from improper or unauthorized manipulation. *[CNSSI 4009, NIST SP 800-27-Rev.A]*

LEAST PRIVILEGE: principle requiring that each subject be granted the most restrictive set of privileges needed for the performance of that subject's authorized tasks. Application of this principle limits the damage that can result from accident, error, or unauthorized use of a component or system. *[CNSSI 4009]*

MALICIOUS CODE: software or firmware intended to perform an unauthorized process that will have adverse impact on the dependability of a component or system. *[CNSSI 4009]*

MISTAKE: An error committed by a person as the result of a bad or incorrect decision or judgment by that person. Contrast with "error", which is used in this document to indicate the result of a "mistake" committed by software (i.e., as the result of an incorrect calculation or manipulation).

QUALITY: the degree to which a component, system or process meets its specified requirements and/or stated or implied user, customer, or stakeholder needs and expectations. *[IEEE Std 610.12-1990, ISO/IEC 9126]*

PROBLEM: often used interchangeably with anomaly, although problem has a more negative connotation, and implies that the anomaly is itself or results from a flaw, defect, fault, error, or failure. *[NIST SP 500-209]*

RELIABILITY: The probability of failure-free (or otherwise satisfactory) software operation for a specified/expected period/interval of time, or for a specified/expected number of operations, in a specified/expected environment under specified/expected operating conditions. *[ANSI STD-729-1991, IEEE Std 610.12-1990, ISO/IEC 9126, NASA-GB-A201, NASA-GB-8719.13, NIST SP 500-209, TM 5-698-2]*

RISK: the possibility or likelihood that a particular threat will adversely impact a system by exploiting a particular vulnerability. *[CNSSI 4009]*

ROBUSTNESS: the degree to which a component or system can function correctly in the presence of invalid inputs or stressful environmental conditions, including inputs or conditions that are intentionally and maliciously created *[IEEE Std 610.12-1990]*

SAFETY: Persistence of dependability in the face of accidents or mishaps, i.e., unplanned events that result in death, injury, illness, damage to or loss of property, or environmental harm. *[IEEE Std 1228-1994, Rushby]*

SANDBOXING: a method of isolating application-level components into distinct execution domains, the separation of which is enforced by software. When run in a "sandbox", all of the "sandboxed" program's code and data accesses are confined to memory segments within that "sandbox". In this way, "sandboxes" provide a greater level of isolation between executing processes than can be achieved when processes run in the same virtual address space. The most frequent use of sandboxing to isolate the execution of untrusted programs (e.g., mobile

code, programs written in potentially unsafe languages such as C) so that each program is unable to directly access the same memory and disk segments used by other programs, including the application's trusted programs. Virtual machines (VMs) are sometimes used to implement sandboxing, with each VM providing an isolated execution domain. *[NIST SP 800-19]*

SECURE STATE: the condition in which no subject can access another entity in an unauthorized manner. *[CNSSI 4009]*

SECURITY POLICY: a succinct statement of the strategy for protecting objects and subjects that make up the system. The system's security policy describes precisely which actions the entities in the system are allowed to perform and which ones are prohibited. *[Anderson, NIST SP 800-27-Rev.A, Tanenbaum* et al*]*

SECURITY: Protection against intentional subversion or forced failure. Preservation of the three subordinate properties that make up "security", i.e., availability, integrity, and confidentiality. In terms of dependability, security preserves dependability (including all of its subordinate properties) in the face of threats to that dependability. Security manifests as the ability of the system to protect itself from external faults that may be accidental or deliberate (i.e., attacks). According to Bruce Schneier in *Beyond Fear,* "Security is about preventing adverse consequences from the intentional and unwarranted actions of others." *[ISO/IEC 9126, Sommerville]*

SERVICE: A set of one or more functions, tasks, or activities performed to achieve one or more objectives to benefit a user (human or process).

SOFTWARE ASSURANCE: *(This term has multiple valid definitions. Please refer to Section 2.1.)*

SOFTWARE-INTENSIVE SYSTEM: a system in which the majority of components are implemented in/by software, and in which the functional objectives of the system are achieved primarily by its software components. *[IEEE Std 1471-2000]*

STATE: (1) a condition or mode of existence that a system or component may be in, for example the input state of a given channel; (2) the values assumed at a given instant by the variables that define the characteristics of a component or system. *[IEEE Std 610.12-1990]*

SYSTEM: a collection of components organized to accomplish a specific function or set of functions. *[IEEE Std 610.12-1990]*

THREAT: any entity, circumstance, or event with the potential to harm the software system or component through its unauthorized access, destruction, modification, and/or denial of service. *[CNSSI 4009, NIST SP 800-27-Rev.A]*

TRUSTWORTHINESS: logical basis for assurance (i.e., justifiable confidence) that the system will perform correctly, which includes predictably behaving in conformance with all of its required critical properties, such as security, reliability, safety, survivability, etc., in the face of wide ranges of threats and accidents, and will contain no exploitable vulnerabilities either of malicious or unintentional origin. Software that contains exploitable faults or malicious logic cannot justifiably be trusted to "perform correctly" or to "predictably satisfy all of its critical requirements" because its compromisable nature and the presence of unspecified malicious logic would make prediction of its correct behavior impossible. *[DHS, Laprie* et al, *Neumann]*

USER: any person or process authorized to access an operational system. *[CNSSI 4009]*

VERIFICATION AND VALIDATION (V&V): the process of confirming, by examination and provision of objective evidence, that:

- each step in the process of building or modifying the software yields the right products (verification). Verification asks and answers the question "Was the software built right?" (i.e., correctness);

- the software being developed or modified will satisfy its particular requirements (functional and non-functional) for its specific intended use (validation). Validation asks and answers the question "Was the right software built?" (i.e., suitability).

In practical terms, the differences between verification and validation are unimportant except to the theorist. Practitioners use the term V&V to refer to all of the activities that are aimed at making sure the software will function as required. V&V is intended to be a systematic and technical evaluation of software and associated products of the development and maintenance processes. Reviews and tests are done at the end of each phase of the development process to ensure software requirements are complete and testable and that design, code, documentation, and data satisfy those requirements. Independent V&V is a process whereby the products of the software development life cycle are reviewed, verified, and validated by an entity that is neither the developer nor the acquirer of the software, which is technically, managerially, and financially independent of the developer and acquirer, and which has no stake in the success or failure of the software. *[IEEE Std. 610.12-1990, ISO 9000, ISO/IEC 12207, NASA-GB-A201, NASA Glossary, NIST SP 500-234]*

VULNERABILITY: a development fault or other weakness in deployed software or exploited with malicious intent by a threat with the objective of subverting or incapacitating (violation of integrity, achieved by modification, corruption, destruction, or security policy violation)—often to gain access to the information it handles—or to force the software to fail (denial of service). Vulnerabilities can originate from flaws on the software's design, defects in its implementation, or problems in its operation. *[Avizienis* et al*, CIAO, CNSSI 4009, NIST SP 800-27-Rev.A, Sommerville, Veríssimo* et al*]*

WEAKNESS: a flaw, defect, or anomaly in software that has the potential of being exploited as a vulnerability when the software is operational. A weakness may originate from a flaw in the software's security requirements or design, a defect in its implementation, or an inadequacy in its operational and security procedures and controls. The distinction between "weakness" and "vulnerability" originated with the MITRE Corporation Common Weaknesses and Exposures (CWE) project (http://cve.mitre.org/cwe/about/index.html). *[Avizienis* et al*, CIAO, CNSSI 4009, NIST SP 800-27-Rev.A, Sommerville, Veríssimo* et al*]*

A.2. Abbreviations and Acronyms

The following list provides the amplifications of all abbreviations and acronyms used in this document.

AA.....Application Area

ACE .....Application Consulting Engineering

ACM .....Association for Computing Machinery

ACSM/SAR .....Adaptive Countermeasure Selection Mechanism/Security Adequacy Review

AEGIS.....Appropriate and Effective Guidance for Information Security

AJAX .....Asynchronous JavaScript and XML

ANSI .....American National Standards Institute

AOD.....Aspect Oriented Design

AOM .....Aspect Oriented Modeling

AOP .....Aspect Oriented Programming

AOSD.....Aspect Oriented Software Development

API .....Application Programming Interface

ARM .....Automated Requirements Measurement

ASAP .....Application Security Assurance Program

ASD ..... Assistant Secretary of Defense; Adaptive Software Development

ASP ..... Active Server Page; Agile Software Process

ASSET ..... Automated Security Self-Evaluation Tool

AT&L ..... Acquisition, Technology And Logistics

AusCERT ..... Australian Computer Emergency Response Team

AVDL ..... Application Vulnerability Definition Language

BIA ..... Business Impact Analysis

BPCP ..... BITS Product Certification Program

BS ..... British Standard

C&A ..... Certification And Accreditation

C3I ..... Command, Control, Communications and Intelligence

C4II ..... Command, Control, Communications, Computers, Intelligence Integration

C4ISR ..... Command, Control, Communications, Computer, Intelligence, Surveillance, and Reconnaissance

CAML ..... Categorical Abstract Machine Language

CBK ..... Common Body of Knowledge

CC ..... Common Criteria

CCTA ..... Central Computer and Telecommunications Agency

CEO ..... Chief Executive Officer

CERIAS ..... Center for Education and Research in Information Assurance and Security

CERT/CC ..... CERT Coordination Center *(CERT originally amplified to Computer Emergency Response Team, but is now a trademark.)*

CGI ..... Common Gateway Interface

CHACS ..... Center for High Assurance Computer Systems

CHL ..... Charles H. LeGrand

CIAO ..... Critical Infrastructure Assurance Office

CISP ..... Cardholder Information Security Program

CLASP ..... Comprehensive, Lightweight Application Security Process

CLR ..... Common Language Runtime

CM ..... Configuration Management

CMM ..... Capability Maturity Model

CMMI ..... CMM-Integrated

CMU ..... Carnegie Mellon University

CNSS ..... Committee on National Security Systems

CoBIT ..... Control OBjectives for Information and related Technology

COCOMO ..... COnstructive COst MOdel

COMBINE ..... COMponent-Based INteroperable Enterprise
CONOPS ..... CONcept of OPerationS
CORAS ..... Consultative Objective Risk Analysis System
CORBA ..... Common Object Request Broker Architecture
Corp. ..... Corporation
COSECMO ..... COnstructive SEcurity Cost MOdel
COTS ..... Commercial-Off-The-Shelf
CRAMM ..... CCTA Risk Analysis and Management Method
CSAD ..... Certified Secure Application Developer
CSE ..... Center for Software Engineering
CTMM ..... Calculative Threat Modeling Methodology
CVE ..... Common Vulnerabilities and Exposures
CWE ..... Common Weaknesses and Exposures
D.C. ..... District of Columbia
DbC ..... Design by Contract
DCID ..... Director of Central Intelligence Directive
DHS ..... Department of Homeland Security
DIACAP ..... DoD Information Assurance Certification and Accreditation Process
DITSCAP ..... DoD Information Technology Security Certification and Accreditation Process
DLL ..... Dynamic Link Library
DoD ..... Department of Defense
DoS ..... Denial of Service
DREAD ..... Damage potential, Reproducibility, Exploitability, Affected users, Discoverability
DSB ..... Defense Science Board
DSDM ..... Dynamic System Development Method
DSS ..... Defense Security Service
e.g. ..... exempla grata, *Latin term which means "provided as an example"*
EAL ..... Evaluation Assurance Level
EC-Council ..... Council of Electronic Commerce Consultants
ECSP ..... EC-Council Certified Secure Programmer
EE ..... Enterprise Edition
EU ..... European Union
FAA ..... Federal Aviation Administration
FDD ..... Feature-Driven Development
FIPS ..... Federal Information Processing Standard

FIRM ..... Fundamental Information Risk Management
FISMA ..... Federal Information Security Management Act
FITSAF ..... Federal Information Technology Security Assessment Framework
FMF ..... Flexible Modeling Framework
FSR ..... Final Security Review
FTP ..... File Transfer Protocol
GAISP ..... Generally Accepted Information Security Principles
GAO ..... General Accountability Office
GOTS ..... Government-Off-The-Shelf
HTML ..... HyperText Markup Language
HTTP ..... HyperText Transmission Protocol
HTTPS ..... HyperText Transmission Protocol-Secure
i.e. ..... id est, *Latin term which means "that is"*
I/O ..... Input/Output
IA ..... Information Assurance
IAVA ..... Information Assurance Vulnerability Alert
IBM ..... International Business Machines
iCMM ..... Integrated CMM
ICSA ..... International Computer Security Association
ICSE ..... International Conference on Software Engineering
ID ..... IDentifier
IDE ..... Integrated Development Environment
IEC ..... International Electrotechnical Commission
IEEE ..... Institute of Electrical and Electronics Engineers
IESE ..... Institute Experimentelles Software Engineering
Inc. ..... Incorporated
INFOSEC ..... INFOrmation SECurity
IOCCC ..... International Obfuscated C Code Contest
IRAM ..... Information Risk Analysis Methodologies
ISF ..... Information Security Foundation
ISO ..... International Standards Organization
ISS ..... Internet Security Systems
ISSA ..... Information Systems Security Association
ISSP ..... Information Systems Security Program
IST ..... Information Society Technologies
IT ..... Information Technology

ITIL ..... Information Technology Infrastructure Library

IV&V ..... Independent Verification and Validation

JAD ..... Joint Application Design (*or* Development)

Jan. ..... January

JMU ..... James Madison University

JPL ..... Jet Propulsion Laboratory

JSA-TP4 ..... Joint Systems and Analysis Group Technical Panel 4

JSP ..... Java Server Page

JTC ..... Joint Technical Committee

JVM ..... Java Virtual Machine

LAAS-CNRS ..... Laboratoire d'Analyse et d'Architecture des Systèmes-Centre National de la Recherche Scientifique

LaQuSo ..... Laboratory for Quality Software

LD ..... Lean Development

LDAP ..... Lightweight Directory Access Protocol

M.S. ..... Master of Science

MARCORSYSCOM ..... Marine Corps Systems Command

MDA ..... Model Driven Architecture

MDR ..... MetaData Registry

MITRE ..... Massachusetts Institute of Technology Research and Engineering

ML ..... MetaLanguage

MSDN ..... MicroSoft Developer Network

NASA ..... National Aeronautics and Space Administration

NCSC ..... National Computer Security Center

NCST ..... National Cyber Security Taskforce

NIACAP ..... National Information Assurance Certification and Accreditation Process

NII ..... Networks and Information Integration

NIST ..... National Institute of Standards and Technology

No. ..... Number

NRL ..... Naval Research Laboratory

NSA ..... National Security Agency

NSC ..... National Security Council

OCL ..... Object Constraint Language

OCTAVE ..... Operationally Critical Threat, Asset, and Vulnerability Evaluation

OCTAVE-S ..... OCTAVE Secure

OMB ..... Office of Management and Budget

OMG ..... Object Management Group

OSD ..... Office of the Secretary of Defense

OVAL ..... Open Vulnerability and Assessment Language

OWASP ..... Open Web Application Security Project

PA ..... Practice Area

PABP ..... Payment Application Best Practices

PBT ..... Property-Based Tester; Property-Based Testing

PCI ..... Payment Card Industry

Perl ..... Practical extraction and report language

PHP ..... *a recursive acronym that amplifies to* PHP Hypertext Processor*; PHP originally stood for Personal Home Page*

PKI ..... Public Key Infrastructure

PSM ..... Platform Specific Model

PTA ..... Practical Threat Analysis

QDSC ..... Qualified Data Security Company

RAII ..... Resource Acquisition Is Initialization

RAISE ..... Rigorous Approach to Industrial Software Engineering

RBAC ..... Role Based Access Control

RCM ..... Reliability-Centered Maintenance

RDBMS ..... Relational Database Management System

RDT&E ..... Research, Development, Test and Evaluation

REVEAL ..... Requirements Engineering VErification and vALidation

RM-ODP ..... Reference Model for Open Distributed Processing

RSSR ..... Reducing Software Security Risk

RUP ..... Rational Unified Process

RUPSec ..... Rational Unified Process Secure

S&S ..... Safety and Security

S3 ..... Secure Software Systems

SAL ..... Standard Annotation Language

SAL ..... Standard Annotation Language

SAMATE ..... Software Assurance Metrics and Tool Evaluation

SARA ..... Security Auditor's Research Assistant

SC ..... SubCommittee

SCADA ..... Supervisory Control And Data Acquisition

SCR ..... Software Cost Reduction

SDL ..... Security Development Lifecycle

SDLC ..... Software Development Life Cycle; System Development Life Cycle

SECURIS ..... *(model-driven development and analysis of)* SECURe Information Systems

SEI ..... Software Engineering Institute

SELinux ..... Security-Enhanced Linux

Sept. ..... September

SESS ..... Software Engineering for Secure Systems

SiSo ..... Sichere Software (Secure Software)

SLAC ..... Stanford Linear Accelerator Center

SLAM ..... Software-specification, Language, Analysis, and Model-checking

SOA ..... Service Oriented Architecture

SOAP ..... Simple Object Access Protocol

SoS ..... Security of Systems

SOUP ..... Software Of Unknown Pedigree

SP ..... Special Publication

SPRINT ..... Simplified Process for Risk IdeNTification

SQL ..... Structured Query Language

SQM ..... Software Quality Management

SQUARE ..... System QUAlity Requirements Engineering

SRI ..... Science Research International

SSAI ..... Software Security Assessment Instrument

SSE-CMM ..... System Security Engineering CMM

SSL ..... Secure Socket Layer

ST&E ..... Security Test and Evaluation

STD ..... Standard

Std. ..... Standard

STRIDE ..... Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, Elevation of privilege

SwA ..... Software Assurance

TCP ..... Transmission Control Protocol

TDD ..... Test Driven Development

TFM ..... Trusted Facility Manual

TICSA ..... TruSecure International Computer Security Assocation

TLS ..... Trusted Layer Security

T-MAP ..... Threat Modeling based on Attacking Path Analysis

TOE ..... Target of Evaluation

TPM ..... Trusted Platform Module

TSP.....Team Software Process

U.S. .....United States

UC-Berkeley .....University of California at Berkeley

UC-Davis .....University of California at Davis

UK.....United Kingdom

UML.....Unified Modeling Language

UMLSec.....*a security-enhanced UML profile*

URI.....Uniform Resource Identifier

URL .....Uniform Resource Locator

USC.....University of Southern California

US-CERT.....United States Computer Emergency Response Team

USD .....UnderSecretary of Defense

USSTRATCOM.....United States STRATegic COMmand

Va.....Virginia

VM.....Virtual Machine

Vol. .....Volume

VPN .....Virtual Private Network

WASC.....Web Application Security Consortium

Wisdom.....Whitewater Interactive System Development with Object Models

WS .....Web Service

XHTML .....eXtensible HyperText Markup Language

XML.....eXtensible Markup Language

XP .....eXtreme Programming

A.3. References for this Appendix

[Anderson] Ross J. Anderson: *Security Engineering: A Guide to Building Dependable Distributed Systems* (Wiley Computer Publishing, 2001)

[ANSI STD-729-1991] American National Standards Institute (ANSI) and Institute of Electrical and Electronic Engineers (IEEE): Standard Glossary of Software Engineering Terminology (1991)

[Avizienis *et al*] Algirdas Avizienis, 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, January-March 2004)
http://techreports.isr.umd.edu/reports/2004/TR_2004-47.pdf

[CIAO] White House Critical Infrastructure Assurance Office: Critical Infrastructure Glossary of Terms and Acronyms
http://permanent.access.gpo.gov/Websites/www.ciao.gov/ciao_document_library/glossary/c.htm

[CNSSI 4009] Committee on National Security Systems (CNSS) Instruction No. 4009: *National Information Assurance (IA) Glossary* (Revised June 2006)
http://www.cnss.gov/Assets/pdf/cnssi_4009.pdf

[DHS] Joe Jarzombek, Director for Software Assurance, DHS National Cyber Security Division: "Software Assurance: A Strategic Initiative of the U.S. Department of Homeland Security to Promote Integrity, Security, and Reliability in Software—Considerations for Modernization in Advancing a National Strategy to Secure Cyberspace" (27 October 2005)
http://www.omg.org/news/meetings/workshops/ADM_2005_Proceedings_FINAL/K-2_Jarzombek.pdf - *or* -
http://www.incits.org/tc_home/CS1/2005docs/cs1050021.pdf

[DoD 500.59-M] Modeling and Simulation Glossary
https://www.dmso.mil/public/resources/glossary/ - *or* -
http://www.dtic.mil/whs/directives/corres/text/p500059m.txt

[IEEE Std 610.12-1990] IEEE *Standard Glossary of Software Engineering Terminology*
http://www.swen.uwaterloo.ca/~bpekilis/public/SoftwareEngGlossary.pdf

[IEEE Std 1012-1986] IEEE Standard for Software Verification and Validation Plans

[IEEE Std 1228-1994] IEEE Standard 1228-1994, Software Safety Plans

[IEEE Std 1044-1993] IEEE *Standard Classification for Software Anomalies*
http://www.swtest.jp/secure/std1044-1993.pdf

[IEEE Std 1471-2000] IEEE Standard 1471-2000, Recommended Practice for Architectural Description of Software-Intensive Systems

[IFIP WG 10.4] Jean-Claude Laprie, International Federation for Information Processing (IFIP) Working Group (WG) 10.4 - Dependable Computing and Fault Tolerance (editor): *Dependability: Basic Concepts and Terminology* (Springer Verlag, 1992)

[ISO 9000] Quality Management

[ISO/IEC 9126] Software Engineering - Product Quality

[ISO/IEC 12207] Information Technology - Software Life Cycle Processes

[Landwehr *et al*] Carl E. Landwehr, Alan R. Bull, John P. McDermott, and William S. Choi, Naval Research Laboratory Center for High Assurance Computing Systems: "A Taxonomy of Computer Program Security Flaws" (revision of version originally published in *ACM Computing Surveys*, Vol. 26, No. 3, September 1994)
http://chacs.nrl.navy.mil/publications/CHACS/1994/1994landwehr-acmcs.pdf

[Laprie *et al*] Algirdas Avizienis, Jean-Claude Laprie, and Brian Randell: Fundamental Concepts of Dependability (Research Report No. 01145, Laboratoire d'Analyse et d'Architecture des Systèmes-Centre National de la Recherche Scientifique [LAAS-CNRS], April 2001)
http://www.cs.ncl.ac.uk/research/pubs/trs/papers/739.pdf

[NASA-GB-8719.13] NASA *Software Safety Guidebook*
http://www.hq.nasa.gov/office/codeq/doctree/871913.pdf

[NASA-GB-A201] NASA *Software Assurance Guidebook*
http://satc.gsfc.nasa.gov/assure/agb.txt

[NASA Glossary] NASA Software Assurance Glossary
http://sw-assurance.gsfc.nasa.gov/help/glossary.php

[Neumann] Peter G. Neumann, SRI International:"Trustworthy Systems Revisited" ("Inside Risks #188", *Communications of the Association for Computing Machinery [ACM]*, Vol. 49, No. 2, February 2006)

[NIST SP 500-209] NIST Special Publication 500-209, "Software Error Analysis" (March 1993)
http://hissa.nist.gov/SWERROR/

[NIST SP 500-234] NIST Special Publication 500-234, "Reference Information for the Software Verification and Validation Process" (29 March 1996)
http://hissa.nist.gov/HHRFdata/Artifacts/ITLdoc/234/val-proc.html

[NIST SP 800-19] NIST Special Publication 800-19, "Mobile Agent Security"
http://csrc.nist.gov/publications/nistpubs/800-19/sp800-19.pdf

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

[Rushby] John Rushby, SRI International Computer Security Laboratory: Critical System Properties: Survey and Taxonomy (Technical Report CSL-93-01, May 1993 revised February 1994; original version published in *Reliability Engineering and System Safety*, Vol. 43, No. 2, 1994)
http://www.csl.sri.com/~rushby/papers/csl-93-1.pdf - *or* -
http://www.csl.sri.com/papers/csl-93-1/

[Sommerville] Ian Sommerville: *Software Engineering* (Addison Wesley, 7th edition, 2004)
http://www.comp.lancs.ac.uk/computing/resources/IanS/SE7/index.html

[Szyperski] Clemens Szyperski: *Component Software: Beyond Object-Oriented Programming* (New York: ACM Press and Addison-Wesley, 1998)

[Tanenbaum *et al*] Andrew S. Tanenbaum and Maarten Van Steen: *Distributed Systems: Principles and Paradigms* (Prentice-Hall, 2001)
http://www.prenhall.com/divisions/esm/app/author_tanenbaum/custom/dist_sys_1e/

[TM 5-698-2] Department of the Army Technical Manual 5-698-2, "Reliability-Centered Maintenance (RCM) for Command, Control, Communications, Computer, Intelligence, Surveillance, and Reconnaissance (C4ISR) Facilities" (3 May 2003)
http://www.usace.army.mil/usace-docs/armytm/tm5-698-2/toc.htm

[UML] Object Management Group: Unified Modeling Language: Superstructure, Section 8, Components (Version 2.0, August 2005)

[Veríssimo *et al*] Paulo Esteves Veríssimo, Nuno Ferreira Neves, and Miguel Pupo Correia: "Intrusion-Tolerant Architectures: Concepts and Design" (*Architecting Dependable Systems: Springer Verlag Lecture Notes in Computer Science*, Vol. 2677, 2003)
http://www.di.fc.ul.pt/sobre/documentos/tech-reports/03-5.pdf

APPENDIX B. FOR FURTHER READING

This appendix provides a listing of resources to information for further reading on software security topics. These resources are organized according to their subject matter. URLs are provided for those references that are available on the World Wide Web or on a generally accessible private network. Offline resources, such as books, are listed without URLs, unless there is a Web page containing useful information about or an online excerpt from the book.

> *NOTE: With a very few exceptions, resources already listed in the References listed at the ends of chapters or other appendices of this document are not repeated here.*

B.1. Books

Anderson, Ross J.: *Security Engineering—A Guide to Building Dependable Distributed Systems* (John Wiley & Sons, 2001)
http://www.cl.cam.ac.uk/~rja14/book.html

Berg, Clifford J.: *High Assurance Design: Architecting Secure and Reliable Enterprise Applications* (Addison Wesley, Nov 2005)

Cerven, Pavol: *Crackproof Your Software: Protect Your Software Against Crackers* (No Starch Press, First Edition, 2002)

Dowd, Mark, and John McDonald: *The Art of Software Security Assessment: Identifying and Avoiding Software Vulnerabilities* (Pearson Professional Education, October 2006)

Gary McGraw: *Software Security: Building Security In* (Addison-Wesley Professional, 2006)

Hoglund, Greg, and Gary McGraw: *Exploiting Software: How to Break Code* (Addison-Wesley Professional, 2004)
http://www.exploitingsoftware.com/

Howard, Michael, and David LeBlanc, Microsoft Corporation: *Writing Secure Code, Second Edition* (Microsoft Press, 2003)
http://www.microsoft.com/mspress/books/5957.asp

Howard, Michael, and Steve Lipner: *The Security Development Lifecycle* (Microsoft Press, May 2006)

Howard, Michael, John Viega, and David LeBlanc: *19 Deadly Sins of Software Security* (McGraw-Hill Osborne Media, 2005)
http://books.mcgraw-hill.com/getbook.php?isbn=0072260858&template=

Huseby, Sverre H.: *Innocent Code: A Security Wake-up Call for Web Programmers* (John Wiley & Sons, 2004)
http://innocentcode.thathost.com/

Jürjens, Jan: *Secure Systems Development in UML* (Springer, First Edition, 2004)

Kaspersky, Kris: *Hacker Disassembling Uncovered* (A-List Publishing, 2003)
http://www.cert.org/books/secure-coding/index.html

Okada , M., B. Pierce, A. Scedrov, H. Tokuda, and A. Yonezawa (editors): *Software Security—Theories and Systems* (Revised Papers from the Mext-NSF-JSPS International Symposium (ISSS 2002, Tokyo, Japan, 8-10 November 2002); Springer Series: Lecture Notes in Computer Science, Vol. 2609, 2003;)
http://www.springer.com/sgw/cda/frontpage/0,11855,4-164-22-2319046-0,00.html

Schumacher, Markus, Eduardo Fernandez-Buglioni, Duane Hybertson, Frank Buschmann, and Peter Sommerlad: *Security Patterns: Integrating Security and Systems Engineering* (John Wiley & Sons, March 2006)

Seacord, Robert: *Secure Coding in C and C++* (Addison Wesley Professional/SEI Series in Software Engineering, 2005)
http://www.cert.org/books/secure-coding/index.html

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

Van Wyk, Kenneth R., and Mark G. Graff: *Secure Coding: Principles and Practices* (O'Reilly & Associates, 2003)
http://www.securecoding.org/

Viega, John, and Gary McGraw: *Building Secure Software: How to Avoid Security Problems the Right Way* (Addison-Wesley Publishing Company, 2001)
http://www.buildingsecuresoftware.com/

Viega, John, and Matt Messier: *Secure Programming Cookbook for C and C++* (O'Reilly & Associates, 2003)
http://www.secureprogramming.com/

Whittaker , James, and Herbert Thompson: *How to Break Software Security* (Addison Wesley, 2003)

B.2. Publications

U.S. Air Force Software Technology Support Center: *CrossTalk: The Journal of Defense Software Engineering*—Software Security issue (October 2005)
http://www.stsc.hill.af.mil/crosstalk/2005/10/index.html

DoD Software Tech News—Secure Software Engineering issue (July 2005, Vol. 8, No. 2)
http://www.softwaretechnews.com/stn8-2/stn8-2.pdf

IsecT *NOTICEBORED*—Secure Software Development issue (Issue 30, November 2005)
http://www.noticebored.com/NB_newsletter_on_secure_devt.pdf

National Cyber Security Taskforce: *Processes to Produce Secure Software*, Vol. I
http://www.cyberpartnership.org/software%20development%20life cycleFULL.pdf

National Cyber Security Taskforce: *Processes to Produce Secure Software*, Vol. II
http://www.cyberpartnership.org/Software%20Pro.pdf

B.3. Web Sites Devoted to Software (or Application) Security

DHS-sponsored BuildSecurityIn Web portal (at US-CERT)
https://buildsecurityin.us-cert.gov/

Microsoft Security Developer Center
http://msdn.microsoft.com/security/

Apple Developer Connection: Secure Coding Guide
http://devworld.apple.com/documentation/Security/Conceptual/SecureCodingGuide/index.html

Michael Howard's (Microsoft) Web Log
http://blogs.msdn.com/michael_howard/archive/2005/07/11/437875.aspx

Secureprogramming.com: Recipes for Cryptography, Authentication, Networking, Input Validation & More
http://www.secureprogramming.com/

Palisade Application Security Intelligence
http://palisade.paladion.net/

B.4. Archives and Libraries

SANS InfoSec Reading Room: Securing Code
http://www.sans.org/rr/whitepapers/securecode/

SANS InfoSec Reading Room: Application and Database Security
http://www.sans.org/rr/whitepapers/application/

Matt Bishop, University of California at Davis: Papers
http://nob.cs.ucdavis.edu/~bishop/papers/index.html

Professor Gerhard Popp, Universität Innsbruck Institut für Informatik: Publications
http://www4.informatik.tu-muenchen.de/~popp/publications/

Open Seminar on Software Engineering: Security and Privacy
http://openseminar.org/se/modules/15/index/screen.do

Security Innovation: Whitepapers, Articles, and Research Reports
http://www.securityinnovation.com/resources/whitepapers.shtml

Ounce Labs: Library
http://www.ouncelabs.com/library.asp

Advosys Consulting: Writing Secure Web Applications
http://advosys.ca/papers/Web-security.html

Carnegie Mellon University Software Engineering Institute Publications
http://www.sei.cmu.edu/publications/publications.html

B.5. Individual Reports, Whitepapers, etc.

A White Paper on Software Assurance (Revised Document—Nov. 28, 2005; *sponsored by the DoD Software Assurance Tiger Team initiative*)
http://adm.omg.org/SoftwareAssurance.pdf

Jayaram K R and Aditya P Mathur, Purdue University: Software Engineering for Secure Software—State of the Art: A Survey (CERIAS TR 2005-67)
https://www.cerias.purdue.edu/tools_and_resources/bibtex_archive/view_entry.php?bibtex_id=2884

Premkumar Devanbu, UC-Davis, Stuart Stubblebine, CertCo, Inc.: Software Engineering for Security: A Roadmap
http://www.softwaresystems.org/future.html

Axel van Lamsweerde, Peter Van Roy, Jean-Jacques Quisquater: MILOS: Méthodes d'Ingénierie de Logiciels Sécurisés *(in French)*
http://www.info.ucl.ac.be/people/PVR/MILOSabstr.rtf

SAP AG Secure Software Development Guide
https://www.sdn.sap.com/irj/servlet/prt/portal/prtroot/docs/library/uuid/3cae2f4d-0701-0010-ccae-e8489f4c26ce

Shari Lawrence Pfleeger and Charles P. Pfleeger: Chapter entitled "Program Security" in *Security in Computing*, 3rd Edition (Prentice-Hall, 2003)
http://www.phptr.com/articles/article.asp?p=31782&seqNum=7&rl=1

NIST: SP 800-64, *Security Considerations in the Information System Development Life Cycle*
http://csrc.nist.gov/publications/nistpubs/800-64/NIST-SP800-64.pdf

William Yurcik: National Missile Defense: The Trustworthy Software Argument (*Computer Professionals for Social Responsibility Newsletter*, Vol. 19, No. 2, Spring 2001)
http://archive.cpsr.net/publications/newsletters/issues/2001/Spring/yurcik.html

Why is some software more secure than others?
http://irccrew.org/~cras/security/securesoftware.html

B.6. Organizations Devoted to Software (or Application) Security

DHS National Cyber Security Division US-CERT
http://www.US-CERT.gov/

National Cyber Security Partnership (NCSP): Software Lifecycle Task Force
http://www.cyberpartnership.org/init-soft.html

Object Management Group (OMG) Software Assurance Special Interest Group (SwA SIG)
http://swa.omg.org/

Application Security Industry Consortium
http://www.appsic.org/

Open Web Application Security Project
http://www.owasp.org/index.jsp

Secure Software Forum
http://www.securesoftwareforum.com/

Web Application Security Consortium (WASC)
http://www.webappsec.org/

B.7. Recurring Events Devoted to Software (or Application) Security

DHS/DoD co-sponsored Software Assurance Forum *(semiannual)*
https://buildsecurityin.us-cert.gov/

Software Security Summit
http://www.s-3con.com/

Software Engineering for Secure Systems (SESS) *(in conjunction with the International Conference on Software Engineering (ICSE)*
http://homes.dico.unimi.it/~monga/sess06.html

OWASP AppSec Conference *(semiannual, once in U.S., once in Europe)*
http://www.owasp.org/index.php/Category:OWASP_AppSec_Conference

Object Management Group: Software Assurance Workshop
http://www.omg.org/news/meetings/SWA2007/call.htm

IEEE International Symposium on Secure Software Engineering (ISSSE)
http://www.jmu.edu/iiia/issse/

IEEE Workshop on Secure Software Engineering Education and Training
http://www.jmu.edu/iiia/wsseet/

B.8. Software Assurance and High-Confidence Software Resources

NASA Software Assurance Technology Center (SATC)
http://satc.gsfc.nasa.gov/

National Coordination Office for Networking and Information Technology Research and Development: High Confidence Software and Systems Coordinating Group (HCSS CG)
http://www.nitrd.gov/subcommittee/hcss.html

Federal Aviation Administration (FAA) Office of the Assistant Administrator for Information Services & Chief Information Officer (AIO): Documents
http://www.faa.gov/aio/common/documents.htm

U.S. Naval Research Laboratory Center for High Assurance Computer Systems
http://chacs.nrl.navy.mil/

European Workshop on Industrial Computer Systems (EWICS) Technical Committee 7 on Reliability, Safety, and Security: System Security Subgroup
http://www.ewics.org/docs/system-security-subgroup

APPENDIX C. RECONCILING AGILE METHODS WITH A SECURE DEVELOPMENT LIFE CYCLE

Agile proponents have proposed and debated various ways in which agile methods and agile teams may be adaptable to address software security needs, particularly with regard to capturing non-functional security requirements, generating and executing appropriate test cases, and planning and staffing agile development projects, are discussed below.

C.1. Agile Methods and Security Requirements Engineering

It has been suggested by agile proponents that as long as software's requirements explicitly express the issues that make software "critical"—with that criticality driving specific requirements for security—agile methods should be able to be used successfully to develop that software. Allowing that agile methodologies have indeed matured to the extent that agile security requirements capture is now possible, it is only necessary for those requirements with the highest priority to be identified as such at the beginning of the project; from that point, any agile method will ensure that the highest priority requirements are assigned a comparably high level of testing with emphasis on the critical issues. Moreover, not only agile methods but *any* software development method will fail to adequately address requirements for security unless those requirements are explicitly stated.

Some agile proponents suggest that if security is clearly designated as important, an agile method may actually be more likely than a non-agile method to incorporate security into the software's specification and demonstrate it early in the life cycle. The need to explicitly define requirements for security is increasingly conventional wisdom that is not unique to agile methods. No matter what development methodology used (even if no defined method is used at all), addressing security needs as early in the development process as possible will always be more effective and less costly than waiting to "shoehorn" security into a system late in its life cycle.

The difficulty in capturing security requirements in agile development projects may originate, at least in part, from the reliance on "user stories" as the means for expressing those requirements. A user story specifies how a user interacts with the system to create value. To a great extent, user stories are similar to UML use cases, and hey share the same inadequacies in terms of capturing requirements related to security requirements that manifest as constraints on the behavior of the *system*, rather than constraints on the behavior of the user. Because of their purely user-centric view, use stories can be inadequate for capturing security requirements that must pervade the entire system, versus those that can be associated with specific usage scenarios.

Techniques for capturing constraint requirements are not well documented in the agile literature. To reconcile the need for such requirements with the reliance on user stories, Security requirements must be expressed as "technical requirements" in a format already accepted by the agile community; only if this can be achieved, will agile developers be able to address such security/constraint requirements using existing agile processes. One approach could be the creation of user stories that would be similar to the modeling of cross-cutting security aspects in AOM: each constraint user story would be evaluated as a component of every other user story, just as cross-cutting aspects are propagated to other areas of aspect oriented systems.

C.2. Security Test Cases and Test Driven Development

Agile proponents suggest that, as long as the necessary non-functional security requirements are captured, as described above, the associated security-relevant test cases will be defined and exercised during the software's iterative testing cycle, with all security reviews and tests needed to prove that the software's security objectives have been satisfied iterated throughout that testing cycle.

Agile security engineering would need to employ a highly iterative process in which security objectives were translated into automated security test cases before implementing the system began. These security test cases would elaborate the system security objectives and characterize the secure behaviors of the software, "secure" in

this instance being whatever was considered "secure enough" by the customer, consistent with standard risk management approaches.

It might be necessary to extend the concept of the user story to include "malicious user stories", comparable to the misuse or abuse cases proposed for extending UML. While misuse/abuse case modeling is not directly supported as part of agile requirements definition, in practical terms, such models could be defined using a tool such as FIT/FitNesse to produce executable acceptance tests. In the case of the abuse case tests, the expected result of each test would be the desired secure result (e.g., failure of unauthorized user to access system). This approach would ensure the system was exercised to demonstrate it behaved securely under all anticipated abuse scenarios. Customers (in the Agile Manifesto sense) who value security properties would have to write acceptance tests that measure those properties in the implemented software. Such tests will be more difficult to write, and are likely to may require testing tools and environments that are more complex than JUnit (a testing tool favored by many agile developers) to accomplish. Research is under way to improve mechanisms for testing non-functional security properties.

C.3. Role of Security Experts

Because agile methods emphasize collective responsibility and discourage specialization within a team, they do not adapt well to the idea of having a specific security expert on the team. For agile developers to achieve secure software the security awareness and skills of all team members would have to be raised. However, it is unlikely that the need for external security experts can be completely eliminated. At a minimum, such experts would be needed both to train, coach or mentor the other team members, and to perform the independent audits/certifications that are considered essential to achieving security assurance. Tappenden *et al* suggest that the role of the security engineer in agile security engineering would also be to advise and coach the customer about risks, to better enable the customer to prioritize those risks.

C.4. Planning for Security

The Agile Manifesto approach to planning does not allow for the amount of planning that has proven necessary in software development projects that need to achieve high levels of security. The problem again lies in the user story, which represents the unit of planning granularity in the Agile "planning game" of XP and other agile methods. Because user stories as currently conceived do not adequately capture security constraint requirements, planners of agile projects must find some other way to allow realistic scheduling for the implementation, testing, and assurance verification of the software's cross-cutting security aspects and constraints.

Customers of agile process-developed software should be made to recognize that security adds business value to that software. Within the framework of the "planning game", prioritization and planning of the necessary security properties and constraints will directly depend on the value a given security property or constraint is seen as adding to the software. The security risk manager responsible for coaching customers to recognize and state their security needs to influence those customers to include security among, if not at the top of the list of, the concerns that drive their prioritization of requirements.

C.5. References for this Appendix

Summary of the First eWorkshop on Agile Methods (April 8, 2002), Part 3: Applicability of Agile Methods for Critical, Reliable, and Safe Systems
http://fc-md.umd.edu/projects/Agile/Summary/Summary3.htm

Panel on Secure Agility and Agile Security (JavaPolis 2005)
http:// www.javapolis.com/confluence/display/JP04/Secure+agility+and+agile+security?decorator=printable

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

A. Tappenden, P. Beatty, and J. Miller, University of Alberta, and A. Geras and M. Smith, University of Calgary: "Agile Security Testing of Web-Based Systems via HTTPUnit" (*Proceedings of the AGILE2005 Conference,* July 2005)
http://www.agile2005.org/RP4.pdf

APPENDIX D. STANDARDS FOR SECURE LIFE CYCLE PROCESSES AND PROCESS IMPROVEMENT MODELS

The security-enhanced standards for life cycle processes described in this appendix are less specific and higher-level than the lifecycle processes described in Section 4. This is generally due to the nature of such standards, which are intended to reflect a broad international consensus and thus are intentionally relatively generic, broadly applicable, and devoid of detail or specificity about their implementation.

The security-enhanced process improvement models/capability maturity models (CMMs) described here are, like all CMMs, are most often adopted at the enterprise level, rather than at the level of the individual development team or project.

In both cases, these standard processes and models will be of most interest to the higher-level managers responsible for their selection and adoption across the enterprise.

D.1. IEEE Standard 1074-2006, Developing Software Project Life Cycle Processes

IEEE Standard 1074-2006, the revision of the IEEE Std. 1074-1997, *Developing Software Project Life Cycle Processes*, supports appropriate prioritization of security and building of appropriate levels of security controls into software and systems. The new standard accomplishes this by adding a small number of security activities to the SDLC defined in the earlier version of the standard.

IEEE Std. 1074-1997 formed the basis for developing ISO/IEC 12207.1 - *Standard for Information Technology—Software life cycle processes* and 12207.2 - *Software life cycle processes—Life cycle data.* The main objective of both IEEE Std. 1074-1997 (previously) and ISO/IEC 12207 (currently) is to define a quality-driven SDLC process; for that reason, neither standard contains specific security guidance, though ISO/IEC 12207 does suggest the need for security activities, or provides references to security-specific standards (an omission attributed to the fact that no such standards related to life cycle processes existed at the time the above standards were adopted).

The team that undertook revision of IEEE 1074 started their work by analyzing two ISO/IEC 17799:2000 *Code of Practice for Information Security Management* and ISO/IEC 15408 *Common Criteria for Information Technology Security Evaluation.* The objective of their analysis was to identify additional security activities and artifacts that should be added to the Project Activities defined in IEEE 1074-1997. The result is a new Project Life Cycle Process framework that elevates the visibility and priority of security to a level that appropriately addresses compelling business need.

IEEE Std. 1074-2006 aligns with several quality assurance and improvement standards:

- ISO TL 9000, *Quality Management Systems—Telecommunications.* P1074-2005 requires the definition of a user's software life cycle consistent with this standard;
- ISO 9001, *Quality Management Systems—Requirements*, Section 7.1, "Planning of product realization";
- ISO/IEC 9003 (superseded by ISO 9001:2000);
- Capability Maturity Model Integration (CMMI) Organizational Process Definition (OPD), which requires the establishment of standard processes, life cycle model descriptions, tailoring criteria and guidelines, and establishment of an measurement repository and process asset library);
- ISO/IEC 15288, *Systems Engineering Life Cycle;*

- ISO/IEC 12207, *Software Life Cycle*.

By contrast with the new IEEE Std. 1074-2006, however, the earlier quality-driven standards offer no actionable guidance on ensuring the appropriate prioritization of security. Nor are they structured to support project or product security measurement.

IEEE 1074-2006 is an improvement on ISO/IEC 12207 in particular in that it generates the security information needed to document security risks and solutions throughout the SDLC. The new standard leverages Common Criteria assurance principles and assets, creates the comprehensive security profile needed for evaluation of software integrity, and covers a number of security areas not addressed in ISO/IEC 12207, including generating information essential to secure operations. For example, while ISO/IEC 15288 and 12207 acknowledge change control as an important life cycle activity, neither flags it as the highest-risk activity an organization can undertake. By contrast, IEEE Std. 1074-2006 focuses the majority of its attention on the security risk inherent in system and software change.

IEEE Std. 1074-2006 is intended to ensure control of project security risk and output of quality deliverables that appropriately address business priority and security risk. The standard provides a systematic approach to defining specific security requirements for each discreet life cycle activity, including the ongoing audit and product maintenance/improvement processes. The standard also specifies what is needed to produce the high-quality security artifacts to during each life cycle activity.

The guidance IEEE Std. 1074-2006 provides is clear, actionable, and structured to be easily adapted for tools-based conformance measurement. The standard supports validation of security acceptability through testing, and requires product accreditation by an independent security/integrity auditor or officer. The standard also enhances training activities for security.

D.2. ISO/IEC 15026, Systems and Software Assurance

Sub-Committee (SC) 7 of the Joint Technical Committee (JTC) 1/Sub-Committee (SC) 7 of the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) undertook a project to revise ISO/IEC 15026—*System and Software Engineering—System and Software Assurance*. This revision was intended to add three core processes for planning, establishing, and validating assurance cases for software-based systems, in order to establish the necessary level of assurance of software developed via the systems and software engineering life cycle processes defined in ISO/IEC 12207 and ISO/IEC 15288.

The revised ISO/IEC 15026 would define life cycle process expectations and their associated outcomes for systems and software that must satisfy critical requirements. The standard would also specify the activities that must be undertaken to ensure that systems and software are acquired, developed, and maintained in such a way that is likely to satisfy those critical requirements, and to provide assurance that the implemented system or software does in fact satisfy these requirements throughout its life cycle.

In the most recent draft of the revised standard, the following life cycle processes expectations and their associated outcomes are addressed:

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

No assumption was made as to the specific nature of “critical requirements”. Instead, the standard was intended to apply to whatever risk dimensions are relevant to the system or software (e.g. safety, security, prevention of financial loss). The standard was meant to occupy a position between domain-specific standards—such as IEC 61508 for the safety of programmable electronic systems, and ISO/IEC 15443, Parts 1-3, *Framework for IT*

Security Assurance—and system and software life cycle framework standards, ISO/IEC 15288—*System Life Cycle Processes* and ISO/IEC 12207—*Software Life Cycle Processes.*

The JTC 1/SC 7 effort to define a revised ISO/IEC 15026 recently ended (as of July 2006). The separate effort to harmonize the processes that make up ISO/IEC 12207 and 15288 is ongoing, with the intended result to be a single standard engineering life cycle process definition for software-intensive systems into which the revised 15026 assurance activities were intended to be "dropped in".

It is unclear at present whether the work on revising ISO/IEC 15026 will continue within JTC 1/SC 7, or whether the ISO/IEC work to date on defining assurance activities for the development life cycle will be transferred over to IEEE, and continue under their auspices.

D.3. Proposed Safety and Security Extensions to CMMI/iCMM

The Safety and Security Extension Project Team established by the FAA and the DoD produced the draft report *Safety and Security Extensions to Integrated Capability Maturity Models* (September 2004) defining a safety and security application area to be used in combination with the SEI's CMMI or the FAA's iCMM to achieve process improvements in the safety and security of software produced by CMMI- or iCMM-guided software development life cycle processes.

The Safety and Security Application Area (AA) Extension is intended to implement practices within the relevant Process Areas (PAs) of the iCMM and CMMI. The practices defined are derived from existing Defense (U.S. DoD and UK Ministry of Defence), NIST, and ISO/IEC security and safety standards. The methodology used to map these practices to the appropriate iCMM or CMMI PAs requires, in some cases inserting iCMM PAs into CMMI when no comparable CMMI PA exists, or when an existing CMMI PA is inadequate for achieving the recommended safety or security practices.

Table D-1 maps the iCMM and CMMI PAs to the Safety and Security Application Area (S&S AA) practices recommended by the international team draft report. The report goes on to provide extensive information on the activities, typical work products associated with each AA, and provides recommended practices for achieving the objectives of each AA as it is integrated with the iCMM or CMMI process of the organization.

New activities added to CMMs were derived from several sources, including ISO/IEC 21827: SSE-CMM and ISO/IEC 17799: Code of Practices for Information Security Management. Sixteen of the activities in the Proposed Safety and Security Extensions document have, in turn, been integrated into the revision of ISO/IEC standards on systems and software assurance.

Table D-1. Safety and Security Extensions to iCMM/CMMI

iCMM PA	CMMI PA	S&S AA Practice
PA 22: Training	Organizational Training	AP01.01: Ensure S&S Competency
PA 19: Work Environment	Work Environment	AP01.01: Ensure S&S Competency AP01.02: Establish Qualified Work Environment AP01.05: Ensure Business Continuity AP01.11: Objectively Evaluate Products
PA 17: Information Management	(add iCMM PA 17 to CMMI)	AP01.03: Control Information AP01.12: Establish S&S Assurance Argument
PA 10: Operation and Support	(add iCMM PA 10 to CMMI)	AP01.04: Monitor Operations and Report Incidents AP01.10: Develop and Deploy Safe and Secure Products and Services

Table D-1. Safety and Security Extensions to iCMM/CMMI cont'd

iCMM PA	CMMI PA	S&S AA Practice
PA 13: Risk Management	Risk Management	AP01.05: Ensure Business Continuity AP01.06: Identify S&S Risks AP01.07: Analyze and Prioritize Risks AP01.08: Determine, Implement, and Monitor Risk Mitigation Plan AP01.14: Establish a S&S Plan
PA 00: Integrated Enterprise Management	Organizational Environment for Integration Organizational Innovation and Deployment (add iCMM PA 00)	AP01.05: Ensure Business Continuity AP01.09: Identify Regulatory Requirements, Laws and Standards AP01.13: Establish Independent S&S Reporting AP01.14: Establish a S&S Plan AP01.16: Monitor and Control Activities and Products
PA 01: Needs PA 02: Requirements	Requirements Development Requirements Management	AP01.09: Identify Regulatory Requirements, Laws, and Standards AP01.10: Develop and Deploy Safe and Secure Products and Services
PA 03: Design PA 06: Design Implementation	Technical Solution	AP01.10: Develop and Deploy Safe and Secure Products and Services
PA 08: Evaluation	Verification Validation	AP01.11: Objectively Evaluate Products AP01.12: Establish S&S Assurance Argument
PA 15: Quality Assurance and Management	Process and Product Quality Assurance	AP01.12: Establish S&S Assurance Argument AP01.13: Establish Independent S&S Reporting AP01.16: Monitor and Control Activities and Products
PA 11: Project Management	Project Planning Project Monitoring and Control Integrated Project Management Quantitative Project Management	AP01.01: Ensure S&S Competency AP01.13: Establish Independent S&S Reporting AP01.14: Establish a S&S Plan AP01.16: Monitor and Control Activities and Products
PA 16: Configuration Management	Configuration Management	AP01.16: Monitor and Control Activities and Products
PA 18: Measurement and Analysis	Measurement and Analysis	AP01.16: Monitor and Control Activities and Products
PA 05: Outsourcing PA 12: Supplier Agreement Management PA 09: Deployment, Transition, and Disposal	Supplier Agreement Management Integrated Supplier Management	AP01.15 Select and Manage Suppliers, Products, and Services AP01.10: Develop and Deploy Safe and Secure Products and Services
PA 21: Process Improvement	Organizational Process Focus	AP01.16: Monitor and Control Activities and Products

D.4. ISO/IEC 21827, Systems Security Engineering Capability Maturity Model

The Systems Security Engineering Capability Maturity Model (SSE-CMM) is a process reference model for improving and assessing the maturity of the security engineering processes used to produce information security products, trusted systems, and security capabilities in information systems. The scope of the processes addressed by the SSE-CMM encompasses all activities of the system security engineering life cycle, including concept definition, requirements analysis, design, development, integration, installation, operation, maintenance, and decommissioning. The SSE-CMM includes requirements for product developers, secure systems developers and integrators, and organizations that provide computer security services and/or computer security engineering, including organizations in the commercial, government, and academic realms.

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

The SSE-CMM is divided into two dimensions: domain and capability. On the domain side, practices are organized in a hierarchy of process categories, process areas, and base practices. The SSE-CMM augments project and organizational process areas from the Systems Engineering (SE) CMM with the security engineering process areas depicted in Table D-2:

Table D-2. SSE-CMM Security Engineering Process Areas and Goals

Security Engineering Process Area (PA)	Goals of PA
Administer security controls	Ensure that security controls are properly configured and used.
Assess impact	Reach an understanding of the security risk associated with operating the system within a defined environment.
Assess security risk	Identify system vulnerabilities and determine their potential for exploitation.
Assess threat	Reach an understanding of threats to the security of the system.
Assess vulnerability	Reach an understanding of the system's security vulnerabilities.
Build assurance argument	Ensure that the work artifacts and processes clearly provide the evidence that the customer's security needs have been met.
Coordinate security	Ensure that all members of the project team are aware of and involved with security engineering activities to the extent necessary to perform their functions; coordinate and communicate all decisions and recommendations related to security.
Monitor security posture	Detect and track internal and external security-related events; respond to incidents in accordance with policy; identify and handle changes to the operational security posture in accordance with security objectives.
Provide security input	Review all system issues for security implications and resolved those issues in accordance with security goals; ensure that all members of the project team understand security so they can perform their functions; ensure that the solution reflects the provided security input.
Specify security needs	All applicable parties, including the customer, reach a common understanding of security needs.
Verify and validate security	Ensure that the solutions satisfy all of their security requirements and meet the customer's operational security needs.

On the capability side, the model identifies capability (maturity) levels from zero to five. Higher levels imply increased organizational support for planning, tracking, training, etc., which leads to more consistent performance of the domain activities. This support is captured in a set of common features and generic practices for each level.

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

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

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

improvement and in selecting suppliers. The appraisal process is outlined in Chapter 4 of the SSE-CMM Model Description Document and in the SSE-CMM Appraisal Method Description.

Specific ways in which SSE-CMM's creators envisioned it would be used include:

1. **Acquisition/Source Selection**

    - Acquirers: Would use the SSE-CMM in specifying capability requirements in the statement of work (SOW) for a new information system, so as to gauge whether bidders have the necessary capability to successfully perform security engineering for the new system.

    - System developers would perform self-assessments against the SSE-CMM capability assessment criteria. The acquirer may accept a developer's self-assessments, or if system criticality, data sensitivity, and cost warrant, may perform its own assessment of the developer. The system developer may also specify SSE-CMM profiles for subcontractors, hardware and software suppliers, and service providers.

2. **System Development**

    - System developers would ensure the institutionalization of their security engineering processes, and would consistently produce appropriate evidence that their security products were correctly engineered and met the customer's security needs. Because repeatable results are a hallmark of capability maturity, the customer would be able to expect that the developer's previous security engineering successes would be repeated. Moreover, new products could be easily incorporated into the system because the SSE-CMM was used to expedite the passage of those products through evaluation.

3. **Security Assessment**

    - System developers would ensure that security requirements were appropriate before system development began. The SSE-CMM would guide the developer in the capture, verification, and validation of security requirements. The process areas "Verify and validate security" and Coordinate security" in particular would help the developer ensure that the security requirements were met as the system was being developed.

    - Because the SSE-CMM ensures that necessary assurance evidence was produced during the course of system development, security assessment of the developed system by certifiers or evaluators would be accomplished more quickly and easily. The security assessor could take the developer's security engineering capability into account as a basis for confidence in the quality of the system's security engineering and in its integration with other engineering disciplines.

4. **Operation and Sustainment**

    - Operators and sustainers would be guided by the SSE-CMM in the continued secure operation of the system in a secure manner. Configuration changes would be managed, changing threats and risks would be addressed, and the security posture would monitored on an ongoing basis.

D.5. Trusted Capability Maturity Model

The Trusted Capability Maturity Model (TCMM), an integrated reference model derived from the software assurance principles contained in the Missile Defense Agency Strategic Defense Initiative Office's (SDIO's) Trusted Systems Development Methodology (TSDM; see note) and the SEI's CMM for software (SW-CMM).

The TCMM was first defined by a team formed by NSA and the SEI in 1995 (Version 2.0 was published in 1996) as a technique for establishing quantitative measures of process maturity and of compliance with developmental principles that increase the trustworthiness of software. NSA's goal for the project was to replace several assessments (e.g., process-based, product-oriented, etc.) with a single assessment, thus significantly reducing the length and expense of the post-development testing and evaluation cycle.

TCMM was also meant to provide more reliable, higher-quality software systems—especially life-critical applications such as military and financial systems. Consistent with the TSDM, TCMM activities were inserted throughout the software life cycle to reduce the ability of developers to introduce incorrect logic and other into software, either intentionally or inadvertently.

The TCMM revised several SW-CMM Key Process Areas (KPAs) and introduced one new KPA, "Trusted Software Development", comprising several new practices that did not fit within any existing SW-CMM KPA. The TCMM practices focused on improving the development environment and management and organizational processes/activities (and on measuring that improvement).

The TCMM differed considerably from the NSA's subsequent SSE-CMM (described below), which eventually supplanted it, and focused on controlling system development processes and activities specifically directed towards creation and maintenance of trusted systems.

> *NOTE: The TSDM, formerly called the Trusted Software Methodology (TSM), was developed by the SDIO in the mid 1980s to increase software assurance by strengthening the development process. The estimated size (in millions of lines of code) of the proposed SDIO software systems meant that typical software error densities would have rendered those systems inoperative. To reduce the software error rate, the TSDM described a development and assurance process for establishing high degree of confidence that the software produced would be free from inadvertent errors and malicious code. The TSDM outlined 25 Trust Principles (documented in the SDIO's 2 July 1993 TSM Report), each of which has an associated rationale, set of compliance requirements, list of applicable trust classes, and list of associated requirements (with useful references) for activities similar to those addressed in the trust principle. A number of TSDM concepts were adapted and adopted by later trusted and secure development methods.*

D.6. References for this Appendix

Bar Biszick-Lockwood: IEEE P1074-2005: Roadmap for Optimizing Security in the System and Software Life Cycle (2005)
http://www.qualityit.net/Resources/WhitePapers/IEEEP1074-2005-RoadmapForOptimizingSecurityInTheSystemAndSoftwareLifeCycle.pdf

Bar Biszick-Lockwood: The Benefits of Adopting IEEE P1074-2005
http://qualityit.net/Resources/WhitePapers/BenefitsOfAdoptingIEEEP1074-2005.pdf

FAA Integrated Process Group: Safety and Security Extensions to Integrated Capability Maturity Models (September 2004)
http://faa.gov/ipg/news/docs/SafetyandSecurityExt-FINAL.doc - *or* -
http://faa.gov/ipg/news/docs/SafetyandSecurityExt-FINAL.pdf

CMMI
http://www.sei.cmu.edu/cmmi/

iCMM
http://www.faa.gov/ipg/pif/icmm/index.cfm

International Systems Security Engineering Association (ISSEA) SSE-CMM Project Team: Systems Security Engineering Capability Maturity Model (SSE-CMM) Model Description Document, Version 3.0 (15 June 2003)
http://www.sse-cmm.org/

Rick Hefner, TRW; Ron Knode, Computer Sciences Corp.; and Mary Schanken, NSA: The Systems Security Engineering CMM (*CrossTalk: The Journal of Defense Software Engineering*, October 2000)
http://www.stsc.hill.af.mil/crosstalk/2000/10/hefner.html

Karen Ferraiolo and Victoria Thompson, Arca Systems, Inc.: Let's Just Be Mature about Security! Using a CMM for Security Engineering (*CrossTalk: The Journal of Defense Software Engineering*, August 1997)
http://www.stsc.hill.af.mil/crosstalk/frames.asp?uri=1997/08/security.asp

ISO/IEC 12207, Software Life Cycle Processes
http://www.abelia.com/docs/12207cpt.pdf - *and* -
http://www.software.org/quagmire/descriptions/iso-iec12207.asp

ISO/IEC 15288, System Life Cycle Processes
http://www.software.org/quagmire/descriptions/iso-iec15288.asp

Integration with System Engineering: ISO/IEC 15288 (2002) Systems engineering - systems life cycle processes
http://www.processforusability.co.uk/Usability_test/html/integration.html

ISO/IEC 16085 (originally IEEE Std 1540), Software Risk Management Process
http://psmsc.com/UG2001/Presentations/06IEEEStandard1540.PDF

ISO/IEC 15939, Software Measurement Process
http://www.software.org/quagmire/descriptions/iso15939.asp

SixSigma
http://www.motorola.com/content/0,,3070-5788,00.html

APPENDIX E. SOFTWARE AND APPLICATION SECURITY CHECKLISTS

Checklists are at the heart of software inspections. Product checklists house the strongly preferred indicators that set the standard of excellence for the organization's software products. They fuel the structured review process and form the standard of excellence expected for the software product. Some of the aspects of the software's security that are considered in checklists include:

- **Completeness:** Checking focuses on traceability among software product artifacts of various types including requirements, specifications, designs, code, and test procedures. Completeness analysis may be assisted by tools that trace the components of a product artifact of one type to the components of another type. Completeness analysis of predecessor and successor artifacts reveals what sections are missing and what fragments may be extra. A byproduct of the completeness analysis is a clear view of the relationship of requirements to the code product: straightforward (one to one), simple analysis (many to one), and complex (one to many).

- **Correctness:** Checking focuses on reasoning about programs by answering informal verification and correctness questions derived from the prime constructs of structured programming and their composite use in proper programs. Input domain and output range are analyzed for all legal values and all possible values. State data is similarly analyzed. Adherence to project-specified disciplined data structures is analyzed. Asynchronous processes and their interaction and communication are analyzed.

- **Style:** Checking is based on project-specified style guidance. This guidance is expected to call for block structured templates. Naming conventions and commentary are checked for consistency of use along with alignment, highlighting, and case. More advanced style guidance may call for templates for repeating patterns and semantic correspondence among software product artifacts of various types.

- **Rules of construction:** Checking focuses on the software system's architecture and the specific protocols, templates, and conventions used to carry it out. For example, these include interprocess communication protocols, tasking and concurrent operations, program unit construction, and data representation.

- **Multiple views:** Checking focuses on the various perspectives and view points required to be reflected in the software product. During execution many factors must operate harmoniously as intended including initialization, timing of processes, memory management, input and output, and finite word effects. In building the software product, packaging considerations must be coordinated including program unit construction, program generation process, and target machine operations. Product construction disciplines of systematic design and structured programming must be followed as well as interactions with the user, operating system, and physical hardware.

The following publicly-available programming and application security checklists are relatively widely used for checking for one or more of these aspects. These checklists should help developers assess the key security aspects of their software at various stages of its life cycle.

- NASA Software Security Checklist — *contact Kenneth McGill, NASA JPL IV&V Center at:* Kenneth.G.McGill@nasa.gov

- DISA: Application Security Checklist Version 2 Release 1.8 (14 March 2006) http://iase.disa.mil/stigs/checklist/app-security-checklist-v2r18-14mar06.doc

- Open Web Application Security Project (OWASP): Web Application Penetration Checklist v1.1 http://www.owasp.org/documentation/testing/application.html

- Visa U.S.A.: CISP Payment Application Best Practices checklist
  http://usa.visa.com/download/business/accepting_visa/ops_risk_management/cisp_Payment_Application_Best_Practices.pdf

- Djenana Campara: Secure Software: A Manager's Checklist (20 June 2005)
  http://www.klocwork.com/company/downloads/SecureSoftwareManagerChecklist.pdf

- Australian Computer Emergency Response Team (AusCERT) Secure Unix Programming Checklist (July 2002 version)
  http://www.auscert.org.au/render.html?it=1975

- Don O'Neill Consulting: Standard of Excellence Product Checklists *(including a Security Checklist)*
  http://members.aol.com/ONeillDon2/special_checklist_frames.html

APPENDIX F. SECURITY CONCERNS ASSOCIATED WITH OUTSOURCED AND OFFSHORE SOFTWARE DEVELOPMENT

The following excerpts from three government reports, two from the Government Accountability Office (GAO) and one from the Defense Science Board (DSB), that discuss key concerns associated with offshore outsourcing of software development by commercial software firms.

F.1. Excerpt from GAO Report on Defense Department Acquisition of Software

SOURCE: General Accountability Office (GAO): "Defense Acquisitions: Knowledge of Software Suppliers Needed to Manage Risk"

The Department of Defense (DoD) is experiencing significant and increasing reliance on software and information systems for its weapon capabilities, while at the same time traditional DoD prime contractors are subcontracting more of their software development to lower tier and sometimes nontraditional defense suppliers. Because of economic incentives and business relationships, these suppliers are using offshore locations and foreign companies to complete various software development and support tasks. DoD is also using more commercial-off-the-shelf (COTS) software [Footnote 1] to reduce development costs and allow for frequent technology updates, which further increases the number of software developers with either direct or indirect access to weapon system software. The increased dependence on software capability, combined with an exposure to a greater variety of suppliers, results in more opportunities to exploit vulnerabilities in defense software. Software security, including the protection of software code from hacking and other malicious tampering, is therefore an area of concern for many DoD systems. As DoD's need for software increases, knowledge about foreign suppliers [Footnote 2] in software development is critical for identifying and mitigating risks. Multiple national security policies and other guidance recognize the inherent risks associated with foreign access to sensitive information and technology. For 2001, the Defense Security Service (DSS) reported a significant increase in suspicious attempts by foreign entities to access U.S. technology and information, with one-third of that activity coming from foreign government-sponsored or affiliated entities. While both U.S.- and foreign-developed software are vulnerable to malicious tampering, DSS specifically noted a concern with the potential exploitation of software developed in foreign research facilities and software companies located outside the United States working on commercial projects related to classified or sensitive programs. As foreign companies and individuals play a more significant role in software development activities, the need for knowledge to manage associated risks also increases....

Results in Brief:

DoD acquisition and software security policies do not require program managers to identify and manage the risks of using foreign suppliers to develop weapon system software. The primary focus of the acquisition policies is to acquire weapon systems to improve military capability in a timely and cost-effective manner. Despite the inherent risks associated with foreign access to sensitive information and technology, the guidance allows program officials discretion in managing risks related to foreign involvement in software development. Other requirements and policies intended to mitigate information system vulnerabilities focus primarily on operational software security threats, such as external hacking and unauthorized access to information systems, but not on insider threats such as the insertion of malicious code by software developers. While recent DoD initiatives, such as the establishment of working groups to evaluate software products and security

processes, may help to increase DoD's focus on software security and may lead to the development and identification of several potential software security best practices, they have yet to be implemented in weapon acquisition programs.

Given broad discretion, program officials for 11 of the 16 software intensive weapon systems we reviewed did not make foreign involvement in software development a specific element of their risk management efforts. As a result, the program offices had varying levels of awareness of the extent of software developed by foreign suppliers on their systems. Program officials generally did not consider the risks associated with foreign suppliers substantial enough to justify specific attention, and instead focused their resources on meeting software development cost and schedule goals while ensuring software functionality and quality. In addition, most of the program offices relied on their defense contractors to select software subcontractors and ensure that best available software development practices were being used, such as peer review and software testing. Without specific guidance from program officials, contractors primarily focused their software development efforts on meeting stated performance requirements, such as software quality and functionality, rather than mitigating potential software development security risks associated with using foreign suppliers. Contractors that excluded foreign suppliers from their programs' software development did so to avoid the additional costs and resources needed to mitigate the risks of using such suppliers.

As the amount of software on weapon systems increases, it becomes more difficult and costly to test every line of code, and DoD cannot afford to monitor all worldwide software development facilities or provide clearances for all potential software developers. Program managers who know more about potential software development suppliers early in the software acquisition process will be better equipped to include software security as part of source selection and risk mitigation decisions. Therefore, we are making three recommendations to the Secretary of Defense to ensure such knowledge is available to address risks attributable to software vulnerabilities and threats. In written comments on a draft of this report, DoD agreed with our findings that malicious code is a threat that is not adequately addressed in current acquisition policies and security procedures and stated that the department is working to strengthen software related risk management activities. However, DoD only partially agreed with our recommendations over concerns that responsibility for mitigating risks would be placed on program managers and software assurance experts alone. We made adjustments to our recommendations to acknowledge the value of having other DoD organizations involved in software security risk management.

Background

To protect the security of the United States, DoD relies on a complex array of computer-dependent and mutually supportive organizational components, including the military services and defense agencies. It also relies on a broad array of computer systems, including weapon systems, command and control systems, financial systems, personnel systems, payment systems, and others. These systems are, in turn, connected with other systems operated by contractors, other government agencies, and international organizations. In addition, performance requirements for weapon systems have become increasingly demanding, and breakthroughs in software capability have led to a greater reliance on software to provide more weapon capability when hardware limitations are reached. As such, DoD weapon systems are subject to many risks that arise from exploitable software vulnerabilities. Software code that is poorly developed or purposely injected with malicious code could be used to disrupt these and other DoD information systems, and potentially others connected to the DoD systems.

DoD has reported that countries hostile to the United States are focusing resources on developing information warfare strategies. For example, a DSS report noted that

in 2001 there was a significant increase in suspicious attempts by foreign entities to access U.S. technology and information and that trend is expected to continue. Information systems technology was the most sought after militarily critical technology by these entities. Forty-four countries were associated with attempts at accessing U.S. information technology, with 33 percent of the activity coming from foreign government-sponsored or affiliated entities. Because the U.S. defense industry is at the forefront of advanced design and development of weapon systems that include militarily critical technologies, access is sought after for industrial and financial purposes. Access to these technologies by potential adversaries could enhance the performance of their military systems and may be used to counter U.S. capabilities. DSS specifically noted a concern with exploitation or insertion of malicious code with the use of foreign research facilities and software development companies located outside the United States working on commercial projects related to classified or sensitive programs.

Multiple requirements and guidance are in place to ensure the protection of U.S. national security interests. They generally acknowledge the inherent risk associated with foreign access to classified and export-controlled information and technology by establishing procedures to manage such access. For example, the National Industrial Security Program Operation Manual [Footnote 3] establishes mandatory procedures for the safeguarding of classified information that is released to U.S. government contractors. It generally limits access to U.S. citizens with appropriate security clearances and establishes eligibility policies for U.S. contractors determined to have foreign ownership, control, or influence. Further, an additional DOD directive requires programs containing classified military information to have controls to prevent the unauthorized release of this information to foreign recipients. [Footnote 4] In addition, the International Traffic in Arms Regulations (ITAR) controls foreign access to defense articles and services through the establishment of the export license and authorization process. U.S. entities, including defense contractors, may apply to the Department of State for authorization to export controlled information and technology to qualified foreign recipients, which is determined through the approval or denial of license requests.

DoD estimates that it spends about 40 percent of its Research, Development, Test, and Evaluation budget on software--$21 billion for fiscal year 2003. Furthermore, DoD and industry experience indicates that about $8 billion of that amount may be spent on reworking software because of quality-related issues. Carnegie Mellon University's Software Engineering Institute (SEI), recognized for its expertise in developing best practices for software processes, has developed models and methods that define and determine organizations' software process maturity. Better software development practices are seen as a way to reduce the number of software defects and therefore improve overall software quality, but alone the practices cannot be expected to address malicious software development activities intended to breach security. To underscore the importance of securing software-related products, the Office of the Assistant Secretary of Defense (Networks and Information Integration) and the Federal Aviation Administration Chief Information Office are co-sponsoring, with the involvement of the Department of Energy, the National Aeronautics and Space Administration, and SEI, a project aimed at developing ways to provide safety and security assurance extensions to integrated software maturity models.

Footnotes:

[1] COTS software is that which is not specifically developed for military use and instead purchased "as-is" from an outside vendor.

[2] For the purposes of this report, a foreign supplier is defined as any foreign company or foreign national working for companies either in the United

States or abroad. It encompasses both prime contractors and subcontractors performing work under those contracts.

[3] The *National Industrial Security Program Operating Manual*, DOD 5220.22-M, prescribes specific requirements, restrictions, and other safeguards necessary to prevent unauthorized disclosure of classified information to U.S. contractors.

[4] DOD Directive 5230.11, "Disclosure of Classified Military Information to Foreign Governments and International Organizations", June 16, 1992.

F.2. Excerpt from GAO Report on Offshoring of Services

SOURCE: GAO Report to Congressional Committees: "Offshoring of Services: An Overview of the Issues"

Experts express varying degrees of concern that offshoring could pose security risks, including increased risks to national security, critical infrastructure, and personal privacy. Underlying these disagreements are unresolved questions about the extent to which offshore operations pose additional risks than outsourcing services domestically and the extent to which U.S. laws and standards apply and are enforceable for work conducted offshore.

Some Concerns Have Been Raised That Offshoring May Pose Increased Risks to National Security and Critical Infrastructure, Though Some Experts Contend That Offshoring May Not Pose Additional Major Risks:

Some security and offshoring experts, including the Department of Defense (DoD), have raised concerns that offshoring could pose increased risks to national security and critical infrastructure, but others believe that offshoring will not. National security concerns relate to government programs and systems involved in national defense, particularly military and intelligence operations. Critical infrastructure concerns relate to systems and structures owned by either government or private entities that are essential to the country, such as utilities, transportation, and communications networks. [Footnote 53]

One concern raised by security experts is that offshoring the development of software used in defense systems could pose additional security risks, specifically, that foreign workers with hostile intentions could obtain critical information or introduce malicious code into software products that could interfere with defense or infrastructure systems. There are currently few explicit restrictions on the type of services work that can be sent offshore. [Footnote 54] DoD's Defense Security Service has analyzed this issue and identified concerns with the potential exploitation of software developed in foreign research facilities and software companies for projects related to classified or sensitive programs. [Footnote 55] We have reviewed DoD's management of software developed overseas for defense weapons systems as well. [Footnote 56] Our report noted that multiple requirements and guidance acknowledge the inherent risks associated with foreign access to classified or export-controlled information and technology and are intended to protect U.S. national security by managing such access. However, we found that DoD does not require program managers of major weapons systems to identify or manage the potential security risks from foreign suppliers. For instance, DoD guidance for program managers to review computer code from foreign sources not directly controlled by DoD or its contractors is not mandatory. In addition, DoD programs cannot always fully identify all foreign-developed software in their systems.

Private-sector groups and government officials have raised similar concerns about the added security risks posed by offshoring to U.S. non-military critical

infrastructure, such as nuclear power plants, the electric power grid, transportation, or communications networks. For example, some have noted that sensitive but unclassified information, such as the plans of important U.S. utilities or transport networks, could be sent to foreign locations where it could be released improperly or made available to hostile foreign nationals. Other concerns relate to the offshoring of software development and maintenance. Software security experts in the public sector--including DoD and the Central Intelligence Agency--have expressed concern that organizations and individuals with hostile intentions, such as terrorist organizations and foreign government economic and information warfare units, could gain direct access to software code by infiltrating or otherwise influencing contractor and subcontractor staff, and then use this code to perpetrate attacks on U.S. infrastructure systems or conduct industrial or other forms of espionage. Security experts also note that critical infrastructure systems rely extensively on commercial off the shelf (COTS) software programs that are developed in locations around the world. These programs include exploitable vulnerabilities and potentially even malicious code that can allow indirect access to infrastructure systems to cause the systems to perform in unintended ways. Thus, some experts believe that ongoing use of COTS software modules, whether developed offshore or not, as well as offshoring of software-related services could increase the risk of unauthorized access to critical infrastructure code in comparison to in-house development and maintenance of proprietary programs and code.

Security experts also express concerns about longer-term effects of offshoring. For instance, some note that continued offshoring of certain products might make the U.S. dependent on foreign operations for critical civilian or military products, and therefore vulnerable if relations between the U.S. and those countries become hostile. Another concern is the ability to control access to certain civilian technologies with military uses when work on these technologies takes place in foreign locations. Some fear that offshoring certain high-tech work may lead to the transfer of information and technology that could be used by foreign entities to match or counter current U.S. technical and military superiority. The U.S. can control exports of such dual-use technologies by requiring firms to obtain an export license from the Department of Commerce before they can be worked on in foreign locations or by foreign nationals. We have reviewed some aspects of this export licensing program and found key challenges to Commerce’s primary mechanism for ensuring compliance with export licenses. [Footnote 57]

Some representatives of business groups contend that offshoring may not pose major increased security concerns for a variety of reasons. Some believe that protections currently in place are adequate to manage the added risks posed by offshoring. Currently, the Department of Defense has mandatory procedures to safeguard classified information that is released to U.S. government contractors, and firms that offshore certain work related to military technologies are required to obtain export licenses from either the State or Commerce departments. [Footnote 58] In addition, some argue that foreign workers in offshore locations do not necessarily pose added security risks, relative to U.S. workers in domestic outsourced operations, because domestic workers could also improperly handle information. Some foreign affairs experts also argue that offshoring could have positive effects on national security. They contend that increased international trade may reduce the threat of international tensions because countries with integrated economies have a stake in one another’s well-being....

Proposals to Protect Security:

Proposals to address concerns about security seek to reduce the added risk that information sent to foreign locations could be used in ways that could impair U.S. national security, critical infrastructure, or personal privacy. Proposals include

restrictions on certain types of work with security implications and strengthening standards governing how information is handled.

Protecting National Security and Critical Infrastructure:

Concerns that offshoring could pose increased risks to national security or critical infrastructure have led to proposals to restrict some services work from being sent to foreign locations or performed by foreign nationals and to improve security standards for work that is performed offshore, including the following proposals:

- Requiring that certain projects involving defense acquisitions or military equipment be performed exclusively in the U.S.
- Requiring that work on critical infrastructure projects such as electricity grids or pipelines be done within the U.S.
- Increasing the standards and review procedures that apply to use of offshore services. For example, GAO has previously recommended that DoD adopt more effective practices for developing software and increasing oversight of software-intensive systems, such as ensuring that risk assessments of weapons programs consider threats to software development from foreign suppliers. [Footnote 68]

Footnotes:

[53] Presidential Decision Directive/NSC-63 on Critical Infrastructure Protection defines critical infrastructure as "those physical and cyber-based systems essential to the minimum operations of the economy and government. They include, but are not limited to, telecommunications, energy, banking and finance, transportation, water systems, and emergency systems, both governmental and private. "

[54] There are some restrictions in annual defense appropriations and authorization acts requiring certain DOD procurements to be performed by U.S. firms, for instance research contracts in connection with weapons systems and the Ballistic Missile Defense Program. See GAO, Federal Procurement: International Agreements Result in Waivers of Some U.S. Domestic Source Restrictions, GAO-05-188 (Washington, D.C.: Jan. 26, 2005); also Defense Federal Acquisition Regulation Supplement, Subpart 225.70.

[55] Defense Security Service, Technology Collection Trends in the U.S. Defense Industry 2002 (Alexandria, Va.: 2002).

[56] GAO, Defense Acquisitions: Knowledge of Software Suppliers Needed to Manage Risks, GAO-04-678 (Washington, D.C.: May 25, 2004).

[57] GAO, Export Controls: Post-Shipment Verification Provides Limited Assurance That Dual-Use Items Are Being Properly Used, GAO-04-357 (Washington, D.C.: Jan. 12, 2004). This GAO review did not look at controls over service inputs specifically but found weaknesses in Commerce's post-shipment verification checks for confirming that controlled items sent to countries of concern arrive at their proper location and are used in compliance with the conditions of export licenses.

[58] The State Department manages the regulation of defense articles and services, while the Commerce Department manages the regulation of dual-use items with both military and commercial applications. In most cases, Commerce's

controls are less restrictive than State's. GAO has reviewed this export-control system, and found that Commerce has improperly classified some State-controlled items. See GAO, Export Controls: Processes for Determining Proper Control of Defense-Related Items Need Improvement, GAO-02-996 (Washington, D.C.: Sept. 20, 2002).

[68] See GAO-04-678.

F.3. Excerpt from Defense Science Board Report on Globalization and Security

SOURCE: Office of the Under Secretary of Defense for Acquisition and Technology: "Final Report of the Defense Science Board Task Force on Globalization and Security", specifically Annex IV: "Vulnerability of Essential U.S. Systems Incorporating Commercial Software"

The principal risks associated with commercial acquisition lie in the software area, where heavy reliance on commercial software--often developed offshore and/or by software engineers with little if any allegiance to the United States--is almost certainly amplifying DoD's vulnerability to adversary information operations.

Commercial software products--within which malicious code can be hidden--are becoming foundations of DoD's future command and control, weapons, logistics and business operational systems (e.g., contracting and weapon system support). Such malicious code, which would facilitate system intrusion, would be all but impossible to detect via traditional testing, primarily because of commercial software's extreme complexity, and the ever-increasing complexity of the systems into which commercial software is being incorporated. Moreover, cyber-aggressors need not be capable of, or resort to implanting malicious code to penetrate commercial software-based DoD systems. They can readily exploit inadvertent and highly transparent vulnerabilities (bugs, flaws) in commercial software products that have been incorporated into DoD systems.

Risk management of systems incorporating commercial software is not currently practiced assiduously at every stage of a system's life cycle (i.e., from concept development through operations and maintenance). Moreover, accountability for system integrity is neither fixed nor resting at a sufficiently authoritative level.... As currently constituted, the personnel security system is ill-suited to mitigate the growing risks associated with commercial software acquisition. There exists today only a hint of the aggressive, focused counter-intelligence program that is required.

F.4. References for this Appendix

GAO: "Defense Acquisitions: Knowledge of Software Suppliers Needed to Manage Risk" (GAO-04-768, May 2004)
http://www.gao.gov/docdblite/summary.php?rptno=GAO-04-678&accno=A10177

GAO: Report to Congressional Committees by the United States Government Accountability Office: "Offshoring of Services: An Overview of the Issues" (GAO-06-5, November 2005)
http://www.gao.gov/docdblite/summary.php?rptno=GAO-06-5&accno=A42097

Office of the Under Secretary of Defense for Acquisition and Technology: "Final Report of the Defense Science Board Task Force on Globalization and Security"—Annex IV: "Vulnerability of Essential U.S. Systems Incorporating Commercial Software" (December 1999)
http://www.acq.osd.mil/dsb/reports/globalization.pdf

APPENDIX G. SECURE DEVELOPMENT PRINCIPLES AND PRACTICES

The following practical suggestions for improving the security of development practices throughout the life cycle are intended to be implemented in the short term to start improving the security of software. There will continue to be shortfalls and gaps in security practices, as security analysis and metrics remain a difficult problem. Research is currently under way to address these issues: more secure processes and analysis tools are emerging; in addition, several organizations are working on providing security metrics to further aid in the analysis of security practices and software security.

This section is not intended to be comprehensive or exhaustive. A large and rapidly increasing number of resources, both online and in print, exist on all of the major topics covered in this section. As a quick reference, we feel this section will be helpful to developers, either to reinforce the knowledge they have gained from more extensive reading and study, or to introduce new ideas for them to investigate further.

G.1. Developing for the Production Environment

Unless the software process or component being developed is explicitly intended to perform a trusted function (e.g., cryptography-related, "trusted regrading" in a cross domain solution, reference monitor in a trusted operating system), it should be developed to require *no* "trusted user" (root or administrator) privileges. Moreover, it should be developed in a way that ensures that moving it to the production environment will have no negative impact on its ability to operate correctly within the constraints of that production environment.

Every process in a software system, including a trusted system, should be written to require the absolute minimum permissions and privileges that will enable it to run correctly (consistent with the principle of "least privilege"). Nor should the software be written to expect access to any directory except that in which it itself will be installed. It should also be developed so it can run correctly given the most restrictive environment configuration settings (e.g., all unused and high-risk network and system services disabled).

Developers will be less likely to write software that expects excessive permissions, unnecessary directory access, etc., if the security settings in the development environment mirror those in the production environment to the greatest extent possible. This means setting up the software's directory and its processes' permissions to mirror those expected in the production environment. In addition, the development environment should be configured to enforce strict controls on developer access to artifacts that he/she did not create: each developer should be granted write access to his/her own files only, to prevent him/her from inadvertently (or in the case of a malicious developer, intentionally) overwriting or deleting another developer's work. A virtual machine environment, such as VMWare, Microsoft Virtual PC, or University of Cambridge's Xen for Linux, can make it easier to set up a development host in which different development activities, tools, and artifacts are isolated.

G.2. Software Security Requirements Engineering

In their paper, "Core Security Requirements Artifacts" (available on the BuildSecurityIn portal; see Appendix B:B.3), Moffett, Haley, and Nuseibeh observe that, "Although security requirements engineering has recently attracted increasing attention, ...there is no satisfactory integration of security requirements engineering into requirements engineering as a whole." The authors go on to study a process by which software security requirements were decomposed, through several risk analysis and modeling steps, from a system-level statement of system security needs or goals.

Software security requirements must be regarded as part of the overall systems engineering problem, and not in isolation. Security goals, unlike functional goals, cannot be discharged by the one-time specification of a function or constraint. Security must be considered at every iteration of the development life cycle.

It is critical that the software's security requirements be as complete as possible. To accomplish this, it is first important to establish a complete and accurate statement of the system-level security needs and goals. If all of those goals are explicitly addressed during the process of decomposing the security requirements for the software, completeness of those requirements is more likely.

To avoid inconsistent or inaccurate specification of security requirements, the first step in the requirements process should be for the system's stakeholders to reach agreement on how to define key terms, such as confidentiality, availability, integrity, recovery, restricted access, etc.

Eliciting requirements from the stakeholders should involve direct communication, e.g., structured interviews, but is also likely to require reviews of artifacts identified or provided by the stakeholders. In the case of security, such artifacts will probably include organizational security policies, standards, etc., which may have to be carefully interpreted to understand their specific implications for software. For example, requirements for authentication, authorization, and accountability of users may need to address both human users and software entities (processes, Web services) that *act* as (or on behalf of) users. Factors to consider in the elicitation and capture of security requirements include:

- **Deriving interpreted requirements from stakeholder sources:** Sources could be from policy, laws, standards, guidelines, or practices documents;
- **Risk assessment of the functional requirements specification:** This initial risk analysis should form an important source of security requirements for the system;
- **Need for definition of the security context for the operation of the software:** Specifically, this means developing a threat model and an attacker profile that will clarify the security context provided by the environment in which the software will operate;
- **Risks associated with risk assessments:** When considering attacks, the metrics used to measure probability and impact can have wide variances. Over the life of the software, there may be significant changes in the threat environment, both in terms of the types of attacks to which the software will be subject, and the probabilities of those attacks occurring. Use of detailed attack information in the risk analysis could soon render the results of the analysis obsolete. This obsolescence could then reduce the correctness and completeness of security requirements and design. Moreover, the usual approach to dealing with expected cost of failure (i.e., *expected loss = [loss] x [probability of the event]*) is not helpful for events with low probabilities but significant impacts/consequences; many intentional attacks on software are low probability-high consequence events.

A key determination to be made through the risk assessment is the required level of assurance the application must demonstrate, given the levels of both security risk and business risk to which it will be exposed. This assurance level will be used to inform the criteria for selecting the components, interface techniques, and security protections that will be used in the software system. A discussion of design and security assurance appears in Appendix G:G.3.3.6.

Requirements elicitation methods such as IBM's Joint Application Design (JAD—http://www.umsl.edu/~sauter/analysis/JAD.html) and NASA's Automated Requirements Measurement (ARM—http://satc.gsfc.nasa.gov/tools/arm/) can be used to augment structured interviews, threat modeling, attack trees and misuse and abuse cases, to determine an accurate, comprehensive set of security requirements for software. Note that the requirements engineering activities within CLASP are specifically geared towards the specification of requirements for *secure* software systems.

A specific software requirements methodology that has been cited as useful when specifying requirements for secure software systems is Software Cost Reduction (SCR) developed by the U.S. Naval Research Laboratory (NRL). The SCR toolset contains tools for specifying, debugging, and verifying system and software

requirements. These tools have proved useful in detecting errors involving safety properties in software specifications for the International Space Station, a military flight guidance system, and a U.S. weapons system. More recently, researchers at NRL applied the SCR toolset to the specifications of requirements involving security properties. Their findings indicated that that SCR was very helpful in refining and verifying the requirements specification, but significantly less so in the initial capture and correct formulation of those requirements. Nor did the toolset provide much support for test case generation to enable source code validation based on specification-based testing.

More recently, the team at the University of Southern California Center for Software Engineering (USC CSE) that invented the Constructive Cost Model (COCOMO) for costing software development projects began work on COSECMO, a new model expressly intended for costing secure software-intensive systems. COSECMO introduces a new cost driver, and defines guides for setting other COCOMO drivers when costing the development of a secure system. The USC CSE team is also developing a model for estimating the cost to acquire secure systems, and is evaluating the effect of security goals on other models in the COCOMO family of costing models.

Two systems requirements engineering methods may also prove adaptable to the problem of software security requirements engineering: Security Quality Requirements Engineering (SQUARE) and Requirements Engineering Verification and Validation (REVEAL). SQUARE is a nine-step process developed by Professor Nancy Mead of the CMU SEI to capture requirements associated with safety and survivability in information systems and applications. The methodology explicitly addresses the need to capture and validate the safety and security goals of the system, and entails developing both use cases and misuse/abuse cases and attack trees. SQUARE also provides for prioritization and categorization of safety and security requirements, cost and impact analyses, and budgeting. While SQUARE defines nine steps, it is inherently flexible enough to allow steps to be "lumped together", as has been demonstrated by SEI's own use of the methodology. SQUARE promotes ongoing communication with the software's stakeholders to ensure that their needs/goals are accurately reflected in the prioritized, categorized requirements that result.

REVEAL emerged from lessons learned by Praxis High Integrity Systems in their use of requirements engineering when developing critical systems. Praxis recognized that elements of requirements engineering permeate throughout the development life cycle, because requirements evolve and change as the system is developed. For this reason, requirements capture (which includes both revision of existing requirements and capture of new requirements) must be integrated with key design decisions and tradeoffs, and safety and security analyses. REVEAL provides a methodology to accomplish this, and its activities are integrated with project management and systems engineering activities.

G.2.1. Requirements for Security Properties and Attributes versus Security Functions

Functional security requirements are by and large, though not exclusively, constraints on how the software's functionality is allowed to operate. For example, a software security requirement correlating to the functional requirement "the software shall accept the user's input" might be "the software shall authenticate the user before accepting the user's input". Similarly, the security requirement correlating to the functional requirement "the software shall transmit the data over a network socket" could be "the software shall call its encryption process to encrypt the data before transmitting that data over a network socket".

Additional security requirements include the administration of user security data. In modern enterprise architectures, data must be shared among a collection of systems-requiring organizations to manage user authorization and authentication in a consistent fashion across multiple and distributed systems. An important security requirement is the amount of time it takes to add, remover, or update an employee's information across all systems within the enterprise.

Security properties and attributes, by contrast, are either constraints on the behavior of the software as it performs its functions, e.g., "the software shall not write to memory any input that has not passed the software's input

validation function"); or they may be more general constraints on the software's design or implementation characteristics, e.g., "the software shall contain no functions other than those explicitly specified", or "any unspecified function present in the system must be completely isolated and contained so that it cannot be inadvertently executed", or "the software will contain no covert channels".

> *NOTE: Because it would likely prove unrealistic to verify satisfaction of the requirement that "the software shall contain no functions other than those explicitly specified" in software has integrated or assembled from acquired or reused components, for such software it would be more practical to specify "the software must perform no functions other than those explicitly specified".*

Most requirements for software security combine security functions and security properties or attributes. For example, the nonfunctional requirement that "the software system must not be vulnerable to malicious code insertions" could be coupled with a functional security requirement that "the software system shall validate all input from untrusted users before writing the input to memory"—a measure that would go a long way towards preventing cross-site scripting attacks and buffer overflows that result in overflow of malicious code from a memory buffer onto the system's execution stack.

G.2.2. Analysis of Software Security Requirements

Analysis of the requirements specification should include:

1. **Internal analysis,** or verification, to determine whether the requirements for the software's security properties and attributes are complete, correct, and consistent with the other security requirements, and with the functional, safety, performance, etc., requirements of the software. Obviously, the need for requirements verification is not limited to security requirements.

2. **External analysis,** or validation, to answer the following questions:

    a. Do the software's security requirements adequately reflect applicable stakeholder sources (e.g., laws, standards, guidelines, and best practices)?

    b. Are the security requirements a valid refinement of the system security goals? Are there any software security requirements that are in conflict with the system's security goals?

The objective of the requirements analysis is to produce a set of reconciled security requirements for the software that will then be added to the software requirements specification, and captured in its requirements traceability matrix; requirements "discarded" during the conflict resolution should be documented to explain to stakeholders why the requirements were omitted. The requirements traceability matrix should be augmented with a dependency matrix that illustrates all associations between the software's security requirements and its other requirements.

The review team for the requirements specification should include a security analyst/engineer to ensure that all necessary security requirements are captured and documented clearly, paying special attention to capturing nonfunctional requirements for the software's security properties and attributes. These may include a requirement that the software must be able to resist entering any state that will leave it vulnerable to attack, with associated functional requirements, such as the software must validate the length and format of all input to ensure that the input conforms with required ranges and values, the software must perform specific functions when it detects patterns associated with specific exceptions and errors, the software must perform an orderly shut down if it cannot isolate itself from the source of bad input, etc. To aid in requirements analysis, the NASA Software Assurance Technology Center developed the ARM tool (http://satc.gsfc.nasa.gov/tools/arm/) that aids in assessing the quality of a requirements specification.

Before the requirements analysis begins, the requirements specification to be reviewed should be checked into the Configuration Management (CM) system and designated as the "baseline". Any modifications required as a result of review findings can then be applied to the baseline. This approach will minimize the likelihood of a rogue

developer being able to delete requirements or insert malicious requirements into the specification, since it will be easy to compare the pre-analysis baseline specification against the post-analysis revision.

The security requirements for the software should emerge from the results of the initial risk analysis (including threat modeling and vulnerability assessment) of the software's initial functional and interface requirements and Concept of Operations (CONOPS). Additional security requirements may be identified, and existing requirements modified, based on iterative risk analyses later in the software development life cycle.

The software's security requirements will act as constraints on its design. Risk management throughout the remainder of the life cycle will focus on iteratively re-verifying and adjusting the understanding of the baseline risk analysis, and on verifying the adequacy, in terms of minimizing assessed risk to an acceptable level, of the emerging software system's security properties and attributes and associated functionalities. During unit, integration, and system testing, the software's implementation (individual components, system subsets, and whole system) should be checked against its relevant security requirements to verify that the implementation satisfies those requirements and also that it adequately mitigates risk. Automated checks can be useful for determining whether requirements are internally consistent and complete in achieving the software's security objectives (e.g., to comply with security policy, to counter likely threats, to adequately mitigate known risks).

For high assurance and high confidence software, more comprehensive analyses of security requirements, including inspections and peer reviews, can more accurately assess whether the desired security properties/attributes have been captured clearly, and in terms that can be easily translated into a design and implementation. Using an executable specification language to capture security requirements can enable the analyst to exercise those requirements, ensuring that they both capture their intent and reflect reality.

G.3. Architecture and Design of Secure Software

Once the requirements specification has been analyzed to determine the security properties and attributes that the software must possess and the associated security functions that it must perform, the secure system architecture should be developed to reveal and depict any overlaps of security properties/attributes among individual components, and to capture "fall backs" in the case of a compromise of any of a given component's security properties, attributes, or functions that are relied on by other components.

The software architecture should include countermeasures to compensate for vulnerabilities or inadequate assurance in individual components or inter-component interfaces. For example, the architecture for a software system integrated from acquired or reused components may include a wrapper to filter excessively long input on behalf of a component written in C or C++ that does not perform its own range checking.

A secure architecture and high-level design are crucial to the correct detailed specification of the means by which the software will implement its required security functions and manifest its required security properties and attributes. An omission or error in the software architecture that is iterated into its detailed design and implementation may introduce vulnerabilities that will be both more difficult to assess after the software has been implemented, and more expensive to resolve during the later life cycle phases.

G.3.1. Security Modeling

Security models are abstractions that capture, clearly and simply, the security properties and attributes and risk areas of the software. These models enable the developer to experiment with different variations and combinations of component security properties, attributes, and functions, to assess their comparative effectiveness in addressing risk. By organizing and decomposing the whole software system into manageable, comprehensible parts, security modeling can help the developer pinpoint and specify design patterns for software's security functions.

Modeling frameworks can help capture the architectural patterns that specify the structure and security-relevant behaviors of the whole software system. Collaboration frameworks are similarly helpful for capturing the design patterns that specify sets of abstractions that work together to produce common security-relevant behaviors.

Specifically, software security modeling enables the developer to:

- Anticipate the security-relevant behaviors of the individual software components, behaviors related to the interactions among components, behaviors of the whole system as it interacts with external application-level and environment-level entities.
- Identify software functions that may need to be modified or constrained, and inter-component interfaces that may need to be filtered or protected.
- Detect errors and omissions in the assumptions that informed the software requirements, architecture, and design to make necessary adjustments to the design before implementation begins;
- Identify known vulnerabilities and failure modes in the software and architectural-level countermeasures to ensure that the most vulnerable components cannot be compromised or exploited;
- Verify the security properties and attributes of all components;
- Identify conflicts between any component's expectations of the security properties, attributes, or services of any other component, and try out alternative integration/assembly options to eliminate those conflicts;
- Identify conflicts between the individual components' assumptions and the whole system's assumptions about the execution environment security properties, attributes, services, and protections, again enabling the exercise of alternative integration/assembly options, and the identification of needed countermeasures to mitigate and constrain the impacts of irresolvable conflicts;
- Analyze the security implications of different architectural, design, and implementation decisions for software in development;
- Analyze the security implications of new or changed stakeholder requirements or threat models for software in development or deployment;
- Analyze the security impacts of software updates (including patches and new versions) or substitutions for software in deployment;
- Determine the distribution and interactions of the security functions within the software system, and the interactions of those functions with entities external to the system, with the main objective being to reveal any conflicts between the "as desired" (architectural) security functions the and "as implemented" security functions so as to make necessary adjustments to the software risk analysis.
- For software that will be assembled or integrated from components, verify that the individual components behave consistently and ensure that certain end-to-end security properties and attributes are consistently demonstrated by the components when they interoperate within the integrated software system. One critical property, for example, is the conformance of the whole system's operation with governing security policy. Another is the ability of the system to resist or withstand then recover from attacks.

The software's security models should be revalidated at each phase of the life cycle to account for changes iterated back into an earlier life cycle phase's artifacts necessitated by discoveries made during a subsequent phase. For example, changes may be needed in the software architecture to reflect adjustments to the detailed design made during the implementation phase to address difficulties implementing the original design.

Various modeling methods can be used to reveal component and whole-system security properties and attributes, and to provide the basis for comparing the attack surfaces of similar components under evaluation. Some modeling methods that can be adapted to security modeling were discussed in Section 4.3.3.

G.3.2. Security in the Detailed Design

Design decisions require tradeoffs between security and other desirable attributes such as usability, reliability, interoperability, performance, backward compatibility, cost-effectiveness, etc. A design that minimizes the software's exposure to external threats may also hinder required interactions between the software and any external services or users. Designing software to be secure sometimes seems like an exercise in managing complexity. In fact, most software security faults have causes that are simple, straightforward, well understood, and easy to prevent in the software's design and to detect during testing. Designing for security means designing software that is able to:

- Proactively detect and prevent attacks that may lead to compromises;
- Reactively minimize (isolate and constrain) the impacts of attacks that do succeed, and terminate or block access by their suspected causing actor to the software.

Mechanisms such as fail-safe design, self-testing, exception handling, warnings to administrators or users, and self-reconfigurations should be designed into the software itself, while execution environment security mechanisms such as application-level firewalls, intrusion detection systems, virtual machines, and security kernels can add a layer of "defense in depth" by filtering or blocking input to the software, monitoring network traffic for relevant attack patterns, and performing other resistance and resilience functions in aid of the software's own security.

Increasingly, security mechanisms are designed as self-contained components or subsystems external to and called by software systems. The compartmentalization of design, with the software's security logic separated from its other functional logic simplifies both the development of the software as a whole, and the assurance verification of its security functions.

The detailed design specification should include a detailed description of how each requirement pertaining to a security property or attribute will be accomplished through the combination of associated security functions (including control functions and protection functions), what the interfaces of those functions are, and what inputs and outputs they must process. For example, a requirement that the software not be able to enter a vulnerable state would be expressed in the design by including a function to perform input validations check input to ensure it contains no patterns or errors that could trigger a fault (such as a buffer overflow) that would force the software to enter an insecure state.

Once completed, the security architecture and design specification should be analyzed to confirm that they satisfy the specified security requirements. Ensuring that the security architecture and design map directly to the software's security requirements specification will make it easier to identify exploitable faults and insecure behaviors early in the life cycle, and will facilitate proactive changes and tradeoffs at the architecture or design level that can prevent, or at least minimize and constrain the impact of, the exploitation of those faults and insecure behaviors. Addressing such problems early in the life cycle is far less expensive than waiting to observe compromises caused by simulated attacks against the system during penetration and fault injection testing or post-deployment vulnerability scanning, at which point they can be addressed only reactively, through patching, inelegant workarounds, or disabling of the insecure portions of the software until the faults can be remedied.

If the software is to be deployed in multiple execution environments with different robustness levels, the detailed design should be flexible enough to allow for the system to be reconfigured and, if necessary, re-implemented to accommodate unavoidable substitutions, exclusions, or (in some environments) additions of environment-level security mechanisms and protections relied on by the software for defense in depth. In addition, the software's architecture and design should include enough self-protecting characteristics to compensate for lack of

environment-level protections in less robust execution environments, to maintain at least an acceptable minimum level of attack resilience and attack resistance in the software itself.

In environments in which security policies change frequently, or in which the software may need to support multiple security policies, the design should delay the binding of security decisions, whenever possible implementing those decisions as installation-time or runtime configuration options, rather than "hard-wiring" them in the software's source code. This will contribute to the goal of making the system more extensible. While extensibility is necessary, it also increases risk, particularly when extensibility is preserved in the software after it has been deployed. Extensibility of software in deployment provides attackers with built-in hooks and mechanisms to change the behavior of the software system. (Firefox extensions and browser plug-ins are good examples of software enabled by built-in extensibility features in deployed software, in this case browser applications.)

The system's design specification should include a set of hierarchical diagrams and possibly pseudocode (at least for from-scratch software) depicting the modules that will implement the security functions that will assure its required security properties and attributes. This information should be comprehensive and sufficiently detailed so as to enable skilled developers to implement these functions with minimal additional guidance.

If the software is to be implemented through assembly or integration, all of the different candidate design options for the system's components should be documented in the design document. Then these design options should be exercised as part of the selection process for the acquired or reused components, to determine which specific set of components in which particular configuration best satisfies the whole-system requirements (including the security requirements). The aspects of the design that will be implemented using acquired or reused software then form the basis for defining the evaluation criteria (including security evaluation criteria) used in selecting those components.

Before submitting it for review, the design specification should undergo a thorough risk analysis by a security analyst on the design review team. The risk analysis should determine whether the security aspects of the design are coherent, and whether all of the software's specified security property/attribute and associated functional requirements have been satisfied. The requirements traceability matrix should be updated at this phase to reflect any changes in the security elements of the system that may have emerged during the design review.

Before each design review begins, design documentation and pseudocode to be reviewed should be checked into the CM system and designated as the "baseline". Any modifications required as a result of review findings can then be applied to the baseline. This approach will minimize the likelihood of a rogue developer being able to incorporate intentional faults into the design, since it will be easy to compare the baseline design against the modified design. See Appendix G:G.9 for more information on secure configuration management

The following sections describe some specific principles that should be considered to achieve a more secure software design.

G.3.2.1. Isolating Trusted Functions

Trusted functions should be placed in separate modules that are simple, precise, and verified to operate correctly. Such modules—even small ad hoc scripts—should not rely on any global state and should minimize any architectural dependencies; such reliance increases complexity and reduces the ability to easily trace the trusted module's flow of control. Only precisely defined interfaces should be used for communication between trusted and untrusted modules. These interfaces should not directly reference any internal variables.

G.3.2.2. Avoiding High-Risk Services, Protocols, and Technologies

Services, protocols, and technologies (as well as specific products) that are frequently the topic of vulnerability and bug reports, newsgroups, mailing lists, blogs, etc., should never be used in a software system without security countermeasures that can be positively verified to constrain or prevent the risky behaviors of those services/protocols/technologies.

The truism "where there's smoke, there's fire" is particularly true for software with vulnerabilities that are frequently publicized. Even if the software's supplier is able to (seemingly) produce a continuous stream of security patches for that software, unless absolutely no more secure alternative exists, why would any developer or software project manager want to take on the triple headache of intensive patch management, risk management, and configuration control that such insecure software is going to require throughout the lifetime of the system in which it is used?

In those instances in which a high-risk service, protocol, or component is absolutely required and no lower-risk alternative exists, the system must be designed to limit the potential impact on the system's security if (when) the insecure service/protocol/technology misbehaves or is compromised. Noninterference, containment, and filtering mechanisms such as security wrappers, constrained execution environments, and application firewalls are some of the countermeasures that may prove effective for this purpose.

The risks of mobile code and mobile agent technologies are particularly nebulous at this stage. Both industry best practices and many government agencies' security policies limit developers to using only "approved" mobile code technologies. These approved technologies are generally those that introduce only a low level of risk to the software and its environment, often because they run within a virtual machine that constrains their behavior. Even lower-risk mobile code technologies are often required to be used only in conjunction with code signatures, i.e., the mobile code program (e.g., Java applet) is digitally signed to indicate that it originated from a trusted source. The recipient of the code validates the code signature, and if the source is recognized and trusted, the code is executed; otherwise it is discarded.

Unfortunately, code signing is not yet universally supported. Nor can it prevent a determined attacker from executing malicious or insecure mobile code; the attacker can simply subvert the code signature process, then sign the mobile code that contains the malicious or insecure logic.

G.3.2.3. Defining Simple User Interfaces to Sensitive Functions

User interfaces to all functions should always be as simple and intuitive as possible, but such simplicity and intuitiveness is critical for user interfaces to security functions and other trusted functions, such as those that implement user identification and authorization, user registration, user-initiated digital signature or encryption, or user configuration of system's security parameters. Intuitive, easy to use interfaces will reduce the likelihood that users will make mistakes in using these security/trusted functions.

Poorly designed user interfaces make it easier for the user to choose an insecure configuration default than to set up secure configuration parameters. Poorly designed interfaces increase the likelihood that users will try to shut off or bypass security functions. For this reason, all user interfaces to trusted and security functions should be non-bypassable, to prevent resourceful users from easily "getting around" the user interface to directly access sensitive data used or updated by the system, or to directly invoke the system's trusted functions (e.g., via a system shell or other command line interface).

User interfaces that accept data, such as HyperText Markup Language (HTML) or eXtensible HyperText Markup Language (XHTML) forms, should be designed to limit the size and format of data that a user can enter, especially if the system receiving that data includes C or C++ code or runtime libraries. This is true especially if the code/libraries have not been verified to be free of buffer overflow vulnerabilities. Pre-validation of input on

the client can help guide the user in entering correctly formatted, bounded data, and in filtering out potentially overflow-inducing input.

The trouble is that client-side input validation can be easily bypassed by a resourceful user, and even valid client-originated data may be tampered with in transit to the server. For this reason, client-side input validation does not override the need for thorough server-side input validation. Client-side input validation should be seen as having one very specific goal—to help the user correctly use the interface while possibly reducing server workload in response to erroneous data. However, all server-side interfaces must be designed and implemented as if there was no client-side input validation whatsoever.

The user interfaces of the system's trusted and security functions should be prototyped early in the development life cycle, and that prototype demonstrated to the system's intended users, ideally with their hands-on participation. This demonstration serves both to educate the users in what they will or will not be allowed to do when they use the system, and to get their feedback on how intrusive, confusing, or difficult they may find the interfaces to be.

The results of this demonstration should be compared against the requirements for the interfaces, to determine whether any of the users' negative feedback reflects a failure of the interface in question to satisfy its specification. Analysis of the demonstration findings should also determine whether the interfaces encourage appropriate user behaviors, and should ensure that they prevent unauthorized behaviors.

For any acquired or reused software that performs a security/trusted function or sets up the system's security configuration, the intuitiveness and ease of use of the user interfaces should be an evaluation criterion.

G.3.2.4. Eliminating Unnecessary Functionality

Software systems, including those assembled from acquired or reused software, should contain only those functions they require to accomplish their processing objectives. Similarly, the runtime environment should contain only those software libraries, routines, etc., that are explicitly called by the system.

Even function, routine, or library that "could prove useful at some later date" should be eliminated until that later date arrives. Such elements too often provide attackers with exploitable entry points into the system. Also, because "dormant" functions are not expected to be used, the administrator who configures the system's logging/auditing or intrusion detection may overlook them, enabling an attacker who does manage to exploit them to go undetected. There are a tools available that help the developer navigate through source code trees to detect dormant functions that are never invoked during execution.

For acquired or reused software, eliminating unused functions will require either removing that functionality (this may be a possibility with open source software), or completely disabling or isolating the "dormant code" to prevent the unused functions from being either intentionally invoked or accidentally triggered by users of or processes that interface with the software product within the assembled/integrated system.

However, when a function in a commercial or open source is not used in the system, it still should be assiduously patched and maintained. Even if it affects only a "dormant" function, every patch/update should be applied: otherwise, if the function is activated later, it will still contain avoidable vulnerabilities.

G.3.2.5. Designing for Availability

Several software fault tolerance methods have been proven to improve program robustness and may also prove helpful in improving program resistance to intentional denial of service (DoS) attacks. These methods include:

- **Simplicity:** Even if avoiding complexity in individual software components is not practical, simplicity can be achieved at a higher level. The design option used to integrate/assemble the individual components into a whole software system, i.e., the system's architecture and design, should explicitly minimize

complexity.

Each component should have a single input path as few output paths as possible. Acquired or reused components should be configured with all features/functions not used in the system disabled (or "turned off"). In addition, the interfaces that trigger the execution of a specific function within the component should never cause the component to behave insecurely or unsafely.

Complexity of a software system goes beyond simply the interfaces between components. For a large software-intensive system, there may be complex dependencies among subsystems, potential feedback loops among components, and contention among components over shared resources. In such complex systems, it becomes increasingly difficult to identify the exact source of an error, which may be necessary to determine an appropriate response. The design of the software-intensive system as a whole should minimize such dependencies.

- **Graceful degradation:** In many cases, to remain available in the face of a DoS attack, software will need to alter its operation to a degraded state. Graceful degradation ensures that when an intentional fault condition occurs, the software does' not immediately shut down (or crash), but instead continues to operate in a reduced or restricted manner. In the case of critical or trusted functions, individual degradation or shut down may not be acceptable. In such cases, the orderly shutdown of the whole system should be initiated as soon as a fault in one of those functions is detected.

- **Redundancy:** Availability of a critical component is most often achieved by deploying redundant copies or instantiations of that component on different physical hosts. This helps ensure that an attack on a given copy/instantiation of on one host will not impact availability of the component to the software system as a whole, and thus the availability of the software itself.

  Each critical software component should be designed so that if it is subjected to a DoS attack, it will implement automatic fail-over to a redundant copy/instantiation of the same component. In Web applications and other session-oriented software programs, assured session fail-over is another factor that must be considered. Web applications can be designed to prevent users from experiencing transient data loss by using stateless rather than stateful data objects.

  Critical software components, which are identified using risk analysis, should be designed so that neither intentional performance degradation nor automated fail-over will cause the loss of user sessions active at the time of degradation or fail-over. In addition, software should be designed to reject all *new* user session requests during such failures. Session fail-over must be designed into the software, ensuring that both the user's session and transient data are preserved, so the user experiences no interruption in service.

- **Diversity:** Reliance on redundancy alone, while it may be a good strategy for continuity of operations, will not achieve robustness against intentional failures for the same reason that simply breeding more hemophiliacs will not increase the amount of clotting factor in any of their bloodstreams. Unless the component that is multiplied to create redundancy is also inherently robust against DoS, the attacker (human or worm code) will simply be able to successfully target the backup components using the same attack strategy.

  Diversity combined with redundancy helps ensure that the backup or "switchover" component to which the software system fails over cannot be successfully attacked using the same attack strategy that brought down the failed component. Diversity can be achieved through practices such as *n*-version programming, reliance on standard interfaces rather than particular technologies, so that the software's components can distributed across different host platforms, and implementation of interface abstraction layers that make it easier to host multiple copies of the software on different platforms.

Diversity is not without drawbacks: while not all components or systems in a diverse architecture will be susceptible to a single attack, there will likely be a wider variety of exploitable faults in the system as a whole. This gives attackers more opportunities to discover the faults and take advantage of them. However, each fault will only affect a single system or component, rather than the whole network. As with all security strategies, some risk analysis should be used when deciding how to implement diversity.

G.3.2.6. Designing for Assurance

The choice of software, interface techniques, and protection techniques used in the design to satisfy the security requirements for the system must reflect the required level of assurance for assembly technique as demonstrated as it operates within the system. The common practice of using lower assurance software and techniques in software systems that are subject to higher level risks creates a disconnect that will leave the system vulnerable. The appropriate approach is to use software and techniques that provide a level of assurance commensurate to the level of risk to which the system will be exposed in production.

The design can attempt to increase the assurance of the system overall, despite lower assurance in the individual software products, by augmenting the techniques in those software products with protections that result from the way in which the products are assembled, as well as using isolation and constraint techniques (e.g., security zones, sandboxing) and filtering mechanisms (e.g., wrappers). However, if these "augmentations" fail to eliminate the vulnerabilities that arise from disconnects between software product assurance and system risk, it may be necessary to replace the lower assurance products and techniques with higher assurance alternatives (which may be more expensive). In software requiring high assurance, more components will probably need to be developed from scratch to avoid vulnerabilities in acquired or reused components.

G.3.2.7. Designing by Contract

"Design by Contract" (DbC) enables the designer to express and enforce a contract between a piece of code ("callee") and its caller. This contract specifies what the callee expects and what the caller can expect, for example, about what inputs will be passed to a method or what conditions that a particular class or method should always satisfy. DbC tools usually require the developer to incorporate contract information into comment tags, then to instrument the code with a special compiler to create assertion-like expressions out of the contract keywords. When the instrumented code is run, contract violations are typically sent to a monitor or logged to a file. The degree of program interference varies. The developer can often choose a range of monitoring options, including:

1. Non-intrusive monitoring by which problems are reported, but execution is not affected;
2. Throwing an exception when a contract is violated;
3. Performance of user-defined action in response to a contract violation.

DbC can enforce security boundaries by ensuring that a software program never accepts inputs known to lead to security problems, or never enters a state known to compromise security. The developer can start creating an infrastructure that provides these safeguards by performing unit testing to determine which inputs and conditions would make the software program vulnerable to security breaches, then write contracts that explicitly forbid these inputs and conditions. The developer then configures program so that whenever the conditions specified in the contract are not satisfied, the code fires an exception and the requested action (for example, a method invocation) is not allowed to occur. When this infrastructure is developed after thorough unit testing, it provides a very effective last layer of defense.

G.3.3. Systems Engineering Approaches to Protecting Software in Deployment

Defense in depth countermeasures external to the software that they are intended to protect (e.g., by blocking attack patterns or otherwise counteracting frequently-targeted vulnerabilities in that software) are rapidly

becoming less and less effective. This is because attackers are becoming increasingly knowledgeable about the inner workings of these high-value targets, and thus are able to craft increasingly more effective techniques to bypass or subvert them.

Furthermore, reliance on external countermeasures is predicated on the ability to trust that those countermeasures will remain dependable when they themselves are targeted by attackers. This means trusting that the developers of those countermeasures did their job correctly (and securely), and the installers also. A successfully targeted vulnerability in a "trusted" defense in depth countermeasure could severely compromise not only the countermeasure, but the security of any software that relied on it for protection.

These concerns reinforce the importance of developing software that is secure in and of itself. As Hurricane Katrina so tragically taught us, no matter how high one builds a levee, it can never completely guarantee that the houses below sea level that rely on that levee to keep them dry will be as safe as houses that were built on higher ground. Software that contains no exploitable weaknesses cannot be compromised by attacks that target those vulnerabilities. Secure software should never rely predominantly, let alone solely, on external countermeasures to protect it. Instead, such countermeasures should be limited to providing defense in depth (1) to counteract new threats (or newly discovered vulnerabilities) until the software can be patched (short term) and reengineered (longer term) to be robust against them; (2) when software is deployed in a higher-risk operating environment than it was designed to run in.

The remainder of this section describes some defense in depth countermeasures that can be used to help protect software that is not adequately robust.

G.3.3.1. Secure Operating Platforms

Secure operating platforms are not really "development tools" but rather varieties of operating system-level, kernel-level, or hardware-level protections, enhancements, or variants that use various approaches to either isolate untrustworthy software from trusted software running on the system and/or to protect all software on the system from attacks by outsiders.

Some of these technologies are still in the research phases. The examples given below are all actually available, either commercially or open source:

- **Kernel-level protections:** Address specific vulnerabilities, such as buffer overflow vulnerabilities, by adding security protections to memory/page management, or preventing stack execution.

- **Hybrid kernel- and processor-level protections:** Address buffer overflow vulnerabilities by enabling security protections for memory page management that prevent stack and heap execution. Because of incompatibility with some applications, some operating systems allow users or applications to disable hardware buffer overflow protection; ideally, it should be always enabled. While server processors and operating systems have provided hardware protection against buffer overflows for several years, consumer-level processors have begun shipping with this protection as well—which will reduce the effectiveness of buffer overflow attacks in the future.

- **Security-enhanced operating systems, operating system security-enhancements:** Add security features to existing operating systems to address various vulnerabilities and/or add security functionality.

- **Hardened operating systems:** Intended as platforms for highly sensitive software systems and security applications, e.g., firewalls, intrusion detection systems, virtual private network (VPN) servers. The intent is similar to the minimized kernels described below, i.e., to disable, securely configure, or remove problematic and unnecessary services and resources that could be exploited by attackers.

- **"Minimized" kernels and microkernels:** Modified versions of existing operating systems from which problematic and unnecessary features have been removed to produce a small, well-behaved environment that provides only the minimum core set of services and resources needed by the software systems that run on them. Developers who write for these kernels need to be fully aware of what services and resources are missing, to ensure that their software does not rely on those services. Determining whether an acquired or reused component can run on one of these systems can be a challenge.

  An emerging type of microkernel is the secure microkernel, also known as "separation kernel", "secure hypervisor", or "secure μkernel". Further information on this technology appears in Appendix I.

- **Trusted operating systems:** Built from scratch to implement a trusted computing base, mandatory access control policy, and a reference monitor that enables stronger isolation of processes and data stored at different mandatory access levels, and strongly constrains accesses and interactions among those entities.

- **Trusted hardware modules:** Similar in intent to virtual machines (VMs) and sandboxes, provides even stronger hardware-enforced isolation of processes, in terms of their interactions with other processes, and their access to data. Technologies produced by the U.S. DoD's Anti-Tamper Program and the commercial Trusted Platform Modules (TPMs) from various vendors are examples.

In Unix and Linux systems in which no VM or application framework is being used, a constrained execution environment can be achieved by implementing a "*chroot* jail" in which the program to be constrained is executed.

G.3.3.2. Secure Application Frameworks

As with constrained execution environments, application development frameworks provide execution environment security to software that runs within those frameworks. Unlike constrained execution environments, however, these frameworks also provide software libraries, prepackaged applications, and other resources to help the developer by providing standard implementations of functionality that many software systems have in common.

Use of a framework enables developers to avoid having to custom-develop protocol implementations and other functions, for example functions used in all Web services, such as XML parsing and service registration. This enables the developer to focus instead on implementing the functions that are unique to the application being developed. Secure application frameworks provide to both trusted and untrusted applications a set of security functions and services, security protections, and execution environment access controls that enable those applications to:

1. Execute with confidence that their integrity and the integrity of their resources (both at the individual component and whole-system levels) will be protected. In Java EE the integrity-assurance mechanism is the JVM. In .NET, the integrity-assurance mechanism is Code Security.

2. Interoperate with a set of predefined security services and interfaces to those services that are provided to programs that execute in the framework, thus eliminating the need for custom-development those security services and interfaces, or the need to acquire them from suppliers and integrate/assemble them.

Application frameworks include software libraries (or APIs to libraries) that implement standard protocols, and provide the logic of and APIs to other common functions, including common security functions such as authentication and authorization, SSL/TLS, HTTPS, cryptography, RBAC, etc. The frameworks also provide development tools to assist developers in implementing applications to run in those frameworks.

Several well-known application frameworks implement security models that can, to some extent, provide a basis for building secure distributed applications. In reality, however, only the extended implementations of Java EE

platforms (such as IBM Websphere, Oracle 9i Application Server, and others) provide a rich enough set of security services to be leveraged effectively by applications.

A middleware framework can minimize the likelihood of security problems being introduced through improper integration of application components. Both Java EE and .NET use Java as their "managed code architecture" to control the behavior of client and server application code that runs within the framework. Unlike Java EE, which retains Java's essential portability, .NET is a proprietary framework that runs only on Microsoft Windows operating systems (though some open source and commercial .NET implementations for other platforms are emerging). .NET components provide little or no interoperability to application components running on other operating systems/in other frameworks.

However, as application systems become increasingly loosely coupled with the adoption of the SOA model, the dependency on application frameworks such as .NET and Java Enterprise Edition (Java EE) for the application's security model will prevent developers from being able to make assumptions about the platform that a particular called service might be running on. Software may be developed to run on Java EE, but the security model implied by that design choice may be rendered irrelevant if the application becomes a component of a larger multi-platform distributed system. To this end, IBM, Microsoft, and VeriSign have developed the WS-* suite of Web Services protocols. These protocols provide mechanisms that enable Java EE, .NET, and other SOA frameworks to support security policies and trust frameworks separate from the application framework's underlying security model.

G.3.3.3. Content Filtering Appliances

- **Application firewalls:** Application-level firewalls extend the network firewall model up to the application layer, to detect and block common Web-based attacks, such as cross-site scripting and SQL injection. Commercial and open source products come in both software-only and hardware/software "appliance" forms.

- **eXtensible Markup Language (XML) firewalls and security gateways:** XML firewalls (increasingly referred to as XML security gateways) protect against exploitation of XML vulnerabilities by acting as an XML proxy to perform such checks and filtering functions as well-formedness checks, buffer overrun checks, schema validations, content filtering, and denial of service protection. XML security gateways augment firewall features with access control and secure routing functions. As with application firewalls, can be software-only, or "appliance".

G.3.3.4. Browser Integrity Plug-Ins

Browser integrity plug-ins are software components that can be linked into a browser to enable the browser to provide integrity protections to the browser program itself (versus security plug-ins that implement or interface to the security functions to be performed from within the browser). In general, such plug-ins augment the browser's constrained execution environment for mobile code by providing functionality for code signature validation, or by performing client-side input validation.

G.4. Implementing Secure Software

For software to be secure it must avoid faults in its implementation that introduce vulnerabilities regardless of whether the majority of development involves either from-scratch coding or integration/assembly of acquired or reused software components. For software built from scratch , achievement of security in the implementation phase will focus mainly on avoiding faults that manifest as vulnerabilities. Security in implementation of software systems assembled or integrated from acquired or reused components will focus on implementing countermeasures and constraints to deal with known vulnerabilities in the individual components and their "plug-and-play" interfaces.

G.4.1. Using Implementation Tools to Enhance Software Security

Development tools alone cannot guarantee that the software they are used to produce will be secure. Developers who have a true and profound understanding of the security implications of their development choices will be more successful in writing secure software in C, C++, Visual Basic, JavaScript, or AJAX using general-purpose development tools (e.g., tools that enforce good software engineering, editors that highlight and require correction of errors in the code as it is being written) than will security-ignorant developers who use only "safe" languages and arsenals of "secure" development tools.

This said, when a developer is security-aware, he/she will not only use practices that are consistent with that awareness, but will choose tools that help enforce those secure software conceptualization and implementation practices. Used with security in mind, the categories of implementation tools described in this section should help developers achieve at least one of the following objectives:

- Reduce the presence or exposure of exploitable faults and weaknesses in the software;
- Implement in-deployment software security countermeasures;
- Constrain the extent of insecure software behaviors so they do' not affect other software, data, or environment components.

NOTE: Tools that support earlier life cycle activities (specification, design) were discussed at relevant points in Sections 3 and 4.

G.4.1.1. Secure Use of Standard Compilers and Debuggers

The level of type checking for C and C++ can be increased by turning on as many compilation flags as possible when compiling code for debugging, then revising the source code to compile cleanly with those flags. In addition, strict use of American National Standards Institute (ANSI) prototypes in separate header files will ensure that all function calls use the correct types. Source code should never be compiled with debugging options when compiling and linking the production binary executable. For one thing, some popular commercial operating systems have been reported to contain critical vulnerabilities that enable an attacker to exploit the operating system's standard, documented debug interface. This interface, designed to give the developer control of the program during testing, remains accessible in production systems, and has been exploited by attackers to gain control of programs accessed over the network to elevate the attacker's privileges to that of the debugger program.

Many C/C++ compilers can detect inaccurate format strings. For example, GCC supports a C extension that can be used to mark functions that may contain inaccurate format strings, and the /GS compiler switch in Microsoft's Visual C++ .NET can be used to flag buffer overflows in runtime code. Many other compilers offer similar facilities.

G.4.1.2. "Safe" Programming Languages

A programming language that supports good coding practices and has few inherent vulnerabilities is more likely to be used securely than a language with critical security faults or deficiencies. C and C++ are more difficult to use securely than Java, Perl, Python, C# and other languages that have embedded security-enhancing features such as built-in bounds checking, "taint mode", and in some cases their own security model (e.g., the JVM, the C# Common Language Runtime [CLR]).

For software that is not performance-critical, the performance advantages of C/C++ should be weighed against the potential for buffer overflow risks. Avoiding buffer overflow is not even remotely the only concern for programmers. It is quite possible to write insecurely in languages with built in bounds checking, taint mode, and their own security model. In particular, input validation should be performed regardless of the language in which the system is written. While C and C++ are notoriously prone to buffer overflows and format string attacks,

software written in other languages may be susceptible to parameter tampering, command injection, cross-site scripting, SQL injection and other compromises that exploit user input to the system. Regardless of the language used, all user input (including input from untrusted processes) should be validated.

All commands and functions known to contain exploitable vulnerabilities or otherwise unsafe logic should be avoided. None of the obscure, unfamiliar features of a language should be used unless (1) those features are carefully researched to ensure the developer understands all of their security implications; (2) the required functionality cannot be achieved in any other way.

Shell scripting languages, device languages, etc., should never be used in application-level software. "Escaping to shell" from within application-level software creates an interface that is much sought after by attackers, because it is an interface that gives the attacker a direct path to system-level functions, files, and resources. Instead of embedding shell script, device command strings, etc., in the system, the developer should use trustworthy APIs to the required system or device functions, data, or resources. If the developer uses an add-on library or file input/output library, the portion of the software that uses that library should be compartmentalized, and all of the system's accesses to those libraries should be logged/audited.

Much research has been done to produce secure variants of C and C++ and other languages, and to define new programming languages that contain few if any vulnerable constructs. Java is probably the most successful example of a type-safe programming language that also prevents pointer manipulation and has built-in garbage collection. These features, along with Java's built in virtual machine (the Java Virtual Machine, or JVM) sandboxing mechanism for isolating the transformation and execution of untrusted byte code were conceived in part to ensure that Java would be free of the security deficiencies found in C and C++. Microsoft's C#, similarly, attempts to extend C++ concepts into a safer (and more modern) language structure. This said, Java and C# could potentially produce less secure code, because the intermediate production of byte code means that the developer is one step removed from the actual functioning of the object code, and has no visibility into memory allocation.

Languages designed with security as their main objective have not been particularly successful, and for the most part remain in the realms of research and academia. Examples of security-oriented languages include Hermes, CCured, SafeC, Fail-Safe C, Cyclone, Vault, E, and Oz-E.

G.4.1.3. "Safe" Compilers, Compiler Extensions, and Runtime Libraries

The focus of safe compilers and compiler extensions is on compile-time checking, sometimes with code optimization, to flag and eliminate code constructs and errors that have security implications (e.g., pointer and array access semantics that could generate memory access errors), and to perform bounds checking of memory references to detect and prevent buffer overflow vulnerabilities in stacks and (sometimes) heaps. The majority of safe compilers and compiler extensions are intended for use with C and C++, and focus exclusively on avoiding buffer overflows.

Compiler extensions that use stack canaries (a measure first introduced in StackGuard), as well as assembler preprocessors, can help reduce susceptibility of C and C++ programs to stack overflows. Heap overflows may be counteracted using a *malloc()* debugger.

Safe software routine libraries detect the presence at link time of calls by the software to unsafe runtime library functions (such as those known to be vulnerable to buffer overflow attacks). These unsafe *functionModes* are replaced with safe versions or alternatives. As with the safe compilers, most safe libraries are for C or C++ and focus on replacing library routines that are prone to buffer overflow. One of the first safe libraries was Libsafe.

There are compile-time and runtime tools that take different approaches, such as providing runtime protection against buffer overflows in binaries that are executed under a particular operating system, or that perform static or dynamic compile-time and runtime security analyses of source code and/or compiled binaries.

G.4.1.4. Code Obfuscators

Code obfuscators implement a software confidentiality technique that protects intermediate code (e.g., Java byte code) or runtime-interpreted source code (e.g., Perl, PHP, Python, JavaScript, AJAX scripting code) against decompilation and other forms of reverse engineering. Obfuscation may also be used to protect intellectual property, e.g., by preventing source code from being viewed and copied using a browser's "view source" function.

G.4.1.5. Content Filtering Programs

The intent of content filtering programs is to identify and remove, transform, or isolate input or output that is suspected of containing malicious content, such as worms or Trojan horses, "unsafe" constructs, such as buffer overflow inducing data strings, or commands strings that enable unauthorized users to illicitly escalate their privileges. Such filters may be implemented in the software during its development or may be applied as systems engineering defense in depth measures in the software's execution environment. "In development" content filtering programs include:

- **Security wrappers:** By and large, wrappers must be developed from scratch, though a few have emerged as open source offerings. Many are still in the realm of research.

- **Input validation filters:** Like security wrappers, input validation filters are still predominantly custom-built, with a few made available under open source licenses by academics.

"In deployment" content filtering "appliances" were discussed in Appendix G:G.3.3.3.

G.4.2. Coding from Scratch

> *NOTE: The authors have chosen the term "code from scratch" rather than "custom development", because in the commercial software industry, unlike government, "custom" implies "one-off". Some commercial developers refer to "coding from scratch" as "green field" development. What the term is meant to indicate is software that is written rather than reused or acquired (i.e., "developmental" rather than "non-developmental" software).*

Little if any application-level software is ever entirely coded from scratch. Even software that is truly "from scratch" is influenced by acquired and reused software "enablers", such as APIs, development tools, plug-ins, languages, libraries, utilities, configuration management tools, and execution environment software (middleware, operating systems, firmware). Security for software coded from scratch needs to focus both on secure coding and on protecting the from-scratch code from any negative security impacts imposed on it by the acquired or reused software to which it will be exposed throughout its development and deployment.

The remainder of this section describes specific practices for secure coding. These coding practices are mainly language-neutral, unless otherwise noted. This document does not address security issues associated with obscure or obsolete programming languages, or with assembler and hardware languages. With the few exceptions of language-specific practices, the techniques described in this document should help improve the security of code written from scratch, regardless of the language used.

G.4.2.1. Minimizing Size and Complexity, Increasing Traceability

Secure code is efficient. It includes all of its required functions, along with robust input validation and exception handling. It does not contain any unused functions. All unnecessary software should be removed from the source code base before compilation. The smaller and simpler the code base is, the easier it will be to assure the security of the software.

Secure code implements its functions in the smallest number of lines of code possible while maintaining readability and analyzability. Using multiple small, simple, single-function modules instead of one large, complex module that performs multiple functions will make the system easier to understand and document, thus making it easier to verify the security and correctness of the individual component and of the system as a whole. When used correctly, object inheritance, encapsulation, and polymorphism are all techniques that can greatly simplify code. All processes should be written with only one entry point and as few exit points as possible. To the extent possible, the system should be implemented with minimal interdependencies, so that any process module or component can be disabled when not needed, or replaced if found to be insecure or a better alternative is identified, without affecting the operation of system as a whole.

In code that implements security functions, trusted functions, critical functions, or otherwise sensitive functions the number of faults in the code can be significantly reduced by reducing the size of the software modules that implement those functions. Structured programming, avoidance of ambiguities and hidden assumptions, and avoidance of recursions and *GoTo* statements that blur the flow of control are effective techniques for achieving code simplicity and minimizing code size. Complex functions that would need to be implemented by large software modules should be analyzed and divided into multiple small, simple functions that can be implemented by small software modules.

G.4.2.2. Coding for Reuse and Maintainability

Secure code is often reusable. The features of code that make it reusable-simplicity, comprehensibility, traceability-are the same features that help make it secure. To achieve code "elegance", the developer should first write a comprehensive code specification in language that is clear and direct enough to enable another developer to take up coding where the first developer left off, or to maintain the code later. However, there may be mismatches in the design or architecture of the software and the reusable code that may prevent reuse.

Developers should never assume that their source code will be self-explanatory. All source code should be extensively commented and documented, reviewed, and tested to ensure that other developers and maintainers can gain a complete and accurate understanding of the code, which will enable them to reuse or modify it without introducing exploitable faults or weaknesses.

The main driver for reuse is the expense involved in custom (or bespoke) development of large "one-off" software systems. Unfortunately, reused software has a history of falling short of expectations. Secure reuse of source code is particularly challenging, because extensive security evaluations and reviews are needed to determine the suitability (in terms of security properties and attributes and secure interfaces and behaviors) of a particular component, not only in isolation, but when combined with components with which it was never designed to interact.

Most software that is reused was not specifically written to be reused. According to Butler Lampson in "Software Components: Only the Giants Survive," software that is most often reused does not come in the form of small, flexible source code modules; it comes in the form of large, complex binary executables. This is especially true of software reused in Web and database applications. The functionality and attributes of browsers, Web servers, and database management systems reflect a strictly defined, not-very-flexible set of usage and execution environment assumptions. Reusing such software in a way that satisfies the security property and attribute requirements of the new system into which it will be assembled/integrated can be very difficult when those property and attribute requirements conflict with the original assumptions under which the reused components were developed.

G.4.2.3. Using a Consistent Coding Style

A consistent coding style should be maintained throughout the system's code base, regardless of how many developers are involved in writing the code. Coding style includes the physical appearance of the code listing, i.e., indentation and line spacing. The physical appearance of the code should make it easy for other developers and

code reviewers to read and comprehend the code. The entire development team should follow the same coding style guide.

Coding style should be considered as an evaluation criterion for open source software, particularly for software that will implement trusted and high-consequence functions. Trusted software that will undergo code security reviews for C&A or other approval processes will be at an advantage if it exhibits a clear and consistent coding style.

G.4.2.4. Keeping Security in Mind when Choosing and Using Programming Languages

Refer to Appendix G:G.4.1.2.

G.4.2.5. Avoiding Common Logic Errors

Regardless of how careful a coding effort is, the resulting software program is virtually guaranteed to contain at least a few logic errors. Such bugs may result from design mistakes or deficiencies or from implementation errors (e.g., typos). Useful techniques for avoiding the most common logic errors include:

- **Input validation:** Input from users or untrusted processes should never be accepted by the system without first being validated to ensure the input contains no characteristics, or malicious code, that could corrupt the system or trigger a security exploit or compromise;

- **Compiler checks:** To ensure that correct language usage has been adhered to, and to flag "dangerous" language constructs, such as overflow-prone calls in C and C++.

- **Analysis to ensure conformance to specification:** For software systems that require high assurance, this should be a formal analysis, with particular attention paid to the system's conformance to its security requirements, both those specifying required security properties and behaviors, and those specifying required security functions and protections.

- **Type checking and static checking:** Both types of checks expose consequential (security-relevant) and inconsequential faults. The developer must then distinguish between the two, to ensure that he handles the security-relevant faults appropriately.

- **Logic modeling and comprehensive security testing:** Many logic errors cannot be identified effectively except through testing. Logic modeling provides the basis for developing appropriately comprehensive and revealing security test scenarios, test cases, and test oracles.

G.4.2.6. Consistent Naming

A common cause of security faults is incorrect developer use of aliases, pointers, links, caches, and dynamic changes without re-linking. To reduce the likelihood of such problems, developers should:

1. Treat aliases symmetrically. Every alias should be unique, and should point to only one resource;

2. Be cautious when using dynamic linking, to avoid unpredictable behaviors that result from runtime introduction of components;

3. Minimize use of global variables. When such variables are necessary, give the variables globally-unique names;

4. Clear caches frequently;

5. Limit variables to the smallest scope possible. If a variable is used only within a single function or block, that variable should be declared, allocated, and deallocated only within that function or block;

6. Deallocate objects as soon as they are no longer needed. If they will be needed again later, they can be reallocated at that time. Use language idioms, such as RAII (Resource Acquisition Is Initialization) in C++, to automatically enforce this convention.

G.4.2.7. Correct Encapsulation

Incorrect encapsulation can expose the internals of software procedures and processes by revealing (leaking) sensitive information or externally inducing interference. Correct encapsulation is achieved through a combination of:

- Effective system architecture;
- Effective programming language design;
- Effective software engineering;
- Static checking;
- Dynamic checking;
- Effective error handling, with generic (uninformative) error messages sent to the user, while full error information is logged.

G.4.2.8. Asynchronous Consistency

Timing and sequencing errors, such as order dependencies, race conditions, synchronization, and deadlocks, can threaten secure execution of the software. Many timing and sequencing errors are caused by sharing state information (particularly real time or sequence-order information) across otherwise disjoint abstractions.

> *NOTE: Disjoint abstractions are abstractions, such as classes in object oriented software systems that are unrelated in any way and possibly in conflict.*

An example of a related abstraction is “circle and ellipse”. Both objects are round geometric shapes. By contrast, a class named “CustomerTable” in an object-oriented database application is a disjoint abstraction, because “customer” and “table” are objects that have nothing in common.

The dissimilarity of two objects in a disjoint abstraction may result in a conflict when the two attempt to interact (Example: Object A is programmed to wait up to 5 seconds after requesting a validation from Object B and, if it doesn’t receive it, to terminate. However, Object B is programmed to give precedence to any validation request it receives from Object C. This causes it to interrupt processing of Object A’s validation request, with the result that it takes up to 10 seconds to return the response awaited by Object A, which has terminated in the meantime).

Synchronization and sequencing errors may be exploitable by attackers (i.e., in the above example, the attacker may use a man-in-the-middle spoofing attack to pose as Object C and issue the request that intentionally interrupts Object B’s handling of the validation request from Object A). In some cases, it may cause a process to fail suddenly (“crash”) into an insecure state, with the result that the software core dumps sensitive data held in memory. If the process that fails belongs to a security component relied on to protect other components or data, the failure may enable the attacker to bypass the security protection provided by the failed security component.

To avoid sequencing and synchronization errors in software developers should:

- Make individual transactions atomic (non-interdependent).

- Use multiphase commits for data "writes".
- Use hierarchical locking to prevent simultaneous execution of processes.

G.4.2.9. Safe Multitasking and Multithreading

Multitasking and/or multithreading in programs that run on operating systems that support multitasking and/or multithreading can improve the system's performance. However, multitasking/multithreading can also increase the software's complexity, making it harder to analyze and verify the software's correctness and security. Multitasking also increases the likelihood of deadlocks, which occur when two tasks or threads both stop executing at the same time, each waiting for the other to release a resource or terminate. Even when only a single thread is used, processes that communicate with one another and share resources—the complexity of the software increases.

If a program performs multitasking or multithreading, the code reviewer should carefully analyze its operation to ensure that its simultaneous processing of tasks and threads does not create conflicts in the usage of system resources such as memory or disk addresses. All tasks and threads should be synchronized to prevent such conflicts. As with all structured programs, each task should contain only one entry point and one (or very few) exit point(s).

G.4.2.10. Implementing Adequate Fault (Error and Exception) Handling

Robust software often contains more error- and exception-handling functionality than program functionality. Error and exception handling can be considered secure when the goal of all error and exception handling routines is to ensure that faults are handled in a way that prevents the software from entering an insecure state. The software should include security-aware error and exception handling capabilities, and should perform validation of all inputs it receives—including inputs from the environment—before using those inputs. Input validation will go along way towards preventing DoS, for example DoS resulting from buffer overflows in software written in, or interfacing with libraries written in, C or C++.

The software's error and exception handling should be designed so that whenever possible, the software will be able to continue operating in a degraded manner (with reduction in performance or acceptance of fewer [or no] new inputs/connections) until a threshold is reached that triggers an orderly, secure termination of the software's execution. The software should never throw exceptions that allow it to crash and dump core memory, or leave its caches, temporary files, and other transient data exposed.

See Appendix G:G.3.2.5 and G.4.4 for more information on software availability.

G.4.3. Secure Interaction of Software with Its Execution Environment

Critical software anomalies, many of which have significant security implications, usually result from incorrect assumptions about the software's execution environment, or from a misunderstanding of the interfaces between the software and its environment. Environmental uncertainties can complicate the developer's ability to identify the point in processing at which the software entered an insecure state, and can make it difficult to determine what adjustments are needed to the software's operation to return it to a secure state. The precise understanding of the software's environment during all of the software's potential states—including normal operation, operation under attack, and operation during partial or complete environment failure and partial or complete software failure—is also critical for developing realistic software security test cases and scenarios.

The software must be able to respond effectively to changes in its execution environment without becoming vulnerable to compromise. For example, if the software relies on an application-layer firewall to filter out potentially dangerous input, and that firewall fails, the software should be able to detect that failure and immediately begin rejecting all new input until the firewall is again detected to be online.

No matter how accurate his original assumptions about the execution environment, the developer should not implement software that relies on its environment for security services or protections by hard-coding the software's interfaces (including formats of inputs it expects and outputs it produces) to those environment services/protections. Instead, the developer should use standard interfaces that can be reconfigured, modified, or substituted at compile-time or runtime. That way, if any environment component relied on by the software is replaced, or the software itself is moved to a different execution environment, such changes could be easily accommodated without introducing unanticipated vulnerabilities into the software.

Formal methods can provide a common vocabulary through which software developers and systems engineers can communicate about the execution environment in which the software will operate. Executable specifications for rapid prototyping, especially those with a user-manipulated interface, can enable the developer to explore his assumptions about the execution environment, and can reveal unrecognized security requirements for the software/environment interfaces. The following are specific considerations that will help ensure secure interaction between software and its execution environment.

G.4.3.1. Safe Reliance on Environment-Level Security Functions and Properties

By providing certain security services—such as public key encryption and digital signature, virtual machine sandboxing, application firewall filtering and proxying, process/data isolation by file system access controls, etc.—the software's execution environment (middleware, operating system, hardware platform, and network infrastructure) is often relied on to enable or augment the software's own self-protecting features.

For example, in a Web application, Secure Socket Layer (SSL) or Trusted Layer Security (TLS) encryption at the transport layer may be invoked by the software by using the HyperText Transmission Protocol-Secure (HTTPS) protocol to create an encrypted session "pipe" over which the client can transmit user supplied entries into an HTML form to a Web server application. Moreover, an application-layer firewall may intercept that data on route from client to server, to perform input validations on the data, either to augment input validation in the server by filtering out some of the bad data the server would otherwise receive or, in the case of a deficient acquired or reused (including legacy) server component, to compensate for that server's inadequate (or nonexistent) input validation.

As noted earlier, however, even if environment-level components are relied on by a software system to perform security functions on its behalf, or to protect it from attack, the software itself should include sufficient error and exception handling to prevent it from entering an insecure state or having its security properties otherwise compromised should any of the relied-on environment components fail. Without such error and exception handling, such compromises are highly likely given the likelihood of the software's vulnerabilities becoming exposed in the absence of the environment-level protections.

G.4.3.2. Separating Data and Program Control

Program data is required to be both readable and writeable-for most general purpose software. By contrast, the program control information is unlikely to change, so it can be read-only. By logically separating program data and control information, software is prevented from manipulating itself, whether purposefully or due to the actions of a malicious entity. One widely used example of data and control separation is the Harvard computer architecture, in which data and program control information are stored in two physically separated memories.

There are a wide variety of techniques and options available for separating program data and control information from the processor level to the application level. Some example techniques are:

- Storing data files created by the software in a completely separate location in the file system from programs executed by (or within) the software;

- For software systems that must accept mobile code, mobile agents, or other programs that are remotely downloaded, they can implement very restrictive constrained execution environments (e.g., sandbox) for that downloaded code to prevent it from accessing other areas of the execution environment, or the software's data files or binary executables;

- Storing executable files in a dedicated, tightly access-controlled directory isolated from the data directory to prevent modification;

- Encrypting executable files and relying on a trusted encryption function at startup;

- Taking advantage of computing hardware that uses the Harvard architecture.

One common situation where lack of data and program control separation has led to security incidents are auto-executing macros in documents. Numerous computer viruses have taken advantage of macros to infect users' systems. As such, macros should never be embedded in files created by the software system unless those macros have first been thoroughly tested to verify that they do not change the security posture of the execution environment or gain unauthorized access to its data. As with mobile code, embedded macros should be limited to executing within a constrained environment.

G.4.3.3. Trustworthy Environment Data

Security is more easily achieved for software that runs within frameworks that have been verified to provide only trustworthy environment data. For example, Java EE components run within "contexts" (e.g., System Context, Login Context, Session Context, Naming and Directory Context, etc.) that can be relied on to provide only trustworthy environment data to Java programs at runtime.

Even if a framework is' not used, file system access controls can be configured to protect the configuration data stored locally with the software. Only the administrator should be granted *write* and *delete* privileges to that configuration data, while *read* privileges are limited to the administrator and the software itself. No other actor ever requires access to the system's configuration data. (Noteworthy exception: Client-side users may need to be granted at least a subset of administrator privileges for their own client systems.) If file system access controls alone are not sufficient to protect the software's configuration files, those files may be stored in encrypted form, and the software system extended to invoke the necessary decryption service whenever it needs to reference the files (e.g., at start-up time).

The software's configuration information may be stored remotely, on a trusted server elsewhere on the network (e.g., in the Lightweight Directory Access Protocol [LDAP] directory from which a public key-enabled system retrieves certificates and public keys). The software's requests to access that remote directory should be sent over an encrypted (e.g., SSL/TLS) connection.

If this directory approach is used with software that is likely to be cloned, the configuration data should not be copied directly from the software to the directory. Instead, the software should be prevented from returning its configuration data to the remote directory over the same communications path by which it earlier retrieved the configuration data from that directory. If the software requires read-back verification of the configuration data it receives, the data and/or connection should be encrypted to prevent the clear-text version of the data from being "sniffed" in transit.

> *NOTE: Cloning is the act of creating an identical copy of the existing system, including the software system and its execution platform. Cloning may be performed to create or refresh a test system to ensure it is identical to the production system, before testing software updates. Cloning may also be performed to move an existing software system to a new platform, or to duplicate it on another platform(s). Cloning does more than simply copy the executable; it also updates all of the software's configuration files to accommodate its hosting on the new platform. Cloning of private cryptographic information (such as private X.509*

certificates) may cause a security violation; for this reason, new private cryptographic information should be created for the clone.

G.4.3.4. Presuming Client Environment Hostility

Always assume that a client will run in the most hostile execution environment possible. Browsers and other clients cannot be trusted to perform security-critical functions, because they are too easily reconfigured, corrupted, and extended by the client's users. Server applications, portals, and proxy agents that interact with clients should be implemented to protect themselves against attacks from clients that have been subverted, hijacked, or spoofed.

G.4.3.5. Safe Interfaces to Environment Resources

Nearly every programming and scripting language allows application-level programs to issue system calls that pass commands or data to the underlying operating system. In response to such calls, the operating system executes command indicated by the system call, and returns the results to the software along with various return codes that indicate whether the requested command was executed successfully or not.

While system commands may seem like the most efficient way to implement an interface to the underlying operating system, a secure application will never issue a direct call to the underlying operating system, or to system-level network programs such as Sendmail or File Transfer Protocol (FTP). Not only does each application call to a system-level function create a potential target for attack, whenever a system call is issued by the software, the homogeneity of the system's design is reduced, and its reliability diminishes.

Application-level programs should call only other application-layer programs, middleware, or explicit APIs to system resources. Applications should not use APIs intended for human users rather than software nor rely on a system-level tool (versus an application-level tool) to filter/modify their own output.

All references to system objects should be made securely. For example, call-outs and filename references should specify the full pathname of the system resource being called/the file being referenced, e.g., */usr/bin/sort* rather than *../../sort*. Using full pathnames eliminates the possibility that the wrong program may be called, or executed from the wrong directory (e.g., a directory in which a Trojan horse is stored at the location where the calling program expected to find a valid program).

The discussion in Appendix G:G.8.9 of the need for secure software installation configuration documentation provides information about documenting the software's required environment configuration.

G.4.4. Implementing Software to Remain Available

When software performs unexpected actions, such actions could not only cause a critical fault (a failure), but could violate the software's specified security properties, making it vulnerable to compromise. Robust software not only operates correctly, it cannot be easily induced to fail.

Robust programming is intended to produce robust software. Robust programming requires the software's behaviors to fall within the bounds of its design specification, regardless of the nature of its execution environment or the input it receives. Robust programming adheres to four principles:

1. Defensive programming
2. Information hiding
3. Anomaly awareness
4. Assuming the impossible

These four principles are discussed below. Refer back to Appendix G:G.3.2.5 for a discussion of designing software that will remain available.

G.4.4.1. Defensive Programming

Defensively programmed software is software that:

1. Does not rely on any parameters that are not self-generated;
2. Assumes that attempts will be made to subvert its behavior, directly, indirectly, or through manipulation of the software to violate a security policy.

All software processes should expect and be able to handle the problems listed in Table G-1 below. These objectives are particularly important when developing code that will be combined (as in a library), used in a variety of contexts (as with system calls), or downloaded (as with mobile code).

Table G-1. Software Errors or Faults and Suggested Remediations

Expected Problem	How the Software Should Handle the Problem
All input and parameters received by the software will be faulty or corrupt.	The software should validate all inputs and parameters are validated before use to filter out those that are incorrect, bogus, or malformed. The list of references in Appendix G:G.10 includes a number of resources describing how to implement input validation in a variety of programming and scripting languages.
The execution environment differs significantly from the environment for which the software was designed (a frequent problem with acquired and reused software).	Hidden assumptions should be recognized, and the software should be designed to defend against attempts to exploit such assumptions to force its components to behave insecurely.
The results the software receives from any other component or function it calls will contain at least one error.	The software should be designed and implemented to handle unexpected and unlikely, even presumably "impossible", events. All possible errors and exceptions should be anticipated.
The software contains vulnerabilities, not all of which will be identified before the software is deployed.	The software should be designed and implemented to contain and limit the damage that could be caused by vulnerabilities in any of its components. It should provide feedback to the developer (during debugging and testing) and the administrator (during operation) to assist in their rapid detection of such vulnerabilities.

G.4.4.2. Information Hiding

Information hiding and information abstraction provide well-defined interfaces through which the software can access the information and functions while concealing the specifics of implementing the data structures and functions from the program. Information hiding and abstraction are an implementation of the principle of least privilege. Software should not be granted privileges that enable it to access library routine internals that are expected to remain consistent across calls, because the software would then be able to alter those routine internals without using the library's interfaces.

By preventing the software from accessing library internals directly, an extra layer of defense is introduced into the software. For example, given a library that uses a pointer to an internal data structure representing a file, if the calling software is able to access the data in that structure by simply dereferencing the pointer, the software could accidentally (or deliberately) modify the data in that data structure and, thus, cause the library functions to fail. Robust library functions should never return pointers or indices into arrays that the calling software could use to access internal data directly. Instead, the library functions should hide the true pointers or indices in a token. Hiding data structures also has the benefit of making the software or library more modular.

G.4.4.3. Anomaly Awareness

In most distributed systems, components maintain a high level of interaction with each other. Inaction (i.e., lack of response) in a particular component for an extended period of time, or receipt from that component of messages that do not follow prescribed protocols, should be interpreted by the recipient as abnormal behavior. All components should be designed or retrofitted to recognize abnormal behavior patterns that indicate possible DoS attempts. This detection capability can be implemented within software developed from scratch, but must be retrofitted into acquired or reused components (e.g., by adding anomaly detection wrappers to monitor the component's behavior and report detected anomalies).

Early detection of the anomalies that are typically associated with DoS can make containment, graceful degradation, automatic fail-over, and other availability techniques possible to invoke before full DoS occurs. While anomaly awareness alone cannot prevent a widespread DoS attack, it can effectively handle isolated DoS events in individual components, as long as detected abnormal behavior patterns correlate with anomalies that can be handled by the software as a whole.

G.4.4.4. Assuming the Impossible

Events that seem to be impossible rarely are. They are often based on an expectation that something in a particular environment is highly unlikely to exist or to happen. If environment changes or the program is installed in a new environment, those events may become quite likely.

The use cases and scenarios defined for the program need to take the broadest possible view of what is possible. The software should be designed to guard against both likely and *un*likely events. Developers should make an effort to recognize assumptions they are not initially conscious of having made, and should determine the extent to which the "impossibilities" associated with those assumptions can be handled by the software.

G.4.5. Making Software Resistant to Malicious Code

Malicious code comes in the form of byte code, executable code, or runtime-interpretable source code that is either embedded, via insertion or modification, within a deployed web application or web service, or to invalid applications or services to which a user or consumer service request is redirected when it attempts to access a valid web page or web service. In both cases, the objective is to replace or augment the service's valid logic with malicious code that intentionally subverts or prevents the service's intended operation.

Malicious code attacks are most often achieved by including executable commands in what is expected to be non-executable input. In some cases, these commands provide pointers (links) to locations at which malicious code resides and from which, as soon as the location is accessed, it will be automatically downloaded to the targeted system.

Logic bombs, time bombs, trapdoors, and backdoors are forms of malicious code that are either planted by malicious developers or delivered as Trojan horses via another attack vector such as a virus or worm payload or planted by an attacker who has gained the necessary level of write-access to the targeted software's host. Increasingly, attackers are augmenting these more traditional forms of malicious code with malicious implementations of surreptitious software mechanisms that originally had valid (if sometimes ethically questionable) purposes, such as spyware, adware, and bots.

Unlike external attackers, malicious developers can exploit their deep knowledge of how a particular host will be configured in deployment, and how the software and its environment will interact. The threat of developer-inserted malicious code has become increasingly prominent as more and more of the components from which software systems are constructed are acquired rather than built from scratch, by overseas development organizations who may be under the influence or control of terrorist organizations or adversarial governments.

Developers can use several techniques and host-based tools to make their software more resistant to infection and corruption by malicious code after deployment. These techniques fall into four categories:

1. Programmatic countermeasures;
2. Development tool countermeasures;
3. Environment countermeasures;
4. Add-on countermeasures.

Some anti-malicious code countermeasures are applicable for binary executables, while others can be used only on source code. Software-level countermeasures should be used in conjunction with host-based and network-based countermeasures, such as virus scanners, filtering routers and firewalls, and (possibly) newer techniques such as virus throttling (developed by Hewlett Packard).

Most development-time countermeasures focus on producing software that is either not vulnerable to runtime insertions of malicious code, or that resists the insecure state changes that executing malicious code will attempt to force in the targeted software. By contrast, most runtime countermeasures either prevent malicious code from being inserted in deployed programs, or prevent already-inserted malicious code from executing, most often by altering some aspect of the runtime environment to something other than what is expected by the malicious code.

The four categories of countermeasures are discussed below.

G.4.5.1. Programmatic Countermeasures

Programmatic countermeasures are techniques and mechanisms built into the program itself at development time, including:

- Input and output controls that detect and filter potentially malicious input: An example of such controls are the input validation routines that verify proper data types, syntax, and input lengths, to prevent buffer and stack overflows and command injection attacks;

- Implementation of security checkpoints within the program that cannot be bypassed by users;

G.4.5.2. Development Tool Countermeasures

Using specific tools, or use of specific techniques with general development tools, in software development can help protect the software against runtime malicious code insertions. These countermeasures include:

- **Type-safe programming languages:** Type-safe languages such as Java, Scheme, ML (MetaLanguage), or F# ensure that operations are only applied to values of the appropriate type. Type systems that support type abstraction let programmers specify new, abstract types and signatures for operations that prevent unauthorized code from applying the wrong operations to the wrong values. In this respect, type systems, like software-based reference monitors, go beyond operating systems in that they can be used to enforce a wider class of system specific access policies. Static type systems also enable offline enforcement through static type checking instead of each time a particular operation is performed. This lets the type checker enforce certain policies that are difficult with online techniques. Refer back to Appendix G:G.4.1.2 for more information on safe languages;

- **Safe versions of libraries and languages:** Most safe libraries and languages (or language variants) are intended to help avoid problems with buffers, pointers, and memory management. *Safestr* in C, for instance, provides a consistent and safe interface to string-handling functions, the root of many security vulnerabilities. However, it requires some effort to recode any string handling that the program may do, converting it to use the new library. If the program will operate in a particularly vulnerable environment,

it may be prudent to consider at a minimum implementing a virtual machine on the host system to contain and isolate the program, or possibly reimplementing the program in a language that includes its own security model and self-protecting features (e.g., Java, Scheme, Categorical Abstract Machine Language [CAML]) rather than C or C++. Refer back to Appendix G:G.4.1.4 for more information on "safe" libraries;

- **Hardened versions of system calls:** Since all programs must use system calls to transfer data, open files, or modify file system objects, limiting the system calls that a program is able to invoke allows untrusted programs to execute while limiting the damage they can do. Kernel-loadable modules, on systems that support them, can be used to extend the protections surrounding untrusted programs, limiting further the damage that can be done by a subverted program;

- **Secure compilation techniques:** Compilers can be modified to detect a maliciously modified stack or data area. A simple form of this protection is the stack canary, which is put on the stack by the subroutine entry code and verified by the subroutine exit code generated by the compiler. If the canary has been modified, then the exit code terminates the program with an error. Another protection that can be implemented in a compiler is randomization of variables and code positions in memory, particularly the randomization of the location of loaded libraries. Refer back to Appendix G:G.4.1.1 for a discussion of secure use of compilers.

G.4.5.3. Environment-Level Countermeasures

Environment-level countermeasures are usually runtime countermeasures implemented at the execution environment level, such as:

- Access control-based constraint mechanisms that prevent malicious code from being introduced into a system, intentionally or unintentionally, by users. Example: *chroot* "jail" on a Unix or Linux system;

- Constrained execution environments that minimize the scope of damage that malicious code and other untrusted processes can affect, if it does manage to enter the system. An example is to run all untrusted programs within a virtual machine (VM). The VM is, in fact, a standard feature of Java, which executes all untrusted programs within the JVM, in effect "sandboxing" those programs so that they cannot directly access the software's trusted programs or resources. A VM can rely on specific resources (memory, hard drive space, virtual network interface, etc.) that it can access directly, but it is completely unable to affect the programs and resources that reside outside the VM;

- Program shepherding, to monitor control flow transfers, prevent execution of malicious data or modified code, and to ensure that libraries are entered only through exported entry points (thus, restricting control transfers based on instruction class, source, and target). Program shepherding also provides sandboxing that cannot be circumvented, allowing construction of customized security policies;

- Altered program memory maps implemented by modifying the default protection bits applied to a program's stack, and, additionally, other memory regions. Each page in a computer system's memory has a set of permission bits describing what may be done with the page; the memory management unit of the computer, in conjunction with the kernel, implements these protections: altering the memory map requires no changes to the protected programs and, if successful in thwarting an exploit, the result will be a protection fault and termination of the vulnerable program. This mechanism has no performance impact on the protected programs themselves, but may incur overhead within the operating system. As it requires a modification to the operating system, this protection is not portable. Please also note that altering program memory maps only protects the stack, not the heap or program data;

- Monitoring and filtering to detect and prevent undesirable state changes in the execution environment. Such filtering will help identify suspicious state changes in the software's execution environment by

taking "snapshots" of key environment attributes before and after executing untrusted software (e.g., mobile code) that may contain malicious logic, and monitoring unexpected differences in environment state during the program's execution. Such state changes are often the earmarks of malicious code attacks. One approach entails the following actions:

1. Configure a filtering router to pass traffic between a test system, on which is hosted the software and its intended execution environment, and the network.

2. Install network analysis tools on the filtering router.

3. Snapshot the software's execution environment to develop a detailed picture of its known, trusted behavior.

4. Disconnect or isolate the test system from the network.

5. Install the untrusted program suspected to contain malicious code;

6. Record and analyze all changes in environment behavior during the untrusted program's execution. If the tester determines that all recorded changes to the environment and system states are neither unauthorized nor unexpected, it can be reasonably concluded that the particular untrusted software is "safe".

G.4.5.4. Add-on Countermeasures

Add-on countermeasures may be added to the program at development time or runtime, e.g., in the form of wrappers, plug-ins, etc. Add-on countermeasures include:

- Code signatures to provide validated evidence that the code came from a trustworthy source. Code signing helps protect software programs by enabling the user or process that will execute the code to verify where the code came from and whether the code has been modified (i.e., tampered or corrupted) from its original state. Note that, while code signatures make it possible to verify that an program comes from a trusted source in an uncorrupted form, it cannot guarantee that the code was' not malicious to begin with, or is otherwise error-free;

- Code specifically designed to prevent the exploitation of covert channels by communicating program components;

- Filtering of system calls, whereby a monitoring program is implemented that requires system calls invoked by an untrusted program to be inspected and approved before being allowed to continue. The monitor program makes decisions about the validity of system calls by knowing in advance what the untrusted program is supposed to do, where it is expected to manipulate files, whether it is expected to open or listen to network connections, and so on. Valid behavior of the untrusted program is coded in a profile of some kind, which is referenced when the untrusted program executes. This defense, though, will affect the performance of programs run under the watch of a monitor but will not affect other programs.

G.5. Secure Assembly of Reused and Acquired Components

Like all software, a system assembled/integrated from reused and acquired components must be explicitly designed to satisfy its functional and non-functional security requirements. It must then be implemented to conform to that design, and the implementation must exhibit its required security properties. A component-based software system that does not conform to its specifications, regardless of how well it is engineered, is both incorrect and insecure by definition.

Multiple components are likely to exist that can satisfy the specified requirements for certain portions of a software system. This means a number of integration/assembly options may need to be modeled to provide a framework in which candidate components can be combined to evaluate which combinations of components most effectively satisfy the whole system's requirements. The best grouping of components can then be selected, and integrated/assembled according to the candidate framework that, when implemented with the selected components, conforms most closely to requirements. A key consideration in deciding whether to use acquired or reused software components, and which components to use, is the security criteria used in the evaluation and selection of those components.

Use of components influences software system architecture and design in the following ways.

1. Integration/assembly options must be based on explicit and implicit assumptions about how a given component will interact with other components. Components often depend on other components (and even on specific versions of other components) for their successful operation. The suppliers of commercial (COTS, government-off-the-shelf [GOTS] and shareware), freeware, and legacy binary components (but not open source components) virtually always retain the data rights to their components' source code. These suppliers intend for their components to be used without modification. Binary components must be approached as "black boxes" whose functions, properties, and interfaces can only be changed through reconfiguration, to the extent supported by the component, or by external means such as filters, firewalls, and wrappers.

2. Designers need to recognize that each component's supplier is likely to put competitive advantage far ahead of security considerations when scheduling new component releases and defining their content. In the prioritization and scheduling of patches for discovered/reported security vulnerabilities, the supplier will most likely focus on those vulnerabilities that are the least expensive to patch, and those which get the worst or most enduring publicity.

3. Recognizing that as soon as a component is selected for use in the software system, the security of the software system is, at least in part, at the mercy of the component supplier's priorities and security decisions, the system designer should make sure the selected integration/assembly option is flexible enough to easily accommodate:

    - new versions of current components,
    - substitutions of one supplier's component for another,
    - reconfigurations of components,
    - insertion of countermeasures to mitigate security faults introduced with new versions/components or vulnerabilities not yet patched by suppliers.

    The ideal design will be as generic as possible: it will reflect the *roles* of the components in the software system rather than the specific selected components, and will rely on standard rather than proprietary interfaces.

A key assumption throughout this document is that software components have inherent security properties. This has particular implications for multi-purpose software systems, such as Web services. A Web service is an example of a software system that will be used in multiple instances, each of which may have a different set of security requirements. In order to operate effectively, the Web service will need to possess all of the security properties and capabilities required for each of the instances in which it will operate. For a SOA to work effectively, it may be necessary to delay the binding of security decisions within the Web service until runtime, to minimize the "hard coding" of security properties and policy decisions that will prevent the service from adapting to the needs of the different operational instances under which it must run.

G.5.1. Component Evaluation and Selection

It is critical that the developer's understanding of a component's security properties is not excessively influenced by or wholly reliant on a supplier's claims for a product. A critical focus of the system engineering process, then, must be the thorough and accurate security evaluation of all acquired or reused software to ensure that those components do not include any exploitable faults, misbehaviors, or other weaknesses. A component security evaluation should include the following steps:

1. Establish the functional and security design constraints to be used to narrow the range of possible components to be evaluated;

2. Select a methodology by which design tradeoffs can be quantified so that a measurement of the component's fitness for secure use can be computed;

3. Analysis and testing to determine which candidate components best satisfy the requirements for the security properties and attributes and associated functions required for those components given their roles in the larger software system;

4. Analysis, testing, and prototyping to determine for each integration/assembly option, which components behave the most securely, when combined with the others selected for that option;

5. Selection of the most secure components and integration/assembly option, then given those selections, adjustment of the software system's architecture and design to mitigate residual risk to the greatest extent feasible.

The security functions and properties/attributes of acquired or reused components reflect certain implicit and sometimes explicit assumptions made by that software's supplier, including assumptions about the original supplier's specification of security requirements for the component, the operational contexts in which it is expected to be used, and the presumed business processes of its users. The supplier's assumptions rarely match all of the security requirements, contexts, and operational processes (current or anticipated) of the role the component is intended to fill in the integrated/assembled software system.

Much of the selection process for acquired or reused software should focus on determining and mitigating the impact of conflicts between supplier assumptions and integrator assumptions. For SOUP components, determining supplier assumptions may be particularly difficult, especially if the software is not well-documented. For this reason, the availability of good technical documentation should be a key decision factor in the selection of SOUP components. This may entail adjusting the security expectations and operational processes of the integrated software system's intended users and stakeholders to accommodate the ability to use a desirable acquired or reused component for which a conflict arises between its supplier's assumptions and its users expectations.

G.5.2. Secure Component Assembly/Integration

During the implementation phase, those elements of the design that are to be implemented through from-scratch development may be translated into a machine-readable format using a code generation tool, or they may simply be used as a frame of reference for coding from scratch . All programming tools (compilers, interpreters, debuggers, and automatic code generators) will be selected with an eye towards simplifying the assurance of the software's required security properties and, conversely, minimizing risk that the tools themselves will introduce vulnerabilities into the software produced by them. The programming language(s) to be used will be selected with similar considerations in mind.

The security properties and other emergent properties exhibited by a component-based system should not conflict with the properties exhibited by the system's individual components. It is commonly believed that a system composed of sophisticated components (e.g., COTS applications) can only be as secure as the least secure of

those components. In reality, however, component assembly/integration techniques, such as sandboxing during the execution of an insecure component, can be used to isolate vulnerable components to minimize their impact on the security of the system as a whole. For example, if Component A is vulnerable to buffer overflows, a new Component B could be created with the sole purpose of providing an interface between the required service provided by of Component A and the other system components that use that service. Once Component B is established, all direct interactions between the other components and Component A would be disallowed or disabled.

There should be ongoing analysis throughout the integrated/assembled system's lifetime to assure that its security requirements remain adequate, and that it continues to satisfy those requirements correctly and completely even as acquired or reused components are patched, updated, and replaced. See Appendix G:G.9 for more information on secure configuration management of software containing acquired or reused components.

G.5.3. Influence of Reused and Acquired Software on from-Scratch Code

There is a misapprehension among some developers that as long as they write all their own software from scratch, and never use acquired or reused components, they completely control the security of their software. As demonstrated in Figure 5-1, even software built from scratch will not be completely unaffected by acquired or reused components.

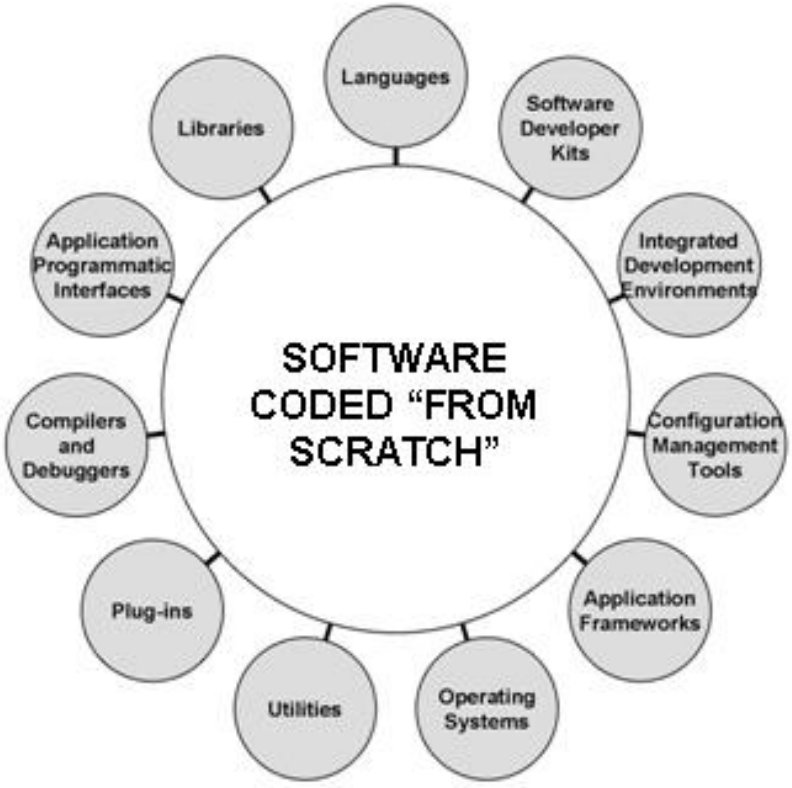

Figure 5-1. Acquired and Reused Enablers for from-Scratch Software

The only exception to the rule that all software is subject to the influence of acquired or reused programs, tools, and environment components are the handful of high-consequence embedded systems that:

1. Incorporate their own operating systems;
2. Are hosted on purpose-built hardware;

3. Are built entirely by custom-developed tools, including purpose-built compilers and debuggers.

Given that this is a very small subset of modern software, it is safe to say that for the vast majority of software, no matter how much rigor and care is applied to the coding of the software itself, any of the acquired or reused tools or components in its development or execution environments could introduce vulnerabilities into the software at compile time, during installation, or at runtime.

The security of these acquired and reused tools and components then is critical to assuring the security of the software itself. Code security review alone will never be sufficient on its own for determining how secure from-scratch code will be once compiled and deployed. Security testing should also include techniques that exercise and observe the behavior of the compiled, linked binary software as it runs in its target environment

G.6. Software Security Testing

Software security testing assesses the way that software systems behave and interact, and of how the components within a software system behave and interact. The main objectives of software security testing are to:

1. Locate exploitable vulnerabilities in the software or failures to meet security requirements;
2. Demonstrate the continued secure behavior of software in the face of attempts to exploit known or suspected vulnerabilities in the software.

Assessing the security of a software module, component, or whole program, while not necessarily technically complex, relies on a multifaceted approach involving a variety of technologies and techniques. The main objective will be to ensure that the software has sufficient safeguards for the threat environment and is' not vulnerable to attack. The DHS BuildSecurityIn portal (https://buildsecurityin.us-cert.gov/) provides a number of articles on security testing. These articles go much further into the motivations behind software security testing as well as the resources available to software security testers.

Security testing can also help verify that the software exhibits its required security properties and behaviors. Requirements verification should not be the main objective, however. Testing that is limited to verifying requirements compliance will not necessarily ensure that the software is secure: it will only ensure that the software is as secure as its security requirements are adequate. If the software's security requirements are inadequate, the security of the software will also be inadequate. For this reason, as with the rest of the software life cycle, software security testing should be essentially risk-based rather than requirements-based.

Risk-based testing focuses on misuse and abuse cases within software. More than simple negative testing, these efforts demand that the tester "think like an attacker" and attempt to invalidate assumptions made by the software and underlying libraries, frameworks, platforms, and operating systems. This type of testing is likely to reveal luring attacks on privileged components, overflow-type attacks, and other attacks where normal negative testing would not. Test techniques that are well suited to supporting risk-based testing include security fault injection and penetration testing.

Software security testing, then, is not the same as testing of the correctness and adequacy of the software's security functions. While such tests are extremely important, they are correctly a subset of the software's overall functional and interoperability testing, because security functions are a subset of the software's overall functionality, and the interfaces to those functions are a subset of the software's overall interface set.

The security of the software itself (versus its security functionality) is what is often overlooked during requirements-based testing, if only because the security of the software—i.e., its lack of exploitable vulnerabilities and continued secure behavior in the face of intentional faults—is seldom captured as requirements in the software specification. However, failure to document nonfunctional security requirements for software must not be used as an excuse not to test the security of that software. Improving the security of the software life cycle is an evolutionary process, and it is often easier to make a case for adding the necessary due diligence early in the life

cycle if it can first be demonstrated, through security testing that is risk-driven rather than requirements-driven, that security problems exist in the software directly resulting from the failure to capture security behaviors and properties in the requirements specification.

Even if such requirements have been captured, the assumptions on which those requirements are based when they were specified may change by the time the software is ready for testing. While it will be true that a complete specification that includes these nonfunctional security requirements will allow for some security properties and behaviors to be tested as part of the overall testing to verify the software's requirements compliance, the fact is that nonfunctional software requirements will always be inadequate. This is because the threat environment in which the software operates is very unlikely to remain static. New attack strategies are virtually guaranteed to emerge during the time between the specification and the testing of the software. New versions of acquired or reused software components introduced after the original acquisition evaluation may harbor new vulnerabilities.

For these reasons, software security testing that is risk-based should always be included in the overall test plan for the software. Besides, in purely practical terms, the scenarios, tools, and techniques used for verifying nonfunctional security requirements compliance and those used for risk-based testing are very likely to be identical. The important thing is to ensure that the test cases exercise aspects of the software's behavior that may not be demonstrated through requirements-based testing alone.

G.6.1. Objective of Software Security Testing

The main objective of software security testing is to verify that the software exhibits the following properties and attributes:

1. **Ability to remain in a secure state in the face of intentional faults:** The integrity or availability of the software, and the confidentiality of its sensitive data and/or processes, should not be compromised as the result of an intentional fault, i.e., the type of fault typically caused by an attempt to tamper with the software's running executable, or by a denial of service attack against the software. At worst, if the software is compromised, it should include error/exception handling logic that immediately isolates, constrains, and minimizes the impact of the compromise.

2. **Absence of exploitable weaknesses:** The software contains no faults, backdoors, dormant or hidden functions, or other weaknesses that can be exploited by an attacker (human or malicious code) to successfully compromise the security properties of the software itself, of the data it accesses or provides access to, or of any part of its execution environment. Additionally, the Web content itself can be displayed in an insecure manner, such as displaying sensitive information like Social Security numbers. Not all Web content is under control of the developer as content can come from an external database or some other source. However, content that the developers can control should not allow attackers to gain access to sensitive information. Conversely, developers should ensure that is required to be displayed is accessible—such as a reference to a privacy policy.

3. **Predictably secure behavior:** The software performs its specified functions—and only those functions—without compromising the security properties and attributes of the software itself, its environment, or its data: This includes ensuring that no "dormant" functions in the software can triggered, either inadvertently during normal execution, or intentionally by an attack pattern.

4. **Security-aware error and exception handling:** The software performs safe error and exception handling. The software does not react to either unintentional anomalies or intentional faults by throwing exceptions that leave it in an unsafe (vulnerable) state. Furthermore, the software's exception and error handling functions have been designed and implemented to recognize and safely handle all anticipated security errors and exceptions. Testing should demonstrate that the software responds appropriately to anticipated (or envisioned) abnormal situations, including both unintentional anomalies and intentional faults, and should focus particularly on those that occur during software startup, shutdown, error detection

and recovery. Vulnerabilities are most likely to result from the software's incorrect handling of these sensitive changes in its processing states.

5. **Appropriate implementation of security requirements:** The software meets defined security specifications. If the software can maintain a secure state after a failure, contains no exploitable faults or weaknesses, has predictably secure behavior, and implements security-aware error and exception handling, but fails to satisfy its security requirements it may still be vulnerable after it is deployed. For example, software that is otherwise secure may fail to validate the code signature on mobile code it downloads before executing that code, as specified by its requirements, which exposes it to a higher risk of compromise if that mobile code turns out to be malicious.

A sixth objective is to ensure that the software's source code has been stripped of any of the dangerous constructs described in this section.

The ultimate findings of software security testing should provide enough evidence to risk managers to enable them to make informed decisions about how the software should be deployed and used, and to determine whether any constraints must be applied in deployment to counteract security problems introduced during development until those problems can be rectified through reengineering.

G.6.1.1. Finding Exploitable Faults and Weaknesses in Software

As Herbert H. Thompson and James A. Whittaker observe in their book How to Break Software Security, security faults are different from other types of faults in software. Traditional non-security faults usually result from specification violations: the software failed to do something it was supposed to. Security faults, by contrast, typically manifest as additional behavior: the software does something additional that it was not originally intended to do. Finding security faults in software is particularly difficult, because such faults are often masked by the software doing what it is supposed to.

Software testers are trained to look for missing or incorrect output, and traditional software testing is intended to verify correct behavior. As a result software testing tends to overlook side-effect behaviors that are not clearly apparent when observing correct behavior.

A security-aware software tester should not only verify the correctness of functionality, but should also pay attention to functionality's side effects and security impacts. These side effects and impacts may be subtle and hidden from view. For example, they may manifest as unexpected file writes or registry entries, or more obscurely in the addition of extra packets of cleartext data appended to an encrypted data stream.

To verify that software is free of security faults and weaknesses, security analysts, code security reviewers, and software security testers should participate as appropriate in the series of analyses, reviews and tests the software undergoes throughout the life cycle. Reviewing software for exploitable faults/weaknesses may be a daunting task, regardless of whether or not the source code is available for analysis. To this end, a number of vendors provide security testing tools that may assist the security reviewer. Using security testing tools, a semi-automated review can be performed in which the tool can be used to find some of the faults and give the reviewer a better idea of what portions of the software may require a more diligent or manual review. Similarly, as discussed in Appendix G:G.6.3, a number of organizations provide security testing as a service.

To the greatest extent possible, testers should employ available security-oriented automated code review and testing and vulnerability scanning tools, including those that operate on source code and on runtime executables. However, some testers do not completely understand the tools they are using—or may not completely trust these tools. In either case, the results of these tests may not be used fully. Testers that do not fully understand the tools may be unable to distinguish false positives from true positives, while testers that do not trust the tools may overestimate the possible number of false negatives—vulnerabilities the tools do not detect.

Testing of software as it is prepared for deployment and of software after deployment can reveal vulnerabilities that emerge because of changes over time in the software's threat model. Whole life cycle security reviews and tests should be performed on *all* software, not just software that performs security functions or enforces security properties. Defects in a non-security-related software component may not manifest as vulnerabilities in that component itself, but may cause the component to interface insecurely with a contiguous component, thus potentially compromising the security of that second component; if that second component is one that performs security functions or enforces security properties on behalf of the whole software system, such an exploit could compromise the whole system.

Standard analyses and testing of software do not always consider certain aspects of software designs, implementations, or configurations to be "incorrect" or "bad", because such problems do not affect the correctness of the software's normal operation, but only manifest when intentionally exploited by an attacker. The "security-enhancement" of software analysis and testing, then, should focus on expanding the list of what are considered security issues to include the issues described in the sections below.

Security testing should include a phase- and test target-appropriate combination of security test techniques adequate to detect this full range of vulnerability types. The combination of techniques used at each test phase should be adequate to provide the required security assurances needed by the software's risk manager and, in the case of whole-system testing, its security certifier. Other tests that may be run during these test phases include feature tests, performance tests, load tests, stability tests, stress tests, and reliability tests. See Appendix G:G.6.3 and G.6.4 for discussions of "white box" testing (testing that requires source code) and "black box" testing (testing of binary executables).

G.6.1.1.1. User-Viewable Source Code or Byte Code

If a client includes a function such as a browser's "view source" function, the plaintext source code or byte code made visible to the user by that function should be examined to ensure that it cannot be exploited in a reconnaissance attack, either to help the attacker perform social engineering, or to increase his/her knowledge of the software functions that could be exploited, or the directory structures that could be targeted, by other focused attacks. Content that should not be included in viewable plaintext source code or byte code includes:

- Comments that include personal information about the code's developers (compromise of privacy);
- Pathname references that reveal significant details about the directory structure of the host running the software and detailed version information about the development tools used to produce the code and the environment components of the host on which the software runs. These are particularly useful, as they indicate to the attacker whether the version of commercial or open source software used contains a published known vulnerability. For example, embedded SQL queries in database applications should be checked to ensure they do not refer to a specific RDBMS version, and Web-based code should be checked for information about the brand name and versions of development tools used to produce, or runtime-interpret or compile the code.

G.6.1.1.2. Inadequate or Inaccurate of Input Validation

Validation of both unintentionally or maliciously malformed input is the most effective way to prevent buffer overflows. Programs written in C or C++ in particular should be checked to ensure they contain correctly implemented input validation routines that prevent incorrect program responses to input with unexpected sizes or formats.

G.6.1.1.3. Server-side Reliance on Client-side Data or Security Functions

Server-side reliance on client-originated data can make server applications vulnerable to attacks in which client-side data is altered before or in transit from client to server to compromise server-side security properties or

functions, e.g., to subvert authentication checking or to gain access to confidential server-side data. Security analysis should be performed to ensure that server-side software validates all data originating from a client, even if the client validated that data first.

G.6.1.1.4. Insecurity of or Excessive Reliance on Environment-Level Interfaces

Attackers often attempt to impersonate application-level and environment-level functions, inputs, and outputs to reference other application-level functions, resources, and data stores. Redirection methods and messaging functions may also be targeted. Insecure interactions are those that are undertaken without the interacting entities establishing:

- One another's identity;
- The trustworthiness of the mechanism and path by which the interaction will occur.

To the extent possible, the software should be designed to be self-contained, with its dependency on its execution environment for ensuring its correct, secure operation kept to a minimum. Even if environment "defense in depth" measures are used to protect the software from attack, the software itself should contain necessary error and exception handling logic to ensure its security in the case that an attack manages to breach the environment-level protections. Refer to Appendix G:G.4.3 for information on secure interactions between application-level and environment-level software.

G.6.1.1.5. Logic that Enables Unauthorized Permission Escalation

Successful attempts to reference software that has higher server-side permissions, or to exploit race conditions, can enable the attacker to identify lax permission enforcement or authentication checking. Logic that can be exploited to accomplish any of these events can be detected by examining code that has higher permissions than those of the examiner.

G.6.1.1.6. Data Delivery Methods that are Vulnerable to Subversion

Attackers often attempt to subvert data in transit from client to server (or from consumer to provider), and to analyze server/provider responses to the subverted data. Their objective is to determine whether input validation checking by the software prevents the subverted data from being accepted. Subversive data delivery methods include replay attacks and other session-oriented attacks (e.g., hijacking).

G.6.1.1.7. Security Functions/Methods that are Vulnerable to Subversion

The robustness of the software's resistance to subversions of its security methods is critical. These include subversions that attempt to:

- Bypass system security processes;
- Impersonate a valid, authenticated user (human or process);
- Access another user's functions or data;
- Access functions or data for which the user has insufficient privileges.

The system's user segregation methods and server-side/provider-side responses to failed authentication and access attempts are two areas in which exploitable faults/weaknesses frequently occur.

G.6.1.1.8. Inadequate Handling of Security Faults

The overall robustness of a well-designed system is partially predicated by the robustness of its security handling procedures at the code or programming language level. C++ and Java, for example, inherently provide convenient

and extensible exception handling support which includes "catching" exceptions (faults that are not necessarily caused by flaws in externally-sourced inputs or violations of software-level constraints) and errors (faults that are caused by external inputs or constraint violations; errors may or may not trigger exceptions).

Developers who "reactively extend" native exception handlers, e.g., by pasting exception classes into source code after an error or fault has been observed in the program's testing or operation, need to be careful not to rely solely on this approach to exception handling, otherwise the result will be a program that will fail to capture and handle any exceptions that have not yet been thrown during testing or operation. Exception handling should be proactively designed to the greatest extent possible, through careful examination of code constraints as they occur during the concept and implementation phases of the life cycle.

The developer should list all predictable faults (exceptions and errors) that could occur during software execution, and define how the software will handle each of them. In addition, address how the software will behave if confronted with an unanticipated fault or error. In some cases, potential faults may be preempted in the design phase, particularly if the software has been subjected to sufficiently comprehensive threat modeling, while developers could preempt additional faults during pseudo-coding by cautiously examining the logical relationships between software objects and developing "pseudo" exception handling routines to manage these faults.

G.6.1.1.9. Insecure Installation

To be effective, the software's installation-time configuration parameters should be as restrictive as possible, to make sure the software is as resistant as possible to anticipated attacks and exploits. Installation documentation should stress use of secure procedures, not just initially, but for every component installed after deployment (e.g., patches, updates, additional software). If the option is supported, the operating system should be configured to require explicit administrator or user approval before any software component is able to be installed.

G.6.1.1.10. Other Security Issues

Some other shortcomings in software that can be exploited by attackers include:

- Lack of encoding on dynamic content sent to users;
- Poorly-implemented native code use and dynamic code use;
- Non-standard or poor entropy for encryption/decryption;
- Poorly implemented security functions, such as cryptographic, authentication, and secure session management functions.

G.6.1.2. Using Attack Patterns in Software Security Testing

Software receives input from many different sources. Software applications, for example, receive input from human users, operating-system kernels, file systems, middleware services, and other applications. By duplicating the attack patterns that are typically associated with different interfaces to the software under examination, the tester may uncover many vulnerabilities in that software. From a testing point of view, the most revealing attack patterns are those that involve receipt of inputs through multiple interfaces.

During test planning, attack patterns should be investigated and used in defining the test case scenarios to be exercised. Further, attack patterns should form the basis for much of the specific input that will be used in black box fault injection and penetration tests.

Information on real world attack patterns can be gleaned from the information published by bug tracking databases (such as BugTraq), security incident reporting services (such as the U.S. CERT, Purdue University

CERIAS CERT, and others), and vulnerability advisory services (such as those listed earlier in Table 2-2). More information on the use of attack patterns in security testing can be found on the DHS BuildSecurityIn portal.

G.6.2. Timing of Security Tests

Software security testing has traditionally been performed only after the software has been fully implemented and, in many cases, deployed. Such testing entails a "tiger team" attempting to break into the installed system by exploiting well-known vulnerabilities, and after a successful penetration documenting the likely causes of vulnerabilities so the software can be reconfigured, improper file permissions can be reset, or patches can be applied.

The problem with this approach—and specifically the timing of it—is that it happens too late in the software life cycle, enabling attackers to stay a step ahead. Patches, in particular, are problematical: They are often ignored or postponed by administrators (this is particularly true of patches for commercial and open source software functions not used in the software system). While they may fix some vulnerabilities, they can have an adverse effect on the patched software's functionality, and can introduce new vulnerabilities or re-expose old ones.

Software engineering analysis and testing techniques used by the software safety community have revealed that a large proportion of software faults result from human mistakes made during the software's requirements, design, and implementation phases, and thus are likely not to be effectively mitigated by installation-time or maintenance-time countermeasures. Additionally, security mistakes are likely to be repeated throughout the affected software artifacts when those mistakes are not recognized and understood by the artifacts' creators. For this reason, security analysis and testing must begin far earlier in the software development life cycle.

Some late-stage "tiger team" security testing can be useful, either in determining whether the fully implemented, integrated software is ready to ship, or whether the deployed software is ready for Security Acceptance. Recognizing that such late-stage testing is necessarily constrained to identifying patterns of residual security vulnerabilities to remediate them—with the predictable high impact on cost and schedule that late-stage changes represent—before the software is distributed or installed.

Security testing is best performed throughout the software system's life cycle. However, much of today's software is deployed with little or no security testing at any point in the life cycle. For such software, even after it has gone into production, use of techniques discussed in Appendix G:G.6.3 and G.6.4 for white-box and black-box testing can provide valuable insight into the residual vulnerabilities in the software that may be mitigated through application security measures, wrappers, or patching. Indeed, the results of other security reviews and tests, such as risk analysis of new or changed requirements and post-incident forensic analysis, can help developers effectively identify and prioritize the defense in depth protections that can be applied to software in production to reduce the likelihood that residual faults will be exploited and/or to minimize the damage if they are.

This said, the most effective approach to security testing will begin identifying security problems as early in the life cycle as possible, where they can be remediated with the lowest impact on schedule and cost. On the other hand, late-stage "tiger team" testing is better to no security testing at all, which will result in release of vulnerable software. Security testing based on defined security use cases, misuse cases, and abuse cases should be performed iteratively throughout the development life cycle.

Figure 5-2 shows a suggested distribution of different security test techniques throughout various life cycle phases.

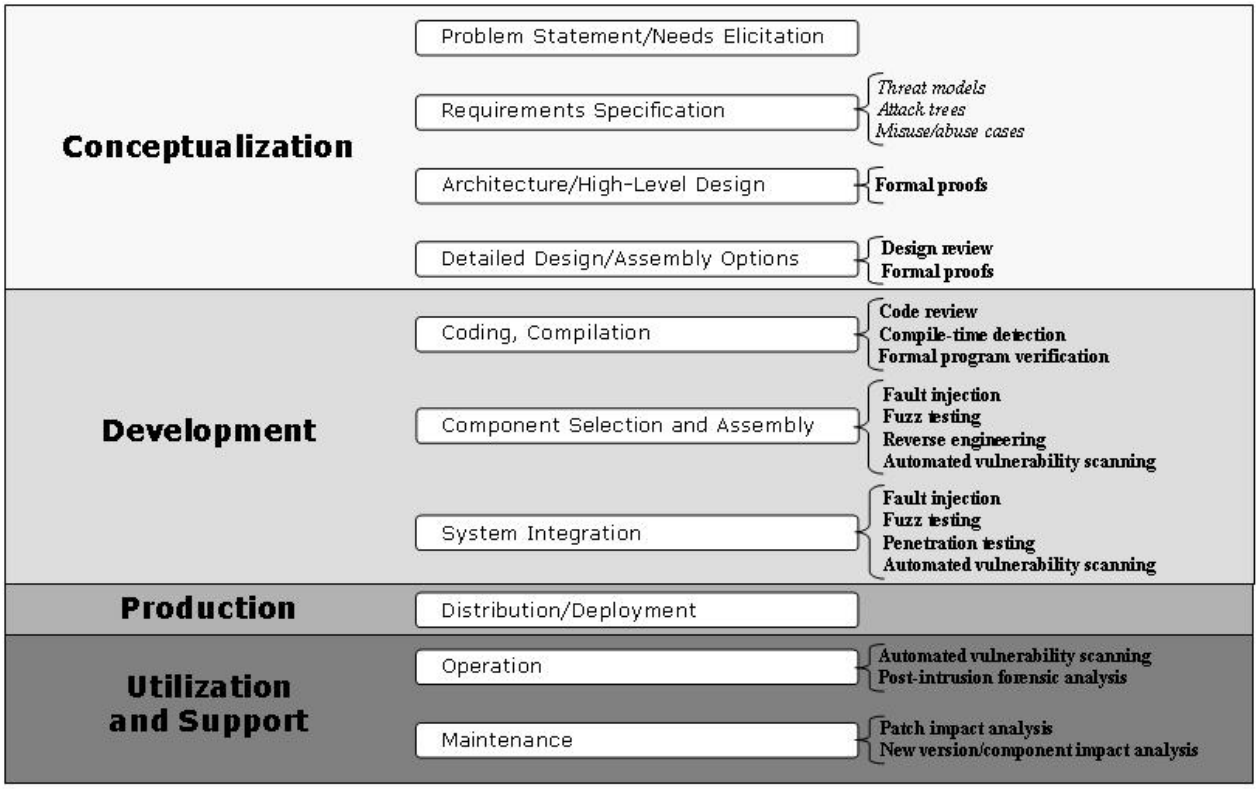

Figure 5-2. Suggested Distribution of Security Test Techniques in the Software Life Cycle

The distribution of security tests throughout the life cycle includes:

- Security reviews of requirements, architecture, design specifications, and documentation of development process and controls;
- Source code security review of from-scratch modules, open source software (the latter as part of the evaluation of that software), and, whenever possible, outsourced commercial software and reused components. Third-party code reviewers may be used when in-house expertise is unavailable or to review outsourced components;
- "Black box" security analysis and testing of binary acquired or reused software under consideration for assembly/integration, as part of the software's evaluation and selection process;
- Assembly/integration option prototype testing to demonstrate secure behavior of components under various design options, and to determine the security of interfaces between components, and between the integration/assembly and external entities in the execution environment. The test results should enable a final determination of the best assembly option to be made.
- Integration security testing of the final selected and implemented integration/ assembly option. The security testing at this stage is in addition to the functional testing of the software system's security functions, which is properly a subset of the system's overall functional correctness testing (i.e., testing of the correct operation and behavior of all trusted functions, security functions, and embedded security protections). Integration security testing should occur within a test environment that to the greatest extent possible mirrors the target operational environment in which the software will be installed and run, in terms of the presence of all anticipated execution environment and infrastructure components, and realistic representative test data.

The system's Test Plan should incorporate a Security Acceptance Plan, which contains suites of security test cases (including misuse/abuse cases), and defines:

1. Test data and test oracle (if one is to be used) necessary to fully demonstrate that the system is ready to go into production use;

2. Testing tools (static and dynamic) to be used and the analyses of tool output to be performed.

In preparation for testing, the software, test plan, test data, and test oracles (if used) are migrated from the development environment into a separate, isolated test environment. All security test cases should be run to ensure the adherence of the assembled/integrated system to all of its security requirements (including those for security properties/attributes, secure behaviors, self-protecting characteristics, and not just security functionality). Particular attention should be paid to the security of interfaces within the software system, and between the system and external (environment and user) entities.

G.6.2.1. Security Test and Evaluation (ST&E)

If the system of which the software is a component is to be certified and accredited, the Security Acceptance Plan will be a prerequisite of the ST&E phase of government Certification and Accreditation processes and of many non-government system security auditing processes, and its format and contents will be defined, in some cases along with the specific test techniques to be used, in the C&A methodology governing the system, e.g., FISMA, NIST, NIACAP, DITSCAP, DIACAP, DCID 6/3, or another C&A or system security auditing process.

ST&E in support of security C&A is performed on the installed system of which the software is a part. All of the relevant measures for preparing the software for secure deployment should be addressed before the ST&E begins. The process used to achieve the system's security acceptance or approval to operate will culminate in a risk analysis. This risk analysis will be used to identify any additional safeguards or countermeasures that may need to be applied to remediate or mitigate the impact of security deficiencies found during the ST&E, and to establish a risk baseline for the system. This risk baseline will be compared against the risk analysis results of the next iteration of the system, produced during the maintenance phase.

All test cases in the ST&E's Security Test Plan are run, and the appropriate stakeholder (user representative or system certifier) reviews the test results to verify that the test suite has been executed with satisfactory results. Once the system is approved to operate and the user accepts delivery, all artifacts of the accepted system, including software source code and binaries, documentation, and acceptance test results are "locked" in the CM system and archived for future reference during system maintenance.

G.6.2.2. Configuration Management and Software Testing

Ongoing post-deployment vulnerability scanning and penetration testing of the operational system is performed as part of the software's ongoing risk management. Other security tests, in support of impact analyses for security patches, updates, new releases, and component substitutions also occur after deployment, as part of the software's overall maintenance plan. The results of each test should be checked into the CM system as soon as the test is completed. When an individual component (unit) or the whole system has completed each test phase, it should also be checked into the CM system.

G.6.3. "White Box" Security Test Techniques

The following tests and reviews are performed on source code. These techniques should be performed as early and as often in the life cycle as possible to allow time to correct a small code unit before it is integrated into the larger code base.

White box tests are also more effective when done first on granularly small components, e.g., individual modules or functional-process units, etc., then later on the whole software system. Iterating the tests in this way increases

the likelihood that faults within the smaller components will be detected during the first set of tests, so that the final whole-system code review can focus on the "seams" between code units, representing the relationships among and interfaces between components.

White box tests should not be limited to from-scratch code, but should also be performed on all open source components used in the software system. Efforts should be made to request source code reviews—or results from such reviews—from outsourced or other commercial components.

G.6.3.1. Code Security Review

A code review, or audit, investigates the coding practices used in the software. The main objective of such reviews is to discover security faults and potentially identify fixes. Because source code security reviews depend on the expertise of the reviewers, they are best performed by security experts on the development team, the organization's risk management or accreditation team, or an expert independent entity. To reduce required review time, reviews are often performed only on the code that risk analysis identifies as the most vulnerable.

The techniques for performing code security reviews range from manual to fully-automated. In a manual review, the reviewer inspects all code without the assistance of automated code review tools, until faults and other vulnerabilities are found. This is a highly labor-intensive activity, and tends to produce its most complete, accurate results very early in the process, before reviewer fatigue sets in. In a fully-automated review, the automated code review tool performs all of the code inspection, and the reviewer's job is to interpret the results.

The problem with the fully automated review is that it can be only as complete as the list of patterns the tool is programmed to scan for. Also, automated code review is unable to identify all vulnerabilities in relationships between different sections of code, but tools are beginning to support identifying these relationships. Between manual and fully-automated code review lies semi-automated review, in which the reviewer uses automated tools to assist in the manual inspection. These tools locate those portions of the code that contain certain patterns, enabling the reviewer to narrow the focus of manual inspection on the tool-highlighted portions of the code. Additionally, automated tools can scan very large code bases in a short time when compared with manual review, while providing consistent results and metrics.

Code review can also focus on suspected malicious code to detect and pinpoint signs and locations of malicious logic. For suspected malicious code written in shell script, reviewers should look for:

- Presence of comments indicating that the code is exploit code;
- Code that allows for logging into the host;
- Code that changes file ownership or permissions.

For suspected malicious code written in C, reviewers should look for comments indicating exploit features, and/or portions of the code that are complex and hard-to-follow, including embedded assembler code (e.g., the *_asm_* feature, strings of hexadecimal or octal characters/values). Submissions to the International Obfuscated C Code Contest (IOCCC) and the Underhanded C Contest from Binghamton University show how complex source code can be written such that even a skilled reviewer may not be able to determine its true purpose.

G.6.3.1.1. Direct Code Analysis

Direct code analysis extends manual code review by using tools that focus on predefined security property requirements, such as non-interference and separatability, *persistent_BNDC*, non-inference, forward-correctability, and non-deducibility on outputs. One of drawbacks of direct code analysis is that it requires a high level of resources and is not particularly scalable, so it is best performed during the acquired or reused software evaluation phase rather than later in the life cycle.

There are two categories of direct code analysis: static and dynamic. In static analysis, the source code and/or executable is examined without being executed to detect insecure code by analyzing all possible executions rather than just test cases—one of the benefits of this is that static analysis can be performed prior to compilation or installation. Dynamic analysis attempts to discover errors and failures during execution by monitoring the variables.

G.6.3.1.2. Property-Based Testing

Property-based testing is a technique that comes from formal methods. It narrowly examines certain properties of source code, such as security properties to prove that no predictable vulnerabilities exist, and to determine which assumptions about the program's behavior are correct. Property-based testing is usually performed after software functionality has been established, and includes analysis and inspection of the software's requirements and design specifications and implemented source code to detect vulnerabilities in the implemented software.

Property-based testing is usually limited to examination of the small subset of the overall code base that implements the software's trusted functions; this limitation in focus is necessary because property-based testing requires complete, systematic analysis of the code to validate all of its security properties. Furthermore, to be effective, property-based testing must also verify that the tests themselves are complete.

G.6.3.2. Source Code Fault Injection

Source code fault injection "instruments" software source code by non-intrusively inserting changes into a program, then compiling and executing the instrumented program to observe how its state changes when the instrumented portions of code are executed. In this way, source code fault injection can be used to determine and even quantify how software behaves when it is forced into anomalous circumstances, including those instigated by intentional faults. It is particularly useful in detecting incorrect use of pointers and arrays, use of dangerous calls, and race conditions.

Source code fault injection is most effective when used iteratively throughout the code implementation process. When new threats (attack types and intrusion techniques) are discovered, the source code should be re-instrumented with faults representative of those new threat types.

G.6.3.2.1. Fault Propagation Analysis

Fault propagation analysis involves two techniques for fault injection testing of source code, extended propagation analysis and interface propagation analysis. The objective of both techniques is not only to observe individual state changes as a result of faults, but to trace the propagation of state changes throughout the code tree that result from any given fault. In order to perform fault propagation analysis, a fault tree must be generated from the program's source code.

- Extended propagation analysis entails injecting faults into the fault tree, then tracing how the injected fault propagates through the tree, in order to extrapolate outward and anticipate the overall impact a particular fault may have on the behavior of the whole software module, component, or system.

- In interface propagation analysis, the focus is shifted from perturbing the source code of the module or component itself to perturbing the states that propagate via the interfaces between the module/component and other application-level and environment-level components. As with source code fault injection, in interface propagation analysis anomalies are injected into the data feeds between components, enabling the tester to view how the resulting faults propagate and to discovery whether any new anomalies result. In addition, interface propagation analysis enables the tester to determine how a failure of one component may affect neighboring components, a particularly important determination to make for components that either provide protections to or rely on protections from others.

G.6.3.3. Compile-Time and Runtime Defect Detection

Compile-time detection and runtime detection rely on the compiler to detect and flag, or in some cases eliminate, faults in the code that were not detected during code review and that could make the compiled software vulnerable to compromises. A simple version of compile-time detection occurs in all basic compilers: type checking and related program analysis. While these checks prove useful for detecting simple faults, they are not extensive or sophisticated enough to detect more complex vulnerabilities.

By contrast, some compilers include extensions to perform full program verification to prove complex security properties; these verifications are based on formal specifications that must be generated prior to compilation. Program verification is most often used to detect errors or "dangerous usages" in C and C++ programs and libraries, such as usages that leave the program vulnerable to format string attacks or buffer overflows. Some compile time verification tools leverage type qualifiers. These qualifiers annotate programs so that the program can be formally verified to be free of recognizable vulnerabilities. Some of these qualifiers are language-independent and focus on detecting "unsafe" system calls that must be examined by the developer; other tools detect language-specific vulnerabilities (e.g., use of buffer overflow prone library functions such as *printf* in C).

Still other tools perform "taint analysis", which flags input data as "tainted" and ensures that all such data are validated before allowing them to be used in vulnerable functions.

G.6.4. "Black Box" Security Test Techniques

"Black box" tests are performed on the binary executable of the software, and in essence "poke at" the software from the outside to observe its outputs and behaviors, rather than examine the actual code that implements it. For acquired and reused black box components, black box tests are the only tests available. However, they are also valuable for testing executables (from-scratch or open source) compiled from source code, because they enable the tester to see the software "in action", to confirm assumptions made during white box testing about a given component's behavior when it receives various inputs from other components, its environment, or its users. Black box testing also enables the tester to observe the software's behavior in response to a wide variety of user inputs and environment configuration and state changes, including those that indicate potential attack patterns. These kinds of interactions between the software and its external environment and users are impossible to achieve during white box testing: in code review they can only be imagined, and in fault injection only roughly simulated.

G.6.4.1. Software Penetration Testing

In software penetration testing, testers target individual binary components or the software system as a whole to determine whether intra or inter-component vulnerabilities can be exploited to compromise the software system, its data, or its environment resources. Penetration testing may reveal critical security vulnerabilities overlooked in functional testing. To be effective, penetration testing needs to be more extensive and complex than less sophisticated (and less costly) black box security testing techniques, such as fault injection, fuzzing, and vulnerability scanning, that would provide much the same results.

The penetration test scenarios should focus on targeting potential security problems in the software's design, and should include tests that reproduce the threat vectors (attacks, intrusions) determined to be either most likely or most damaging by the software's risk analysis, as well as worst-case scenarios, such as a hostile authorized user. Ideally, the test planner will have access to the requirements and design specifications, implementation documentation, source code (when available), user and administrator manuals, and hardware diagrams.

G.6.4.2. Security Fault Injection of Binary Executables

Fault injection of binary executables was originally developed by the software safety community to reveal safety-threatening faults that could not be found through traditional testing techniques. Safety fault injection is used to induce stress in the software, create interoperability problems among components, and simulate faults in the

execution environment. Security fault injection uses similar techniques to simulate faults that would result from intentional attacks on the software, and from unintentional faults that could be exploited by an attacker.

Security fault injection is most useful as an adjunct to security penetration testing, enabling the tester to obtain a more complete picture of how the software responds to attack. Security fault injection involves data perturbation, i.e., alteration of the type of data the execution environment components pass to the software system, or that the software system's components pass to one another. Fault injection can reveal the effects of security faults on the behavior of the components themselves and on the software system as a whole.

Environmental faults in particular are useful to simulate because they are most likely to reflect real-world attack scenarios. However, injected faults should not be limited to those simulating real-world attacks. As with penetration testing, the fault injection scenarios exercised should be designed to give the tester as complete as possible an understanding of the security of the behaviors, states, and security properties of the software system under all possible operating conditions.

G.6.4.3. Fuzz Testing

Fuzz testing is similar to fault injection in that invalid data is input into the software system via the environment, or input by one process into another process. Fuzz testing is implemented by tools called fuzzers, which are programs or script that submit some combination of inputs to the test target to reveal how it responds. Fuzzers are generally specific to particular types of input. The fuzzer provides the software with semi-random input, usually created by modifying valid input. As with the other test techniques, effective fuzz testing requires the tester to have a thorough understanding of the software system targeted and how it interfaces with its environment. Because most fuzzers are written to be used with a specific program, they are not easily reusable. However, their value lies in their specificity: they can often detect security faults that more generic tools such as vulnerability scanners and fault injectors cannot.

G.6.4.4. Reverse Engineering Tests: Disassembly and Decompilation

Reverse engineering of binary executables is achieved through disassembly or decompilation. In disassembly, the assembler code is reconstructed from the binary executable to allow the tester to find the types of security-relevant coding errors and vulnerabilities that can be detected in assembler-level code. Disassembly is difficult because the engineer must have a thorough understanding of the specific assembler language generated by the disassembler, and in the fact that it is far more difficult for a human reviewer to recognize security faults in assembly language than in higher level languages.

By contrast with disassembly, decompilation generates high-level source code from the executable binary. The decompiled source code can be subjected to a security code review and other white box tests. However, the source code generated through decompilation is rarely as navigable or comprehensible as the original source code; thus security code review of decompiled code will be significantly more difficult and time consuming than review of the original source code.

Both techniques are likely to be practical only when strictly limited to trusted or other very high-value software that are considered to be at very high risk (in terms of likelihood of being targeted by attack, or under high suspicion of containing critical vulnerabilities, e.g., because of suspicious pedigree). Many commercial software products use obfuscation techniques to deter reverse-engineering; such techniques can increase the level of effort required for disassembly/decompilation testing, making such techniques impractical. In other cases, the commercial software's distribution license may explicitly prohibit reverse-engineering.

G.6.4.5. Automated Application Vulnerability Scanning

Automated application vulnerability scanning employs commercial or open source scanners that search the executing application for behaviors and input/output patterns that match patterns associated with known vulnerabilities stored in the scanning tool's database of vulnerability "signatures" (comparable to virus signatures

in virus scanning tools). The vulnerability scanner is, in essence, an automated pattern-matching tool, similar in concept to an automated code review tool. And like the automated code review tool, it is difficult for vulnerability scanners to either to take into account the increased risk associated with aggregations of individual vulnerabilities, or to identify vulnerabilities that result from particular unpredictable combinations of behaviors or inputs/outputs. However, many vulnerability scanners attempt to provide some mechanism for aggregating vulnerabilities.

In addition to signature-based scanning, many Web application vulnerability scanners attempt to mimic techniques used by attackers to "probe" a Web application for vulnerabilities—in a manner similar to fuzz testers. This technique is often referred to as automated stateful application assessment. These scanners can provide a range of service, from using test user log-in information for determining whether access policies are being appropriately followed to supplying invalid and/or malicious input to the application in an attempt to subvert it. Like signature-based scanners, these scanners can detect only known classes of attacks and may not be able to take into account the increased risks associated with aggregations of individual vulnerabilities.

Vulnerability scanners can be either network- or host-based. Network-based scanners run remotely from the target application over a network, while host-based scanners must be installed locally on the same machine as the target application. Host-based scanners are generally able to perform more sophisticated analyses, such as verification of secure configurations. While the scanning process is automated, because the tools usually have a high false positive rate, the tester must have enough application development and security expertise to meaningfully interpret the results and identify the true vulnerabilities.

As with virus scanners, these tools are based on whatever vulnerability signatures the supplier is aware of at the time the tester purchased/obtained the tool. Thus, the tester must be vigilant about frequently downloading updates to the vulnerability signature database (two important evaluation criteria when selecting a vulnerability scanner are how extensive the tool's signature database is, and how often the supplier issues updates). In some cases, vulnerability scanners provide information and guidance on how to mitigate the vulnerabilities detected by the tool.

Vulnerability scanners are most effectively used in the initial security assessment of acquired or reused binary software, to reveal common vulnerabilities that are unfortunately prevalent in commercial software. In addition, vulnerability scanning prior to application penetration testing can be useful to ensure that the application does not include straightforward common vulnerabilities, thus eliminating the need to run penetration test scenarios that check for such obvious vulnerabilities.

G.6.5. Forensic Security Analysis

Forensic security analysis is performed after a deployed executable software has been compromised, to determine what vulnerabilities in the software's functions or interfaces were exploited by the attacker. Unlike pre-deployment security testing, the focus of forensic analysis is to identify and analyze specific proven vulnerabilities, rather than to search for faults that may or may not exist. Forensic analysis comprises three different analyses: intra-component, inter-component, and extra-component. Intra-component forensic analysis is used when the exploited vulnerability is suspected to lie within the component itself.

The tools supporting this analysis provide the analyst with static and dynamic visibility into the behaviors and states of the component (including the source code, if available), to pinpoint where the vulnerability may lie. Inter-component forensics are used when the location of the vulnerability is suspected to lie in the interface between two components. The supporting tools for such an analysis provide insight into the communication or programmatic interface mechanisms and protocols used between the components, and also reveal any incompatibilities between the implementation of those mechanisms/protocols from component to component.

Extra-component analysis is used when the vulnerability is suspected to lie in the whole system's behavior or in some aspect of its environment. Supporting tools for this type of analysis provide insight into the audit and event logs for the software system and its surrounding environment. Records of system-level security-relevant behavior

are also analyzed to reveal points of vulnerability in the software's configuration and interaction with its environment that appear to have been targeted by the attacker.

G.6.6. Software Security Testing Tools

Software security testing tools are designed to help the tester verify that software programs:

1. Have been implemented in strict conformance with their specifications, including performing all of their specified security functions; and

2. Operate correctly and securely, i.e., resist attempted compromises or exploits targeting expected (by the attacker) vulnerabilities, or attempts by attackers to discover unknown vulnerabilities.

Most security testing tools implement a particular testing technique or method, such as "white box" (also known as "clear box" or "crystal box") code review, and "black box" vulnerability scanning, penetrating testing, and security-oriented fault injection. Unfortunately, no single tool exists that can automatically, comprehensively assess the security of a software application, service, or program. Current commercial and open source vulnerability assessment, security fault injection, and code security review tools find only a portion of common basic faults in non-complex software systems, source code files, and execution environments, and provide general advice on better coding and deployment practices. Such tools are probably most useful during the initial security assessment of acquired or reused software and of the various component assembly options, as well as during the security testing of the whole application.

Typically, security testing tools are designed to support in a particular phase, technique, or method of testing. Tools specifically geared towards application-level software testing should be augmented by tools for verifying the security (lack of vulnerabilities, secure configuration) of the software's execution environment, including the environment's middleware (database management systems, application frameworks, Web servers, etc.), operating system, and networking components. This is particularly important to determine whether a security vulnerability revealed through dynamic security analysis of executing software (versus static analysis of code) originated in the software system itself or in its environment.

The main categories of software and application security testing tools most widely available, both commercially and open source, are listed below.

- Compiler error checking and safety enforcement tools
- Formal verification tools, including model checkers, theorem provers, constraint-based and static analysis tools, and property-based testers
- Manual or semi-automated code review assistants, source code scanners, dynamic code analysis tools
- Source code fault injectors
- Byte code scanners
- Reverse engineering tools, including decompilers, disassemblers, and binary scanners
- Binary fault injectors
- Fuzzers
- Penetration testing tools
- Execution environment scanners, including network, operating system, database, and Web server scanners
- Application scanners

- Source code fault injection tools
- Other relevant tools, such as brute force testers, buffer overrun detectors, and input validation checkers.

For listings and descriptions of specific commercial and open source tools in each category, the reader is encouraged to visit the NIST Software Assurance Metrics and Tool Evaluation (SAMATE) Website. The NIST SAMATE Project, funded by DHS, is chartered to is evaluate of a wide range of software security testing tools, measuring the effectiveness of those tools, and identifying gaps in tools and methods. One of the artifacts of SAMATE is an evolving taxonomy for organizing and categorizing software tools for evaluation purposes. This taxonomy will also support the development of detailed specifications by tool suppliers of tool classes and functions. The SAMATE taxonomy includes tools used across the software life cycle, including:

- Conceptualization/business needs analysis (e.g., use cases, changes to current systems);
- Requirements and design (e.g., consistency, completeness, and compliance checking);
- Implementation;
- Assessment and acceptance (internal and external);
- Operation (e.g., operator training, auditing, penetration testing).

Information on the metrics activities of the SAMATE program appears in Appendix G:G.6.7.

G.6.6.1. NSA Center for Assured Software

The NSA Center for Assured Software (CAS) was established in November 2005 as a focal point for Software Assurance issues in the DoD. With the overarching mission of eliminating exploitable vulnerabilities from critical DoD information systems, the CAS is spearheading collaboration efforts across government and academia to research, coordinate, and prioritize software assurance activities throughout the industry.

In addition to a broader collaboration role, the CAS is attempting to establish methodologies and leverage tools for comprehensively evaluating software trustworthiness, including COTS and government-developed software utilized in critical deployed DoD systems and networks. For IA-enabled products, the CAS specifically evaluates software through the National Information Assurance Partnership (NIAP) Common Criteria Evaluation and Validation Scheme (CCEVS). As a result of its evaluation activities, the CAS hopes to offer recommendations to the DoD with respect to changes in policy and standards that may permanently improve or enhance the level of assurance of commercial or government-developed software.

In order to produce a consistently repeatable and reasonably accurate process for evaluating software, the CAS is searching for properties common to all software that provide a degree of indication of levels of assurance, including degree of confidence that software:

- Will securely and appropriately perform its intended functions;
- Will not perform any unauthorized functions;
- Does not contain implementation flaws that could be exploited.

The CAS software evaluation process has been defined to include five distinct phases:

1. **Acceptance:** Determine whether tools or techniques exist for evaluating the software under evaluation;
2. **Extraction/Inspection:** Apply tools and techniques that extract relevant metadata from the software;
3. **Analysis:** Apply tools and techniques that query the extracted metadata for indicators of assurance (i.e. assurance properties);

4. **Meta-analysis:** Integrate outputs from analytical tools to rank the relevance of the identified indicators;

5. **Reporting:** Transform analytical results into comprehensible reports.

The CAS software evaluation process is still evolving. The CAS program plans to establish a public Web portal in the near future.

G.6.6.2. NASA Software Security Assessment Instrument

The open source National Aeronautics and Space Administration (NASA) Software Security Assessment Instrument (SSAI) was developed by engineers from NASA and from University of California at Davis (UC-Davis), under the auspices of the Reducing Software Security Risk (RSSR) program (November 2000-December 2005) at the NASA Jet Propulsion Laboratory (JPL) Independent Verification and Validation (IV&V) Center.

The SSAI verifies and validates the security of software and the processes by which it is developed and maintained. The key elements of the SSAI are:

- Software Security Checklists for verifying the security of development and maintenance phase activities of the software life cycle and of activities associated with the external release of software;

- Property-Based Tester (PBT);

- Flexible Modeling Framework (FMF) for modeling security specifications;

- Training slides on software security for managers, software system engineers, and developers.

The SSAI has been successfully used by NASA's IV&V Center with the cooperation of PatchLink Corporation to verify the security properties of the PatchLink Corporation Java-based Unix Agent. The findings of that security assessment are being used by PatchLink to improve the security of their product. To obtain a copy of the SSAI, send email to Kenneth McGill, RSSR Research Lead at the NASA JPL IV&V Center at: Kenneth.G.McGill@nasa.gov.

G.6.7. Software Security Metrics

[This is a placeholder for a discussion of software security metrics, to appear in a future revision of this document.]

G.6.7.1. NIST Software Assurance Metrics and Tool Evaluation Project

The DHS-sponsored NIST SAMATE Project was established with two primary objectives:

1. Develop metrics to gauge the effectiveness of existing software assurance tools;

2. Assess current software assurance methodologies and tools to identify deficiencies that may introduce software vulnerabilities or contribute to software failures.

Specifically, NIST (i.e. SAMATE) was tasked by DHS to address concerns related to assessing and establishing "confidence" in software products. This confidence would be achieved through use of well-established metrics to quantify the level of assurance of a given software product with respect to its security, correctness of operation, and robustness. The software products to be assessed include the software assurance tools used to develop, test, deploy, and support other software products. Measurement of confidence of these tools would focus on assessing the accuracy of the tools' report output and quantifying the incidence of false positives and negatives. Because oif the variability of features and capabilities across tools, SAMATE is also defining a standard testing methodology that will provide a structured and repeatable baseline for evaluating software tool effectiveness.

As noted in Appendix G:G6.6, central to SAMATE's roadmap and guiding philosophy is the notion of a robust taxonomy for defining classes of tool functions. To ensure maximum vendor community buy-in, the SAMATE Project has defined a roadmap for producing their planned artifacts. This roadmap incorporates community involvement through multiple communications means, including workshops and an email group (listserv) for helping tool developers, researchers, and users prioritize particular tool functions and to define metrics for measuring the effectiveness of these functions.

Artifacts of the SAMATE Project will include:

- Taxonomy of software weaknesses, faults, and vulnerabilities;
- Taxonomy and survey of software assurance tools, including tool functional specifications;
- Reference data set, metrics, and measurements for use in the evaluation of software assurance tools;
- Detailed functional tests for tool evaluation;
- Identification of tool functional gaps and research requirements.

These and other artifacts of the SAMATE program will be posted to the SAMATE Website at

G.6.8. Outsourcing of Software Security Testing

Increasingly, organizations are offering security testing as a service. Sometimes referred as "red-teaming" or "tiger-teaming", independent testing is useful as the reviewer is not biased in how the system operates—potentially discovering vulnerabilities that may be overlooked by a reviewer closely involved with developing the system. In a similar vein, vulnerabilities may be overlooked because the independent reviewer ignores functionality that may not be documented or obvious to an outside observer. Security testing services are available at all points in the life cycle.

When using third-parties to perform independent testing, it may be important to consider whether they are needed at all points in the life cycle. If not, consider which points they may be most useful. While important vulnerabilities may be discovered early on, it may be more cost effective to use an in-house security expert and/or train some or all developers in software security concepts. Regardless, outsourcing security testing provides an independent observers who are well-trained in software security concept a chance to uncover vulnerabilities that would otherwise go unnoticed. In addition, these observers can also greatly enhance internal security reviews to ensure that the software has as few vulnerabilities as possible.

G.7. Preparing the Software for Distribution and Deployment

All software artifacts and initial production data should be "cleaned up" as necessary to remove any residual debugging "hooks", developer "backdoors", sensitive comments in code, overly informative error messages, etc. that may have been overlooked during the implementation phase (note that as secure development practices become more deeply ingrained, such items will not be introduced into the code in the first place), and to change any default configuration settings, etc., not already addressed prior to whole-system testing. All software should be distributed in a default configuration that is as secure as possible, along with a set of secure configuration instructions that explain the risk associated with a change to each secure default when the software is deployed.

The installation configuration for the software should be different from the development environment parameters, to prevent developers from having visibility into the details of the deployed software in its operational environment.

Before delivering the software to the deployment team or acceptance test team, the developer should undertake the activities described below to ensure the software is as secure as possible when preparing it for distribution/shipment and installation/deployment.

G.7.1. Removing Debugger Hooks and Other Developer Backdoors

Before deploying the software operationally, the developer should remove from the code all developer backdoors and debug commands, and change all default settings for all components (source or binary).

Debug commands come in two distinct forms: explicit and implicit. The developer must learn to recognize both types of debug commands and to remove them from all source code, whether from-scratch or open source, before the binary is compiled.

For acquired or reused binary components, as part of the component's security evaluation, the integrator may wish to ask the supplier whether such a scan was performed on the component's source code before it was compiled and distributed as a binary executable.

G.7.1.1. Explicit Debugger Commands

All code should be checked before the software is installed to ensure that it does not contain commands designed to force the software into debug mode.

Web applications should be implemented to validate the content of name-value pairs within the URL or Uniform Resource Identifier (URI) submitted by the user, as such URLs/URIs sometimes include embedded commands such as *debug=on* or *Debug=YES*. For example, consider the following URI:

```
http://www.creditunion.gov/account_check?ID=8327dsddi8qjgqllkjdlas& Disp=no
```

An attacker may intercept and alter this URI as follows:

```
http://www.creditunion.gov/account_check?debug=on& ID=8327dsddi8qjgqllkjdlas&Disp=no
```

with the result that the inserted "debug=on" command would force the application into debug mode, enabling the attacker to observe its behavior more closely, to discover exploitable faults or other weaknesses.

Debug constructs can also be planted within the HTML, XHTML, or Common Gateway Interface (CGI) scripting code of a Web form returned from a client to a server. To do this, the attacker merely adds another line element to the form's schema to accommodate the debug construct, then inserts that construct into the form. This would have the same as the URL/URI attack above.

G.7.1.2. Implicit Debugger Commands

Implicit debugger commands are seemingly innocuous elements placed by the developer into from-scratch source code, to make it easier to alter the software state to reduce the amount of testing time. These commands are often left in the software's comment lines.

If the source code containing the implicit debugger commands is user-viewable, such as that of Web pages in HTML/XHTML, JSP, or ASP, as well as CGI scripts, such embedded commands can be easily altered by an attacker with devastating results. When using JSP or ASP, these comments may be available to users and may provide an attacker with valuable information.

For example, consider an HTML page in which the developer has included an element called "*mycheck*". The name is supposed to obscure the purpose of this implicit debugger command:

```
<!- begins ->
<TABLE BORDER=0 ALIGN=CENTER CELLPADDING=1 CELLSPACING=0>
<FORM METHOD=POST ACTION="http://some_poll.gov/poll?1688591" TARGET="sometarget"
MYCHECK1="666">
```

```
<INPUT TYPE=HIDDEN NAME="Poll" VALUE="1122">
<!- Question 1 ->
<TR>
<TD align=left colspan=2>
<INPUT TYPE=HIDDEN NAME="Question" VALUE="1">
<SPAN class="Story">
```

Attackers are usually wise to such obfuscation attempts (which constitute "security through obscurity"; please note that security through obscurity is inadequate to hinder any but the most casual novice attackers).

The developer should text-search all user-viewable source code files to locate and remove any implicit debug elements before compiling the code in preparation for deployment.

G.7.2. Removing Hard-Coded Credentials

Basic authentication should never be used in Web applications, even over SSL/TLS-encrypted connections. Not using basic authentication should alleviate any need for hard-coded credentials in HTML or XHTML pages. Regardless of the reason the credentials may have been hard-coded during development, for obvious reasons, they must be sought out and removed from all user-viewable source code before the software system is installed.

G.7.3. Removing Sensitive Comments from User-Viewable Code

Comments left in user-viewable source code should never reveal sensitive or potentially exploitable information, such as information about the file system directory structure, the software's configuration and version, or any type of security-relevant information. Of particular concern are comments within source code that can be viewed by attackers using a Web browser.

The following types of information should never be included in user-viewable source code comments:

- Paths to directories that are not explicitly intended to be accessed by users,
- Location of root,
- Debugging information,
- Cookie structures,
- Problems associated with the software's development,
- Developers' names, email addresses, phone numbers, etc.,
- Release and version numbers of commercial and open source components,
- Hard-coded credentials,
- Any other information that could be exploited by an attacker to target the software, its data, or its environment.

Comments are included in user-viewable source files in one of the following ways:

- **Structured comments:** Included regularly by members of large development teams at the top of the viewable source code page, or between a section of scripting language and a subsequent section of markup language, to inform other developers of the purpose or function implemented by the code.

- **Automated comments:** Automatically added to viewable source pages by many commercial Web generation programs and Web usage programs, such comments reveal precise information about the version/release of the package used to auto-generate the source code—information that can be exploited by attackers to target known vulnerabilities in Web code generated by those packages. (Note that *httpd* restricts what can be included in a filename, unless the Web server has *exec* disabled.)

- **Unstructured comments:** Informal comments inserted by developers as memory aids, such as "The following hidden field must be set to 1 or XYZ.asp breaks" or "Do' not change the order of these table fields". Such comments represent a treasure trove of information to the reconnaissance attacker.

The following HTML comments represent security violations:

<!—#exec cmd="rm -rf /"—>

<!—#include file="secretfile"—>

A simple filter can be used to strip out all comments from user-viewable source code before the code is installed. In the case of automatically-generated comments, an active filter may be required to remove comments on an ongoing basis, as the code is likely to be maintained using the same package that originally generated the code, and so is likely to include more undesirable comments in regenerated versions of the code.

G.7.4. Removing Unused Calls

A code walk-through should be performed in which all calls that do not actually accomplish anything during system execution are identified and removed from the source code. Examples of such calls are those that invoke environment processes or library routines/functions that are no longer present, or that have been replaced.

G.7.5. Removing Pathnames to Unreferenced, Hidden, and Unused Files from User-Viewable Source Code

If the software includes user-viewable source code, all pathname/URI references that point to unused and hidden files need to be removed from user-viewable source code, so as to prevent enumeration attacks in which the attacker searches for files or programs that may be exploitable or otherwise useful in constructing an attack.

G.7.6. Removing Data-Collecting Trapdoors

Federal government policy, as set out in Office of Management and Budget (OMB) Director Jacob J. Lew's Memorandum for the Heads of Executive Departments and Agencies (M-00-13, 22 June 2000), states that cookies must not be used on federal government Web sites, or by contractors operating Web sites on behalf of federal agencies, unless certain conditions are met:

1. There is a compelling need to gather the data on the site;

2. Appropriate, publicly disclosed privacy safeguards have been implemented to handle information extracted from cookies;

3. The head of the agency owning the Web site has personally approved use of data collecting cookies.

This policy was particularized for U.S. DoD in the Office of the Secretary of Defense (OSD) memorandum, dated 13 July 2000, "Privacy Polices and Data Collection on DoD Public Web Sites" (13 July 2000).

In addition to removing any non-policy-compliant cookies that may have inadvertently been left in the Web application, the source code should be carefully vetted before deployment to ensure that it does not contain any other kinds of "Web bugs", spyware, or trapdoor programs (particularly no malicious trapdoors) the intent of

which is either to collect or tamper with privacy data, or to open a back-channel over which an attacker could collect or tamper with such data.

NOTE: A Web bug is a graphic on a Web page or in an email message that is designed to monitor who is reading the Web page or email message. Web bugs are often invisible because they are typically only one pixel-by-one pixel (1x1) in size. They are represented as HTML <IMG> tags. Here is an example:

```
<img src="http://ad.doubleclick.net/ad/ pixel.whoreads/NEW" width=1 height=1 border=0>
<IMG WIDTH=1 HEIGHT=1 border=0 SRC="http://user.preferences.gov/ping?
ML_SD=WebsiteTE_Website_1x1_RunOfSite_A ny&db_afcr=4B31-C2FB-
10E2C&event=reghome&group=register&time=2002.10.27.20.5 6.37">
```

All <IMG> tags in HTML/XHTML code should be checked to ensure that they do not implement Web bugs.

G.7.7. Reconfiguring Default Accounts and Groups

A lot of commercial software is preconfigured with one or more default user (and sometimes group) accounts, such as "administrator", "test", "guest", and "nobody". Many of these accounts have widely-known default passwords, making them subject to password guessing attacks. Passwords should be changed on all default accounts. All unused accounts should be deleted immediately upon installing commercial and open source software. The hardening procedure appropriate to the commercial software and its environment should be performed. Web and database application vulnerability scanners should be run, if possible, to detect any commonly used default passwords that may have been overlooked. If possible without "breaking" its operation, a Web server's "nobody" account should be renamed to something less obvious.

G.7.8. Replacing Relative Pathnames

If the software calls a library or another component, that call should be explicit and specific. The software should not be written to call a relative pathname or to rely on a search path. Doing so will make network-based software vulnerable to cross-site scripting and similar malicious code attacks. For the same reason, whenever possible, full pathnames should be used for URL/URIs and other file paths that users will reference. These pathnames should not simply point to the relative pathname of the current directory. Instead, the full directory path should be included in the pathname.

Use of relative pathnames can also make software vulnerable to directory traversal attacks. Because the current directory is always searched first, a malicious program or library routine hidden in that directory may be triggered when, in fact, the user thought he was accessing a different directory where a trusted program was expected to reside. As search rules for dynamic link libraries (DLLs) and other library routines become more complex, vulnerabilities will be more easily introduced into system routines that can parse filenames that contain embedded spaces.

G.7.9. Defining Secure Installation Configuration Parameters and Procedures for the Software and Its Execution Environment

Even if software itself has been made secure for distribution, it still may be vulnerable to malicious attacks if installed in an execution environment that is insecure. The documentation delivered with the software should provide adequate information to guide the administrator in configuring the environment controls and protections that the software has been designed to rely upon.

Faulty configuration is the source of some significant security problems. When software systems are installed, it is important to make the initial installation configuration secure and make the software easy to reconfigure while keeping it secure. During software distribution, *write* access to the software and its data in configuration management system should be denied to all programmers and end users to ensure the integrity of the software when it is installed.

The software should be delivered with a set of installation and configuration routines, procedures, and tools that will help ensure that the transfer of software and data from the development environment into the production environment is secure. These procedures should include post-installation test procedures that double check the installation and configuration to ensure the appropriate procedures were followed accurately.

In some organizations, secure configuration guides and scripts are produced, or existing third-party guides are mandated, by an operational security team, rather than by the software's developer. In such organizations, it is the developer's responsibility to ensure that the software as designed and implemented does not presume the availability or require the presence of any environment-level services or interfaces that are not supported in the mandated environment configuration guidelines.

In both cases, the developer is still responsible for providing the software's unique configuration requirements to whatever team is responsible for "locking down" the execution environment. If environment configuration guides are mandated by a security team, the developer-produced guidelines should identify any additional environment constraints that the software expects to establish and sustain secure operation. In addition, the software's configuration requirements should document any circumstances in which the violation of a mandated configuration parameter is truly unavoidable, for example when the software truly could not be implemented in a way that would enable it to run correctly with a particular environment service or interface disabled.

The installation and configuration procedures for the software system should include the following instructions for the installer/administrator:

1. **Configure restrictive file system access controls for initialization files:** Many software systems read an initialization file to allow their defaults to be configured. To ensure that an attacker cannot change which initialization file is used, nor create or modify the initialization file, the file should be stored in a directory other than the current directory. Also, user defaults should be loaded from a hidden file or directory in the user's home directory. If the software runs on Unix or Linux and is *setuid/setgid,* it should be configured not to read any file controlled by a user without first carefully filtering that file as untrusted input. Trusted configuration values should be loaded from a different directory (e.g., from `/etc` in Unix).

2. **Validate all security assumptions:** When installing, the administrator should be guided in verifying that all security assumptions made by the software are valid. For example, the administrator should check that the system's source code, and that of all library routines used by the software, are adequately protected by the access controls of the execution environment (operating system) in which the software is being installed. Also, the administrator should verify that the software is being installed on the anticipated execution environment before making any assumptions about environment security mechanisms and posture.

3. **Configure restrictive access controls on target directories:** The administrator should configure the most restrictive access control policy possible when installing the software, only adjusting those restrictions as necessary when the software goes into production. Sample "working users" and access rights for "all configurations" should never be included the software's default configuration.

4. **Remove all unused and unreferenced files from the file system:** The administrator should remove all unnecessary (unused or unreferenced) files from the software's execution environment, including:

    - Commercial and open source executables known to contain exploitable faults;
    - Hidden or unreferenced files and programs (e.g., demo programs, sample code, installation files) often left on a server at deployment time;
    - Temporary files and backup files stored on the same server as the files they duplicate;
    - DLLs, extensions, and any other type of executable that is not explicitly allowed.

NOTE: If the host operating system is Unix or Linux, the administrator can use a recursive file grep *to discover all extensions that are not explicitly allowed.*

G.7.10. Trusted Distribution

While trusted distribution as defined in *A Guide to Understanding Trusted Distribution in Trusted Systems* (NCSC-TG-008, also referred to as the "Dark Lavender Book") will not be required for most software, several of the principles and practices of trusted distribution are widely applicable to ensure the integrity of software distributions by reducing the number of opportunities for malicious or nefarious actors to gain access to and tamper with the software after it has shipped.

Many features of trusted distribution have, in fact, become standard mechanisms for protecting commercial software from tampering while in transit from supplier to consumer, including tamperproof or tamper-resistant packaging and read-only media, secure and verifiable distribution channels (e.g., HTTPS downloads, registered mail deliveries), and digital integrity mechanisms (such as hashes and code signatures).

The following practices will help preserve the integrity of the software, including the installation routines and tools shipped with it:

- **Strong authentication for installation and configuration routines, tools, and interfaces:** Neither the software itself, nor its installation routines, should be installable under a default password. Instead, each software distribution should be assigned a unique strong password. This password should be sent to the purchaser via a different distribution path and mechanism, and at a different time, than that used to distribute the software, e.g., in an encrypted email or in a document mailed via the postal service. It should not be shipped with the software itself.

- **Strong default access controls on installed programs:** The default privileges assigned to the software's executable files should be execute-only for any role other than "administrator".

- **Clear, secure configuration interfaces:** The sample configuration file, if there is one, should contain sufficient, clear comments to help the administrator understand exactly what the configuration does. If there is a configuration interface to the software, the default access rights to this interface should disallow access to any role other than "administrator".

G.8. Keeping Software Secure After Deployment

Most of what the developer will be concerned with is establishing security in the software throughout its development. However, there are development activities related to the post-deployment support and re-engineering of the software—activities that occur after the first version of the software has been deployed (installed) in its operational environment (i.e., gone into production). Systems engineering measures that will be relied on to protect software in deployment should be determined during the system's architecture and design phases, and were discussed in Appendix G:G.3.3.

G.8.1. Support by Suppliers of Acquired or Reused Software

Before acquiring and using a COTS or GOTS component or whole system, the consumer should ensure that the supplier (commercial or contractor) will commit to provide ongoing security support services, including a feedback mechanism for easy consumer reporting of observed security issues, and a security advisory service for supplier reporting to the customer of other discovered security issues, with information about the standard time-frame and process to be used to address consumer-reported security issues. When a security issue in commercial software originates from a software fault, incorrect configuration, or insecure default setting, the associated security advisory should assign a widely-recognized (and ideally machine-readable) designation to that reported security issue (e.g., a Common Vulnerabilities and Exposures [CVE] number, Application Vulnerability

Definition Language [AVDL] designation, or Open Vulnerability and Assessment Language [OVAL] indicator). The supplier should also maintain a publicly available data store containing descriptions of the common types of software faults that the suppliers' development methods and tools have reduced or eliminated from each of the supplier's software products.

G.8.2. Post-Release Support and Re-engineering

> *NOTE: The term "maintenance", frequently used to describe the activities the phase of the software life cycle after its initial deployment, is problematic. "Maintenance" as it is commonly understood fails to reflect the high degree and frequency of changes to which modern software is subject after deployment. For this reason, this document uses the term "post-deployment support and re-engineering" instead of "maintenance".*

After the software has been accepted and is operational, it will need to be modified, at a minimum to apply supplier-provided security patches to commercial and open source software packages, or to swap in the latest releases of that software. In all likelihood, changes will also be made to in response to new or altered user requirements, policies, technologies and standards, threats, or discovered vulnerabilities or faults/bugs. Web applications and Web services are particularly dynamic. Content is continually being altered, new features are added, orchestrations and choreographies of services are redefined—in some instances continuously.

G.8.2.1. Updates of Risk Analysis and Software Documentation

Each time a software system is changed, even if the change is the application of a security patch, a risk is imposed that it will no longer be secure (or at least no longer at the accepted risk level). Even the simplest of changes could introduce a fault that could be exploited to compromise the security of the software system, its data, or its execution environment. Even if the system is "frozen", vulnerabilities are likely to emerge over time even in the most secure environments, vulnerabilities that are inadvertently introduced by changes in the system's surrounding infrastructure or execution environment, or in other software entities with which it interacts.

Any changes to individual components, a component sub-assembly, or to the whole assembly of the software system must not be made in an ad hoc fashion. The requirements driving the change should be captured in a revision of the baseline requirements specification. The changes themselves, including all patches, should be captured in a revision of the design documentation, which should then go through the a security design review. The changed software should undergo security testing to demonstrate that the software system has not been made vulnerable by the change. These reviews and tests may focus more on the differences in functionality and behavior between the previous and current versions, and any changes in the security of interfaces between the changed and unchanged aspects of the integrated system.

Before the modified software goes into production, a security risk analysis should be performed to ensure that the new version does not introduce an unacceptable level of risk; the risk analyst should refer to the baseline risk analysis done when the system was first accepted/approved to operate.

While the waterfall life cycle model specifies only one post-deployment "maintenance" phase, the spiral model and other iterative models provide several distinct phases for post-release support; some of these models treat post-deployment re-engineering and changes to the software system as entirely new development projects. Regardless of which model is used, the same security considerations and practices should be applied to changes made to software after its initial deployment.

G.8.2.2. Countermeasures to Software Aging in Operational Software

Software programs that are required to execute continuously are subject to software aging. Software ages because of error conditions, such as memory leaks, memory fragmentation, memory bloating, missed scheduling deadlines, broken pointers, poor register use, build-up of numerical round-off errors, and other error conductions that accumulate over time with continuous use. Software aging manifests by increasing the number of failures that result from deteriorating operating system resources, unreleased file locks, and data corruption. Software aging

makes continuously-running software a good target for DoS attacks, because such software is known to become more fragile over time.

Software aging can occur in acquired or reused software, such as Web servers, database management systems, or public key infrastructure (PKI) middleware, as well as in software developed from scratch. As the number of software components that are "always on" and connected to the publicly accessible networks (e.g., the Internet) increases, the possibility of DoS attacks targeted at likely-to-be-aging programs also increases. Attackers who are able to guess that a particular system is constantly online can exploit this knowledge to target the kinds of vulnerabilities that manifest as a result of software aging. Two techniques that have proven effective against the problem of software aging are described below.

G.8.2.1.1. Software Rejuvenation

Software rejuvenation is a proactive approach that involves stopping executing software periodically, cleaning internal states, and then restarting the software. Rejuvenation may involve all or some of the following: garbage collection, memory defragmentation, flushing operating system kernel tables, and reinitializing internal data structures. Software rejuvenation does not remove bugs resulting from software aging but rather prevents them from escalating to the point where the software becomes significantly fragile and easily targeted.

While rejuvenation incurs immediate overhead because some services become temporarily unavailable, these brief "outages" can be scheduled and predicted, and can help prevent lengthy, unexpected failures caused by successful DoS attacks. The critical factor in making scheduled downtime preferable to unscheduled downtime is determining how often a software system must be rejuvenated. If unexpected DoS could lead to catastrophic results, a more aggressive rejuvenation schedule might be justified in terms of cost and availability. If unexpected DoS is equivalent to scheduled downtime in terms of cost and availability, then a reactive approach might be more appropriate.

G.8.2.1.2. Software Reconfiguration

By contrast with proactive rejuvenation, the reactive approach to achieving DoS-resistant software is to reconfigure the system after detecting a possible attack, with redundancy as the primary tool that makes this possible and effective. In software, reconfiguration implements redundancy in three different ways:

1. Independently-written programs that perform the same task are executed in parallel, with the developers comparing their outputs (this approach is known as *n*-version programming);
2. Repetitive execution of the same program while checking for consistent outputs and behaviors;
3. Use of data bits to "tag" errors in messages and outputs, enabling them to be easily detected and fixed.

The objective of software redundancy is to enable flexible, efficient recovery from DoS, independent of knowledge about the cause or modus operandi of the DoS attack. While robust software can be built with enough redundancy to handle almost any failure, the challenge is to achieve redundancy while minimizing cost and complexity. More extensive discussion of redundancy appeared in Appendix G:G.3.3.5.

In general, reconfiguring executing software for recovery from a failure in another part of the software system should only be performed if it can be accomplished without impacting users. However, there are cases when a high priority component fails and requires resources for recovery. In this scenario, lower priority components' execution should be delayed or terminated and their resources reassigned to aid this recovery, resulting in intentional degradation.

G.9. Activities and Considerations that Span the Life Cycle

The following sections provide guidance on security enhancing those activities that span multiple phases of the software development life cycle.

G.9.1. Secure Configuration Management (CM)

Diligent CM of the software development and testing artifacts is critical to ensure the trustworthiness of those artifacts throughout the development life cycle, 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*, 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 problems 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 life cycle.

Figure 5-3 depicts a secure configuration change control process and how configuration items, and, in particular, the source code, binary executables, and documentation, are protected within the CM system throughout the software development life cycle.

G.9.1.1. CM and Patch Management

In large enterprises, patching of software can become a difficult and expensive logistical exercise. Many patches require some amount of downtime for the software-intensive system, which means that the updates must be applied within specific timelines. If any part of the update process fails, the system may be unavailable for longer than intended.

To minimize downtime associated with patching, and to ease the burden on systems administrators, software should be designed with the assumption that patches will be an inevitability. Even with the most stringent secure software development process, there remains the possibility that the resulting software will need to be updated. Furthermore, software that can automatically apply a patch with little or downtime is more likely to have those updates applied. Nevertheless, many organizations have strict security policies regarding incoming software updates, so any automated updates or version notification should be configurable, giving such organizations the ability to integrate all software patch management into a single process.

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

In the case of security patches, developers can never be sure when or even if the supplier of a particular software product will release a needed security patch for a reported vulnerability that might render a selected component otherwise unacceptable for use in the software system.

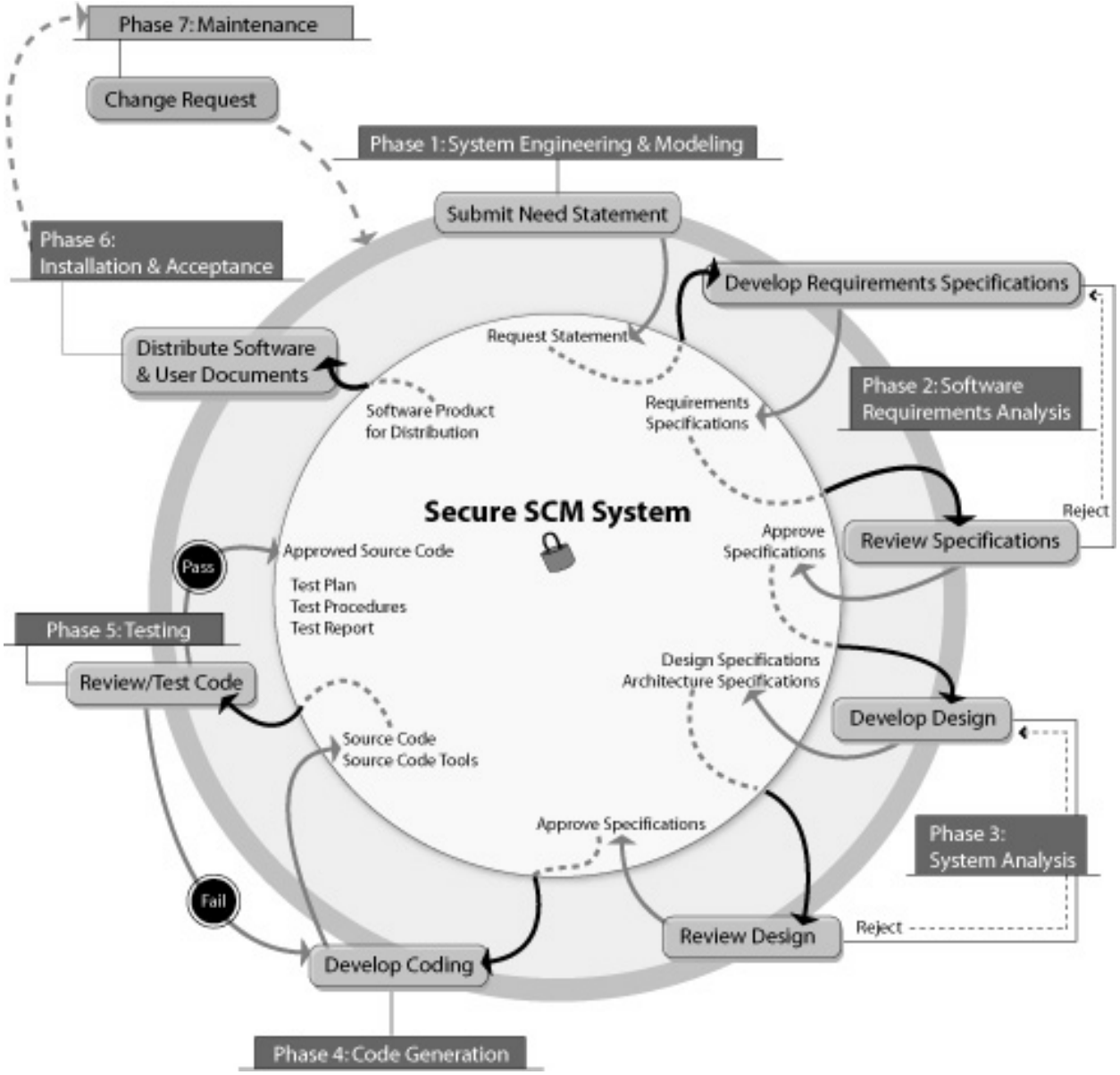

Figure 5-3. Secure CM in the Software Development Life Cycle

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

Announced security-enhancements of acquired or reused software should be considered as early in the development life cycle as possible so that risk management and C&A requirements can be addressed for any new releases, versions, or updates that are expected to have impacts on the software system's security overall. It is particularly important to determine whether the replacement of any software product that performs a trusted function or security function. If it does, its replacement—even only to move to a later version of the same product—will entail recertifying and reaccrediting the whole software system.

On the other hand, if a particular component is not kept up to date according to the supplier's release schedule, the configuration manager and developer will both need to keep track of the minutiae of the supplier's maintenance and support agreement, to determine if there is a point at which the non-updated version of the software will no longer be supported by that supplier. The risks associated with using unsupported software will then have to be weighed against the risks of adopting a new version of the software, or replacing it with an alternative product. The supplier's willingness to support older versions for a fee may be something worth negotiating during acquisition, as is supplier willingness to include modifications to counteract security vulnerabilities found during assembly/integration versus after deployment into the product's standard code base.

To keep abreast with vendor patches, new releases, and upgrades and to be informed with the latest software vulnerabilities and issues, the configuration manager should keep track of vulnerability reports issued by the US-CERT, U.S. DoD's Information Assurance Vulnerability Alert (IAVA) program, the CVE database, and other reliable sources of software product vulnerability information. The configuration manager should download all necessary patches indicated by those vulnerability reports, and work with the risk manager to determine the impact of adopting those patches.

It is extremely critical to determine and understand how the security of the assembled/integrated system may be affected by new behaviors or interfaces introduced in the updated software. This is particularly true because suppliers often embed non-security-related features in their security patches, i.e., features that will later appear in the next full release of the software. Unfortunately, these as-yet-undocumented features are seldom announced when delivered with the patch. Detecting such features can present a significant challenge to developers who must understand their potential impact on a component's behavior when interfacing with other components.

In order to reduce the number and increase the accuracy of interim impact analyses of patches and updates, it may make sense to horde several updates, new releases, and patches to several components for some fixed period, so as to be able to analyze them all at once and in conjunction with each other. While this approach may increase risk by postponing patching, it may also reduce risk by ensuring that as many updated components as possible can be analyzed in tandem, to observe their actual behaviors and interfaces before adopting them, or determining that they cannot be adopted due to unacceptable security impacts.

CM, of course, must include the tracking and control of all acquired or reused fixes, patches, updates, and new releases. The configuration manager needs to create and follow a realistic schedule by which new versions of the software system incorporating the updated software can be impact-analyzed, integrated/assembled, tested, recertified/re-accredited (if necessary), and in the case of production software, deployed.

The security patch management solution used during the pre-deployment phases of the development life cycle (versus in the post-production phases) must be flexible enough to allow for such impact analyses to be performed prior to integration/assembly of the patched acquired or reused software. If scripts are used to automate software installations, the scriptwriter should be sure that the scripts do not overwrite security patches already installed on the target systems. For systems running Microsoft software, the Baseline Security Analyzer available from Microsoft should be run when developing installation scripts.

G.9.1.2. Using CM to Prevent Malicious Code Insertion During Development

Uncontrolled software development life cycle 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 PKI 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 life cycle 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 those that enable buffer overflows and race conditions. Software developers should be given awareness training in the types of problems to look for during security code reviews and tests (See Section 4.4 for more information on security education for developers.)

Malicious developers who intentionally include development faults in their code have plausible deniability: they can always claim that the faults were the result of simple mistakes. Work to eradicate such insertions 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 life cycle 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.

G.9.2. Security Documentation

Proactive documentation of software systems is imperative, especially for software systems that undergo C&A or certification against the Common Criteria. Security documentation should not be separate from the rest of the system's documentation. The process of generating security documentation should be fully integrated into the overall development life cycle, and the content of the resulting documents should be fully incorporated into the general system documentation. For example, there should not be a separate security requirements specification: security requirements should be included in the main system requirements specification. Similarly, the security user's manual should not be a separate document: security procedures should be integrated throughout the general user's manual for the software system. The only exception may be the Trusted Facility Manual (TFM) that may be required for some software systems, but not for others, and which should be separate from the general operator and administrator manuals.

G.9.3. Software Security and Quality Assurance

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);

- **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.

In addition to these activities, quality assurance practitioners should periodically audit the correctness of the performance of these procedures.

G.10. References for this Appendix

Goertzel, Karen Mercedes, Theodore Winograd, Holly Lynne McKinley, Alice Goguen, Jaime Spicciati, Shawn Gunsolley, Dina Dywan, Toby Jones, and Patrick Holley: Defense Information Systems Agency (DISA) Application Security Developer's Guides (Draft Version 3.1, October 2004)

McGraw, Gary: *Software Security: Building Security In* (Addison-Wesley Professional, 2006)

Howard, Michael, and David LeBlanc, Microsoft Corporation: *Writing Secure Code, Second Edition* (Microsoft Press, 2003)
http://www.microsoft.com/mspress/books/5957.asp

Howard, Michael, John Viega, and David LeBlanc: *19 Deadly Sins of Software Security* (McGraw-Hill Osborne Media, 2005)
http://books.mcgraw-hill.com/getbook.php?isbn=0072260858&template=

Seacord, Robert: *Secure Coding in C and C++* (Addison Wesley Professional/SEI Series in Software Engineering, 2005)
http://www.cert.org/books/secure-coding/index.html

Van Wyk, Kenneth R., and Mark G. Graff: *Secure Coding: Principles and Practices* (O'Reilly & Associates, 2003)
http://www.securecoding.org/

Viega, John, and Gary McGraw: *Building Secure Software: How to Avoid Security Problems the Right Way* (Addison-Wesley Publishing Company, 2001)
http://www.buildingsecuresoftware.com/

Viega, John, and Matt Messier: *Secure Programming Cookbook for C and C++* (O'Reilly & Associates, 2003)
http://www.secureprogramming.com/

Premkumar Devanbu, UC-Davis, Stuart Stubblebine, CertCo, Inc.: Software Engineering for Security: A Roadmap
http://www.softwaresystems.org/future.html

Axel van Lamsweerde, Peter Van Roy, Jean-Jacques Quisquater: MILOS: Méthodes d'Ingénierie de Logiciels Sécurisés *(in French)*
http://www.info.ucl.ac.be/people/PVR/MILOSabstr.rtf

David Wheeler, Institute for Defense Analysis: Secure Programming for Linux and Unix HOWTO—Creating Secure Software
http://www.dwheeler.com/secure-programs/

Joint Application Design/Development (JAD)
http://www.umsl.edu/~sauter/analysis/JAD.html

Automated Requirements Measurement (ARM)
http://satc.gsfc.nasa.gov/tools/arm/

Software Cost Reduction (SCR)
http://chacs.nrl.navy.mil/5540/personnel/heitmeyer/

Edward Colbert, Danni Wu, Yue Chen, and Barry Boehm, University of Southern California Center for Software Engineering: Cost Estimation for Secure Software and Systems (2004–5 USC CSE, 29 December 2005)
http://sunset.usc.edu/publications/TECHRPTS/2006/usccse2006-600/usccse2006-600.pdf

Yue Chen, Barry W. Boehm, and Luke Sheppard, University of Southern California Center for Systems and Software Engineering: Value Driven Security Threat Modeling Based on Attacking Path Analysis (USC-CSE-2006-607, 16 June 2006)
http://sunset.usc.edu/publications/TECHRPTS/2006/usccse2006-607/usccse2006-607.pdf

Edward Colbert and Danni Wu: COCOMO II Security Extension Workshop Report (20th Annual COCOMO II and Software Costing Forum, 2005)
http://sunset.usc.edu/events/2005/COCOMO/presentations/COSECMOReport.ppt

Security Quality Requirements Engineering (SQUARE)
http://www.cert.org/archive/pdf/05tr009.pdf

Requirements Engineering Verification and Validation (REVEAL)
http://www.praxis-his.com/reveal/

Miroslav Kis, Ph.D., CISSP: Information Security Antipatterns in Software Requirements Engineering
http://jerry.cs.uiuc.edu/~plop/plop2002/final/mkis_plop_2002.pdf

Jonathan D. Moffett, Bashar Nuseibeh, The Open University: A Framework for Security Requirements Engineering (20 August 2003) — http://www.cs.york.ac.uk/ftpdir/reports/YCS-2003-368.pdf

Jonathan D. Moffett, Charles B. Haley, Bashar Nuseibeh, The Open University: Core Security Requirements Artefacts *[sic]* (21 June 2004; Technical Report No. 2004/23)
http://computing-reports.open.ac.uk/index.php/content/download/166/999/file/2004_23.pdf

John McDermott and Chris Fox, James Madison University: Using Abuse Case Models for Security Requirements Analysis
http://www.cs.ndsu.nodak.edu/~vgoel/Security-Engineering/wed-b-1030-john.pdf

Axel van Lamsweerde, Simon Brohez, Renaud De Landtsheer, David Janssens, Université Catholique de Louvain: From System Goals to Intruder Anti-Goals: Attack Generation and Resolution for Security Requirements Engineering
http://www.cs.toronto.edu/~jm/2507S/Readings/avl-RHAS03.pdf - *or* -
http://www.cs.ndsu.nodak.edu/~vgoel/Security-Engineering/avl-RHAS03.pdf

Anthony A. Aaby, Walla Walla College: "Security and the Design of Secure Software", from lecture notes on his book *Software: A Fine Art* (Draft, V.0.9, 21 January 2004)
http://cs.wwc.edu/~aabyan/FAS/book/node76.html - *and* -
http://moonbase.wwc.edu/~aabyan/FAS/book.pdf - *or* -
http://cs.wwc.edu/~aabyan/FAS/book.pdf

Professor Zeller, Saarland University (Germany): Secure Software Design (Lecture SS 2004)
http://www.st.cs.uni-sb.de/edu/secdesign/

Joshua Pauli, Dakota State University; Dianxiang Xu, North Dakota State University: Threat-Driven Architectural Design of Secure Information Systems
http://cs.ndsu.edu/%7Edxu/publications/pauli-xu-ICEIS05.pdf

Joshua Pauli, Dakota State University; Dianxiang Xu, North Dakota State University: Misuse Case-Based Design and Analysis of Secure Software Architecture
http://www.homepages.dsu.edu/paulij/pubs/pauli-xu-ITCC05.pdf

Ivan Fléchais, M. Angela Sasse, Stephen M. V. Hailes, University College London: Bringing Security Home: A Process for Developing Secure and Usable Systems
http://www-dept.cs.ucl.ac.uk/staff/A.Sasse/nspw2003.pdf - *or* -
http://www.softeng.ox.ac.uk/Ivan.Flechais/downloads/nspw2003.pdf

Ivan Fléchais, University College London: *Designing Secure and Usable Systems* (PhD. Thesis, February 2005)
http://www.softeng.ox.ac.uk/Ivan.Flechais/downloads/thesis.pdf - *or* -
http://www.cs.ucl.ac.uk/staff/flechais/downloads/thesis.pdf

Robert J. Ellison and Andrew P. Moore, Carnegie Mellon Software Engineering Institute: Trustworthy Refinement through Intrusion-Aware Design (TRIAD): An Overview (2003)
http://www.cert.org/archive/pdf/triad.pdf

Andrew P. Moore, SEI: Security Requirements Engineering through Iterative Intrusion-Aware Design
http://www.cert.org/archive/pdf/req_position.pdf

Robert J. Ellison and Andrew P. Moore, SEI: Trustworthy Refinement through Intrusion-Aware Design (TRIAD) (March 2003 Technical Report CMU/SEI-2003-TR-002/ESC-TR-2003-002)
http://www.cert.org/archive/pdf/03tr002.pdf

Bertrand Meyer: Applying "Design by Contract" (IEEE Computer, October 1992)
http://doi.ieeecomputersociety.org/10.1109/2.161279

Paul Kocher, Ruby B. Lee, Gary McGraw, Anand Raghunathan, and Srivaths Ravi: Security as a New Dimension in Embedded System Design (*Proceedings of the Design Automation Conference (DAC)*, June 2004) — http://palms.ee.princeton.edu/PALMSopen/Lee-41stDAC_46_1.pdf

SecurityPatterns.org
http://www.securitypatterns.org/index.html

Implementation Schemes for Secure Software
http://anzen.is.titech.ac.jp/index-e.html

Pascal Meunier, Purdue Univ. CERIAS/Symantec Corp.: Overview of Secure Programming
http://projects.cerias.purdue.edu/secprog/

Jim Hranicky, University of Florida: Secure Application Development presentation
http://www.cise.ufl.edu/~jfh/pres/

Christophe Grenier's "Tutorial sur la programmation sécurisée" (Tutorial on Secure Programming; *available in French and English*)
http://www.cgsecurity.org/Articles/SecProg/buffer.html

Hervé Schauer Consultants: Programmation sécurisée et trappes logicielles (Secure programming and logic traps) (tutorial given at Eurosec 2002; *in French*)
http://www.hsc.fr/ressources/presentations/traps/mgp00001.html

Timo Sirainen: Secure, Efficient and Easy C Programming
http://irccrew.org/~cras/security/c-guide.html

Paladion Networks: Best Practices in Input Validation (*Palisade Application Security Intelligence*, December 2004)
http://palisade.paladion.net/issues/2004Dec/input-validation/

John Viega and Matt Messier: *Input Validation in C and C++* (O'Reilly Network, 20 May 2003)
http://www.oreillynet.com/pub/a/network/2003/05/20/secureprogckbk.html

MSDN Library: Visual Basic and Visual C# Concepts: Introduction to Validating User Input in Web Forms
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcon/html/vbconintroductiontovalidatinguserinputinWebforms.asp

Anthony Moore, Microsoft Corporation: User Input Validation in ASP.NET (MSDN Library, July 2000/March 2002)
http://msdn.microsoft.com/library/en-us/dnaspp/html/pdc_userinput.asp?frame=true

Scott MacKenzie, York University (Canada): Java Primer—Input Validation
http://www.cs.yorku.ca/~mack/1011/InputValidation.PDF

Budi Kurniawan: Input Validation with Filters (*JAVAPro,* 13 August 2002)
http://www.ftponline.com/javapro/2002_08/online/servlets_08_13_02/?q=rss

Jordan Dimov, Cigital Inc.: Introduction to Input Validation with Perl (developer.com)
http://www.developer.com/lang/article.php/861781

Input Validation with JavaScript
http://www.macromedia.com/v1/documents/cf31userguide/user/ug090003.htm

Jon Kurz: Dynamic Client-Side Input Validation (*ColdFusion Developer's Journal,* 1 May 2003)
http://coldfusion.sys-con.com/read/41599.htm

Lampson, B. W. "Software Components: Only the Giants Survive," in K. Spark-Jones and A. Herbert (editors): *Computer Systems: Papers for Roger Needham* (Cambridge, U.K.: Microsoft Research, February 2003)

John E. Dobson and Brian Randell, University of Newcastle upon Tyne: "Building Reliable Secure Computing Systems out of Unreliable Insecure Components" (*Proceedings of the 17th Annual Computer Security Applications Conference (ACSAC 01),* 2001)
http://www.cs.ncl.ac.uk/research/pubs/inproceedings/papers/355.pdf

Dick Hamlet, Portland State University, and Dave Mason and Denise Woitz, Ryerson Polytechnic University: "Properties of Software Systems Synthesized from Components" (30 June 2003)
http://web.cecs.pdx.edu/~hamlet/lau.pdf

Davide Balzarotti, Politecnico di Milano, Mattia Monga, Università degli Studi di Milano, and Sabrina Sicari, Università di Catania: "Assessing the Risk of Using Vulnerable Components" (*Proceedings of the First Workshop on Quality of Protection*, September 2005)
http://www.elet.polimi.it/upload/balzarot/publications/download.php?doc=risk-qop05.pdf

Issues in Developing Security Wrapper Technology for COTS Software Products
http://iit-iti.nrc-cnrc.gc.ca/publications/nrc-44924_e.html

David L. Wilson, Purdue University: *Risk Perception and Trusted Computer Systems: Is Open Source Software Really More Secure than Proprietary Software?* (Master's Thesis—CERIAS TR 2004-07)
https://www.cerias.purdue.edu/tools_and_resources/bibtex_archive/view_entry.php?bibtex_id=2686

Don O'Neill, Center for National Software Studies: "Competitiveness vs. Security" (*CrossTalk: The Journal of Defense Software Engineering*, June 2004)
http://www.stsc.hill.af.mil/crosstalk/2004/06/0406ONeill.html

General Accounting Office Report to Congressional Requesters: Defense Acquisitions: Knowledge of Software Suppliers Needed to Manage Risks (GAO-04-678, May 2004)
http://www.gao.gov/new.items/d04678.pdf

Rob Ramer: The Security Challenges of Offshore Development (SANS Institute, 2001)
http://www.sans.org/rr/whitepapers/securecode/383.php

Grant Gross, IDG News Service: "Security Vendor Says Offshore Development Needs Checks; Extra Steps Called for to Ensure Secure Code" (*InfoWorld*, 2 June 2004)
http://www.infoworld.com/article/04/06/02/HNoffshorecheck_1.html

Dan Verton: "Offshore Coding Work Raises Security Concerns" (*Computerworld*, 5 May 2003)
http://www.computerworld.com/managementtopics/outsourcing/story/0,10801,80935,00.html

Adam Nelson: "Outsourcing and Offshore Coders: Good or Evil?" (*TheCodeProject*, 27 October 2003)
http://www.codeproject.com/gen/work/offshore.asp

Michael Rasmussen and Natalie Lambert, Forrester Research: "Security Assurance in Software Development Contracts" (*Best Practices*, 24 May 2004)
http://www.sec-consult.com/fileadmin/Newsletters/newsletter062004.pdf

Ounce Labs, Inc.: Trust, But Verify: "How to Manage Risk in Outsourced Applications" (A Security Topics White Paper; *reader must first register on Ounce Labs' Website to gain access*)
http://ouncelabs.com/abstracts/Outsourced_Applications.asp

NIST Software Assurance Metrics and Tool Evaluation (SAMATE)
http://samate.nist.gov/index.php

NIST SP 500-264: "Proceedings of Defining the State of the Art in Software Security Tools Workshop" (September 2005)
http://hissa.nist.gov/~black/Papers/NIST%20SP%20500-264.pdf

Landon Curt Noll, Simon Cooper, Peter Seebach, Leonid A. Broukhis: International Obfuscated C Code Contest
http://www.ioccc.org/

Scott Craver, Binghamton University: The Underhanded C Code Contest
http://bingWeb.binghamton.edu/~scraver/underhanded/index.html

Adam Shostack: Security Code Review Guidelines
http://www.homeport.org/~adam/review.html

NSA National Computer Security Center (NCSC): *A Guide to Understanding Trusted Distribution in Trusted Systems* (NCSC-TG-008)
http://www.radium.ncsc.mil/tpep/library/rainbow/NCSC-TG-008.pdf

NSA NCSC: *A Guide to Understanding Configuration Management in Trusted Systems* (NCSC-TG-006)
http://www.radium.ncsc.mil/tpep/library/rainbow/NCSC-TG-006.html - *or* -
http://www.radium.ncsc.mil/tpep/library/rainbow/NCSC-TG-006.pdf

Microsoft Baseline Security Analyzer (MBSA)
http://www.microsoft.com/technet/security/tools/mbsahome.mspx

Nadine Hanebutte and Paul W. Oman: "Software vulnerability mitigation as a proper subset of software maintenance" (*Journal of Software Maintenance and Evolution: Research and Practice,* November 2001
http://doi.wiley.com/10.1002/smr.315

V. Castelli, R. E. Harper, P. Heidelberger, S. W. Hunter, K. S. Trivedi, K. Vaidyanathan, and W. P. Zeggert: "Proactive management of software aging" (IBM Journal of Research and Development, Volume 45, Number 2, 2001)
http://www.research.ibm.com/journal/rd/452/castelli.pdf

Mohamed Elgazzar, Microsoft Corporation: Security in Software Localization
http://www.microsoft.com/globaldev/handson/dev/secSwLoc.mspx

APPENDIX H. SOFTWARE SECURITY AWARENESS AND OUTREACH CAMPAIGNS

H.1. DHS Software Assurance Program Outreach Activities

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."

DHS began the Software Assurance 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, process, technology, and acquisition throughout the software life cycle.

The Software Assurance Program's outreach activities are intended to increase software security awareness across government, industry, and academia. These activities include:

- Co-sponsorship (with DoD) of semi-annual Software Assurance Forums for government, industry, and academia to facilitate ongoing collaboration;
- Sponsorship of Software Assurance issues of CrossTalk: The Journal of Defense Software Engineering, and authorship of articles for other journals and magazines, to "spread the word" to relevant stakeholders;
- Provision of free Software Assurance resources (including this document) via the "Build Security In" Web portal at https://buildsecurityin.us-cert.gov;
- Provision of speakers for conferences, workshops, and other events (registered with the DHS Speakers Bureau);
- Support for the efforts of government organizations, industry consortia and professional societies, and national and international standards bodies in promoting Software Assurance. Specifically, DHS participates in the following Standards Committees and Organizations:
    - IEEE Software and Systems Engineering Standards Committee,
    - Object Management Group Software Assurance Special Interest Group,
    - ISO/IEC JTC 1/SC7, Software and Systems Engineering,
    - ISO/IEC JTC 1/SC27, IT Security,
    - ISO/IEC JTC 1/SC22, Programming Languages,
    - NIST.
- Sponsorship of a Business Case Working Group, to help define and tailor Software Assurance messages (including return on investment messages) for different stakeholder communities, including software suppliers, integrators, corporate users, and individual users.

H.2. DoD Software Assurance Tiger Team Outreach Activities

In July 2003, when the Assistant Secretary of Defense for Networks and Information Integration (ASD[NII]) established a Software Assurance Initiative to examine the challenges associated with evaluating the assurance

risks of commercially-acquired software to be deployed in defense and other government environments. The DoD Software Assurance Tiger Team was formed in December of the following year as a partnership between the ASD(NII) and the Undersecretary of Defense for Acquisitions, Technology and Logistics [USD(AT&L)].

With USD(AT&L)/ASD(NII) oversight and brokering, the Software Assurance Tiger Team has extended its reach into several fora in order to coordinate a suite of community-driven and community-adopted Software Assurance methodologies, and to designate leadership roles for advancing the objectives of the Initiative. The underlying goal of the outreach is to "partner with industry to create a competitive market that is building demonstrably vulnerability-free software". This goal requires coordination among industry, academia and national and international partners to address shared elements of the problems related to assessing Software Assurance risk. The main goal of the Tiger Team is to develop a holistic strategy to reduce the DoD's, and more broadly, the federal government's susceptibility to these risks. In addition to USD(AT&L) and ASD(NII), the major organizational participants in the Tiger Team efforts include the National Defense Industrial Association (NDIA), the Government Electronics & Information Technology Association (GEIA), the Aerospace Industries Association (AIA), and the Object Management Group (OMG).

Through its outreach to government and industry stakeholders, the Tiger Team has defined a set of guiding principles that are intended to serve as the foundation of reducing Software Assurance risk. These include:

- Understand the problem(s) from a systems perspective.
- Tailor responses to the scope of the identified risk.
- Ensure responses are sensitive to potential negative impacts, e.g.:
    - Degradation of ability to use commercial software,
    - Decreased responsiveness/ increased time to deploy technology,
    - Loss of industry incentive to do business with DoD.
- Minimize burden on acquisition programs.
- Exploit and extend relationships with:
    - National, international, and industry partners;
    - DoD initiatives, e.g., trusted integrated circuits and Information Assurance.

Having established a shared understanding of DoD/government Software Assurance problems, the Tiger Team also aims to guide science and technology on research and development into technologies that can improve upon the capabilities of existing software development tools, strengthen standards for modularizing software, and enhance the ability to discover and eliminate software vulnerabilities.

H.3. Secure Software Forum Application Security Assurance Program

The Secure Software Forum, a consortium of major commercial software producers, is promoting the creation by organizations that produce application software of Application Security Assurance Programs (ASAP) that embrace the following set of broad principles to improve the security of software, without defining any specific process improvement model or SDLC methodology, or even identifying the required features such a model or methodology would need to accomplish these objectives:

- There must be executive level commitment to secure software.
- Security must be a consideration from the very beginning of the software development lifecycle.
- Secure software development must encompass people, processes, and technology.
- An adoption of metrics to measure security improvements and enforce accountability.

- Education as a key enabler of security improvements.

The Secure Software Forum is now discussing a proposal to develop an ASAP Maturity Model to help drive adoption of ASAPs.

H.4. BITS/Financial Services Roundtable Software Security and Patch Management Initiative

BITS established its Software Security and Patch Management Initiative for its member financial institutions. The Initiative has three primary goals:

1. To encourage software vendors that sell to critical infrastructure industries to undertake a higher "duty of care";

2. To promote the compliance of software products with security requirements before those products are released;

3. To make the patch-management process more secure and efficient, and less costly to software customer organizations.

A fourth goal of the initiative is to foster dialogue between BITS members and the software industry, in order to produce solutions to software security and patch management problems that are effective, equitable, and achievable in the near term.

BITS is acting jointly with the Financial Services Roundtable to encourage that financial services companies that make up their collective membership to:

- Develop best practices for managing software patches;

- Communicate to software vendors clear industry business requirements for secure software products;

- Facilitate CEO-to-CEO dialogue between the software industry and the financial services industry and also critical infrastructure sectors;

- Analyze costs associated with software security and patch management;
  Communicate to the federal government the importance of investing to protect critical infrastructure industries;

- Explore potential legislative and regulatory remedies.

As part of the initiative, BITS has established the BITS Product Certification Program (BPCP) that tests software applications and supporting infrastructure products used by financial institutions against a set of baseline security criteria established by the financial services industry and aligned with the Common Criteria. The BPCP was established in hopes of influencing the vendor community to include security considerations in their development processes, and to improve the security of software products used in the financial services industry. The outcome of a successful BPCP test is a BITS Tested Mark that indicates that the tested product passed BITS BPCP testing.

Together BITS and the Financial Services Roundtable have also issued what they call their Software Security and Patch Management Toolkit. This "toolkit" is a set of documents that provide recommended security requirements, a template for a cover email message, and procurement language for ensuring that security and patch management requirements are incorporated into the procurement documents issued by financial institutions to their potential software suppliers. The "toolkit" also include a set of talking points and background information designed to help

executives of financial institutions communicate about software security and patch management issues with their software suppliers and the media.

Finally, BITS and the Financial Services Roundtable are encouraging the software industry to notify companies of vulnerabilities as early as possible.

H.5. References for this Appendix

Secure Software Forum
http://www.securesoftwareforum.com/index.html

BITS Software Security Toolkit
http://www.bitsinfo.org/downloads/Publications%20Page/bitssummittoolkit.pdf

APPENDIX I. SECURE MICROKERNELS

The main design goals of a secure microkernel are to decrease the size of the core trusted code base, and to put a clear, inviolable interface between the trusted code base and less trusted code. The kernel represents a small trusted system, and can be implemented by hardware, software, or a combination of the two. When implemented by software, it will be very small by contrast with the large system libraries of conventional operating systems or virtual machine monitors.

Secure microkernels are extremely limited in the services they attempt to provide. These usually include hardware initialization, device control, application scheduling, and application partitioning. For purposes of software security, this last feature may be the most important. Even with this limited set of services and security features, the separation kernel can run the services of a conventional operating system or a full virtual machine while maintaining a high level of security.

By enforcing a process (or application) separation policy, the secure microkernel can guarantee that two independently running processes or VM environments will not be able to affect each other (thus the designation "separation kernel"). This process separation (or isolation) ensures that malicious code inserted into one isolation segment of the kernel cannot access or steal resources, corrupt or read files, or otherwise harm the processes and data in another isolation segment.

Secure microkernels are most often used in combination with conventional operating systems, to host VMs, or in embedded systems to directly host application-level software. A number of secure microkernels are system specific. This is particularly true of secure hardware microkernels, such as those used in smart cards, cryptographic devices, and other embedded systems. However, there are some secure microkernels emerging that are intended to be system-independent and thus can be more widely adopted. Table H-1 lists some of the most noteworthy secure microkernels.

Table H-1. Secure Microkernel Implementations

Name	Specifier or Developer	More Info
Multiple Independent Layers of Security (MILS) Partitioning Kernel (see Note below)	Specification: NSA and U.S. Air Force Rome Labs	http://www.afrlhorizons.com/Briefs/Aug06/IF_H_06_06.html, http://www.stsc.hill.af.mil/crosstalk/2005/08/0508Vanfleet_etal.html, http://www.military-information-technology.com/article.cfm?DocID=1537
Flux Advanced Security Kernel (Flask) (now incorporated into SE Linux)	Univ. of Utah, NSA, Secure Computing Corp.	http://www.cs.utah.edu/flux/fluke/html/flask.html
L4 Secure Microkernel (seL4, developed for use in the ERTOS [Embedded RTOS])	National Information and Communications (ICT) Technology Australia (NICTA)	http://l4hq.org/
VFiasco (the formally-verified version of the Fiasco µ-kernel)	Univ. of Dresden (Germany)	http://os.inf.tu-dresden.de/vfiasco/
µSINA (used in the Nizza security architecture)	Univ. of Dresden (Germany)	http://os.inf.tu-dresden.de/mikrosina/
kaneton	French open source project	https://www.kaneton.org/bin/view/Kaneton
Coyotos Microkernel	U.S. open source project	http://www.coyotos.org/docs/ukernel/spec.html
Safe Language Kernel (J-Kernel is one implementation)	Cornell University (sponsored by DARPA)	http://www.cs.cornell.edu/slk/, http://www.cs.cornell.edu/slk/jkernel.html

The now-obsolete Trusted Mach (TMach) operating system was among the first to implement a secure microkernel. Some current operating systems that include secure microkernels are Security Enhanced (SE) Linux, which includes a refinement of the Flask microkernel, and Coyotos, which is an open source refinement of EROS (also open source).

NOTE: MILS-compliant microkernel implementations are found in the DO-178B/ARINC 653-1 compliant real time operating systems (RTOSes) from LynuxWorks (LynxOS-178), Green Hills Software (INTEGRITY), and Wind River Systems (VxWorks). DO-178B is a Radio Technical Commission for Aeronautics (RTCA) standard for development processes used to produce safety-critical applications, while ARINC 653-1 (ARINC Specification 653: Avionics Application Standard Software Interface, Supplement 1, Draft 3) is an Airlines Electronic Engineering Committee (AEEC) standard for safety-critical systems. Both of these safety standards describe safety microkernels that should be used for partitioning and isolating safety-critical processes and non-critical processes in real time aviation systems. In the case of ARINC 653-1, the safety microkernel is called the Application Executive (APEX). As the MILS compliance of the RTOSes cited above suggests, safety microkernels provide most of the features required for separation of privileged and unprivileged processes in trusted operating systems. This is especially true when the microkernel does not have to enforce data separation and information flow security with a higher level assurance than that afforded by operating systems certified at CC EAL 4.

placeholder for back page