Enhancing the Development Life Cycle to Produce Secure Software

A Reference Guidebook on Software Assurance
October 2008

Data&Analysis Center for Software

https://www.thedacs.com/

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. REPORT DATE (DD-MM-YYYY) 2008-10-01

2. REPORT TYPE Technical Report

3. DATES COVERED (From - To)
2008-10-01 - 2008-10-01

4. TITLE AND SUBTITLE
ENHANCING THE DEVELOPMENT LIFE CYCLE TO PRODUCE SECURE SOFTWARE

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Goertzel, Karen Mercedes (editor, principal co-author)
Winograd, Theodore (co-author)
Numerous Other Reviewers

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
DACS Data & Analysis Center for Software, ITT AES, 775 Daedalian Dr., Rome, NY 13441 US

8. PERFORMING ORG REPORT #
DAN 358844

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Technical Information Center (DTIC)/AI, 8725 John J. Kingman Rd., STE0944, Ft.Belvoir, VA 22060 US

10. SPONSOR/MONITOR'S ACRONYM(S)
DTIC

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

12. DISTRIBUTION / AVAILABILITY STATEMENT - DISTRIBUTION STATEMENT A. Approved for public release;

13. SUPPLEMENTARY NOTES .

14. ABSTRACT: The most risk-averse system with a security architecture including layers upon layers of defenses-in-depth can still be vulnerable to violations and compromises if the software that implements those layered defenses is not dependable, trustworthy, and survivable. The reality is this: software has long been, and remains, the weakest link in any information system. The adversaries who attack those systems know this. And they have the expertise, tools, and resources to exploit that knowledge.

Enhancing the Development Life Cycle to Produce Secure Software answers the questions of why software security is important, why so much software is not secure, and the risks posed to systems that contain non-secure software. Enhancing the Development Life Cycle introduces a set of principles to govern risk-aware software engineering, and provides extensive guidance for software practitioners that can help them adapt and enhance their current software life cycle practices to increase the likelihood that the software they produce will be more dependable, trustworthy, and survivable...in other words, more secure.

Benefiting from collaborative contributions and critiques by participants in the Software Assurance Forum, Enhancing the Development Life Cycle provides information intended to prepare its readers to evaluate and choose from among the growing number of secure software development methodologies, practices, and technologies best suited for adoption by their own development organizations to help reshape their life cycle processes and practices..

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:

a. REPORT	b. ABSTRACT	c. THIS PAGE
U	U	U

17. LIMITATION OF ABSTRACT
UU

18. NUMBER OF PAGES: 331

19a. NAME OF RESPONSIBLE PERSON
Thomas McGibbon

19b. TELEPHONE NUMBER (include area code)
315-838-7094

FOREWORD

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 Supervisory Control and Data Acquisition systems to commercial-off-the-shelf applications. 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.*

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.*

Software Assurance has emerged in response to the dramatic increases in business and mission risks that are now known to be attributable to exploitable software, including:

- Dependence on software components of systems despite their being the weakest link in those systems;
- Size and complexity of software that obscures its intent and precludes exhaustive testing;
- Outsourcing of software development and reliance on unvetted software supply chains;
- Attack sophistication that eases exploitation of software weaknesses and vulnerabilities;
- Reuse and interfacing of legacy software with newer applications in increasingly complex, disparate networked environments resulting in unintended consequences and the increase of vulnerable software targets.

The growing extent of the resulting risk exposure is not yet well understood. The number of threats specifically targeting software is increasing, as the majority of today's network- and system-level attacks exploit vulnerabilities in application-level software. These factors combine to the increase of risks to software-enabled capabilities and the vulnerability of software-intensive systems to asymmetric cyber threats. Only by establishing the basis for justifiable confidence in the software that enables their core business operations can the organizations that depend on software-intensive systems trust those systems to continue performing in a dependable, trustworthy manner, even in the face of attack.

Enhancing the Development Life Cycle to Produce Secure Software joins a growing body of software assurance information resources and tools provided through the Department of Homeland Security (DHS) BuildSecurityIn Web portal (https://buildsecurityin.us-cert.gov) that are intended to assist software developers, architects, acquirers, and educators in the improvement and verification of the quality, reliability, and security of the software they produce or procure—and in establishing the justification to use that software with confidence.

Enhancing the Development Life Cycle to Produce Secure Software[1] is intended to complement *Software Security Assurance: A State-of-the-Art Report*,[2] which provides an broad overview of the current methodologies, practices, technologies, and activities engaged in by government, industry, and academia for producing secure software and verifying software's security. *Enhancing the Development Life Cycle* complements *Software Security Assurance* by describing in greater technical depth and detail the security principles and practices that software developers, testers, and integrators can adopt to achieve the twin objectives of producing more secure software-intensive systems, and verifying the security of the software they produce.

---

1 Sponsored by the DHS Software Assurance Program and collaboratively developed through contributions of the Software Assurance Forum working groups, the document is published by the Defense Technical Information Center's Data and Analysis Center for Software (DACS) as a community resource.

2 Goertzel, Karen Mercedes, *et al. Software Security Assurance: A State-of-the-Art Report.* Herndon, Virginia: Information Assurance Technology Analysis Center (IATAC) of the DTIC, 31 July 2007. Accessed 28 January 2008 at: http://iac.dtic.mil/iatac/download/security.pdf

Enhancing the Development Life Cycle benefited greatly from contributions and critiques by participants in the Software Assurance Forum. The Software Assurance Forum was established jointly by the DHS and Department of Defense (DoD) Software Assurance Programs to provide a venue in which relevant initiatives and organizations across the private sector, academia, and government agencies can collaborate and partner to improve the state of the art of software development and acquisition by shaping a comprehensive strategy that addresses people, processes, technologies, and acquisition practices throughout the software life cycle.

DHS established its Software Assurance Program, consistent with the National Strategy to Secure Cyberspace which included action/recommendation 2-14: *"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."*

The co-sponsored Software Assurance Forum provides a public-private venue for stakeholders to discuss relevant issues and promote the objectives of software assurance throughout the development life cycle. The software assurance objectives are:

1. **Dependability (Correct and Predictable Execution):** Justifiable confidence can be attained that software, when executed, functions only as intended;

2. **Trustworthiness:** No exploitable vulnerabilities or malicious logic exist in the software, either intentionally or unintentionally inserted;

3. **Resilience (and Survivability):** If compromised, damage to the software will be minimized, and it will recover quickly to an acceptable level of operating capacity;

4. **Conformance:** A planned and systematic set of multi-disciplinary activities will be undertaken to ensure software processes and products conform to requirements and applicable standards and procedures.

A growing number of software and security engineers have identified and now promote principles, practices, and technologies that support these software assurance objectives. Their experiences have provided real-world evidence that software produced and vetted by such tools and practices exhibit fewer faults and weaknesses that are exploitable as vulnerabilities, while also increasing the software's overall dependability and quality. *Enhancing the Development Life Cycle* describes these principles, practices, and technologies.

Recognizing that no single practice, process, or methodology offers a universal "silver bullet" for software security, the authors of *Enhancing the Development Life Cycle* do not set out to recommend a specific methodology. In this way, *Enhancing the Development Life Cycle* differs from many other works published on secure software engineering, secure programming, secure coding, application security, and similar topics.

What *Enhancing the Development Life Cycle* does do is present software practitioners with background in software security concepts and issues, and describes a set of frequently cited secure software principles and security-enhancing development practices. These are principles and practices that have been demonstrated in software development projects across government, industry, and academia, in the United States (U.S.) and abroad, to aid in the

production, verification, and sustainment of software that is more dependable, more trustworthy, and more resilient than software produced without the benefit of those principles and practices. The information in *Enhancing the Development Life Cycle* is intended to prepare its readers to evaluate and choose from among the growing number of secure software development methodologies, practices, and technologies those best suited for adoption by their own development organizations to help reshape their life cycle processes and practices.

Enhancing the Development Life Cycle 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 the DoD, DHS, or any other federal government organization. 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 and should be addressed to the DACS at swa@thedacs.com and Joe.Jarzombek@dhs.gov.

Joe Jarzombek, PMP
Director for Software Assurance
National Cyber Security Division
Department of Homeland Security

Other software assurance documents are freely available through the Software Assurance Program of the U.S. Department of Homeland Security (DHS) National Cyber Security Division. The documents and resource material can be downloaded via the "Software Assurance Community Resources and Information Clearinghouse" at https://buildsecurityin.us-cert.gov/swa and the "Build Security In" website at https://buildsecurityin.us-cert.gov.

The Software Assurance Forum and Working Groups have provided collaborative venues for stakeholders to share and advance techniques and technologies relevant to software security, and they provide resources and seek ongoing feedback to improve those resources.

Software Security Assurance: A State-of-the-Art Report (SOAR) represents an output of collaborative efforts of software assurance subject matter experts in the Department of Defense (DoD) Information Assurance Technology Analysis Center and Data and Analysis Center for Software, in concert with organizations and individuals in the SwA Forum and working groups. The SOAR provides an overview of the current state of the environment in which software must operate and surveys current and emerging activities and organizations involved in promoting various aspects of software security assurance. The report also describes the variety of techniques and technologies in use in government, industry, and academia for specifying, acquiring, producing, assessing, and deploying software that can, with a justifiable degree of confidence, be said to be secure. The report also presents observations about noteworthy trends in software security assurance as a discipline. Many other SwA resources are provided by the SwA working groups.

"Software Assurance in Acquisition: Mitigating Risks to the Enterprise" was published in October 2008 through the National Defense University Press. This collaboratively developed document provides a reference for security-enhanced software acquisition and outsourcing by incorporating SwA throughout the acquisition process from the acquisition planning phase to contracting, monitoring and acceptance, and follow-on phases. For each phase, the material covers SwA concepts, recommended strategies, and acquisition management tips. It also includes recommended request for proposal and/or contract language and due diligence questionnaires that may be tailored to facilitate the contract evaluation process. A pre-production copy is freely downloadable via the DHS SwA Community Resources and Information Clearinghouse at https://buildsecurityin.us-cert.gov/swa/acqact.html

"Practical Measurement Framework for Software Assurance and Information Security" was published in October 2008 through the Practical Software and Systems Measurement Support Center. It provides an approach for measuring the effectiveness of achieving Software Assurance (SwA) goals and objectives at an organizational, program or project level. It addresses how to assess the degree of assurance provided by software, using quantitative and qualitative methodologies and techniques. This framework incorporates existing measurement methodologies and is intended to help organizations and projects integrate SwA measurement into their existing programs. The document can be free downloaded via the PSM Support Center web site under "Products" then "white papers". The link is: http://www.psmsc.com/Prod_TechPapers.asp. The link directly to the paper is: http://www.psmsc.com/Downloads/TechnologyPapers/SwA%20Measurement%2010-08-8.pdf

CREDITS AND ACKNOWLEDGEMENTS

Enhancing the Development Life Cycle to Produce Secure Software, Version 2.0, October 2008

Authors: Karen Mercedes Goertzel and Theodore Winograd (Booz Allen Hamilton)

Contributors: Holly Lynne McKinley, Patrick Holley, Kwok Cheng, Tom Gullo, Jaime Spicciati, Edward W. Tracy, Scott Jung, Stan Wisseman (Booz Allen Hamilton); Michael Howard, Steven Lipner (Microsoft Corporation), Mark S. Kadrich (Symantec), William Bradley Martin (National Security Agency), William Scherlis (Carnegie Mellon University [CMU])

Editor: Karen Mercedes Goertzel (Booz Allen Hamilton)

Publisher: Data and Analysis Center for Software (part of the Defense Technical Information Center)

Reviewers, Version 2: Rod Chapman (Praxis High Integrity Systems), Mary Ann Davidson (Oracle Corp.), Robert J. Ellison (Carnegie Mellon University), Tim Halloran, Joe Jarzombek (DHS National Cyber Security Division), Nancy Mead (Carnegie Mellon University), John F. Miller (The MITRE Corporation), Michele Moss (Booz Allen Hamilton), Haralambos Mouratidis (University of East London), Peter G. Neumann (SRI International), Garry Poliakov, Samuel T. Redwine (James Madison University), Andras Szakal (International Business Machines [IBM]), Carol Woody (Carnegie Mellon University)

Reviewers, Version 1: Daniel Cross (DHS), Robert Seacord, Robert Ellison, Nancy R. Mead (CMU), Jeremy Epstein (WebMethods), Mandee Khera (Cenzic Inc.), Morana Marco (Foundstone, Inc.), Robert Martin (The MITRE Corporation), C. McLean (Ounce Labs, Inc.), Don O'Neill (Center for National Software Studies), Samuel T. Redwine, Jr. (James Madison University), Robin Roberts (Cisco Systems), Michael J. Sherman (Digital Sandbox, Inc.), Michael Smith (Symantec), Frank Stomp (Wayne State University), Richard Struse (Voxem), Lucinda Gagliano, Shelah Johnson, Edward Tracy, Stan Wisseman (Booz Allen Hamilton).

We also thank the other members of the Software Assurance Forum, particularly the Processes and Practices Working Group, for their insights and suggestions during the initial development and ongoing refinement of this document. And special thanks to Joe Jarzombek, the Director for the DHS Software Assurance Program, for leading the national public-private effort to promulgate best practices and methodologies that promote integrity, security and reliability in software code development.

TABLE OF CONTENTS

LIST OF FIGURES

1 INTRODUCTION

This section provides information on this document's content and intended readers.

> *NOTE: This document is not intended to be comprehensive, nor is it intended to be the only document the reader will ever have to consult about producing secure software. It is, instead, intended to provide a roadmap to help the reader navigate and intelligently select from among the multiplicity of other resources available on the various aspects of software security and assurance.*

1.1 DOCUMENT PURPOSE AND INTENDED AUDIENCE

The main purpose of *Enhancing the Development Life Cycle* is to arm developers, integrators, and testers with the information they need to incorporate security considerations and principles into the practices and processes they use to produce software, and thereby increase the likelihood that the resulting software will be secure. What "secure" means as it relates to software is explained in Section 2.

The intended readers for this document are "software practitioners", who include:

1. Requirements analysts,
2. Architects and designers,
3. Programmers (also known as "coders"),
4. Software integrators,
5. Testers,
6. Maintainers,
7. Software configuration managers,
8. Security experts assigned to work with software development teams (see note),
9. Software project technical leads/managers.

> *NOTE: Whereas this document is intended primarily to familiarize software practitioners with security concepts, principles, and practices, the security expert can read this document to better grasp the relationship between security concepts and principles he/she already understands, how they pertain to software, and how they can be applied to software practices – both areas with which the security expert is likely to be unfamiliar.*

The principles and practices identified in this document can inform the use of a variety of frameworks, models, and standards that provide the infrastructure for repeatable processes and continuous process improvement, including ISO 15288 and Capability Maturity Model Integration. To support the integration of assurance considerations in the development lifecycle, an industry working group created a draft set of assurance goals and practices that harmonize existing practices in the Motorola Secure Software Development Model (MSSDM), the System Security Engineering Capability Maturity Model (SSE-CMM), and other secure

system and software engineering models and experiences. See the Suggested Resource list below for references.

Please note that while the software project manager is listed as an intended beneficiary of the information in this document, issues of security in software project management and software process improvement at the level addressed by CMMs and similar process models both fall outside the scope of this document. There are a number of good resources for software project managers to consult in these areas. These are listed in the Suggested Resources list below.

This document assumes that the reader is already familiar with good general software development concepts, principles, and practices. To benefit fully from the content of this document, the reader should also be familiar with key information security and cyber security concepts, such as "trust", "privilege", "integrity", and "availability". See the Suggested Resources at the end of Section 3.2.1 for some helpful Information Assurance (IA) and Information System Security resources.

CAVEAT ABOUT TERMINOLOGY:
Developers, Designers, Architects

While the authors recognize that there are a number of different roles—those listed above plus others—involved in the production and sustainment of software, we have chosen in this document to frequently use the term *developer* as a general designator for anyone directly involved with the development of software at some point in its life cycle: this includes the requirements analyst, the architect, the designer, the programmer, the software (*vs.* system) integrator, and the maintainer. We can only hope that this imprecision of terminology will not confuse the reader, and that he/she will understand when, for example, within the discussion of requirements engineering we may refer to the developer rather than the (requirements) analyst. We generally refer to the tester separately, to clearly differentiate the function of evaluating software (source code or binary executable) that has already been developed from that of reviewing code as it is still being written.

Also, the authors do not attempt to make a clear distinction between the software (*vs.* system) architect and the software designer—either of whom may be referred to as designer or developer; this is because we feel that the two aspects of software definition tend to dovetail and blur into each other, and thus any attempt at a clear distinction would be neither accurate or helpful.

1.2 DOCUMENT STRUCTURE AND CONTENT

Enhancing the Development Life Cycle consists of nine chapters and seven appendices, each of which is described in Table 1-1 below. The table also indicates which readers (by role) will benefit most from the content of each section and subsection.

Table 1-1. Document structure and content

Section	Content	Who will benefit most from reading?
1	**Introduction: Document purpose, intended audience, structure, and content description**	All
2	**Background: Understanding the problem**	All
3	**Integrating security into the SDLC**	
3.1	Influence on how software comes to be on its security	Project manager
3.2	General software security principles	All
3.2.1	Software assurance, information assurance, and system security	Project managers Requirements analysts Integrator
3.3	Secure development life cycle activities and practices	Project manager
3.4	Secure version management and change control of SDLC artifacts	Configuration manager
3.5	Security assurance cases for software	Project manager
3.6	SDLC methodologies that aid in secure software production	Project manager
4	**Requirements for secure software**	
4.1	The challenge of negative and non-functional requirements	Requirements analyst
4.2	Origins of requirements for secure software	Requirements analyst Project manager
4.3	Deriving requirements that will ensure security of software	Requirements analyst
4.4	Secure software requirements verification challenges	Requirements analyst
4.5	Requirements engineering and security modeling methodologies and tools	Requirements analyst
4.5.1	Attack modeling	Requirements analyst Tester *(test planning)* Requirements analyst

Section	Content	Who will benefit most from reading?
5	**Secure design principles and practices**	
5.1	Secure architecture considerations	Architect
5.2	Secure software design principles and practices	Designer
5.3	Modeling and risk analysis for architecture and design	Architect Designer
5.4	Relationship of security patterns to secure software	Designer
5.5	Execution environment security contraints, protections, and services for software	Architect Integrator
5.6	Secure architecture and design methodologies	Architect Integrator
6	**Secure component-based software engineering**	
6.1	Architecture and design considerations for component-based software systems	Architect Designer Integrator
6.2	Security issues associated with COTS and OSS components	Architect Integrator
6.3	Security evaluation and selection of components	Architect Integrator
6.4	Implementing secure component-based software	Architect Integrator
6.4	Secure sustainment of component-based software	Integrator
7	**Secure coding principles and practices**	Programmer
8	**Risk-based software security testing**	Tester
9	**Secure distribution, deployment, and sustainment**	
9.1	Preparations for secure distribution	Programmer Integrator
9.2	Secure distribution	Program manager
9.3	Secure installation and configuration	Program manager
9.4	Secure sustainment considerations	Program manager Maintainer
App. A	**Abbreviations, acronyms, and definitions**	All
App. B	**Resources and Bibliography**	All

Section	Content	Who will benefit most from reading?
App. C	**Software assurance concerns raised by specific technologies, methodologies, and programming languages**	
C.1	Security concerns associated with Web service software	All *(for application software)*
C.2	Security concerns associated with embedded system software	All *(for embedded software)*
C.3	Formal methods and secure software	All *(for high-consequence software)*
C.4	Security benefits and concerns associated with specific programming languages	Programmers
	Leveraging Design by Contract™ for software security	Programmers
App. D	**Security checklist excerpts**	Integrators *(evaluators of components)* Testers *(test planners)*

At the end of many sections, the reader will find a list of resources that provide more detailed, in-depth explanatory information on the section's topic, or that provide practical guidance on how to implement the different activities, techniques, and practices referred to in the section. All of these resources, as well as those cited in footnotes throughout this document, also appear, listed alphabetically by author (or in the case where no author's name is know, by title), in Appendix B.

> ***NOTE:*** *References throughout this document to* "Software Security Assurance" *refer to Goertzel, Karen,* et al. Software Security Assurance: A State-of-the-Art Report. *Herndon, Virginia: Information Assurance Technology Analysis Center, 31 July 2007. Accessed 28 January 2008 at: http://iac.dtic.mil/iatac/download/security.pdf*

2 BACKGROUND: UNDERSTANDING THE PROBLEM

2.1 WHAT IS SECURE SOFTWARE?

Secure software is not the same as software that performs security-relevant functions. While the performance of security functions is an excellent rationale for ensuring that the software that performs them is secure, the fact that software performs security-relevant functions does not assure the software's own secure behavior and interactions. Security functionality in a software-intensive system is critical to assuring system security, but does very little to assure software security.

A security function on which the system relies for protection will be of little value if the software that has implemented that function contains exploitable weaknesses that can be used to bypass or compromise the dependable operation of that function. By the same token, a system that cannot correctly authenticate its users, control access to its resources, or validate digital signatures when necessary will be non-secure regardless of whether the poorly implemented authentication, access control, or signature validation software contains no exploitable software vulnerabilities. Both of system-level security and software level-security must be assured for the system to be truly secure; moreover, the software that implements the system's security components/functions for it to be possible to assure the system itself is secure.

To be considered secure, software must exhibit three properties:[3]

1. **Dependability:** Dependable software executes predictably and operates correctly under all conditions, including hostile conditions, including when the software comes under attack or runs on a malicious host.

2. **Trustworthiness:** Trustworthy software contains few if any vulnerabilities or weaknesses that can be intentionally exploited to subvert or sabotage the software's dependability. In addition, to be considered trustworthy, the software must contain no malicious logic that causes it to behave in a malicious manner.

3. **Survivability (also referred to as "Resilience"):** Survivable—or resilient—software is software that is resilient enough to (1) either resist (*i.e.,* protect itself against) or tolerate (*i.e.,* continue operating dependably in spite of) most known attacks plus as many novel attacks as possible, and (2) recover as quickly as possible, and with as little damage as possible, from those attacks that it can neither resist nor tolerate.

3 At a system security level, these properties are equally important. However, they would probably be discussed in the terms of of the need for Availability, Integrity, Quality of Protection, and Quality of Service.

The main objective of software assurance is to ensure that the processes, procedures, and products used to produce and sustain the software conform to all requirements and standards specified to govern those processes, procedures, and products.

A number of factors influence how likely software is to consistently exhibit these properties under all conditions. These include:

- **Development principles and practices:** The practices used to develop the software, and the principles that governed its development;
- **Development tools:** The programming language(s), libraries, and development tools used to design, implement, and test the software, and how they were used by the developers;
- **Acquired components:** How commercial-off-the-shelf (COTS) and open source software (OSS) components were evaluated, selected, and integrated;
- **Deployment configuration:** How the software was configured during its installation;
- **Execution environment:** The nature and configuration of the protections provided to higher-level software by its underlying and surrounding execution environment;[4]
- **Practitioner knowledge:** The level of security awareness and knowledge of the software's analysts, designers, developers, testers, and maintainers…or their lack thereof.

---

4 A software program or component may reside at the application layer, the middleware layer, or the operating system layer (including kernel, device drivers, *etc.*), or the firmware layer. For purposes of this document, middleware software is that software that provides services and interfaces to application-level software to provide it with support capabilities that it cannot obtain from the operating system; examples of middleware include database management systems, Web servers, public key encryption software virtual machine monitors, *etc.*

Regardless of the software's level, all software has a "platform" execution environments on which it is directly hosted. This platform environment consists of the hardware platform consisting of hardware and firmware, plus any other software at the level(s) below that of the software being "hosted" in the environment. For example, operating system software has a platform environment that consists solely of hardware and firmware, except in those cases where the operating system is hosted on top of a low-level virtual machine monitor, which acts as middleware between the operating system and the hardware. For middleware-level software the platform execution environment expands to include the underlying operating system software, which provides the middleware with services and interfaces that enable it to access the file system, network interfaces, environment variables, and other middleware-level and application-level processes executing on the same physical host. For application-level software, the execution environment expands further to include middleware. Much of today's software is "networked" or "distributed", which means that its execution environment extends beyond its own platform, to include external network-based infrastructure services, such as certificate and identity management services, directory services, *etc.*

Both research and "real world" experience indicate that correcting weaknesses and vulnerabilities as early as possible in the software's life cycle is far more cost-effective over the lifetime of the software than developing and releasing frequent patches for deployed software.

Experience has also taught that the most effective way to achieve secure software is for its development life cycle processes to rigorously conform to secure development, deployment, and sustainment principles and practices. Organizations that have adopted a secure SDLC process have found almost immediately upon doing so that they have begun finding many more vulnerabilities and weaknesses in their software early enough in the SDLC that they are able to eradicate those problems at an acceptable cost. Moreover, as such secure practices become second nature over time, these same developers start to notice that they seldom introduce such vulnerabilities and weaknesses into their software in the first place.

2.2 WHY DOES SOFTWARE ASSURANCE MATTER?

The reason software assurance matters is that so many business activities and critical functions—from national defense to banking to healthcare to telecommunications to aviation to control of hazardous materials—depend on the on the correct, predictable operation of software. It is safe to say that in today's world, these and myriad other activities and functions would become hopelessly crippled if not completely impossible were the software-intensive systems that they rely on to fail.

This extreme reliance on software makes it a very high-value target for those who wish to exploit or sabotage such activities and functions, whether for financial gain, political or military advantage, to satisfy ideological imperatives, or out of simple malice.

It is this virtually guaranteed presence of flaws and defects that makes software such an easy target for attackers. Software flaws and defects can cause software to behave incorrectly and unpredictably, even when it is used purely as its designers intended. Moreover, a number of software flaws and defects can be intentionally exploited by attackers to subvert the way the software operates—making it untrustworthy—or to sabotage the software's ability to operate—making it undependable.

The inherent complexity of software-intensive systems makes it extremely difficult to establish whether such systems are secure. Even when the secure behaviors and lack of vulnerabilities in individual software components can be verified, such verifications will not enable an analyst to predict whether each component's secure behavior will continue to be exhibited when that component interacts with other components in the system, nor whether new vulnerabilities may emerge from the interactions of those components. Nor can the security of the system as a whole be extrapolated from the security of its individual components.

According to the Computer Emergency Response Team Coordination Center at CMU, most successful attacks on software result from successful exploitation of the software's known vulnerabilities and non-secure configurations.

Software is subject to threats at various points in its life cycle, perpetrated by either insiders—individuals closely affiliated with the organization that is producing, deploying, operating, or maintaining the software, and thus trusted by that organization—or outsiders who have no affiliation-based trust relationship with the organization. These threats can occur:

- **During development** *(mainly insider threats):* A developer may corrupt the software—intentionally or unintentionally—in ways that will compromise that software's dependability and trustworthiness when it is operational.
- **During deployment** *(mainly insider threats):* Those responsible for distributing the software may fail to tamperproof the software before shipping or uploading, or may transmit it over easily intercepted communications channels. The software's installer may fail to "lock down" the host platform, and may configure the software insecurely. The user organization may not only fail to apply necessary patches and updates, but may fail to upgrade to newer, supported versions of the software from which the root causes of such vulnerabilities may have been eliminated.
- **During operation** *(both insider and outsider threats):* Any software system that runs on a network-connected platform will have its vulnerabilities exposed during its operation. The level of exposure will vary depending on whether the network is public or private, Internet-connected or not, and whether the software's environment has been configured to minimize its exposure. But even in highly controlled networks and "locked down" environments, potential threats from malicious insiders (users, administrators, *etc.*) remain, as do potential threats from the software's own untrustworthy behavior.
- **During sustainment** *(mainly insider threats):* Those responsible for addressing discovered vulnerabilities in released software fail to issue patches or updates in a timely manner to correct those vulnerabilities. Moreover, they fail to seek out and eliminate the root causes of the vulnerabilities to prevent their perpetuation in future releases of the software.

NOTE: The secure development principles, practices, and methodologies discussed in this document are intended to help reduce the exposure of software to insider threats during its development process.

2.3 WHICH SOFTWARE MUST ALWAYS BE SECURE?

There are types of software and critical portions of software-intensive systems for which security should always be a high priority. These include software that performs "trusted" functions and software that implements interfaces among software components, between software and network components, and between software and human users.

Software that performs "trusted" functions includes security-critical, safety-critical, and other high-consequence functions the dependability and trustworthiness of which are vital. Such functions include those that perform security policy decision-making and enforcement, protect sensitive data, avoid critical failures, perform critical calculations, measurements, and

configurations (such as targeting calculations in weapons systems, temperature monitoring and control in nuclear power plants, metering in medical life-support systems), *etc.*

Interfaces include remote procedure calls and programmatic interfaces between the software system's own components, modules, and processes, and interfaces between the software and external entities including those in the execution environment, peer software that provides services to or receives services from the software system, and human and software-agent users.

> *NOTE: In the DoD, Intelligence Community, or Federal civilian agencies, the criticality of information system software is indicated to some extent by its Mission Assurance Category (defined in DoD Instruction 8500.2),*[5] *Protection Level (defined in Director of Central Intelligence Directive [DCID] 6/3 "Protecting Sensitive Compartmented Information within Information Systems"),*[6] *or Impact Levels (defined in Federal Information Processing Standard [FIPS] 199).*[7]

2.4 WHAT MAKES SOFTWARE SECURE?

The main characteristics that discriminate the developer of secure software from that of non-secure software are awareness, intention, and caution. A software professional who cares about security and acts upon that awareness will recognize that software vulnerabilities and weaknesses can originate at any point in the software's conception or implementation: from inadequate requirements, to poor design and implementation choices, to inadvertent coding errors or configuration mistakes.

The security-aware software professional knows that the only way these problems can be avoided is through well-informed and intentional effort: requirements analysts must understand how to translate the need for software to be secure into actionable requirements, designers must recognize and choices that conflict with secure design principles, and programmers must follow secure coding practices and be cautious about avoiding coding errors, and finding and removing the bugs they were unable to avoid. Software integrators must recognize and strive to reduce the security risk associated with vulnerable components (whether custom-built, COTS, or open source), and must understand the ways in which those modules and components can be integrated to minimize the exposure of any vulnerabilities that cannot be eliminated.

---

5 DoD Instruction 8500.2, *Information Assurance (IA) Implementation.* 6 February 2003. Accessed 25 January 2008 at: http://www.dtic.mil/whs/directives/corres/html/850002.htm.

6 Director of Central Intelligence Directive 6/3 (DCID 6/3), *Protecting Sensitive Compartmented Information within Information Systems – Manual.* Accessed 28 January 2008 at: http://www.fas.org/irp/offdocs/dcid-6-3-manual.pdf

7 NIST Computer Security Division. *Standards for Security Categorization of Federal Information and Information Systems.* FIPS 199, February 2004. Accessed 26 January 2008 at: http://csrc.nist.gov/publications/fips/fips199/FIPS-PUB-199-final.pdf

The main reason for adding security practices throughout the SDLC is to establish a development process that codifies both caution and intention, so that:

1. Well-intentioned developers are more likely to capture requirements adequately, and far less likely to make poor design choices and make, or at least leave in, inadvertent coding errors as software is being developed;

2. Malicious developers will find it very difficult to surreptitiously plant exploitable vulnerabilities, weaknesses, and malicious logic into software under development;

3. The software that results from the secure SDLC process will be secure in execution, and in its interactions with external entities (users, execution environment, and other software).

The key elements of a secure SDLC process are:

1. **Security criteria in SDLC checkpoints:** Security criteria are included in each SDLC phase's entry and exit checkpoints. The checkpoint at the exit of a phase is intended to ensure that the product(s)—or *artifact(s)*—of that phase (*e.g.,* specification, architecture, design, code) is sufficient as the basis for developing the artifact(s) of the next phase, For example, the exit criteria for the requirements specification should establish that the specification is clear, detailed, and traceable enough that an architecture and design that will satisfy its requirements can, in fact, be defined with sufficient detail and completeness to ensure that software can be implemented that is functionally complete and correct, and able to consistently exhibit all other required properties including security.

   For secure software, the security criteria of the phase's exit checkpoint will collectively ensure that all security concerns for the phase's artifact(s) have been explicitly and sufficiently addressed. In short, the evaluators of artifacts leaving a given phase should be satisfied that those products contain no unacceptable problems (*e.g.,* inadequate or missing requirements, poor design choices, coding errors, unacceptable interfaces, *etc.*); such unacceptable problems include security deficiencies. The evaluations performed at exit checkpoints include peer reviews, requirements and design reviews, code reviews, security tests, and security audits.

   The purpose of a phase-entry checkpoint is to adjust any findings and assumptions made during the previous phase's exit checkpoint as needed to accommodate new knowledge or changed circumstances. For example, a security safeguard may have been included in the software's design based on the assumption that there were commercial or open source implementations of that specified safeguard available. The discovery that such is not the case, and that the safeguard would have to be custom-developed could lead to the entry-phase evaluator instead requesting that the design be reworked to define an alternative mechanism for addressing the security concern that was supposed to be dealt with by the safeguard. Backward propagation of any such design change might also have to be made in the architecture and even the requirements

specification. Or it may be discovered during the design of a safeguard required in the specification that the safeguard raised an irreconcilable conflict between two requirements.

For example, a requirement for handling user input might have specified that unacceptable input data should be allowed into the system then "sanitized". During the design phase, it might be discovered that an input validation mechanism that was intelligent enough to sanitize input would be likely to impose too much processing overhead. In this way, the requirement for input sanitization conflicts with the system's performance requirements. Instead of attempting to rework the design to define a lower-overhead sanitization filter that is still capable enough to prevent admittance of malicious input (to avoid satisfying performance requirements at the expense of security requirements), it might be easier to simply change the specification to require defensive blocking of unacceptable input at its point of entry into the system rather than admitting the unacceptable input, then attempting to intelligently sanitize it.

> *NOTE: In component-based software development, the products of any life cycle phase may, in fact, be acquired from external sources (e.g., COTS or OSS suppliers) rather than custom-developed. The checkpoint evaluations performed on such artifacts fall under the rubric of supplier assurance. Supplier assurance is an element of secure software acquisition and project management, for which resources are listed at the end of this section.*

2. **Secure software principles and practices:** All development processes and their artifacts conform to secure architecture, design, coding and integration, and testing principles and practices, such as those described in later sections of this document.

3. **Adequate requirements:** Elicitation, derivation, and specification of requirements includes adequate, complete requirements for constraints on the software's functionality and behavior ("negative" requirements) as well as non-functional requirements pertaining to development and evaluation processes, operational constraints, *etc.*, to ensure the software's dependability, trustworthiness, and resilience.

4. **Adequate architecture and design:** The architecture and design are carefully reviewed to ensure that they:

    a. Reflect correct developer assumptions about all possible changes that might arise in the software's environment, including changes in the inputs the software receives from the environment and changes in the states of environment components on which the software relies for its correct operation under normal conditions. The anticipated changes should include changes associated with likely attempts to attack and compromise the software, as well as simple anomalies in the environment's behaviors and outputs.

    b. Reflect correct developer assumptions about all possible state changes in the software itself were its active defenses against environment changes to fail. These

assumptions should include assumptions about the software's behavior under *all* possible operating conditions, not just those associated with "normal usage", and should anticipate not only external attacks and anomalies, but also the expected results of the execution of various types of malicious logic embedded within the software itself;

c. Form an adequate basis for implementing software that will:

   i. Operate in a dependable and trustworthy manner in its intended operational environment(s);

   ii. Use only appropriate and secure interfaces to external components/services, administrator consoles, and users;

   iii. Anticipate all of the different state changes that could result from errors or failures triggered by misuse and abuse (*e.g.,* the latter could be characterized by attack patterns that manifest as errors or anomalies in input streams);

d. Address security concerns associated with COTS and OSS components, such as determining means for preventing the inadvertent or maliciously intentional triggering of functions (dormant code) that are not expressly intended to be executed during the operation of the software system. Similarly, in open source and legacy-with-source code components, a determination of how to deal with dead code should be made, *i.e.,* weighing the practicality of removing the unnecessary code *vs.* attempting to wrap or otherwise isolate/constrain it to prevent an attacker (or an anomaly) from accessing and executing it.

5. **Secure coding:** Includes both coding and integration of software components. Coding follows secure coding practices and adheres to secure coding standards. Static security analysis of code is performed iteratively throughout the coding process, to ensure that security issues are found and eliminated before code is released for unit testing and integration.

6. **Secure software integration:** Software units/modules/components are integrated with the following considerations in mind:

   a. Ensuring that all programmatic interfaces and procedure calls are inherently secure or that security mechanisms are added to secure them;

   b. Minimizing the exposure to external access of high-risk and known-vulnerable components.

   Integration security testing should focus on exercising the software thoroughly enough to gain a good idea of any unanticipated non-secure behaviors or vulnerabilities that may arise due to interactions among components.

7. **Security testing:** Appropriate security-oriented reviews and tests are performed throughout the SDLC. Tests plans include scenarios include abnormal and hostile conditions among "anticipated conditions" under which the software may operate, and test criteria include those that enable the tester to determine whether the software satisfies its requirements for dependability, trustworthiness, and survivability under all anticipated conditions. The review and test regime includes a broad enough variety and combination of tests to enable the tester to determine whether the software satisfies its requirements for dependability, trustworthiness, and survivability under all anticipated conditions.

8. **Secure distribution and deployment:** Distribution and deployment follows secure practices, which include:

    a. Fully "sanitizing" the software executable(s) to remove unsafe coding constructs and embedded sensitive data, developer backdoors, *etc.,* before transfer to download site(s) and/or distribution media;

    b. Distribution on media or *via* communications channels that adequately protect the software from tampering when it is *en route* to its purchaser or installer; this may include application of digital signatures or digital rights management mechanisms to prevent tampering or unauthorized/ unlicensed installation;

    c. Default configuration settings that are maximally restrictive, with the configuration guide sufficiently informative and detailed that it enables the installer to make informed risk-based decisions about reconfiguring the software with less-restrictive settings if necessary;

    d. Readable, accurate user, administrator, and installer documentation that clearly explains all required constraints and security features of the software.

9. **Secure sustainment:** Maintenance, vulnerability management, and patch issuance and distribution conform to secure sustainment principles and practices. Software customers are encouraged to apply patches and keep software updated, to minimize unnecessary exposure of vulnerabilities.

10. **Supportive development tools:** Development, testing, and deployment tools and platforms that enhance security of produced artifacts and support secure development practices are used throughout the SDLC.

11. **Secure configuration management systems and processes:** Secure software configuration management and version/change control of the development artifacts (source code, specifications, test results, *etc.*) as a countermeasure against subversion of those artifacts by malicious developers, testers, or other SDLC "insiders";

12. **Security-knowledgeable developers:** The security the knowledge the developer needs will clarify the direct relationship between security principles and software engineering

practices. Security education and training of developers needs to go beyond simply explaining what security concepts and principles are relevant. It needs to address the practical ways in which those concepts and principles directly apply to the engineering practices by which software is conceived, implemented, and sustained. The education/training should be sufficient to enable the developer to distinguish between security-enhancing and security-threatening software practices, and to confidently and skillfully embrace the former.

13. **Secure project management and upper management commitment:** Just as software engineering practices are adapted to include considerations and activities to ensure the security of their resulting artifacts, the process and practices for oversight, monitoring, and control of the software project should be similarly security-enhanced. The suggested resources at the end of this section include resources on security-enhanced software project management.

    The software organization's upper management needs to commit to providing adequate support (in terms of resources, time, business priorities, and organizational culture changes) the adoption and consistent use of secure software processes and practices. This includes providing appropriate tools, mandating secure development standards, promoting secure SDLC practices, providing adequate resources for developer education and training, and institutionalizing management verification of and incentives for secure software practices, as well as discouragement of non-secure practices.

Organizations can insert secure development practices into their SDLC process either by adopting a codified secure software development methodology, such as those discussed in Section 3.6, or through the evolutionary security-enhancement of their current practices. Sections 3-10 of this document provide information that should help organizations identify and begin to integrate secure development practices into their current SDLC process.

SUGGESTED RESOURCES

- Polydys, Mary Linda and Stan Wisseman. *Software Assurance in Acquisition: Mitigating Risks to the Enterprise*, Draft Version 1.0, 10 September 2007. Accessed 30 May 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/908.html?branch=1&language=1
- National Defense Industrial Association System Assurance Committee. *Engineering for System Assurance*, Version 0.90, 22 April 2008. Accessed 30 May 2008 at: http://www.acq.osd.mil/sse/ssa/docs/SA+guidebook+v905-22Apr08.pdf
- Fedchak, Elaine, Thomas McGibbon, and Robert Vienneau. *Software Project Management for Software Assurance: A DACS State-of-the-Art Report*, Data & Analysis Center for Software Report Number 347617, 30 September 2007. Accessed 7 July 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/dhs/906-BSI.html
- Allen, Julia H., Sean Barnum, Robert J. Ellison, Gary McGraw, and Nancy R. Mead. *Software Security Engineering: A Guide for Project Managers.* Upper Saddle Ridge, New Jersey: Addison-Wesley Professional, 2008.

2.5 THREATS THAT TARGET SOFTWARE

A threat to a software-intensive system is any actor, agent, circumstance, or event that has the potential to cause harm to that system or to the data or resources to which the system has or enables access. Threats can be categorized according to their intentionality: they can be unintentional, intentional but non-malicious, or malicious (intentionality is a prerequisite of maliciousness).

While threats in all three categories have the potential of compromising the security of software, only malicious threats are realized by attacks. The majority of attacks against software take advantage of, or *exploit*, some vulnerability or weakness in that software; for this reason, "attack" is often used interchangeably with "exploit", though the BuildSecurityIn *Attack Pattern Glossary* makes a clear distinction between the two terms, with *attack* referring to the action against the targeted software and *exploit* referring to the mechanism (*e.g.,* a technique or malicious code) by which that action is carried out.

Threats to software may be present throughout its life cycle—during its development, deployment, and operation. For software in development and deployment, most threats will be "insider" threats, *e.g.,* the software's developers, testers, configuration managers, and installers/administrators. The threats they pose may be unintentional, intentional but non-malicious, or malicious. Table 2-1 provides examples of threats in each category at the three main stages of the software's life cycle.

Table 2-1. Examples of threats to software throughout its life cycle

Threat Category	Development	Deployment	Operation
Unintentional	Typo in source code by incautious developer changes the specified functionality of the software compiled from that source code. Programmer ignorant of secure coding practices writes a C module that makes unsafe library calls.	Administrator accidentally assigns "world" write permissions to the directory in which the software will be installed.	User is able to enter overlong input because the HTML input form did not validate and truncate the excess characters.
Intentional but not malicious	To satisfy customer's directive to make performance the #1 priority, developer eliminates input validation functions that add performance overhead. Programmer pressured by management to deliver source code under a tight deadline foregoes security code review.	Administrator assigns "root" privileges to a software program that was implemented in such a way that it can only run as root.	Frustrated user repeatedly enters unusual command combinations in an effort to bypass a time-consuming pull-down menu data entry interface; Frustrated user repeatedly refreshes and resubmits the same input data to an application that was not designed to return an acknowledgement that the user's input data had been received.
Malicious	Programmer intentionally includes three exploitable flaws and a backdoor in his source code. Integrator appends a logic bomb to an open source program.	Installer leaves the application's default password unchanged to make life easier for attacker with whom she is colluding. Administrator intentionally configures the application firewall to allow inbound Uniform Resource Locators (URLs) that contain executable content.	Attacker launches a Structured Query Language (SQL) injection attack against a Web-based database application. Developer submits a predefined data string to a Web application that he knows will trigger execution of the logic bomb he planted in that application.

It has been observed that generally good software engineering intended to reduce the number and impact of unintentional and non-malicious threats to software such as honest coding errors, will coincidentally reduce the number of threat-exploitable vulnerabilities in that software to some extent—albeit usually not enough to mitigate risk to an acceptable level.

Similarly, when applied throughout the life cycle with the intention of minimizing the software's vulnerability to malicious threats, the secure development principles and practices in this document should have the coincidental benefit of making the software less susceptible to unintentional and non-malicious threats as well.

Not all attacks directly target the software itself. Table 2-2 lists targets of "indirect attacks", *i.e.,* attacks on elements other than the software itself; the compromise of such elements will likely enable the sabotage or subversion of the software itself.

Table 2-2. Targets for indirect attacks

Target	**Attack and objective**
Software boundary or "surface"	Intentional triggering of an external fault (*e.g.*, in an interface mechanism) at the software boundary (or surface) can leave the software vulnerable to direct attack.
Execution environment	Intentional changes of execution environment state from correct/expected to incorrect/unexpected can result in a misbehavior by the software; such a misbehavior can be exploited as a vulnerability.
Trigger for malicious code	Various events may trigger the execution of malicious code such as time bombs, logic bombs, and Trojan horses may be triggered. These include such innocuous events as the computer clock reaching a certain time, a particular file being opened or closed, or a certain parameter value being received.
External services	Most software today relies on other software services to perform functions on its behalf, such as authentication of users or validation of code signatures, or to provide defense in depth protections (*e.g.*, application firewalls). The compromise or unintentional failure of these external services could cause the software to behave unpredictably, while a compromise or failure of external protections may leave the software vulnerable to direct attack.
Multiple target series	A series of exploits may target a particular combination of known vulnerabilities in one or more software components. This type of attack is particularly effective when the components are assembled in a way that emphasizes one component's reliance on a vulnerable function of another. The vulnerabilities most likely to be targeted are in the components' external interfaces, which provide the attacker with direct access to the vulnerabilities.

2.5.1 Attacks that exploit software vulnerabilities

The most common categories of attacks against software-intensive systems and their software components are described below. In most cases, tools exist that can be used by the attacker to automate these techniques.

- **Reconnaissance attacks:** Help the attacker find out more about the software and its environment, so other attacks can be crafted to be more effective; attackers are particularly interested in release and version information about the software's and environment's COTS and OSS components, because such information reveals whether the software/environment includes components with known vulnerabilities that can be exploited.

  A specific concern is the "zero day" vulnerability, *i.e.,* the window of opportunity that is created for attackers when a vulnerability in a specific software version is discovered and exploited before the software's supplier is able to issue a patch (let alone the delay that often occurs in the application of patches after they are issued). The less information that is readily available to zero-day attackers about the type of software being run, the less easily they can craft targeted attacks.

  This said, there is clearly a balance that needs to be struck between not revealing too much information to attackers, and the need to provide certain information to enable diagnostics, patching, automated support, *etc.* There are high-risk and high-consequence systems for which direct vendor access *via* the network to installed software creates an unacceptable level of risk. For such systems, the acquisition language for COTS software needs to include provisions whereby the supplier agrees to accommodate alternative, offline approaches to providing support, patches, *etc.*;

- **Enabling attacks:** Make it easier to deliver other attacks. Examples of enabling attacks are buffer overflow exploits for delivering malicious code, and privilege escalation attacks;

- **Disclosure attacks:** Reveal data that shouldn't been seen by the attacker (compromise of confidentiality);

- **Subversion attacks:** Tamper with and corrupt the software to change the way it operates (compromise of integrity);

- **Sabotage attacks:** Cause the software to fail, or prevent it from being accessed by its intended users; also known as "denial of service" (compromise of availability);

- **Malicious code attacks:** Insert malicious logic into the software, trigger the execution of malicious code already embedded in the software, or deliver/execute malicious code in the software's execution environment.

NOTE: Appendix A of the National Institute of Standards and Technology (NIST) Special Publication 800-95, Guide to Secure Web Services,[8] *provides informative descriptions of common attacks within these categories that target Web applications and Web services.*

Attackers can tamper with many different types of input data in order to deliver malicious payloads to targeted applications. These are the types of input that should always be vetted by input validation to prevent such attack input from reaching the application; they include:

- Command line parameters,
- Environment variables,
- Universal Resource Locators (URLs) and Identifiers (URIs),
- Other filename references,
- Uploaded file content,
- Flat file imports,
- Hyper Text Transfer Protocol (HTTP) headers,
- HTTP *GET* parameters,
- Form fields (especially hidden fields),
- Selection lists, drop-down lists,
- Cookies,
- Java applet communications.

There are several conduits, or "attack paths", by which an attacker may deliver spurious input or malicious code to targeted software systems. The developer needs to be aware of these so as to minimize their unnecessary exposure in the software system's architecture and design. They include:

- **Network elements,** such as the network services and Transmission Control Protocol ports used to enable communications by or with the targeted system, or network security devices relied on to block or filter undesirable input before it reaches the system;

- **Software elements of the system itself** (application-level and middleware-level), including software services, application programmatic interfaces (APIs), remote procedure calls (RPCs), third-party software components, or embedded back doors, Trojan horses, or malicious code;

---

8 Singhal, Anoop, Theodore Winograd, and Karen Scarfone. *Guide to Secure Web Services.* NIST Special Publication 800-95, August 2007. Accessed 26 January 2008 at: http://www.csrc.nist.gov/publications/nistpubs/800-95/SP800-95.pdf

- **Execution environment elements,** such as vulnerabilities in the operating system, runtime system (including any runtime code interpreters), or virtual machine, interference channels, covert channels, and backdoors and rootkits.

The potential impacts of successful attacks on software-intensive systems and their components include:

- Unexpected or unauthorized software execution;
- Unauthorized access to the software, the resources it relies on, or the data it handles;
- Unauthorized changes to the software, the resources it relies on, or the data it handles;
- Denial of service (the software itself, its resources, and/or its data).

Table 2-3 provides a number of examples, derived from the NIST National Vulnerability Database (NVD),[9] of real-world software security vulnerabilities discovered in 2008 and 2007. These vulnerabilities are particularly noteworthy because they all appear in software components that implement system-level security protections, including firewalls, anti-virus scanners, virtual machines, encryption mechanisms, intrusion detection systems, and others.

9 Accessed 11 September 2008 at: http://nist.nvd.gov

Table 2-3. Real-world software vulnerabilities in system security components

CVE Entry	Description	CVSS Severity
Buffer, Heap, and Stack Overflow Vulnerabilities		
CVE-2008-1518	Stack-based buffer overflow in kl1.sys in Kaspersky Anti-Virus 6.0 and 7.0 and Internet Security 6.0 and 7.0 allows local users to gain privileges *via* an IOCTL 0x800520e8 call.	7.2 (HIGH)
CVE-2008-0858	Buffer overflow in the Visnetic anti-virus plug-in in Kerio MailServer before 6.5.0 might allow remote attackers to execute arbitrary code *via* unspecified vectors.	7.5 (HIGH)
CVE-2008-0312	Stack-based buffer overflow in the AutoFix Support Tool ActiveX control 2.7.0.1 in SYMADATA.DLL in multiple Symantec Norton products, including Norton 360 1.0, AntiVirus 2006 through 2008, Internet Security 2006 through 2008, and System Works 2006 through 2008, allows remote attackers to execute arbitrary code *via* a long argument to the GetEventLogInfo method. NOTE: some of these details are obtained from third party information.	9.3 (HIGH)
CVE-2007-6386	Stack-based buffer overflow in PccScan.dll before build 1451 in Trend Micro AntiVirus plus AntiSpyware 2008, Internet Security 2008, and Internet Security Pro 2008 allows user-assisted remote attackers to cause a denial of service (SfCtlCom.exe crash), and allows local users to gain privileges, *via* a malformed .zip archive with a long name, as demonstrated by a .zip file created *via* format string specifiers in a crafted .uue file.	7.2 (HIGH)
CVE-2007-4620	Multiple stack-based buffer overflows in Computer Associates (CA) Alert Notification Service (Alert.exe) 8.1.586.0, 8.0.450.0, and 7.1.758.0, as used in multiple CA products including Anti-Virus for the Enterprise 7.1 through r11.1 and Threat Manager for the Enterprise 8.1 and r8, allow remote authenticated users to execute arbitrary code *via* crafted RPC requests.	9.0 (HIGH)
CVE-2007-3969	Buffer overflow in Panda Antivirus before 20070720 allows remote attackers to execute arbitrary code *via* a crafted EXE file, resulting from an "Integer Cast Around."	9.3 (HIGH)
CVE-2007-3951	Multiple buffer overflows in Norman Antivirus 5.90 allow remote attackers to execute arbitrary code *via* a crafted (1) ACE or (2) LZH file, resulting from an "integer cast around."	7.5 (HIGH)
CVE-2008-0365	Multiple buffer overflows in CORE FORCE before 0.95.172 allow local users to cause a denial of service (system crash) and possibly execute arbitrary code in the kernel context *via* crafted arguments to (1) IOCTL functions in the Firewall module or (2) SSDT hook handler functions in the Registry module.	7.2 (HIGH)

CVE Entry	Description	CVSS Severity
Buffer, Heap, and Stack Overflow Vulnerabilities		
CVE-2007-6092	Buffer overflow in libsrtp in Ingate Firewall before 4.6.0 and SIParator before 4.6.0 has unknown impact and attack vectors. NOTE: it is not clear whether this issue crosses privilege boundaries.	10.0 (HIGH)
CVE-2007-1005	Heap-based buffer overflow in SW3eng.exe in the eID Engine service in CA (formerly Computer Associates) eTrust Intrusion Detection 3.0.5.57 and earlier allows remote attackers to cause a denial of service (application crash) *via* a long key length value to the remote administration port (9191/tcp).	7.8 (HIGH)
CVE-2008-2935	Multiple heap-based buffer overflows in the rc4 (1) encryption (aka exsltCryptoRc4EncryptFunction) and (2) decryption (aka exsltCryptoRc4DecryptFunction) functions in crypto.c in libexslt in libxslt 1.1.8 through 1.1.24 allow context-dependent attackers to execute arbitrary code *via* an XML file containing a long string as "an argument in the XSL input."	7.5 (HIGH)
CVE-2008-2541	Multiple stack-based buffer overflows in the HTTP Gateway Service (icihttp.exe) in CA eTrust Secure Content Manager 8.0 allow remote attackers to execute arbitrary code or cause a denial of service *via* long FTP responses, related to (1) the file month field in a LIST command; (2) the PASV command; and (3) directories, files, and links in a LIST command.	10.0 (HIGH)
CVE-2005-3768	Buffer overflow in the Internet Key Exchange version 1 (IKEv1) implementation in Symantec Dynamic VPN Services, as used in Enterprise Firewall, Gateway Security, and Firewall /VPN Appliance products, allows remote attackers to cause a denial of service and possibly execute arbitrary code *via* crafted IKE packets, as demonstrated by the PROTOS ISAKMP Test Suite for IKEv1.	7.5 (HIGH)
CVE-2007-2454	Heap-based buffer overflow in the VGA device in Parallels allows local users, with root access to the guest operating system, to terminate the virtual machine and possibly execute arbitrary code in the host operating system *via* unspecified vectors related to bitblt operations.	6.8 (MEDIUM)
CVE-2007-2438	The sandbox for vim allows dangerous functions such as (1) writefile, (2) feedkeys, and (3) system, which might allow user-assisted attackers to execute shell commands and write files *via* modelines.	7.6 (HIGH)

CVE Entry	Description	CVSS Severity
Inadequate Input Validation		
CVE-2008-1736	Comodo Firewall Pro before 3.0 does not properly validate certain parameters to hooked System Service Descriptor Table (SSDT) functions, which allows local users to cause a denial of service (system crash) *via* (1) a crafted OBJECT_ATTRIBUTES structure in a call to the NtDeleteFile function, which leads to improper validation of a ZwQueryObject result; and unspecified calls to the (2) NtCreateFile and (3) NtSetThreadContext functions, different vectors than CVE-2007-0709.	7.2 (HIGH)
CVE-2007-4967	Online Armor Personal Firewall 2.0.1.215 does not properly validate certain parameters to System Service Descriptor Table (SSDT) function handlers, which allows local users to cause a denial of service (crash) and possibly gain privileges *via* unspecified kernel SSDT hooks for Windows Native API functions including (1) NtAllocateVirtualMemory, (2) NtConnectPort, (3) NtCreateFile, (4) NtCreateKey, (5) NtCreatePort, (6) NtDeleteFile, (7) NtDeleteValueKey, (8) NtLoadKey, (9) NtOpenFile, (10) NtOpenProcess, (11) NtOpenThread, (12) NtResumeThread, (13) NtSetContextThread, (14) NtSetValueKey, (15) NtSuspendProcess, (16) NtSuspendThread, and (17) NtTerminateThread.	4.4 (MEDIUM)
CVE-2006-7160	The Sandbox.sys driver in Outpost Firewall PRO 4.0, and possibly earlier versions, does not validate arguments to hooked SSDT functions, which allows local users to cause a denial of service (crash) *via* invalid arguments to the (1) NtAssignProcessToJobObject,, (2) NtCreateKey, (3) NtCreateThread, (4) NtDeleteFile, (5) NtLoadDriver, (6) NtOpenProcess, (7) NtProtectVirtualMemory, (8) NtReplaceKey, (9) NtTerminateProcess, (10) NtTerminateThread, (11) NtUnloadDriver, and (12) NtWriteVirtualMemory functions.	4.9 (MEDIUM)
CVE-2006-5721	The \Device\SandBox driver in Outpost Firewall PRO 4.0 (964.582.059) allows local users to cause a denial of service (system crash) *via* an invalid argument to the DeviceIoControl function that triggers an invalid memory operation.	4.9 (MEDIUM)

CVE Entry	Description	CVSS Severity
	Inadequate Input Validation	
CVE-2008-0928	Qemu 0.9.1 and earlier does not perform range checks for block device read or write requests, which allows guest host users with root privileges to access arbitrary memory and escape the virtual machine.	4.7 (MEDIUM)
	Race Conditions	
CVE-2007-3970	Race condition in ESET NOD32 Antivirus before 2.2289 allows remote attackers to execute arbitrary code *via* a crafted CAB file, which triggers heap corruption.	5.0 (MEDIUM)
CVE-2007-1973	Race condition in the Virtual DOS Machine (VDM) in the Windows Kernel in Microsoft Windows NT 4.0 allows local users to modify memory and gain privileges *via* the temporary \Device\PhysicalMemory section handle, a related issue to CVE-2007-1206.	6.9 (MEDIUM)
	Cross-Site Scripting Vulnerabilities	
CVE-2008-2333	Cross-site scripting (XSS) vulnerability in ldap_test.cgi in Barracuda Spam Firewall (BSF) before 3.5.11.025 allows remote attackers to inject arbitrary Web script or HTML *via* the email parameter.	4.3 (MEDIUM)
CVE-2008-3082	Cross-site scripting (XSS) vulnerability in UPM/English/login/login.asp in Commtouch Enterprise Anti-Spam Gateway 4 and 5 allows remote attackers to inject arbitrary Web script or HTML *via* the PARAMS parameter.	4.3 (MEDIUM)
CVE-2008-1775	Cross-site scripting (XSS) vulnerability in mindex.do in ManageEngine Firewall Analyzer 4.0.3 allows remote attackers to inject arbitrary Web script or HTML *via* the displayName parameter. NOTE: the provenance of this information is unknown; the details are obtained solely from third party information.	4.3 (MEDIUM)
	Use of Non-Secure Methods	
CVE-2008-1116	Insecure method vulnerability in the Web Scan Object ActiveX control (OL2005.dll) in Rising Antivirus Online Scanner allows remote attackers to force the download and execution of arbitrary code by setting the BaseURL property and invoking the UpdateEngine method. NOTE: some of these details are obtained from third party information.	9.3 (HIGH)
CVE Entry	**Description**	**CVSS Severity**
	(Not-so-)Simple Coding Mistakes	
CVE-2007-1366	QEMU 0.8.2 allows local users to crash a virtual machine *via* the divisor operand to the aam instruction, as demonstrated by "aam 0x0," which triggers a divide-by-zero error.	4.9 (MEDIUM)

CVE Entry	Description	CVSS Severity
	(Not-so-)Simple Coding Mistakes	
CVE-2008-1596	Trusted Execution in IBM AIX 6.1 uses an incorrect pathname argument in a call to the trustchk_block_write function, which might allow local users to modify trusted files, related to missing checks in the TSD_FILES_LOCK policy for modifications performed *via* hard links, a different vulnerability than CVE-2007-6680.	7.2 (HIGH)
	Non-Secure Software Distribution	
CVE-2007-3849	Red Hat Enterprise Linux (RHEL) 5 ships the rpm for the Advanced Intrusion Detection Environment (AIDE) before 0.13.1 with a database that lacks checksum information, which allows context-dependent attackers to bypass file integrity checks and modify certain files.	CVSS Severity: 1.9 (LOW)
CVE-2006-1117	nCipher firmware before V10, as used by (1) nShield, (2) nForce, (3) netHSM, (4) payShield, (5) SecureDB, (6) DSE200 Document Sealing Engine, (7) Time Source Master Clock (TSMC), and possibly other products, contains certain options that were only intended for testing and not production, which might allow remote attackers to obtain information about encryption keys and crack those keys with less effort than brute force.	CVSS Severity: 2.6 (LOW)

SOURCE: National Vulnerability Database

SUGGESTED RESOURCES

- *Software Assurance* Common Body of Knowledge (CBK),[10] Sections 2.4-2.7.
- McGraw, Gary and Greg Hoglund. *Exploiting Software: How to Break Code.* Indianapolis, Indiana: Addison-Wesley Professional, 2004.
- Dwaikat, Zaid. "Attacks and Countermeasures". *CrossTalk: The Journal of Defense Software Engineering*, October 2005. Accessed 21 December 2007 at: http://www.stsc.hill.af.mil/crosstalk/2005/10/0510Dwaikat.html
- NIST, National Vulnerability Database (NVD). Accessed 8 September 2008 at: http://nist.nvd.gov

10 Software Assurance Workforce Education and Training Working Group. *SoftwareAssurance: A Curriculum Guide to the Common Body of Knowledge to Produce, Acquire and Sustain Secure Software.* Draft Version 1.2, 29 October 2007. Accessed 10 March 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/940/version/1/part/4/data/CurriculumGuideToTheCBK.pdf?branch=main&language=default

2.5.2 Causes of weaknesses and vulnerabilities

The main reasons that security weaknesses and vulnerabilities are introduced into software by its developers include:

1. **Malicious developers, integrators, testers:** Specifying, designing, implementing, and testing the software so that weaknesses, vulnerabilities, and/or malicious logic are intentionally included and/or allowed to remain in the software when it is distributed/deployed. The problem of malicious software practitioners holds true throughout all the SDLC phases discussed below;

2. **Software practitioner ignorance or negligence:** Failure of developers, integrators, testers, configuration managers, *etc.,* who fail to take advantage of opportunities to learn how to recognize and analyze the security implications of their SDLC choices, assumptions, and practices, to correct the choices that are deterrents to secure software production, and to identify and use tools to support their secure SDLC practices is both irresponsible and unprofessional. So is failure by the development organization to actively provide such opportunities.

   Such knowledge will not only improve the developers' secure software engineering skills, it will enable them to appreciate that the secure software engineering, which on first superficial glance may seem inconvenient and time-consuming, actually makes it easier for them to produce specifications, designs, and code that are better overall, to avoid many integration problems, to get their software through testing more smoothly, and to increase its ease of maintenance—all this in addition to increasing its dependability, trustworthiness, and survivability. The problem of inadequately prepared software practitioners affects all of the SDLC phases discussed below;

3. **Failure to place real value on security:** This goes hand in hand with knowledge, because understanding what's at stake is a key motivator for caring about preventing security problems in the first place. A software organizational culture that values security as much as it values productivity, quality, *etc.,* is more likely to produce secure software than one that sees security as an afterthought or merely a matter of regulatory conformance. This cultural mind shift must extend to the people actually responsible for conceiving, producing, and sustaining the software. This is another problem that affects the entire SDLC;

4. **Inadequate tools:** Use of tools and programming languages that discourage (or at best, do not encourage) secure development practices. This problem pervades all of the SDLC phases discussed below;

5. **Requirements problems:**

   - Specification of incorrect, incomplete, or spurious requirements;

   - Failure to specify requirements for constraining unsafe behaviors;

- Failure to specify requirements for safe behaviors;
- Specification of requirements based on inaccurate or incomplete risk assessments or attack models;
- Inappropriate consideration of business risks (*e.g.,* address cost, schedule, other project constraints) in what are intended to be security-driven trade-off choices;

6. Architecture and design problems:

- Non-secure choices in the architecture and design of the software;
- Failure to design according to secure design principles;
- Failure to include security criteria in the design review;

7. Component assembly/integration problems:

a. Incorrect or ineffective use of environment-level and other extra-component security protections and services;

b. Establishment of inappropriate trust relationships among components, or between components and human users;

c. Acceptance of COTS and OSS components that contain known vulnerabilities or embedded malicious logic;

8. Implementation problems:

- Failure to program according to secure coding principles and practices;
- Sloppy coding that introduces bugs into the code;

9. Testing problems:

- Failure to include security test cases and criteria in the software's test plan;
- Failure to perform security tests sufficient to establish software's satisfaction of security test criteria;
- Lack of automated tools to increase efficiency and accuracy of security testing;
- Incorrect interpretation of security test results;
- Failure to eliminate or mitigate weaknesses, vulnerabilities, and malicious logic discovered during testing;
- Lack of a test environment that is similar enough to the target operational environment to ensure meaningful security test results;

10. Distribution and deployment problems:

- Failure to remove residual backdoors, embedded sensitive data, *etc.,* before handing off executables for distribution and deployment;
- Failure to define and implement adequately restrictive default configurations for the software and its environment;
- Failure to produce clear, accurate installer, administrator, and user documentation;
- Inclusion of security vulnerabilities and other evidence of non-secure coding practices in sample code delivered with the software documentation;
- Failure to encourage customers to abandon non-supported old versions and upgrade to newer, supported versions of the software as they become available;

11. Sustainment problems:

- Failure to issue patches for custom software to address vulnerabilities discovered post-release;
- Failure to perform security impact analyses before issuing patches and other changes to the software during its sustainment;
- Failure to perform security impact analyses on supplier patches/updates to the software's COTS and OSS components;
- Failure to perform root-cause analyses of problems that arise during the software's operation. (NOTE: In the software safety community, such root-cause analyses ensure that safety-related problems are identified and fixed so they cannot happen again. The same should be done for security problems.)

Inadequacies on software project management are another significant source of security problems for the software produced or maintained by those projects. As noted in Section 1.1, software project management security falls outside of the scope of this document.

Various vulnerability databases, taxonomies, and listings online and in print (several of which are identified in *Software Security Assurance*) enumerate and describe vulnerabilities commonly found in popular execution environment and application components (*e.g.,* operating systems, Web servers); most notable among these is The MITRE Corporation's Common Weakness Enumeration (CWE).[11]

11 The MITRE Corporation. Common Weakness Enumeration Website. Accessed 28 January 2008 at: http://cwe.mitre.org

Such vulnerability information resources should be of particular interest to developers of component-based software systems as a basis for vulnerability modeling, and for establishing criteria for security evaluation of third-party components and popular/standard technologies; in the latter case "absence of known vulnerabilities" could be an important evaluation criterion.

2.5.2.1 Does vulnerability avoidance equate to security?

When examined by a reviewer or tester, vulnerabilities often appear identical with other "benign" flaws and defects. The characteristics that distinguish them as vulnerabilities are:

1. Their ability to be exploited in a way that prevents (temporarily or permanently) the software's dependable, trustworthy operation;

2. Their exposure to external access by an attacker.

Recognizing and eliminating or mitigating vulnerabilities in software—and doing so as early in the SDLC as possible—has long been considered the cornerstone of secure software development, and it is important that the software developer be able to recognize the potential of certain design flaws and coding errors ("bugs") to be exploited by an attacker, or to themselves cause execution failures that could leave the software vulnerable.

In particular, it is critical that the developer develop an awareness of known vulnerabilities in the COTS and OSS components and technologies that he/she hopes to use in building the software system, and that he/she avoid using components/technologies that contain known vulnerabilities—or, at least, if there are no better alternatives, implement an effective vulnerability mitigation strategy to minimize the potential for those vulnerable components/technologies to be accessed and their vulnerabilities exploited.

There is a limit, however, to how much "vulnerability avoidance" alone actually contributes to the production of secure software. For one thing, the ability to determine the exploitability of a particular design flaw or coding error is an inexact science at best. Moreover, even if it were possible to accurately flag certain flaws and bugs as vulnerabilities, 100% vulnerability avoidance would still be unachievable, because any attempt to eliminate exploitable flaws and bugs would necessarily address only those that were already recognized as being exploitable. It would not address the multitude of flaws, bugs, features, coding constructs, *etc.*, that no-one has yet tried to exploit…which does not mean that someone won't try to exploit them in future. It is a truism that new, unanticipated threats will continue to emerge faster than anyone's ability to identify, locate, and mitigate the flaws and bugs that might be targeted and exploited by those threats.

It would appear, then, that the only hope lies in considering *all* design flaws and coding errors as potentially exploitable, and in striving to produce software that us 100% free of all flaws and bugs. Unfortunately, this is even less realistic than attempting to eliminate only the exploitable subset of flaws and bugs. Apart from of a very, *very* few small safety-critical systems and embedded systems, statistically error-free software has never been achieved.

Obviously, vulnerability avoidance, while an admirable goal, should not be the sole objective in the attempt to produce secure software. A second objective, both more realistic and more achievable, is to produce software that is able to resist most anticipated attacks, to tolerate the majority of attacks it cannot resist, and to quickly and with minimal damage recover from the very few attacks it can neither resist nor tolerate. This requires both developer and tester to learn to recognize not only straightforward attack patterns, but also the subtle, complex attack patterns that are intended to trigger sequences of interactions among combinations of components or processes that the developer never intended to interact.

A third objective for the developer of secure software is ensuring that the software itself is both dependable in its execution and trustworthy in its behavior.

To be secure software must attain not just one or two but all three of these objectives; focusing on any one objective, such as vulnerability avoidance, to the exclusion of the others will not result in software that is secure.

Fortunately, the security principles and practices described (and the many resources referenced) in this document will go a long way to helping developers produce software that:

1. Is likely to contain far fewer exploitable and non-exploitable flaws and errors than most software;

2. Can resist, tolerate, and recover from attacks that exploit any residual vulnerabilities the developer has not managed to avoid.

SUGGESTED RESOURCES

- *Software Security Assurance*,[12] Sections 3.1, 3.1.1, and Appendix C.
- Thompson, Herbert H. and Scott G. Chase. *The Software Vulnerability Guide.* Boston, Massachusetts: Charles River Media, 2005.
- The MITRE Corporation. Common Weakness Enumeration Website. Accessed 21 January 2008 at: http://cwe.mitre.org/
- Fortify Software. Fortify Taxonomy: Software Security Errors Webpage. Accessed 21 January 2008 at: http://www.fortifysoftware.com/vulncat/
- Johansson, Jesper M. and E. Eugene Schultz. "Dealing with contextual vulnerabilities in code: distinguishing between solutions and pseudosolutions". *Computers and Security*, Volume 22 Number 2, 2003, pages 152-159.

2.6 SECURITY TRAINING, EDUCATION, AND PROFESSIONAL CERTIFICATION FOR SOFTWARE PRACTITIONERS

12 *Op. cit.* Goertzel, Karen Mercedes, *et al. Software Security Assurance.*

Requirements analysts, architects, designers, programmers, integrators, and testers must all be motivated to accept responsibility for the security of the software they are involved in producing or maintaining, and rewarded when they do so. Part of this motivation should take the form of security training sufficient to prepare them to model attacks, recognize vulnerabilities that make software susceptible to compromise by those attacks and architecture and design weaknesses that expose those vulnerabilities, and to specify, design, and implement software that will not be unacceptably vulnerable.

Fortunately, the amount of professional training available to teach software practitioners how to produce software that is dependable, trustworthy, and survivable (*i.e., secure*) is steadily growing. So is the inclusion of software security and assurance concepts and techniques in academic computer science and information systems curricula.

Moreover, several organizations have established professional certification programs for software developers that provide evidence of the developer's knowledge of and expertise in the development of secure software.

The following is a list of professional certifications available in the realm of secure software development.

- SANS Software Security Institute GIAC Secure Programmer certifications. Accessed 21 January 2008 at: http://www.sans-ssi.org/
- International Council of Electronic Commerce Consultants (EC-Council) Certified Secure Programmer (CSP) and Certified Secure Application Developer (CSAD) certifications. Accessed 21 January 2008 at: http://www.eccouncil.org/ecsp/index.htm
- Secure University Software Security Expert Boot Camp certification. Accessed 21 January 2008 at: http://www.securityuniversity.net/classes_SI_SoftwareSecurity_Bootcamp.php
- International Institute of Training, Assessment, and Certification Certified Secure Software Engineering Professional certification. Accessed 17 December 2007 at: http://www.iitac.org/content/view/146/lang,en/
- Ciphent Certified Secure Software Developer certification. Accessed 21 January 2008 at: http://www.ciphent.com/training/certification
- Stanford University Center for Professional Development Software Security Foundations Certificate. Accessed 21 January 2008 at: http://scpd.stanford.edu/scpd/courses/ProEd/compSec/

In addition to these, ISC(2), the organization that sponsors the Certified Information Systems Security Professional (CISSP) certification, is in the process of developing a broader software assurance certification that will be comparable to the CISSP in technical depth and scope.

SUGGESTED RESOURCES

- BuildSecurityIn Training and Awareness resources. Accessed 21 January 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/training.html
- *Software Security Assurance*, Section 7.2.

3 INTEGRATING SECURITY INTO THE SDLC

"Security enhancement" of the software development life cycle (SDLC) process mainly involves the adaptation or augmentation of existing SDLC activities, practices, and checkpoints, and in a few instances, it may also entail the addition of new activities/practices/checkpoints not currently included in the SDLC process. In a very few instances, it may also require the elimination or wholesale replacement of certain activities or practices that are known to obstruct the ability to produce secure software.

Creating a secure development community using collaboration technologies and a well integrated development environment promotes a continues process of improvement and a focus on secure development life cycle principles and practices that will result in the production of more dependable, trustworthy, and resilient software-based systems.

The rest of this section is devoted to describing the life cycle activities and practices that need to be "security-enhanced" to contribute to the production of *secure* software.

NOTE: The SDLC process and constituent phases implied in this discussion are notional only. Neither they, nor the order and linearity with which they are presented should be interpreted as prescriptive or as constituting a particular SDLC process model or methodology. The authors recognize that different methodologies and process models assign different names to these phases. Some omit some of the phases, while others include additional phases.

This said, the authors feel that virtually every software practitioner will understand what we intend by the phases we have designated requirements, architecture and design, implementation, testing, distribution and deployment, *and* sustainment, *and will recognize the activities, practices, and checkpoints commonly included in those phases, whether or not they are called something else, or excluded from the particular methodology/process model used on the reader's software projects.*

SUGGESTED RESOURCES

- Zulkernine, Mohammad and Sheikh Iqbal Ahamed. "Software security engineering: toward unifying software engineering and security engineering". Chapter 14 of *Enterprise Information Systems Assurance and System Security: Managerial and Technical Issues* (Warkentin, Merrill and Rayford B. Vaughn, editors). Hershey, Pennsylvania: Idea Group Publishing, 2006.
- Mouratidis, Haralambos and Paolo Giorgini, editors. *Integrating Security and Software Engineering: Advances and Future Vision.* Hershey, Pennsylvania: Idea Group Publishing, 2007.

3.1 INFLUENCE OF HOW SOFTWARE COMES TO BE ON ITS SECURITY

Software comes into existence in one of four different ways:

1. **Acquisition:** Acquisition refers to purchase, licensing, or leasing of non-developmental software packages and components[13] produced by entities other than the acquirer. Such non-developmental software may be used "as is," or may be integrated or reengineered by the acquirer (or by a third party under contract to the acquirer). Non-developmental software includes shrink-wrapped COTS, GOTS (Government-off-the-Shelf), and Modified-off-the-Shelf software packages, and OSS, shareware, and freeware components. For purposes of this document, obtaining and reusing legacy components is also considered "acquisition" of non-developmental software, even though it does not involve acquisition as defined by the Federal Acquisition Regulation and Defense FAR Supplement.

2. **Integration or assembly:** If software items must be combined to achieve the desired functionality of the system, that system comes into existence through integration or through assembly. The software items to be combined may be non-developmental or custom, or a combination of the two. In some cases, integration may entail custom-development of code to implement interfaces between software items. In other cases, non-developmental items may need to be modified or extended through reengineering (see below). If the software items to be combined are *components* (*i.e.,* self-contained with standard interfaces), the integration process is referred to as *component assembly*.

3. **Custom development**: Custom-developed software is purpose-built for the specific system in which it will be used. It is written by the same organization that will use it, or by another organization under contractor to the user organization. Very few large information systems are completely custom-developed; most are the result of integration and include at least some non-developmental components.

4. **Reengineering**: Existing software is modified so that one or more of its components can be modified/extended, replaced, or eliminated.

Each of the above approaches to software conception (specification through design) and implementation (coding and integration) has its own advantages and disadvantages with regards not only to the software's cost, support, and functional effectiveness, but to the ability to mitigate security risk associated with the software.

---

13 The term "acquisition. in the context of this specific discussion, focuses on how software comes into existence within a given organization or system. For this reason, it does not include acquisition of contractor development services. However, contractors may be responsible for the integration, assembly, custom development, and/or reengineering of software.

The less an organization responsible for generating a final software product is able to establish visibility into all of that product's components, and the less control it has over the how those components are conceived and implemented, the more risk it must assume. The higher the risk, the more ingenious the organization must be about how to mitigate it, because less visibility and control over the components of a software product necessarily means that fewer options are available to mitigate that risk.

3.2 GENERAL SOFTWARE SECURITY PRINCIPLES

The following principles apply to secure software generally, and should guide the decisions made in producing artifacts at every phase of the software life cycle:

- **Minimize the number of high-consequence targets.** The software should contain as few high-consequence targets (critical and trusted components) as possible. High-consequence targets are those that represent the greatest potential loss if the software is compromised, and therefore require the most protection from attack. Critical and trusted components are high-consequence because of the magnitude of impact if they are compromised. *(This principle contributes to trustworthiness and, by its implied contribution to smallness and simplicity, also to dependability.)*

- **Don't expose vulnerable and high-consequence components.** The critical and trusted components the software contains should not be exposed to attack. In addition, known-vulnerable components should also be protected from exposure because they can be compromised with little attacker expertise or expenditure of effort and resources. *(This principle contributes to trustworthiness.)*

- **Deny attackers the means to compromise.** The software should not provide the attacker with the means by which to compromise it. Such "means" include exploitable weaknesses and vulnerabilities, dormant code, backdoors, *etc.* Also provide the ability to minimize damage, recover, and reconstitute the software as quickly as possible following a compromising (or potentially compromising) event to prevent greater compromise. In practical terms, this will require building in the means to monitor, record, and react to how the software behaves and what inputs it receives. *(This principle contributes to dependability, trustworthiness, and resilience.)*

- **Always assume "the impossible" will happen.** 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 software is installed in a new environment, those events may become quite likely. The use cases and scenarios defined for the software should 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. Specifically, developers should always assume that their software will be attacked, regardless of what environment it may

operate in. This includes acknowledgement that environment-level security measures such as access controls and firewalls, being composed mainly of software themselves (and thus equally likely to harbor vulnerabilities and weaknesses), can and will be breached at some point, and so cannot be relied on as the sole means of protecting software from attack. Developers who recognize the constant potential for their software to be attacked will be motivated to program defensively, so that software will operate dependably not only under "normal" conditions, but under anomalous and hostile conditions as well. Related to this principle are two additional principles about developer assumptions:

1. **Never make blind assumptions.** Validate every assumption made by the software or about the software *before* acting upon that assumption.

2. **Security software is not the same as secure software.** Just because software performs information security-related functions does not mean the software itself is secure. Software that performs security functions is just as likely to contain flaws and bugs as other software. However, because security functions are high-consequence, the compromise or intentional failure of such software has a significantly higher potential impact than the compromise/failure of other software.

Two documents developed by the Software Assurance Forum's Working Group on Workforce and Education, *Software Assurance: A Curriculum Guide to the Common Body of Knowledge to Produce, Acquire, and Sustain Secure Software* and *Principles Organization: Towards an Organization for Software System Security Principles and Guidelines* list, explain, and attempt to further clarify through organization a number of other software security principles. These documents can be downloaded from the DHS BuildSecurityIn portal's Software Assurance CBK/Principles Organization Webpage, accessed 28 January 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/resources/dhs/927.html

3.2.1 Software assurance, information assurance, and system security

This sub-section provides the reader with the background necessary to identify the different requirements of software assurance on the one hand and information assurance and information system security on the other.

Software assurance and information system assurance are inherently related. As software is an integral part of a information systems, the security of the software itself is critical to assuring the security of the system. However, the view of security for information systems virtually always focuses on how the information processed by those systems is adequately protected, with little or no consideration given to whether the mechanisms and safeguards relied upon to provide that protection are in and of themselves dependable, trustworthy, and survivable. Security principles at the system level, and the mechanisms that are implemented according to those principles, have become increasingly robust over time, so much so that the attackers who wish to compromise information systems are now switching their attention to that level of the system that remains rich with exploitable vulnerabilities: the software.

So, while software security and system security can be said to be inherently related and mutually supportive—many of the principles discussed in the context of one are equally relevant for the other—the actual implementation and verification of those principles differs, often significantly, at the software level *vs.* the system level. For example, software security relies heavily on the absence of exploitable defects in the source code and the non-exposure of defects in the binary executable. By contrast, system security relies heavily on safeguards and countermeasures, such as cryptography, access controls, and enforcement of security boundaries.

It is important to understand that without assuring the security of the software that implements the vast majority of system-level safeguards and countermeasures, it will not be possible to truly assure the effectiveness of those safeguards and countermeasures. For an illustration of this reality, refer back to Table 2-3, which includes numerous examples of real-world software security vulnerabilities—many of which are considered to have high severity—found in system security components such as antivirus and firewalls.

As noted above, the main goal of information system security is to protect the information processed by the information system from unauthorized disclosure (breach of confidentiality), unauthorized modification (breach of integrity), or unauthorized denial of service (breach of availability). The reason IT, and specifically information systems, must be protected are because it is the technology/system that provides the conduit by which people access, manipulate, and transport information.

From the point of view of an information system's security model, information exists in the form of computer data, which is considered an *object.*[14] That is, data is something passive that gets acted upon by a *subject,* which is something active. A subject can be a human user or a software, firmware, or hardware process acting independently or on a human user's behalf. At any given time, a data object is in one of in four different states:

- **At rest,** *i.e.,* stored on a computer hard drive or other digital medium, from which it may be read by a subject;
- **In transit,** *i.e.,* moving from one physical location to another, most often over a network, *via* a transmission mechanism such as an email message or file transfer;
- **In creation,** *i.e.,* being written to a digital medium for the first time;
- **In transition,** *i.e.,* being overwritten (modified) on or removed (deleted) from the digital medium on which it has been stored.

Information security policy is concerned, in large part, with defining the set of rules by which system subjects are allowed to change the states of data objects in the system. In practical

14 Not to be confused with an "object" in object-oriented terms.

terms, this means defining for every system subject whether, and if so how, it may store, transmit, create, modify, or delete a given data object (or type of data object).

To the extent that the information system allows or prevents that access and manipulation, or performs it on behalf of another actor, the system can be seen as playing the role of "information protector", with its actions governed by the information security policy. To do this effectively, the system must also have a governing policy that defines how the system itself is allowed and not allowed to be accessed and operated as a conduit to the information, and which also defines the ways in which the security mechanisms and countermeasures built into the system must be configured and operate to accomplish the system's role as information protector.

There are three main objectives common to all system security policies and the mechanisms and countermeasures used to enforce those policies:

1. They must allow authorized access and connections to the system while preventing unauthorized access or connections, especially by unknown or suspicious actors;

2. They must enable allowable reading, modification, destruction, and deletion of data while preventing unauthorized reading (data leakage), modification (data tampering), destruction (denial of service), or deletion (denial of service);

3. They must block the entry of content (user input, executable code, system commands, *etc.*) suspected of containing attack patterns or malicious logic that could threaten the system's ability to operate according to its security policy and its ability to protect the information.

In summary, information system security is primarily about protecting *information*. Any concern about protecting the *system* is expressed solely in terms of the system's relationship to and responsibility for protecting that information.

Information system security is achieved mainly through use of a combination of security mechanisms and countermeasures at the network, operating system, middleware, and application layers. Such mechanisms and countermeasures may be preventive or reactive. Preventive measures are intended allow acceptable information access and manipulation to occur while preventing unacceptable access and manipulation. Reactive measures are intended to minimize and recover from the damage that results from the inability to prevent. Preventive measures may include:

- Network-level and data-level encryption;
- Digital signature;
- Firewalls, proxy filters, and security gateways;
- Honeypots, honeynets, and honeybots;
- Intrusion detection and prevention systems;

- Virus scanners and spyware detectors;
- Network traffic monitoring and trend analysis;
- Identification, authentication, and authorization of users;
- Access control of data and resources;
- User activity logging/auditing and non-repudiation measures;
- Mobile code containment;
- Platform virtualization (hardware or software).

Reactive measures may include:

1. Malicious code containment and eradication;
2. Component-level redundancy that allows for automatic switchover from a compromised to a non-compromised version of the redundant component;
3. Patching of vulnerabilities.

Unlike data, software is both an object and a subject, or *actor.* The executable files containing the software code and stored in the file system are objects, with the same four possible states, making it possible to use many of the same data security mechanisms to protect those files. Executing software, however, is an active entity analogous to a human user in terms of the ability to access and manipulate (and disclose, subvert, or sabotage) data, including other software files.

Software as an actor has a very large number of possible states. It also as an access mode that data does not: execution. Note that "executable data" and "active content" are really misnomers for what is, in fact, data in which executable logic – or software – has been embedded. It is only the embedded software portion of active content or executable data that has the potential to become, through its execution, an active subject, or actor. The rest is just data (and thus a passive object).

The fact that software in execution is an actor changes the relationship of that software to data: rather than being data itself, the software becomes an actor upon data. As such, the software must be seen as a potential threat to the data. But because it is also responsible for acting as a conduit to the data, and a manipulator and protector of the data, it is particularly important that the software's operations be dependable, trustworthy, and resilient.

These requirements for software-as-actor are what software assurance is concerned with. These requirements have no direct analogy in information and IT security (though there are similarities in the emerging discipline of Information Integrity Assurance, in which the validity and trustworthiness of the actual semantics/meaning of information's content are what is assured, not just the information's physical/structural integrity, which is what traditional information assurance has meant by "integrity"). Traditionally, practitioners of IT

security and information system security seldom if ever considered the potential for the technology/system itself to act in an untrustworthy manner.

Because of its schizophrenic nature – it is both object and subject – software is made vulnerable not only in its passive state as a set of files, but also in its active state as an executing actor. The vulnerabilities in executing software originate in the poor judgment and mistakes of its developers – the decisions they make about how it is designed, and the errors they introduce as it is implemented.

To a great extent, software assurance focuses on how software behaves in response to its own state changes, which are triggered by external inputs or stimuli or internal triggers, such as the execution of a code segment. Because such states can be triggered by internal events, information system security approaches, which focus on how the system interacts with external entities, are insufficient to correct or mitigate software security problems, which usually originate with flaws, bugs, and weaknesses built into the software itself. This is why the main objective of software assurance is to avoid including such flaws, bugs, and weaknesses in software.

For software developers and testers concerned about software assurance to work effectively with their system engineers concerned about information assurance, they need to:

1. Recognize the subtle differences in how terms shared by the two disciplines need to be interpreted when they refer to "information" and "software-as-object" *vs.* "software-as-actor";

2. Help system engineers understand the software assurance counterparts to terms unique to information assurance.

NIST FIPS 200[15] identifies core security properties for information systems. These same properties can also interpreted as applying to software in the following ways:

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

2. **Integrity:** At all points in the software's life cycle, the software must be protected from unauthorized modifications (corruption, tampering, overwriting, insertion of unintended logic, destruction, or deletion) by valid entities (persons or processes) and from all modifications by invalid, unauthorized entities;

3. **Confidentiality:** Details of the nature of the operational software – its configuration settings, its logic, its interfaces, and in some cases its very existence must not be

15 NIST Computer Security Division. *Minimum Security Requirements for Federal Information Systems.* FIPS Publication 200, March 2006. Accessed 28 January 2008 at: http://csrc.nist.gov/publications/fips/fips200/FIPS-200-final-march.pdf

disclosed to entities not explicitly authorized to know those details. In practical terms, this means the software must be protected from both reconnaissance attacks and reverse engineering. In addition, all software artifacts (specification, source code, binary code, test plans and reports, *etc.*) must be protected from unauthorized disclosure.

Two properties associated with the relationship between an information system and its human users and administrators can also be extended to software-as-actor when it interacts with an external entity. For example, a Web service that interacts with another Web service or portal, an autonomous agent interacting with other autonomous agents, a client interacting with a server, or a peer process interacting with another peer process. In short, these properties apply to "software-as-user":

4. **Accountability:** All security-relevant actions of the software-as-user must be recorded and tracked, with enough information collected to enable attribution of responsibility for those actions. This tracking must be possible both while and after the recorded actions occur;

5. **Non-repudiation:** The software-as-user must not be able to plausibly deny or disprove having performed any action.

For many, the basis for defining system security requirements is the Common Criteria (in the U.S. government, the main basis for such requirements is the governing information assurance and/or information system security standards, directives, *etc.,* to which the government entity is required, by mandate, to conform. For DoD, this is DoD Directive 8500.52. For other federal agencies (outside the Intelligence Community), it is the Federal Information Processing Standards and NIST Special Publications that delineate the system security requirements that will enable systems to satisfy the requirements of the 2002 Federal Information Security Management Act (FISMA). Many non-U.S. government and private sector entities have similar mandates.

What the Common Criteria and these and other information assurance/information system security mandates share in common is that none of them provide explicit guidance on how to address *software* security aspects of an information system. However, since an Evaluation Assurance Level (EAL) in the Common Criteria captures a specific set of security assurance requirements, it is possible to deduce some general properties that software must exhibit to attain the three highest possible EALs:

- EAL 5: The system must be semi-formally designed and tested. This property should also apply to the software design.

- EAL 6: Same as EAL 5, plus the design must be semi-formally verified. As above, this property should also apply to the software design.

- EAL 7: Same as EAL 6, plus the design must be formally verified and formally tested.[16] As above, this property should also apply to the software design.

The suggested resources below include several good introductions to information security and IT security concepts.

16 See Section 3.6.3 for a discussion of the use of formal methods in secure software development.

SUGGESTED RESOURCES

- *Software Security Assurance,* Sections 4 and 5.

- DHS National Cyber Security Division. *IT Security Essential Body of Knowledge.* Final Draft Version 1.1, October 2007. Accessed 28 January 2008 at: http://www.us-cert.gov/ITSecurityEBK/EBK2007.pdf

- NSA. "National Information Security Education and Training Program: Introduction to Information Assurance". Online presentation, 1998. Accessed 28 January 2008 at: http://security.isu.edu/ppt/shows/information_assurance_files/frame.htm —*or*— http://security.isu.edu/ppt/pdfppt/information_assurance.pdf

- NIST. *Information Security Handbook: A Guide for Managers.* Special Publication 800-100, October 2006. Accessed 29 August 2008 at: http://csrc.nist.gov/publications/nistpubs/800-100/SP800-100-Mar07-2007.pdf

- NIST. *Security Considerations in the Information System Development Life Cycle.* Special Publication 800-64 DRAFT Rev 2, June 2004. Accessed 29 August 2008 at: http://csrc.nist.gov/publications/drafts/800-64-rev2/draft-SP800-64-Revision2.pdf

- NIST. *Recommended Security Controls for Federal Information Systems.* Special Publication 800-53 Rev. 2, December 2007. Accessed 29 August 2008 at: http://csrc.nist.gov/publications/nistpubs/800-53-Rev2/sp800-53-rev2-final.pdf

- NIST. *Generally Accepted Principles and Practices for Securing Information Technology Systems.* Special Publication 800-14, September 1996. Accessed 29 August 2008 at: http://csrc.nist.gov/publications/nistpubs/800-14/800-14.pdf

- NIST. *An Introduction to Computer Security: The NIST Handbook.* Special Publication 800-12, October 1995. Accessed 28 January 2008 at: 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.* Report for Congress, Order Code RL32777, 22 February 2005. Accessed 28 January 2008 at: http://fpc.state.gov/documents/organization/43393.pdf

- National Academy of Sciences Computer Science and Telecommunications Board. *Cybersecurity Today and Tomorrow: Pay Now or Pay Later* (excerpts). Washington, D.C.: National Academies Press, 2002. Accessed 28 January 2008 at: http://books.nap.edu/openbook.php?isbn=0309083125 —*or*— http://books.nap.edu/html/cybersecurity/

- Gansler, Jacques S. and Hans Binnendijk, editors: "Information Assurance: Trends in Vulnerabilities, Threats and Technologies". National Defense University Working Paper, May 2003. Accessed 28 January 2008 at: http://www.ndu.edu/ctnsp/IAverMay03.pdf

- Esterle, Alain, Hanno Ranck, and Burkard Schmitt. "Information Security: A New Challenge for the European Union". European Union Institute for Security Studies Chaillot Paper Number 76, March 2005. Accessed 28 January 2008 at: http://www.iss-eu.org/chaillot/chai76.pdf

- Bidgoli, Hossein. *Handbook of Information Security.* New York, NY: Wiley, 2005.

- Saltzer, Jerome H. and Michael D. Schroeder. "The protection of information in computer systems." *Proceedings of the IEEE*, 1975, Volume 63 Issue 9, pages 1278-1308. Accessed 11 September 2008 at: http://www.acsac.org/secshelf/papers/protection_information.pdf —*and*— http://www.ece.rutgers.edu/~parashar/Classes/03-04/ece572/papers/protection.pdf

3.3 SECURE DEVELOPMENT LIFE CYCLE ACTIVITIES AND PRACTICES

The activities, practices, and checkpoints listed in Table 3-1 pertain to individual SDLC phases, and are elaborated upon in Sections 4-9 of this document.

Table 3-1. Activities, practices, and checkpoints to be added to SDLC phases

Phase	Responsible role(s)*	Current Activities	Additional/Enhanced Activities for Secure Software	Section 2.5.2 issues addressed by added/enhanced activities
Whole SDLC	Configuration manager	Version control, change control	Secure CM practices Secure CM tools	Malicious developers, integrators, testers
Requirements	Requirements analyst	Use Case development	Security use case development Misuse Case and Abuse Case development	Requirements based on inaccurate/incomplete risk assessments/attack models
		Requirements modeling	Attack Modeling (also referred to as Threat Modeling)	Requirements based on inaccurate/incomplete risk assessments/attack models
		Functional requirements specification	Specification of requirements for constraints on functionality, with mapping to associated functional requirements (*e.g.*, "All mobile code must be digitally signed. The code signature must be validated before the mobile code is executed. If the code signature cannot be validated, the code must not be executed.") Specification of requirements for additional software security functions (*e.g.*, for code signature validation, input validation, *etc.*) Specification of non-functional requirements to ensure the security of software (*e.g.*, specification of how the software should be built, *e.g.*, through use of formal methods, secure coding standards, *etc.*)	Lack of requirements for safe behaviors Lack of requirements for constraining unsafe behaviors
		Test case definition	Definition of additional test cases for verifying software security	Lack of security test cases in test plan

Phase	Responsible role(s)*	Current Activities	Additional/Enhanced Activities for Secure Software	Section 2.5.2 issues addressed by added/enhanced activities
		Definition and selection of coding standards	Definition and selection of secure coding standards, secure programming languages, and secure development tools	Inadequate tools and languages
Architecture & Design	Architect Designer	Architecture and design reviews	Addition of security criteria in architecture and design reviews	Non-secure design choices Failure to design according to secure principles Failure to include security criteria in architecture/design reviews Inappropriate trust relationships between software and users Malicious developers
Component assembly/ integration	Integrator	Architecture-level trade-off analyses	Security considerations in trade-off analyses	Non-secure choices in architecture Incorrect/ineffective use of extra-component security protections/ services Inappropriate component-component and component-user trust relationships
		Component selection	Security criteria in component evaluation	Acceptance of vulnerable or malicious COTS/OSS components
		Integration testing	Security criteria in integration testing	Incorrect/ineffective use of extra-component security protections/ services Inappropriate component-component and component-user trust relationships

Phase	Responsible role(s)*	Current Activities	Additional/Enhanced Activities for Secure Software	Section 2.5.2 issues addressed by added/enhanced activities
Implementation	Programmer	Code reviews	Static code security analyses Other source code security analyses	Failure to program according to secure principles/practices Bugs in code Malicious developers
		Unit testing	Security criteria in unit testing	Failure to place value on security
		Build process testing	Security criteria build process testing	Failure to place value on security
Testing	Tester	Functional integration and system tests	Binary security tests (*e.g.*, fault injection, fuzzing, binary code scanning, reverse engineering) Automated vulnerability scans Penetration tests	Failure to perform sufficient security tests Lack of tools to support security testing
		Code reviews	Code security reviews (automated/semi-automated whenever possible) Other white box security tests	Failure to perform sufficient security tests Lack of tools to support security testing
Distribution & deployment	Tester Configuration manager Distribution manager Installer	Deployment testing Distribution (online/ offline) Installation configuration	Automated vulnerability scans Penetration tests Secure distribution techniques and technologies Secure installation configuration	Failure to perform sufficient security tests Lack of tools to support security testing Failure to remove backdoors, *etc.* Non-secure configuration defaults

Phase	Responsible role(s)*	Current Activities	Additional/Enhanced Activities for Secure Software	Section 2.5.2 issues addressed by added/enhanced activities
Sustainment	Maintainer Vulnerability manager Auditor	Issues tracking and resolution	Security incident response Vulnerability management, including vulnerability tracking and reporting	Failure to perform root-cause analyses
		Quality assurance/ support meetings	Security experts involved in quality assurance/support meetings Security criteria included in quality assessment criteria	Failure to place value on security
		Maintenance regression testing	Regression tests of security-critical and high-consequence software Vulnerability scans, penetration tests, *etc.* of updated software Security impact analyses of maintenance changes	Failure to perform security impact analyses of all updates
		Test, release, and distribution of updates and patches	Vulnerability management, including vulnerability reporting and patch issuance Security impact analysis of updates and patches prior to release	Failure to perform security impact analyses of all patches Failure to issue timely patches to fix vulnerabilities Failure to encourage customers to abandon old, non-secure versions
		System, code, and operational audits	Security audits of system, code, and operations	Failure to perform root-cause analyses
		(no counterpart)	Forensic analysis of security incidents, including root cause analyses of operational security incidents	Failure to perform root-cause analyses

**In any or all cases, a security expert may assist the responsible party or parties.*

The software assurance community is increasingly engaged with defining methods and associated artifacts for formally establishing the assurance that secure development practices have been followed, secure software principles have been conformed with, and that security has, in fact, been exhibited as a property of the software. The key artifact being defined for this purpose is the software security assurance case. Section 3.5 discusses the purpose of the assurance cases, emerging processes

associated with developing and validating software security assurance cases, and the software developer's role in those processes.

SUGGESTED RESOURCES

- Gilliam, David P. "Security risks: management and mitigation in the software life cycle". *Proceedings of the 13th International Workshop on Enabling Technologies,* Modena, Italy, 14-16 June 2004, pages 211-216.
- Dickson, John B. "Application Security: What does it take to build and test secure software?". Presented at Information Systems Audit and Control Association North Alabama Chapter meeting, 6 November 2006. Accessed 16 January 2008 at: http://www.bham.net/isaca/downloads/20061106_DenimGroup_Secure_SW_LG_Org.ppt

3.4 SECURE VERSION MANAGEMENT AND CHANGE CONTROL OF SDLC ARTIFACTS

One practice that pertains to the entire SDLC is secure configuration management, which includes version management and change control. The artifacts of the software development process—documentation, software code, and developer notes—are generated throughout the SDLC, and may all be susceptible to tampering and malicious insertions unless kept under strict version control that is managed in ways expressly intended to minimize the likelihood, and to maximize the detection, of any attempts to inappropriately access them.

Using a secure Software Configuration Management (SCM) system, strong authentication and authorization of developers, testers, and others who require access SDLC artifacts can be combined with granular access controls to prevent inappropriate accesses. Role-based access control together with separation of duties will prevent programmers from modifying test results (to erase evidence of their software's test failures). Access can be further controlled based on the relationship of the artifact to the life cycle phase in which the person attempting to access it is involved: for example, once the final version of an artifact has been checked into the SCM system, access privileges for the person who developed that artifact can be reset to read-only, to prevent that person from being able to modify the artifact while it is undergoing review or testing.

As each development artifact is completed, the developer is responsible for checking it into the secure SCM system according to the secure check-in process, which includes digitally signing and timestamping the artifact. This checked-in version is then "baselined": critical metadata about the artifact is added which enables the configuration manager to compare the baseline with each subsequent version of the artifact that is checked in. This comparison verifies that only expected portions/areas of the later artifact version have been changed. The timestamp and digital signature also aid in the detection of later, possibly inappropriate substitutions or modifications.

By ensuring that each new version delivered to the SCM can be easily compared against the most recent previous version or baseline, secure version management ensures that as each artifact is modified after peer reviews, other reviews, or tests, the new version can be easily compared against the previous baseline to verify that the only changes in the new version are those that correct problems found during review/testing. If the developer has made any changes that do not directly address review findings/test results, these can be flagged and analyzed, and the developer can be asked to explain and justify them. In this way, version management minimizes the window of opportunity in which malicious developers can tamper with development artifacts (*e.g.,* by inserting malicious logic or intentional vulnerabilities).

In addition to increasing control over development artifacts and minimizing opportunity for malicious tampering and insertions, secure version management also:

- Increases accountability for and traceability of modifications to SDLC artifacts;

- Supports clear identification and security impact analysis of changes from one version to the next of each SDLC artifact;

- Minimizes the likelihood that undesirable changes will "slip in" to the artifact that would ultimately increase the vulnerability of the software.

It is the job of the configuration manager to verify the validity of changes between versions, and also to periodically perform SCM audits.

All tools used to produce and review/test SDLC artifacts also need to be kept under strict secure change control to minimize the chance that they can be tampered with, or that such tampering will go undetected.

Despite being authored in the late 1980s and early 1990s, many of the configuration management (CM) practices for high assurance software described in the National Computer Security Center's *Guide to Understanding Configuration Management in Trusted Systems*, and the CM practices for high integrity software in Section 3.4 of NIST's *Framework for the Development and Assurance of High Integrity Software*, provide an excellent basis for establishing secure version management and change control practices, and for specifying requirements for supporting SCM tools.

A few secure configuration management repositories and toolsets are available commercially, including Oracle Developer Suite 10g Software Configuration Manager, the Secure Protected Development Repository from Sparta, Inc., and MKS Integrity from MKS Inc. SCM tools such as these incorporate many of the strong authentication and authorization, access control, time-stamping, accountability, and other security features not provided in most standard CM systems.

SUGGESTED RESOURCES

- *Software Security Assurance*, Section 5.1.6.

- National Computer Security Center. *A Guide to Understanding Configuration Management in Trusted Systems* (the "Amber Book"). NCSC-TG-006, 28 March 1988. Accessed 19 December 2007 at: http://handle.dtic.mil/100.2/ADA392775 - *and* - http://www.fas.org/irp/nsa/rainbow/tg006.htm

- Wallace, Dolores R. and Laura M. Ippolito. *A Framework for the Development and Assurance of High Integrity Software.* NIST Special Publication 500-223, December 1994, Section 3.4 "Software Configuration Management Process". Accessed 26 January 2008 at: http://hissa.nist.gov/publications/sp223/

3.5 SECURITY ASSURANCE CASES FOR SOFTWARE

There is an increasing emphasis, in the software assurance community, on defining standards for the content and evaluation of security assurance cases for software. The only claim that can be realistically made for software security assurance cases at this point is that they will provide a useful mechanism for communicating information about software risk.

An *assurance case* is intended to provide a basis for justified confidence may take the form of an assurance case, which may be defined as:

> *…a documented body of evidence that provides a convincing and valid argument that a specified set of critical claims regarding a system's properties is adequately justified for a given application in a given environment.*[17]

Assurance cases are intended to document arguments and claims to a degree that is sufficient to reduce uncertainty about the described software's exhibition of its required property. Assurance cases are also intended to provide the evidence necessary to prove the validity of those arguments and claims, to whatever level of confidence that validity must be proved.

Assurance cases as artifacts should be developed in parallel with the other SDLC activities, with artifacts of those activities providing much of the evidence for the assurance case.

The developer's role in assurance case production will be to create development artifacts that are not only adequate for their intended purpose, *i.e.,* producing software, but which also provide evidence that is sufficient and appropriate to support the assurance claims that will be made about that software. Development artifacts contribute to quantifiable statements of evidence that the software achieves its assurance goals. Moreover, the review and testing artifacts generated throughout the lifecycle provide evidence that stakeholders adequately reviewed the development artifacts and processes and evaluated against standards related to the assurance case itself (*i.e.,* secure architecture or coding guidelines).

SDLC artifacts that might be included as evidence to support the assurance case include:

- The software's semi-formal or formal architecture specification;
- The result of a formal or semi-formal requirements traceability analysis and proof that the architecture fulfills the software's specified requirements;
- The software's semi-formal or formal detailed design specification;
- The result of a formal or semi-formal requirements traceability analysis and proof that the detailed design fulfills the software's specified requirements;
- Results of a vulnerability analysis of the software architecture;
- Documentation of the secure SDLC process/methodology in use, including environment, physical security, and personnel controls;

---

17 Ankrum, T. Scott, and Alfred H. Kromholz. "Structured assurance cases: three common standards". Slides presented at the Association for Software Quality (ASQ) Section 509 Software Special Interest Group meeting. 2006 January 23; McLean, VA. Accessed 31 July 2007 at:
http://www.asq509.org/ht/action/GetDocumentAction/id/2132

- Evidence of the consistent use of secure configuration management practices and tools that prevent unauthorized modifications, additions, or deletions to the SDLC artifacts.

As of July 2007, the most mature of the emerging software security assurance case standards, SafSec (developed by Praxis High Integrity Systems for the UK Ministry of Defence), had only been tested in two case studies, with the next stage of evaluation (which involved applying the methodology on a system under development) still underway.

To date, assurance case proponents have based their expectation of the effectiveness of security assurance cases on an extrapolation from the success of safety cases and to a lesser extent from Common Criteria Security Targets. At present, few if any software vendors use formal methods as a standard development practice, or generate development artifacts that security evidence extensive or detailed enough to form the basis for proving the assurance argument in a software assurance case.

It is understandable that most software practitioners are waiting for empirical evidence that assurance cases can or will improve the security of software and/or increase the level of trust between users and software suppliers before taking on the labor-intensive development of these artifacts.

For organizations that develop high-consequence software, assurance cases even in their still-evolving form have the potential to provide a robust and disciplined means of security verification and validation. For these organizations, early adoption of assurance cases may be warranted, using an emerging methodology and supporting tools.

Some significant research efforts are underway to produce methodologies and tools, or to extend, adapt, and refine safety case methodologies/tools to address requirements specific to security assurance cases. In the international standards community (ISO/IEC), there are efforts underway to define a standard for a software assurance process and the required content and structure of assurance cases.

The reader is encouraged to review Section 5.1.4, "Software Security Assurance Cases" in *Software Security Assurance* for a more complete discussion of emerging software security assurance case methodologies and tools support.

SUGGESTED RESOURCES

- BuildSecurityIn Assurance Case resources. Accessed 21 January 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/assurance.html
- *Software Security Assurance*, Section 5.1.4.
- Praxis High Integrity Systems. SafSec: Integration of Safety and Security Certification webpage. Accessed 21 January 2008 at: http://www.praxis-his.com/safsec/index.asp
- McDermott, John. "Abuse-case-based assurance arguments". *Proceedings of the 17th Annual Computer Security Applications Conference*, New Orleans, Louisiana, 10-14

December 2001, pages 366-374. Accessed 11 September 2008 at: http://www.acsac.org/2001/abstracts/thu-1530-b-mcdermott.html

3.6 SDLC METHODOLOGIES THAT AID IN SECURE SOFTWARE PRODUCTION

A number of "secure SDLC methodologies" have been published to take the guesswork out of how to security-enhance one's software development process. Several such methodologies are introduced in 3.5.1.

In addition, some standard development methods have been demonstrated to increase the likelihood that software produced by them will be secure. Most notable among these are Formal Methods, discussed in 3.5.2.

There are also some methods that have been promoted as aiding in secure software development when, in fact, what they have actually been demonstrated to assist in is the specification and implementation of security functionality. The most notable of these, Aspect Oriented Software Development, is discussed in 3.2.5.3.

Finally, because of their popularity, particularly in the private sector, agile methods and the security issues that arise when they are used are also discussed in 3.2.5.4.

3.6.1 Secure SDLC methodologies

Instead of attempting to determine what security activities, checkpoints and considerations need to be added to their current SDLC process, a development team may want to encourage their organization and/or project manager (whichever has the authority to decide such things) to adopt a "security-enhanced" software development methodology that already defines these things.

Development teams who use secure SDLC methodologies should almost immediately notice an improvement in their ability to detect and eliminate vulnerabilities and weaknesses in the software they produce before that software goes into distribution/deployment. As development teams become expert at following a secure SDLC methodology, and codify the methodology's practices over time, they should also notice a marked reduction in the number of vulnerabilities and weaknesses that appear in their software in the first place. The proof will be in the software's ability to pass its various security checkpoints (reviews and tests, such as design and code security reviews, fault injection tests, penetration tests, vulnerability scans, *etc.*).

For development teams that work in organizations and/or for project managers unwilling to adopt a security-enhanced methodology, the practices described in Sections 4-9 of this document should be able to be integrated into any standard life cycle methodology in order to

"security enhance" that methodology in a more *ad hoc* way to enable the team to improve the security of software produced by their project.

Table 3-2 lists some "ready-to-use" SDLC methodologies, with associated information resources, that are intended expressly to produce secure software. The suggested resources at the end of this section also provide more information about these methodologies, and about security-enhanced methodologies in general.

Table 3-2. Secure SDLC methodologies

Methodology	Resources
CLASP	Secure Software Inc. *CLASP: Comprehensive Lightweight Application Security Process.* Version 2.0, 2006. Accessed 17 December 2007 at: http://searchappsecurity.techtarget.com/searchAppSecurity/downloads/clasp_v20.pdf
	Open Web Application Security Project (OWASP) CLASP Project Webpage. Accessed 17 December 2007 at: http://www.owasp.org/index.php/Category:OWASP_CLASP_Project
Security Development Lifecycle (SDL)	Howard, Michael and Steve Lipner. *The Security Development Lifecycle.* Redmond, Washington: Microsoft Press, 2006.
	Lipner, Steve and Michael Howard. "The Trustworthy Computing Security Development Lifecycle". Microsoft Developer Network, March 2005. Accessed 17 December 2007 at: http://msdn2.microsoft.com/en-us/library/ms995349.aspx
	Microsoft SDL Weblog. Accessed 17 December 2007 at: http://blogs.msdn.com/sdl/
McGraw's Seven Touchpoints	McGraw, Gary. *Software Security: Building Security In.* Boston, Massachusetts: Addison-Wesley Professional, 2006.
	Software Security: Building Security In Webpage. Accessed 17 December 2007 at: http://www.swsec.com/
TSP-Secure	Over, James W. "TSP for Secure Systems Development" (presentation). Accessed 17 December 2007 at: http://www.sei.cmu.edu/tsp/tsp-secure-presentation/tsp-secure.pdf
	Dustin, Elfriede. "The Software Trustworthiness Framework". 30 January 2007. Accessed 11 December 2007 at: http://www.veracode.com/Weblog/?p=22
Secure Software Engineering (S2e)	Schneider, Thorsten. *(S2e) Integrated: Process Oriented Secure Software Development Model.* Version 1.0, 2007 [in German]. Accessed 17 December 2007 at: http://model.secure-software-engineering.com/ Schneider, Thorsten. "Secure Software Engineering Processes: Improving the Software Development Life Cycle to Combat Vulnerability". *Software Quality Professional.* Volume 8 Issue 1, December 2006. Available (with free registration) from: http://www.asq.org/pub/sqp/past/vol9_issue1/sqpv9i1schneider.pdf
Secure Tropos	Mouratidis, Haralambos and Paolo Giorgini (2007) "Secure Tropos: A Security-Oriented Extension of the Tropos Methodology". *International Journal of Software Engineering and Knowledge Engineering,* Volume 17 No. 2, April 2007, pages 285-309. Accessed 25 August 2008 at: http://www.dit.unitn.it/~pgiorgio/papers/IJSEKE06-1.pdf

NOTE: The Rational Unified Process (RUP) is a widely-used structured development methodology. Several noteworthy efforts to extend RUP so it can explicitly address security as a

quality-property have focused mainly on the requirements and architecture modeling phases of the life cycle. Papers on security enhancements to RUP are listed in the suggested resources below, as is a case study demonstrating the graceful integration of CLASP with RUP to achieve a secure life cycle methodology.

SUGGESTED RESOURCES

- BuildSecurityIn SDLC Process resources. Accessed 21 January 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/sdlc.html
- *Software Security Assurance*, Section 5.1.8 and Appendix G.
- Reza Ayatollahzadeh Shirazi, Mohammad, Pooya Jaferian, Golnaz Elahi, Hamid Baghi, and Babak Sadeghian. "RUPSec: An Extension on RUP for Developing Secure Systems—Requirements Discipline". *Proceedings of the World Academy of Science, Engineering, and Technology*, Volume 4, February 2005. Accessed 7 July 2008 at: http://www.waset.org/pwaset/v4/v4-51.pdf. *Original Farsi version accessed 7 July 2008 at: http://hamkelasy.com/files/pdfarticles/fani_moh/61013860203_(www.hamkelasy.com).pdf*
- Jaferian, Pooya, Golnaz Elahi, Mohammad Reza Ayatollahzadeh Shirazi, and Babak Sadeghian, "RUPSec: Extending Business Modeling and Requirements Disciplines of RUP for Developing Secure Systems". *Proceeding of the 31th IEEE Conference of EuroMicro*, Porto, Portugal, 31 August-2 September 2005.
- Paes, Carlos Eduardo de Barros and Celso Massaki Hirata. "RUP Extension for the Development of Secure Systems". *International Journal of Web Information Systems.* Volume 3 Issue 4, 2007, pages 293-314.
- Shiva, Sajjan G. and Lubna Abou Shala. "Adding Security to the Rational Unified Process". *Proceedings of the First Annual Computer Security Conference*, Myrtle Beach, South Carolina, 11-13 April 2007. Accessed 7 July 2008 at: http://computersecurityconference.com/Papers2007/CSC2007Shala.doc
- OWASP. *Guide to Building Secure Web Applications*, Version 2.0. Accessed 8 September 2008 at: http://www.owasp.org/index.php/Category:OWASP_Guide_Project#OWASP_Development_Guide_2.0_Downloads

3.6.2 Can agile methods produce secure software?

Much discussion and debate has occurred over whether it is possible for software projects using agile methods to produce secure software. Some of the issues in secure software development that seem to run contrary to the objective of agile development are:

- Whenever a functional requirement changes, an analysis should be done to understand the security impact of that change.

- Changes to functional requirements are likely to spawn changes to requirements for security constraints on those functions.

- Agile methods approach system development by producing individual components of functionality, then integrating them together. Software security is a whole-system property, and even if individual components are secure, the aggregation of those components will not necessarily result in a measurably secure software system.

- To be able to use agile methods to develop secure software will require the extension or adjustment of the existing agile methods to accommodate the necessary security practices and checkpoints. It will also require adapting those security practices and checks so that they more easily "fit" into the agile process.

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

- Do not directly involve the planning of development activities and production of software *vs.* other activities and artifacts, such as security reviews, requirements and design specifications, independent verification and validation (IV&V), *etc.*, which are crucial for most security evaluations and validations. Moreover, the preference for face-to-face over written communication clashes with the requirements of Certifications and Common Criteria evaluations for extensive software documentation and makes IV&V impractical, because independent testers rely on written documentation to understand the system and must, to maintain their independence, avoid any direct contact with the system's developers;

- Require specialist expertise that software developers do not have. Agile methods make it difficult to include security experts or other non-developer personnel on software teams. Moreover, the requirement that anyone involved with the software must be a member of the development team conflicts with the need for security reviewers and testers to be independent from the development team (separation of roles and duties);

- Cannot be performed concurrently with other life cycle activities. For example, agile methods do not easily accommodate IV&V;

- Require security modeling (though agile methods should accommodate "misuser" and "abuser" story development at the same time as user story development, to assist in requirements capture), as well as capture of negative and non-functional requirements;

- Reject late in the lifecycle changes to requirements that prevent the establishment of a security baseline for purposes of Certification and Accreditation or Common Criteria evaluation;

- Focus on any objective other than producing correct software quickly. It is difficult to incorporate other non-functional objectives, such as dependability, trustworthiness, and resilience, into agile projects;

- Cannot accommodate security impact analyses of development decisions throughout the life cycle;

- Limits test case definition and test execution to the context of TDD, not allowing for software security (*vs.* functional security) tests;

- Constrain who works on the project, in what role, and in what environment. Agile projects assume that all developers are trustworthy, and should have equal access to all software and development artifacts, contrary to the security principles such as separation of roles, separation of duties, least privilege, and role-based access control. Furthermore, in an "all developers have same role/access" environment, all developers will require the same (highest possible) level of security background checks and clearances, which can result in additional costs. Finally, Pair Programming (a standard practice in eXtreme Programming and other agile methods) in which one developer sits with another at the same workstation to continually review the second developer's code as it is being written, may not be allowed in certain work environments in which workstation-sharing is not permitted.

Table 3-3 summarizes, for each Core Principle in the Agile Manifesto, whether that Core Principle contributes to secure software development, or whether it obstructs the ability to produce secure software.

Table 3-3. Security implication of Agile Manifesto core principles

	Core Principle	Implication	Explanation
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.	Obstructive	Unless the customer is highly security-aware. Security testing, if done at all, will probably be inadequate unless (1) security is an explicit, high-priority customer requirement; (2) the customer is willing to delay "early delivery" to ensure that sufficient time is given to specifying, verifying, and testing security requirements.
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.	Obstructive	Unless the customer is willing to allow the necessary time to assess the security impact of each new/changing requirement, and to add or change the security constraint requirements and risk mitigations associated with each functional requirement.
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.	Obstructive	Unless the customer prioritizes the need for security higher than the need for rapid delivery.
4	The project will be built around the commitment and participation of motivated individual contributors.	Neutral	Could be obstructive if the individual contributors either resist or are ignorant of security priorities.
5	Customers, managers, and developers must collaborate daily, throughout the development project.	Neutral	Could be contributory if the development team includes include security experts and the customer team includes security stakeholders (*e.g.*, risk managers).
6	Agile developers must have the development environment and support they need.	Neutral	Could be contributory if the development environment includes tools, platforms, processes, and practices expressly intended to produce secure software.
7	Both management and customers will trust developers to get the job done.	Obstructive	Unless the developers are strongly committed and able to ensure (1) their own security knowledge; (2) security practices and checkpoints in their life cycle process, and security tools in their development toolkit.
8	The most efficient and effective method of conveying information to and within a development team is through face-to-face communication.	Obstructive	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.	Obstructive	Unless "working" means not just "functionally correct" but also "dependable, trustworthy, and resilient". If "working" is purely a matter of functional correctness, agile testing cannot allow for vulnerability scanning, penetration testing, or any other non-functional security tests.

	Core Principle	Implication	Explanation
12	Agility is enhanced by continuous attention to technical excellence and good design.	Contributory	Especially when "technical excellence and good design" reflect strong expertise in and commitment to secure software development.
13	Simplicity, which is defined as the art of maximizing the amount of work not done, is essential to successful projects and good software.	Contributory	If simplicity is a property of both the design and code of the software, which will make them easier to analyze and their security issues easier to recognize.

Source: *Software Security Assurance*

Proponents of test-driven development (TDD, also referred to as *continuous testing*)—a cornerstone of all agile methods—believe TDD can contribute to software security by ensuring that software is examined continuously throughout its development and problems are fixed as early in the development process as possible. TDD approaches testing by using automated tools to run a continuous, iterative series of test cases against the code as it is developed.

It has been noted that automation of testing creates a high degree of acceptance, though manual testing is still required to verify some requirements. The problem, however, is that the types of testing that can be practically accommodated "on the fly" in this way does not include the more sophisticated software security tests, such as fault injection with propagation analysis and dynamic code analysis. Even addition of static code security analysis may be seen as impractical within the typical agile development process due to the security expertise and specialized tools it requires.

Some of the researchers who identified these security enhancements are working to codify "secure agile methods". Their approach has been to combine agile software engineering values with a security risk mitigation mentality, enabling secure software engineering to be performed in an agile manner.

One such process (that of Norwegian researcher Gustav Boström) proposes that agile methods in general, and eXtreme Programming (XP) in particular, can and should be adapted to conform to the security engineering practices defined in the SSE-CMM and identified in the Common Criteria. Boström also suggests that the resulting security-enhanced agile method be combined with AOSD to reduce the burden imposed by the continuous refactoring emphasized in all agile methods, thereby offsetting the additional effort imposed by the security engineering practices.

While software security is an important aspect of any software intensive system, there is a danger that the software practitioner will get a false sense of security from the erroneous belief that fixing software security is all that is required to ensure that the software-intensive system is secure. The inherent security—dependability, trustworthiness, survivability—of software components is only one level of security concern: security at the whole-system level, including correctness and effectiveness of security functionality, conformance to Saltzer and Schroeder principles, and all other system-level security concerns become no less critical just because security issues specific to software are finally being addressed.

The question of how well a particular agile development method can be adapted to achieve the objectives of secure software often requires the addition to the method of security training for developers, attack modeling and risk analysis, documentation and review of architecture and design before coding starts, and non-functional security testing.

Table 3-4 lists security enhancements for agile development processes that researchers have investigated and found to be practical and not onerous.

Table 3-4. Recommended security extensions to agile methods

SDLC Phase	Enhancement	Source[18]
General	Increase security awareness of development team, as well as their acknowledged responsibility for producing secure software.	Kongsli
	Add a security expert role to the development team. The security expert will provide ongoing security education, mentoring, and subject matter expert support to other team members.	Wäyrynen *et al.*
Require-ments	Identify and list all high-consequence software components of the system.	
	Develop "misuser" and "abuser" stories or misuse and abuse cases that intentionally or unintentionally compromise those components. These misuser/abuser stories should be developed by defining deviations from user stories/use cases such that the deviations include modeled attacks.	Wäyrynen *et al.*, Kongsli
	Based on the misuser and abuser stories, perform threat analyses, vulnerability analyses, and identify risk to the high-consequence components, to include the estimated cost of recovery from each compromise documented in the misuse/abuse scenarios. Prioritize the scenarios according to level of risk; determine the maximum acceptable level of risk. Identify all scenarios whose compromises exceed that maximum level.	
	Specify requirements that will prevent the each of the misuse and abuse scenarios that exceeds the maximum acceptable level of risk from compromising the software (through prevention, tolerance, or recovery). Define acceptance test cases that verify the effectiveness of the specified preventive measures.	Wäyrynen *et al.*, Kongsli
Architec-ture and Design	Integrate any security functions required for secure software, such as code signing, sandboxing/virtual machines, *etc.*, into the customer's environment as early in the project as possible.	Beznosov
	Adhere to secure design principles.	Davidson

18 See "Suggested Resources" at the end of this section for complete references.

SDLC Phase	Enhancement	Source[19]
Coding	Extend pair programming reviews to consider the security implications of design assumptions/decisions and to seek out and eradicate security flaws in code.	Wäyrynen *et al.*
	Adhere to secure coding standards and principles.	Davidson
Testing	Add code scanners and other security test tools to the testing "arsenal" tools and security testing tools. Ensure that security criteria are included in every test case.	Davidson, Beznosov
	Use the misuse/abuse scenarios as the basis for defining the test case strategy. Use the risk analysis to prioritize the tests to be performed, testing the highest risk/highest consequence items first.	
	When closing an iteration, augment automated customer acceptance tests with "negative testing" for threats identified in the abuse/misuse scenarios.	Davidson

Because agile development is as much a philosophy as a methodology, agile developers who do not already understand the importance of security may overlook security imperatives that are neither explicit nor implicit in the Core Principles of the Agile Manifesto (with which all agile methods except eXtreme Programming are said to be consistent). Appendix C:C.3.2 elaborates the key points in the debate over whether agile methods can ever successfully be security-enhanced.

A methodology that follows the same general principles as agile development is Feature-Driven Development (FDD). FDD differs from agile methods, however, in providing better scalability and support for planning. Table 3-5 summarizes recommended security enhancements for FDD.

19 See "Suggested Resources" at the end of this section for complete references.

Table 3-5. Recommended security enhancements to FDD

SDLC Phase	Enhancements to FDD
Requirements Analysis	• Capture requirements with use cases with security subjects and security objects are identified and added to use cases • Identify and document abuse cases • Develop an Overall Model of the system • Construct candidate classes from use cases • Derive security levels of classes from use cases and incorporate into classes • Build a feature list • Specify abuse case scenarios • Specify countermeasures to prevent abuse cases • Relate use cases to abuse cases to features • Classify features into feature sets based on activity
Design	• Plan by feature • Define order of features to be developed and tested • Prioritize security countermeasures (features) • Design by Feature • Incorporate security elements and security classification into objects as attributes • Sketch sequence diagram of each security feature
Implementation	• Build by feature • Implement in security countermeasure feature priority order, adding the most important security measures first.
Testing	• Test abuse case scenarios that are most sensitive first • Test based on security countermeasure feature priority list

SUGGESTED RESOURCES

- *Software Security Assurance*, Sections 5.1.8.1 and Appendix G.
- Beznosov, Konstantin and Philippe Kruchten. "Towards agile security assurance". *Proceedings of the 11th Workshop on New Security Paradigms,* Nova Scotia, Canada, September 2004. Accessed 21 January 2008 at: http://konstantin.beznosov.net/professional/works/shared/biblio_view.php?bibid=10&tab=home - *and* - http://konstantin.beznosov.net/old-professional/papers/Towards_Agile_Security_Assurance.html
- Peeters, Johann. "Agile Security Requirements Engineering". Presented at the Symposium on Requirements Engineering for Information Security, Paris, France, 29 August 2005. Accessed 21 January 2008 at: http://johanpeeters.com/papers/abuser%20stories.pdf

- Kongsli, Vidar. "Towards agile security in Web applications". *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications,* Portland, Oregon, 22-26 October 2006.

- Wäyrynen, J., Bodén, M., and G. Boström. "Security engineering and eXtreme Programming: an impossible marriage?". In Zannier, C., Erdogmus, H., and L. Lindstrom, editors. *Extreme Programming and Agile Methods—XP/Agile Universe 2004.* Berlin, Germany: Springer-Verlag, 2004. pages 117-128. Accessed 11 September 2008 at: http://is.dsv.su.se/PubsFilesFolder/612.pdf

- Beznosov, Konstantin. "Extreme security engineering: on employing XP practices to achieve "good enough security" without defining it". Presented at the First ACM Workshop on Business Driven Security Engineering, Washington, D.C. (District of Columbia), 31 October 2003. Accessed 4 April 2007 at: http://konstantin.beznosov.net/professional/doc/papers/eXtreme_Security_Engineering-BizSec-paper.pdf

- Davidson, Michelle. "Secure agile software development an oxymoron?" Application Security Tech Target, 25 October 2006. Accessed 4 April 2007 at: http://searchappsecurity.techtarget.com/originalContent/0,289142,sid92_gci1226109,00.html

- Tappenden, A., P. Beatty, J. Miller, A. Geras, and M. Smith. "Agile security testing of Web-based systems *via* HTTPUnit". *Proceedings of the AGILE 2005 Conference,* Denver, Colorado, 24-29 July 2005.

- Heckman, Rocky. "Is Agile Development Secure?". CNET Builder.au. 8 August 2005. Accessed 4 April 2007 at: http://www.builderau.com.au/manage/project/soa/Is_Agile_development_secure_/0,39024668,39202460,00.htm - *and* - http://www.builderau.com.au/architect/sdi/soa/Is_Agile_development_secure_/0,39024602,39202460,00.htm

- Siponen, M., R. Baskerville, and T. Kuivalainen. "Extending Security in Agile Software Development Methods". In Mouratidis, Haralambos and Paolo Giorgini, editors. *Integrating Security and Software Engineering: Advances and Future Visions.* Hershey, Pennsylvania: IGI Global Publishing, 2007.

- Ge, X., R.F. Paige, F.A.C. Polack, H. Chivers, and P.J. Brooke. "Agile Development of Secure Web Applications". *Proceedings of the ACM International Conference on Web Engineering,* Palo Alto, California, 11-14 July 2006.

- Williams, L., R.R. Kessler, W. Cunningham, and E. Jeffries. "Strengthening the Case for Pair-Programming". *IEEE Software.* Volume 17 Issue 4, July/August 2000, pages 19-25. Accessed 11 September 2008 at: http://www.cs.utah.edu/~lwilliam/Papers/ieeeSoftware.PDF —*and*— http://rockfish-cs.cs.unc.edu/COMP290-agile/ieeeSoftware.pdf

- *Boström, Gustav. Simplifying development of secure software—Aspects and Agile methods.* Licentiate Thesis for the Stockholm (Sweden) University and Royal Institute of Technology. undated. Accessed 9 April 2008 at: http://www.diva-portal.org/diva/getDocument?urn_nbn_se_kth_diva-3913-3__fulltext.pdf

3.6.3 Formal methods and secure software development

To the extent that vulnerabilities may result from the functional incorrectness of software implementations, formal methods, by improving software correctness, can also contribute to the improvement of software security. An example of a successful implementation of formal methods to improve software security is type checking, which is an integral feature of Java, C#, Ada95, and other modern programming languages. Type checking increases the detection rate of many flaws and faults at compile time and runtime.

Use of formal methods does not, in and of itself, guarantee that secure software will be produced. What formal methods do provide are a more precise way of specifying and modeling the requirements and design of software, and of verifying that the requirements and design adequately exhibit a given property or set of properties. Formal methods have proven successful in specifying and checking small, well-structured systems such as embedded systems, cryptographic algorithms, operating system reference models, and security protocols. For example, formal proofs were provided for the integrity property of the Trusted Computing Bases of the Honeywell Secure Communication Processor, the Boeing Multi-Level Secure Local Area Network device, and the Gemini Secure Operating System [used in the Gemini Trusted Network Processor] when these systems underwent National Computer Security Center's A1 level-of-assurance evaluations against the Trusted Computer Security Evaluation Criteria) in the mid- and late 1980s.

The research community is striving to automate formal methods more fully, to make them more practical for general use, at this point formal methods remain labor-intensive, even with tool support. For this reason, their practicality is limited to software in which a high level of trust and confidence must be placed (*i.e.*, security critical, safety critical, and other extremely mission-critical and high-consequence software).

There are some formal methods with tool support that have been used successfully to verify the security properties of safety-critical and security-critical software (mainly embedded systems software). These are listed in Table 3-6.

Table 3-6. Formal methods with tool support

Tool	Developer	Website
Correctness by Construction	Praxis High Integrity Systems Ltd.	https://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/sdlc/613.html?layoutType=plain —*and*— http://www.praxis-his.com/services/software/approach.asp
AutoFocus and Quest	Munich University of Technology	http://autofocus.in.tum.de/index-e.html —*and*— http://www4.informatik.tu-muenchen.de/proj/quest/
B-Method	Jean-Raymond Abrial	http://vl.fmnet.info/b/

While their labor-intensiveness and required level of expertise make formal methods impractical for most general-purpose software development, those readers who are interested

in further discussion of how formal methods can be used throughout the SDLC to produce secure software should consult Appendix C:C.3, as well as the suggested resources below.

SUGGESTED RESOURCES

- *Software Security Assurance*, Section 5.1.2.
- Heitmeyer, Constance, Myla Archer, Elizabeth Leonard, and John McLean. "Applying Formal Methods to a Certifiably Secure Software System". *IEEE Transactions on Software Engineering*, Volume 34 Number 1, January 2008, pages 82-98. Accessed 26 February 2007 at: http://chacs.nrl.navy.mil/publications/CHACS/2008/2008heitmeyer-TSE.pdf
- Wing, Jeannette M. *A Symbiotic Relationship between Formal Methods and Security.* Technical Report CMU-CS-98-188, December 1998. Accessed 4 April 2007 at: http://reports-archive.adm.cs.cmu.edu/anon/1998/abstracts/98-188.html
- Croxford, Martin and Roderick Chapman. "Correctness by Construction: a manifesto for high-integrity software". *CrossTalk: The Journal of Defense Software Engineering*, Volume 18 Number 12, December 2005. Accessed 21 January 2008 at: http://www.stsc.hill.af.mil/CrossTalk/2005/12/0512CroxfordChapman.html
- Hall, Anthony and Roderick Chapman. "Correctness by Construction: developing a commercial secure system". *IEEE Software*, January-February 2002, pages 18-25. Accessed 26 February 2008 at: http://www.anthonyhall.org/c_by_c_secure_system.pdf
- Srivastava, Amitabh. "Engineering Quality Software". Presented at the 6th International Conference on Formal Engineering Methods, Seattle, Washington, November 2004. Accessed 26 February 2008 at: http://research.microsoft.com/conferences/icfem2004/Presentations/AmitabhSrivastava.ppt
- Association for Computing Machinery (ACM) Workshops on Formal Methods in Security Engineering. Accessed 26 February 2008 at: http://www.cs.utexas.edu/~shmat/FMSE08/
- Heitmeyer, Constance Heitmeyer. Automatic Construction of High Assurance Systems from Requirements Specifications Web page. Accessed 26 February 2008 at: http://chacs.nrl.navy.mil/personnel/heitmeyer.html
- Breu, Ruth, Klaus Burger, Michael Hafner, Jan Jürjens, Gerhard Popp, Guido Wimmel, and Volkmar Lotz. "Key Issues of a Formally Based Process Model for Security Engineering". *Proceedings of 16th International Conference on Software and Systems Engineering and their Applications*, Paris, France, 2-4 December 2003. Accessed 7 July 2008 at: http://www4.informatik.tu-muenchen.de/~popp/publications/workshops/icssea03.pdf

4 REQUIREMENTS FOR SECURE SOFTWARE

> *NOTE: This section presumes that the reader is already familiar with good requirements engineering practices, and thus focuses only on the additional considerations associated with the need for software to be secure. The Suggested Resources list at the end of this section includes resources on good requirements engineering.*

It is said that most vulnerabilities and weaknesses in software-intensive information systems can be traced to inadequate or incomplete requirements. While this truism really refers to inadequate requirements as the reason why many information systems lack adequate security functionality to protect the information they process, it can also be seen in the context of failure to specify requirements that specify the functions, constraints, and non-functional properties of software that must be dependable, trustworthy, and resilient.

All software shares these three overarching security needs:

- It must be dependable under anticipated operating conditions, and remain dependable under hostile operating conditions.
- It must be trustworthy in its own behavior, and in its inability to be compromised by an attacker through exploitation of vulnerabilities or insertion of malicious code.
- It must be resilient enough to recover quickly to full operational capability with a minimum of damage to itself, the resources and data it handles, and the external components with which it interacts.

Frankly, if software cannot satisfy these three needs, it is of dubious value. What good is software that cannot be relied on to operate as and when it is needed? Thus, these three needs should be addressed by, and should inform, all of the requirements for the software's functionality, behaviors, and constraints. Moreover, the need for security is so inherent to all software that the system development practice of defining security requirements separate from all other requirements is quite absurd when it comes to software requirements. In all cases, "operate securely" is no less imperative than "operate correctly". What constitutes "secure" may differ depending on the criticality of the software's function, and the nature of its intended operational environment. But there is no modern software for which security is not a concern at some point in its life cycle.

Even non-networked embedded software may be subject to sabotage by malicious code insertion during its development. While the prevention of such sabotage clearly drives a requirement for secure software development and security analysis practices, the possibility that embedded malicious code may not be detected before the software is deployed drives a direct requirement for the software's own survivability, *i.e.*, to enable it to continue dependable operation even if the embedded malicious code is executed.

In their paper, "Core Security Requirements Artefacts",[20] Jonathan Moffett *et al.* 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."

It is critical that the software's security requirements be as complete, accurate, and internally consistent, and traceable as possible. At the start, a complete and accurate statement of system-level security goals and needs will be needed; this statement can then be explicitly "decomposed" to derive requirements for secure software. The authors go on to describe a requirements specification process that uses a series of iterative risk analyses and models to decompose system-level security goals into as complete and accurate a set of secure software requirements as possible.

Recognizing that software requirements specifications can be considered *complete* only if they are sufficient to distinguish desired behavior from undesired behavior, it is important for the requirements analyst to consider the impact that anticipated and unanticipated external stimuli and inputs may have on the software's behavior, not just during normal operation, but also when the software or any of its environment resources is in partial or full failure, and when the software comes under attack.

Understanding and anticipating the various hostile conditions to which the software may be subjected is critical to anticipating requirements that will enable the software to remain dependable, trustworthy, and resilient under those conditions. Gaining this understanding is the objective of the various threat/attack modeling techniques described later in this section.

Requirements traceability is an imperative generally for ensuring that all software requirements are not watered-down or lost in the design or implementation phases. As such, requirements traceability is a best practice of all good software development. This said, there is an important consideration for secure software: establishment of traceability between the artifacts that identify and model the threats, attacks, and vulnerabilities that form basis for defining the software's security requirements.

Just as traceability must be established and maintained between the hazard analyses performed for safety-critical systems and the requirements specification for the software

---

20 Moffett, Jonathan D., Charles B. Haley, and Bashar Nuseibeh. "Core Security Requirements Artefacts". Open University Technical Report Number 2004/23, 21 June 2004. Accessed 26 January 2008 at: http://computing-reports.open.ac.uk/index.php/content/download/166/999/file/2004_23.pdf

components of those systems,[21] traceability needs to be maintained between software specifications and the threat/attack and vulnerability analyses and models that informed the security requirements in those specifications.

The software safety community has produced a number of robust tools for providing automated support in the traceability of complex requirements for safety-critical systems. Examples include Praxis High Integrity Systems' REVEAL, Telelogic's DOORS, ChiasTek's REQTIFY, Safeware Engineering's SpecTRM, *etc.* These tools should be investigated for their adaptability to provide requirements traceability for secure software systems.

SUGGESTED RESOURCES

- BuildSecurityIn Requirements Engineering resources. Accessed 21 January 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/requirements.html
- *Software Security Assurance*, Section 5.2.
- Firesmith, Donald G. Security and Safety *Requirements for Software-Intensive Systems.* Boca Raton, Florida: Auerbach Publications, 2008.
- Araujo, Rudolph. "Security Requirements Engineering: A Road Map". *Software Mag.com*, July 2007. Accessed 13 December 2007 at: http://www.softwaremag.com/L.cfm?Doc=1067-7/2007
- Khan, R.A. and K. Mustafa. "Secured Requirement Specification Framework (SRSF)". *American Journal of Applied Sciences*, Volume 5 Number 12, 2008, pages 1622-1629. Accessed 7 July 2008 at: http://www.scipub.org/fulltext/ajas/ajas5121622-1629.pdf
- Petrovic, Mark. "Discovering a Java Application's Security Requirements". *OnJava*, 3 January 2007. Accessed 11 December 2007 at: http://www.onjava.com/pub/a/onjava/2007/01/03/discovering-java-security-requirements.html

4.1 THE CHALLENGE OF NEGATIVE AND NON-FUNCTIONAL REQUIREMENTS

21 For a methodology for requirements traceability in complex safety-critical systems development, see Section 4.0, "Commercial Aircraft Process Relationships", in Mathers, Greg, Joseph K. Simpson, *et al.* "Framework for the Application of Systems Engineering in the Commercial Aircraft Domain", DRAFT Version 1.2a, 28 July 2000. Accessed 25 August 2008 at: http://www.incose.org/ProductsPubs/pdf/techdata/SEApps-TC/FrameworkForApplicOfSEToCommercialAircraftDomain.pdf. A particularly robust methodology is described in USPTO Patent Application 20070250297 (accessed 28 August 2008 at: http://www.freshpatents.com/Method-for-reducing-hazards-dt20071025ptan20070250297.php?type=description). Unfortunately, it is not clear from the publicly-accessible patent description and claims data whether this methodology has been implemented in any requirements management product as of yet.

The following are challenges associated with the capture of negative security requirements, *i.e.,* requirements for security constraints on functionality, and non-functional security requirements, *e.g.,* requirements for software properties or for guidelines or standards governing with software development:

- Perceived security risks posed by certain requirements for functionality as well as interoperability, usability, or performance;
- Perceived security risks posed by other negative or non-functional requirements, *e.g.,* the "use COTS" imperative;
- Security and other policy statements mandating certain functional constraints (*e.g.,* a Web service must be authenticated before being allowed to communicate with another Web service) or development practices (*e.g.,* no use of unsigned mobile code);
- Standards mandating certain technical constraints (*e.g.,* Web service security standards) or development practices (*e.g.,* development process compliance with SSE-CMM);
- Tendency to think about and express security requirements in terms of things to be avoided or prevented.

Attempting to verify that something is avoided or prevented is essentially attempting to prove a negative: it is difficult (if not impossible) to prove that there can be no exceptions to what is demonstrated during testing, *i.e.,* that software will *never* do a certain thing. Moreover, the degree of non-satisfaction of a security requirement that can be tolerated is usually very low (approaching zero). Stakeholders want criteria for security requirements to be very close to "yes/no".

This said, requirements analysts should not self-impose limitations on their imaginations: the though process by which they conceive security requirements should allow for identifying both "actionable, testable" requirements and "non-actionable, non-testable" requirements, *i.e.,* negative and non-functional requirements. Allowing the mental step of specifying such requirements is a critical step towards defining the correlated positive, testable requirements for additional functionality, constraints on specified functional requirements, and SDLC process improvements that will satisfy those "non-actionable, non-testable" requirements.

Negative requirements exist because software's functionality must not be allowed to behave in a way that could lead to the software failing in an insecure state, or otherwise becoming vulnerable to exploitation or compromise. An example of a constraint on a function is "The software must not use input data it has not validated."

Constraints on software functions are intended to minimize the likelihood of non-secure software behaviors and the interactions that are likely to trigger those behaviors. Constraints do this by placing controls on how the software interacts, and on how it behaves in response to interactions.

Such constraints can be said to be violated when the software fails to enforce them, or commands one of its components/processes to behave in a way that violates them. Constraint violations most frequently arise as a result of (1) interactions between software components; (2) interactions between the software and entities in its environment; and (3) the software's failure. Such violations are particularly likely to occur when the external entity the software interacts with is an intelligent malicious adversary (human or malicious process).

It is the developer's job to make sure that such interactions cannot cause or contribute to a constraint violation. As part of the requirements definition process, then, the developer should identify all constraints that are needed to ensure the software always behaves securely. (Later, the developer must also be sure to design the software to enforce those constraints.)

The following is a suggested two-stage process for moving from negative constraint requirements to positive, testable functional security requirements:

- Map each negative and non-functional requirement to one or more positive functional requirements that will enable the negative or non-functional requirement to be satisfied.

- Recast as many negative/constraint requirements as possible positively, describing what is supposed to happen (not what isn't), and providing binary satisfaction criteria—*i.e.,* the function happens or it doesn't—and clear test criteria by which to determine what "the function happens" means.

For example, given the negative requirement "The software must not accept overlong input data", conceive of positive functional requirement(s) that will enable that negative requirement to be satisfied. Such a requirement might be "The software must validate all input to ensure it does not exceed the size specified for that type of input."

Non-functional security requirements specify:

1. Properties the software must exhibit (*e.g.,* its behavior must be correct and predictable; it must remain resilient in the face of attacks);

2. Required level of assurance or risk-avoidance of individual security functions and constraints;

3. Controls and rules governing the processes by which the software will be built, deployed, and operated (*e.g.,* it must be designed to operate within a virtual machine, its source code must not contain certain function calls).

In many cases, non-functional security requirements will be translated not into elements of the software's design, but into guidelines for its development process, or criteria for its testing.

During software requirements analysis decisions can begin to be made about which requirements can probably be satisfied using COTS or OSS components. In this context, both functional and non-functional security requirements need to be analyzed, particularly as COTS/OSS components often fall short of the level of dependability, trustworthiness, and

survivability required in medium- and high-consequence software systems. Recognizing that COTS/OSS components will be used will allow the requirements analyst to specify requirements for additional protections and countermeasures that will enable such components to be used without compromising the security of the system overall. The requirements analyst can also use these requirements as the basis for defining the security evaluation criteria for such components, with the criteria intended to verify that any candidate components do, in fact, achieve at least the minimum acceptable level of security as defined in the requirements specification.

4.2 ORIGINS OF REQUIREMENTS FOR SECURE SOFTWARE

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—especially (semi-)autonomous entities such as Web services and mobile agents—that act as, or on behalf of, users.

The most common sources of security requirements for software are:

- **Stakeholders' expressed security concerns.**

- **Security implications of the functional specification:** A functional specification is intended to answer the question "What does the software need to do to accomplish *x?*" whereas the question that *should* be answered is "What does the software need to do to accomplish *x* securely?" The resulting answers are, in fact, *secure* requirements (contrast this with security requirements, which are requirements for security-relevant functions. In short, functional requirements need to be stated in a way that makes it clear that the functions must not: (1) be vulnerable to anticipated attacks; (2) operate in a way that compromises any part of the software system or any external software entities (environment-level or application-level) with which the system interacts.

  When finding the answer to the "how to do it securely?" question seems difficult (or impossible) it is because the function desired is too dangerous to be performed securely, so that the basic functional requirement conflicts with the secure functional requirement. When such conflicts emerge, the requirements analyst must identify the trade-offs that will allow the dangerous functions to be performed without introducing excessive levels of risk to the system. Clearly, a risk assessment is needed to identify what level of risk would be excessive.

  Unfortunately, most requirements analysts are not software security experts, so their trade-offs often favor preserving basic functional requirements at the expense of security. For this reason, the inclusion in the development team of a software security

expert is important. The security expert can then collaborate with the requirements analyst, who will thereby gain security knowledge, as a protégé from a mentor.

- **Requirements for security functions:** Such requirements should be treated in the same way as all other functional requirements, except that in the prioritization that occurs during the trade-off analyses described above. The analyst needs to clearly understand the level of risk that arises when any security function relied on to protect a system, its assets, and resources, is not itself adequately *secure* in terms of dependability, trustworthiness, and robustness. The requirements should reflect this understanding: *i.e.*, security-critical and other high-consequence functions, and indeed *all* software functions, should be specified in a way that ensures that they do not an unacceptably high level of risk to the software system. For security-critical and high-consequence functions, the maximum level of risk acceptable will be significantly lower than it is for other functions.

- **Compliance and conformance mandates:** The need to comply with security-relevant laws such as Sarbanes-Oxley, the Federal Information Security Management Act (FISMA), and the Health Insurance Portability and Accountability Act (HIPAA), policies such as , directives such as DoD Directive 8500.1, "Information Assurance (IA)" and DCID 6/3, standards such as ISO/IEC 15408:1999, "The Common Criteria for Information Technology Security Evaluation", the National Aeronautics and Space Administration (NASA) Software Assurance Standard, and the World Wide Web Consortium WS-Policy Framework standard, can result in requirements both for security functions and for secure software behavior. For example, there are a number of requirements implicit in Web service security standards regarding how service-to-service interactions should be secured.

- **Secure development and deployment standards, guidelines, and "best practices":** There may be a mandate that the developers must comply with certain coding, security, or deployment standards, such as a requirement that the Java Virtual Machine's security features be leveraged whenever Java is the language used. A mandated set of guidelines for "locking down" the platform on which the software will run carries implicit requirements for how the software is expected to interact with its platform, and the fact that it should not require resources or services that are made unavailable by the secure platform configuration. Even if a development "rule" is not codified in a standard or guideline, the development organization may choose to follow certain accepted "best practices" that can influence the software's requirements, such as a requirement that a Web application not use persistent or unencrypted cookies as authentication tokens. The monitoring of an organization's SDLC processes verify consistent, correct use of secure system and software development standards, guidelines, and practices is a key concern of security-enhanced continuous process improvement models such as MSSDM and SSE-CMM.

- **Attack models and environment risk analysis:** Models of the attacks to which the software is likely to be subjected, along with the findings of a risk analysis of the

environment in which it will run (to include the data it will handle and the larger system of which it will be a component) provide the requirements analyst with a clear picture of the security context in which the software will operate, including the threats the software needs to be guarded against, and the anticipated environment-level security protections and services the developer may be able to leverage (1) to reduce the exposure of the software's vulnerabilities (its "attack surface") to those threats, and (2) to assist in achieving the software's secure operation. Section 2.2.1 discussed attack modeling in detail.

- **Known and likely vulnerabilities in the technologies and COTS and OSS components that, due to preexisting commitments, must be used:** Requirements can emerge from the need to implement countermeasures in the software to minimize exposure of such vulnerabilities. Vulnerability databases, such as the National Vulnerability Database (NVD) operated by NIST, are a good source of information about such vulnerabilities.

SUGGESTED RESOURCES

- Wu, Dan. "Security Functional Requirements Analysis for Developing Secure Software". Qualification Exam Report for University of Southern California, December 2006. Accessed 17 December 2007 at: http://sunset.usc.edu/csse/TECHRPTS/2006/usccse2006-622/usccse2006-622.pdf

4.3 DERIVING REQUIREMENTS THAT WILL ENSURE SECURITY OF SOFTWARE

In developing detailed functional and non-functional non-security requirements questions should be asked to identify associated security requirements. An example of this question-driven process for deriving security requirements appears below.

Example of Question-Driven Requirements Elicitation

Given the following high level functional requirement: **"**The server should store both public-access and restricted Web pages."

- Derive the detailed functional requirements, *e.g.*, "The server should return public-access Web pages to any browser that requests those pages."
- Identify the related constraint requirements, *e.g.*, "The server should return restricted Web pages only to browsers that are acting as proxies for users with authorized privileges sufficient to access those Web pages."
- Derive the functional security requirements, *e.g.*, "The server must authenticate every browser that requests access to a restricted Web page."
- Identify the related negative requirements, *e.g.*, "The server must not return a restricted Web page to any browser that it cannot authenticate."
- Derive more functional security requirements, *e.g.*, "After authenticating the browser, the server must determine whether that browser is authorized (*i.e.*, has necessary privileges) to access the requested restricted Web page."
- Identify the related negative requirement, *e.g.*, "The server must not return a restricted Web page to a browser whose privileges do not authorize it to access restricted Web content."
- Identify any additional implied requirements, *e.g.*, "The server and browser must support the necessary security mechanisms to enable the server to (1) authenticate the browser; (2) determine the browser's authorized privileges and whether those privileges are sufficient for the browser to access the restricted Web content."

The requirements engineering methodology used for specifying secure software should define a systematic approach that does not require onerous security knowledge on the requirements analyst's part to use.

The methodology used to define requirements should support the analyst in:

1. Determining how difficult it will be for each component (actor) of the system to achieve the system's collective security requirements;
2. Determine which components should be assigned which security requirements, avoiding "overloading" any component with more security requirements than it can satisfy;
3. Identifying the most security-critical components of the system (those responsible for satisfying the most security requirements);
4. Performing trade-offs between security requirements that invalidate the system's ability to satisfy its other functional and non-functional requirements. For high-consequence software, this trade-off analysis should include a quantitative analysis of residual security risk to which the system as a whole will be subject if certain security

requirements are waived or otherwise not satisfied. Alternate requirements that reduce that residual risk to an acceptable level can then be defined.

The resulting requirements specification should define as secure a software-intensive system as possible given all other system imperatives (*e.g.,* functionality, performance, usability, *etc.*)

Table 4-1 lists some requirements methodologies that have been used successfully in the specification of secure software. For each methodology, a resource providing information is also provided.

Table 4-1. Requirements engineering methodologies for secure software

Methodology	**Resource(s)**
Requirements Engineering VErification and VAlidation (REVEAL)	• Praxis High Integrity Systems. REVEAL Webpage. Accessed 19 December 2007 at: http://www.praxis-his.com/reveal/
Security Quality Requirements Engineering (SQUARE)	• Mead, Nancy R., Eric D. Hough, Theodore R. Stehney, II. "Security Quality Requirements Engineering (SQUARE) Methodology". Technical Report CMU/SEI-2005-TR-009|ESC-TR-2005-009, November 2005. Accessed 19 December 2007 at: http://www.cert.org/archive/pdf/05tr009.pdf
Trustworthy Refinement through Intrusion-Aware Design (TRIAD)	• Ellison, Robert J. and Andrew P. Moore. "Trustworthy Refinement through Intrusion-Aware Design". Technical Report CMU/SEI-2003-TR-002, March 2003. Accessed 22 March 2008 at: http://www.sei.cmu.edu/publications/documents/03.reports/03tr002.html
Appropriate and Effective Guidance in Information Security (AEGIS)	• Fléchais, Ivan, Cecilia Mascolo, and M. Angela Sasse. "Integrating security and usability into the requirements and design process". *Proceedings of the Second International Conference on Global E-Security, London, United Kingdom, April 2006.* Accessed 21 January 2008 at: http://www.softeng.ox.ac.uk/personal/Ivan.Flechais/downloads/iges.pdf • Fléchais, Ivan. *Designing Secure and Usable Systems*. University of London doctoral thesis, February 2005. Accessed 19 December 2007 at: http://www.softeng.ox.ac.uk/personal/Ivan.Flechais/downloads/thesis.pdf

Methodology	Resource(s)
Knowledge Acquisition in autOmated Specification (KAOS)	• Haley, Charles B. *Arguing Security: A Framework for Analyzing Security Requirements.* Open University doctoral thesis, March 2007. Accessed 23 January 2008 at: http://www.the-haleys.com/chaley/papers/Thesis-Final-DS.pdf • Haley, Charles, Jonathan D. Moffett, Robin C. Laney, and Bashar Nuseibeh. "A Framework for Security Requirements Engineering". *Proceedings of the 2006 Software Engineering for Secure Systems Workshop*, Shanghai China, 20-21 May 2006, pages 35-42. Accessed 17 January 2008 at: http://www.the-haleys.com/chaley/papers/Haley-SESS06-p35.pdf • Haley, Charles B., Robin Laney, Jonathan D. Moffett, and Bashar Nuseibeh, "Security Requirements Engineering: A Framework for Representation and Analysis". *IEEE Transactions on Software Engineering*, October 2007. • Haley, Charles B., Robin C. Laney, and Bashar Nuseibeh. "Deriving security requirements from crosscutting threat descriptions". *Proceedings of the Third International Conference on Aspect-Oriented Software Development*, Lancaster, United Kingdom, 22-26 March 2004, pages 112-121. Accessed 19 January 2008 at: http://oro.open.ac.uk/2491/01/AOSD04-Haley-final.pdf
Aspect Oriented Modeling (AOM)	• Xu, Dianxiang, Vivek Goel and K. Nygard. "An Aspect-Oriented Approach to Security Requirements Analysis". In *Proceedings of the 30th Annual International Computer Software and Applications Conference*, Chicago, Illinois, 17-21 September 2006, Volume 2, pages 79-82.

4.4 SECURE SOFTWARE REQUIREMENTS VERIFICATION CHALLENGES

The analyses performed to verify the correctness, completeness, and consistency of the requirements specification should include:

- **Internal analysis:** Determines whether the requirements for the software's non-functional properties, and security constraints and positive functional requirements derived from them are complete, correct, and consistent with the other functional and non-functional requirements in the specification (the former include requirements pertaining to safety, performance, usability, *etc.*).

- **External analysis:** Will answer the following questions:

    - Do the software's requirements adequately address the concerns of stakeholders, and the relevant mandates of law, regulation, policy, *etc.?*

- Do the constraint and non-functional security requirements represent a valid refinement of the overall system security goals? Do any security-related requirements conflict with those goals?

Exhibiting required properties ("the software must *be*" *vs.* "the software must *do*") requires careful test planning to determine what types and combinations of software behaviors and actions can be said to demonstrate each required property (*i.e.,* what must the software *do*—or not do—for it to demonstrate that it *is* resilient?)

The following is an example of a multi-stage requirement validation process.

1. Flag each functional requirement in the specification.

2. Search the specification to find one or more constraint ("negative") requirements directly pertinent to each functional requirement. For example, given a requirement that "This function must accept Social Security Numbers from users," there should be a constraint requirement along the lines of "This function must reject all inputs that exceed nine digits (*i.e.,* the length of a Social Security Number)."

3. For each constraint requirement, search the specification for at least one corresponding positive requirement for functionality needed to satisfy the constraint requirement. In our example, this might be "The function must perform bounds checking of all input to ensure its length does not exceed nine digits."

If there is not at least one constraint requirement for a function, the lack of constraints needs to be analyzed and justified or the appropriate constraint requirement should be added to the specification. Similarly, if there is not at least one positive functional requirement for every constraint requirement, this lack needs to be analyzed and justified or remedied.

For high assurance and high confidence software, more comprehensive analyses of security requirements, including inspections and peer reviews, may be needed to assess whether the necessary non-functional and constraint requirements have been captured clearly, in terms that can be easily translated into design and test plan. Formal methods—which first entail use of a formal, often executable specification language, to capture the security requirements so that the analyst can later use proof checkers to run proofs and other formal tools to execute models of those requirements—are an effective, albeit resource- and expertise-intensive, method for validating the correctness, completeness, and internal consistency of requirements, as well as their effectiveness in capturing their intent in a realistic manner.

However, formal methods will not be any more helpful than other methods in predicting the potential effectiveness of a given requirement in terms of its contribution to the dependability, trustworthiness, and/or resilience of the software that will be designed and built to that requirement.

4.5 REQUIREMENTS ENGINEERING AND SECURITY MODELING METHODOLOGIES AND TOOLS

At a simplified level, all software programs include five key components, as illustrated in Figure 4-1.

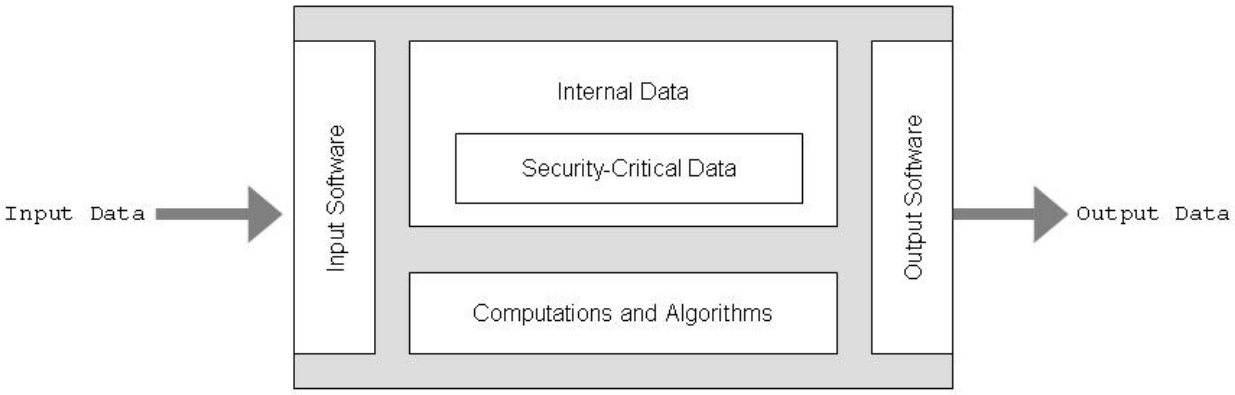


Figure 4-1. Components of software programs

1. **Input software:** Reads data into an internal representation for processing. If the input is cryptographically protected, the input software functionality includes data decryption, integrity verification, *etc.*

2. **Output software:** Software that writes data to an output medium (*e.g.,* Random Access Memory, compact disc, socket) after processing. If the output data is cryptographically protected, then output software functionality includes data encryption, integrity protection, *etc.*

3. **Internal data:** Data initialized by the application, read into an internal representation, or computed within the application (e.g. intermediate values or in preparation for output).

4. **Security-critical data:** A subset of internal data of high value to an attacker, *e.g.,* cryptographic keys, privilege-related data, and other security-critical data.

5. **Computations and algorithms:** Internal program logic that processes the internal data.

This building-block view of software programs clarifies the need to protect each of the five components from attacks in the categories discussed in Section 2.5.1. These protection needs can be decomposed into a set of high-level security requirements that are common to all software programs:

- **Secure the input as it is received and processed by the program:** To protect it from unauthorized access and manipulation, including unauthorized disclosure, tampering, corruption, destruction, and deletion.

- **Secure the output as it is processed and released by the program:** To protect it from unauthorized access and manipulation, including unauthorized interception, rerouting, disclosure, tampering, corruption, destruction, and deletion.

- **Data hiding:** To protect internal data from unauthorized disclosure. In the case of software, we are referring to types of data that tend appear in runtime interpretable source code and scripting code that has not been thoroughly sanitized prior to deployment. Traditional data hiding approaches such as encryption and file system access controls are useless for protecting such data, which should ideally be removed before the code is deployed, or if that is not possible, obfuscated using techniques that will not impede the runtime interpretation of that code.

- **Internal computation/algorithm hiding:** To protect internal program logic from unauthorized disclosure.

- **Tamper resistance:** To protect software code (source, runtime-interpretable, and executable) from unauthorized execution, tampering, corruption, destruction, and deletion.

- **Damage mitigation and recovery:** Entails, at a minimum, ability of software to be isolated from the external cause of the software's failure, then moving the software from a failed state to an acceptable operational state (ideally, this will be the software's pre-failure state). At a system level, damage mitigation also entails terminating all interface and communication paths used by the external cause to access the software and its platform, and monitoring and preventing all further attempts by the external cause to initiate new interface/communication paths to the software or its platform.

SUGGESTED RESOURCES

- BuildSecurityIn Modeling tools resources. Accessed 21 January 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/tools/modeling.html

- Shina, Michael E. and Hassan Gomaab. "Software requirements and architecture modeling for evolving non-secure applications into secure applications". *Science of Computer Programming*, Volume 66, Issue 1, 15 April 2007, pages 60-70.

- Haley, Charles B., Robin C. Laney, Jonathan D. Moffett, and Bashar Nuseibeh. "Using Trust Assumptions with Security Requirements". *Requirements Engineering Journal*, Volume 11 Number 2, April 2006, pages 138-151.

4.5.1 Attack modeling

Software systems may be subjected to a wide variety of attacks that may lead to security being compromised. While numerous in variety, the majority of attacks follow one basic strategy: to build up an attack through a series of smaller and/or atomic attacks (or intrusions). Security

can be modeled as a quantifiable attribute (in the context of Quality of Service and/or of Quality of Protection).

Attack modeling is used to model the nature of an attack. Good attack modeling requires the expertise of practitioners of cyber security, application security, information assurance, *etc.*, particularly penetration testers. (Contrast this with expertise needed for true threat modeling.)

Misuse cases, abuse Cases, security use cases, and other security modeling techniques such as attack trees, attack graphs, anti-models, and attack patterns, enable the analyst to gain a better understanding of the risk areas in the system under consideration by:

- Identifying security objectives to be satisfied by the software system;
- Exploring security threats to be countered by (or on behalf of) the software system;
- Identifying points in the distributed software that are likely to be attacked;
- Defining constraints (*i.e.,* the software must not be able to be compromised by certain threats) needed to achieve the required countermeasures and security objectives;
- Deriving testable (functional) security requirements that will ensure the software enforces the required constraints.

Specifically, there is a need to model:

- Attackers' likely actions (attack graphs and attack trees are useful for this type of modeling; attack patterns can be used as a source for describing specific actions, with graphs/trees depicting the series of generic atomic attacks to which the particular software being considered is subject);
- The software's individual and collective responses to attacker actions (attack response graphs lend themselves best to this type of modeling).

Attack modeling can be done separate from the risk analysis for a specific software system, *i.e.,* one can develop an attack tree or graph that any sufficient threat could execute. The following sections describe methodologies and tools that are useful for modeling the threat environment in which the software will operate.

4.5.1.1 Security use cases, misuse cases, and abuse cases

The same requirements gathering techniques employed to produce use cases that capture the tasks that users perform during "normal" usage of the system can be employed to produce security use cases that identify the users and user tasks associated with security, and to produce misuse cases that depict non-malicious (intentional or unintentional) threats to normal use cases (including security use cases) and abuse cases that depict malicious (intentional) threats to normal use cases. These misuse and abuse cases, in turn, form the basis

for additional security use cases that provide the means to counter or mitigate the threats captured in the misuse and abuse cases.

Misuse cases and abuse cases describe what the software should not do in response to a user's incorrect or malicious use of the system. For each functional use case, the developer should explore the ways in which that function could be deliberately abused or misused, and capture these in misuse/abuse cases. The developer then needs to identify all potential relationships between use cases and misuse/abuse cases, so that he/she can specify security constraints and requirements to prevent a misuse or abuse from compromising the security of each required capability identified in a use case.

On projects in which use cases are used to define functional requirements, adding misuse/abuse cases provides a uniform approach to the specification and analysis of all usage, both intended and adversarial usage.

SUGGESTED RESOURCES

- Hope, Paco, Gary McGraw, and Annie I. Antón. Misuse and abuse cases: getting past the positive. *IEEE Security and Privacy*, May-June 2004, pages 32-34. Accessed 21 January 2008 at: http://www.cigital.com/papers/download/bsi2-misuse.pdf
- Damodaran, Meledath. "Secure Software Development Using use cases and Misuse Cases". *Issues in Information Systems*, Volume VII, Number 1, 2006, pages 150-154. Accessed 13 December 2007 at: http://www.iacis.org/iis/2006_iis/PDFs/Damodaran.pdf
- Sindre, Guttorm and Opdahl, Andreas L. "Eliciting security requirements by misuse cases". *Proceedings of the 37th International Conference on Technology of Object-Oriented Languages and Systems*, Sydney, Australia, 20-23 November 2000, pages 120-131.
- Sindre, Guttorm and Andreas L. Opdahl. "Capturing Security Requirements through Misuse Cases". *Proceedings of the Norsk Informatikkonferanse*, Tromsø, Norway, 26-28 November 2001. Accessed 21 January 2008 at: http://folk.uio.no/nik/2001/21-sindre.pdf
- Sindre, Guttorm and Andreas L. Opdahl. "Templates for misuse case description". *Proceedings of the Seventh International Workshop on Requirements Engineering Foundation for Software Quality*, Interlaken, Switzerland, 4-5 June 2001. Accessed 11 September 2008 at: http://swt.cs.tu-berlin.de/lehre/saswt/ws0506/unterlagen/TemplatesforMisuseCaseDescription.pdf
- Alexander, Ian. "Modelling the interplay of conflicting goals with use and misuse cases". *Proceedings of 8th International Workshop on Requirements Engineering Foundation for Software Quality*, Essen, Germany, 9-10 September 2002, pages 145-15. Accessed 11 September 2008 at: http://ftp.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-109/paper1.pdf

4.5.1.2 Attack patterns

As defined in the Attack Pattern Glossary on the DHS BuildSecurityIn portal,[22] an attack pattern is:

> *…a general framework for carrying out a particular type of attack such as a particular method for exploiting a buffer overflow or an interposition attack that leverages architectural weaknesses…. [A]n attack pattern describes the approach used by attackers to generate an exploit against software.*

An attack pattern, then, describes and codifies the set of actions that constitute some type of attack. Attack patterns are intended as a generic mechanism for capturing and communicating the actionable details on specific types of common attacks that have the potential to affect one or more classes of software. By "generic" we mean that an attack pattern can be developed independent of the specific software system under consideration, resulting in a type of "generic" attack model that holds true for most (but not all) systems against which the attack may be launched. In fact, most existing attack patterns (such as those gathered in the Common Attack Pattern Enumeration and Classification [CAPEC]) are the product of in-depth analyses of specific examples of real world exploits against certain software products executed and used under certain environment and operational conditions.

Derived from the concept of design patterns, but applied in a destructive rather than constructive manner, attack patterns can collectively provide the requirements analyst with a picture of the hostile environment in which the software is likely to operate, so that he/she can:

- Specify the required secure software behavior in response to an attack described by an existing pattern;
- Identify constraints that describe undesired behaviors in response to a described attack;
- Identify positive functional requirements that enable the software to (1) resist or withstand, or (2) mitigate the impact of each described attack.

Following are two examples of attack patterns being leveraged in the specification of requirements for secure software.

Example 1

Given the following high level functional requirement: "The Web server application shall enable the user to purchase merchandise online by submitting a valid credit card number."

Consider the following attack patterns: "buffer overflow" and "command injection"

Identify the required constraint(s): "The software must not accept invalid user input."

---

22 Barnum, Sean and Amit Sethi. "Attack Pattern Glossary", 7 November 2006. Accessed 25 January 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/attack/590.html

Derive the associated functional requirement(s): "The application must validate the input in the credit card number form field to ensure it conforms to the following parameters:

1. "Range: ≥ 13 but ≤ 19 characters
2. "Format: numeric
3. "Encoding: International ASCII"

Identify any implied requirement(s): "The software must reject all input that deviates from the required parameters."

Determine any additional associated requirement(s): In the absence of specific requirements on how the rejection should be handled – *e.g.*, should the input validation routine return an error messages to the user requesting resubmission of the input, with the message providing guidance on acceptable length and format, or should the application itself first determine whether excessive length/incorrect format is caused by inclusion of hyphens between groups of numbers, and if so, should the application filter the input to delete the hyphens and accept the sanitized number that results without returning an error message to the user? – such details will be left up to the designer.

Example 2

Given the following functional requirement: "The Web server application shall return Web pages in response to URLs submitted by browsers."

Consider the attack pattern: "SQL injection"

Identify the required constraint(s): "The application must not accept invalid URLs from browsers."

Derive the associated functional security requirement(s): "The application must validate all URLs received from browsers to ensure that they conform to parameters defined for URLs. These parameters are:

1. "Format: [specifies acceptable prefixes – http:// or https:// followed by an alphanumeric strings followed by a period, another alphanumeric string, followed by a period or a forward slash, followed by another alphanumeric string followed by a forward slash, *etc.*]
2. "Encoding: International ASCII
3. "Content (allowable characters): [specifies "alphanumeric" plus all allowable symbols and the position within a Uniform Resource Locator (URL) in which each symbol may appear]"

Identify related constraint(s): "The input validation routine must reject any URL that does not conform to these parameters."

Determine any additional associated requirement(s): "Should a rejection cause the application to return a standard Hyper Text Markup Language (HTML) error message to the browser, or to simply 'hang' without returning a response?"

Attack patterns are, in fact, of use throughout the software life cycle, and their applicability to other life cycle phases will be discussed in the sections on those phases later in this document. Some specific examples of SDLC artifacts that attack patterns can be used in developing are:

- Misuse and abuse cases;
- Attack trees and attack graphs;
- Test scenarios, especially penetration test scenarios.

SUGGESTED RESOURCES

- BuildSecurityIn Attack Patterns resources. Accessed 21 January 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/attack.html
- Barnum, Sean and Amit Sethi. "Attack Patterns as a Knowledge Resource for Building Secure Software" (whitepaper). Ashburn, Virginia: Cigital, Inc., 2007. Accessed 26 December 2007 at: http://capec.mitre.org/documents/Attack_Patterns-Knowing_Your_Enemies_in_Order_to_Defeat_Them-Paper.pdf
- The MITRE Corporation. CAPEC-Common Attack Pattern Enumeration and Classification Website. Accessed 14 December 2007 at: http://capec.mitre.org/index.html
- Kis, Miroslav. Information Security Antipatterns in Software Requirements Engineering. *Proceedings of the Ninth Conference on Pattern Language of Programs,* Monticello, Illinois, 8-12 September 2002. Accessed 19 December 2007 at: http://jerry.cs.uiuc.edu/~plop/plop2002/final/mkis_plop_2002.pdf

4.5.1.3 Threat modeling

Threat modeling identifies and analyzes threats to a system (rather than the attacks that are used against a system). In threat modeling, the specifics of attacks are not considered. Instead the focus is on identifying and understanding the following four aspects of each threat:

- **Capability** (technical expertise, resources available, opportunity, *etc.*);
- **Intentions** (objectives);
- **History** of previous successful attacks against specific target(s);
- **Targeting** plan for intended, imminent attack(s) against same or different specific target(s).

A classic lesson from military history is "Where there is capability, an adversary may develop intent." It is important that software developers understand this lesson. It is not meaningful to identify and model threats when operating under the false assumption that "nobody would ever do that." Good threat modeling begins with the recognition that if it is possible, it is highly likely that *somebody* will do it.

Threat modeling requires the expertise of the law enforcement or intelligence/ counterintelligence community. Most other organizations are not qualified to model or deal

with threats and should limit their focus to mitigating that vulnerabilities that may be exploited by any threat. In addition, when a threat is identified or suspected, they should take all steps necessary to get law enforcement and/or the intelligence community to do threat removal.

At the software requirements level, true threat modeling is of limited value on its own; it can, however, provide information that forms an extremely useful basis for attack modeling.

On of the most popular attack modeling methodologies was that developed by Microsoft. In reality, Microsoft's Application Threat Modeling methodology combines aspects of attack modeling with that of system security risk analysis. What it does not include is true threat modeling. What Microsoft calls a "threat" is, in fact, an "attack". The methodology focuses on identifying all likely attack patterns, their likelihood of success given the software's vulnerability and exposure to those attacks, and the associated loss anticipated with successful attacks. Microsoft Threat Modeling, as well as the other software attack modeling/risk assessment methodologies in Table 4-1 model attacks, vulnerabilities, and outcomes for the software within the constraints of the following:

- The particular execution environment in which it is expected to operate,
- Its criticality/value to the enterprise,
- Its user's expectations,
- Its requirements for assurance, including security assurance.

The methodologies and tools listed in Table 4-2 are designed to support software and/or system attack modeling and security risk analysis. For each of these, a resource is listed that provides detailed information and (with one exception) access to documentation and, if available, the tool itself.

Table 4-2. Threat modeling tools

Tool	Resource(s)
Application Consulting & Engineering Threat Analysis and Modeling	Microsoft Security Developer Center Application Threat Modeling Webpage. Accessed 25 January 2007 at: http://msdn2.microsoft.com/en-us/security/aa570413.aspx
Calculative Threat Modeling Methodology	Practical Threat Analysis Website. Accessed 25 January 2008 at: http://www.ptatechnologies.com/
Trike	octotrike.org Tools Webpage. Accessed 25 January 2008 at: http://www.octotrike.org/
Consultative Object Risk Analysis System (CORAS)	The CORAS Project Webpage. Accessed 25 January 2008 at: http://coras.sourceforge.net/
Threat Modeling based on Attacking Path (T-MAP)	University of Southern California Center for Systems and Software Engineering. Security Economics and Threat Modeling for Information Technology (IT) Systems—A Stakeholder Value Driven Approach project Webpage. Accessed 25 January 2008 at: http://sunset.usc.edu/csse/research/COTS_Security/index.html *(NOTE: the T-MAP tool cannot be downloaded from this page; those interested in*

	use of T-MAP should send email or call the point of contact listed on the Webpage.) *—and—* Chen, Yue. *Software Security Economics and Threat Modeling Based on Attack Path Analysis: A Stakeholder Value Driven Approach.* University of Southern California Doctoral Dissertation, December 2007. Accessed 25 January 2008 at: http://sunset.usc.edu/csse/TECHRPTS/PhD_Dissertations/files/ChenY_Dissertation.pdf

SUGGESTED RESOURCES

- *Software Security Assurance*, Section 5.2.3.1.
- OWASP Threat Risk Modeling Webpage. Accessed 25 January 2008 at: http://www.owasp.org/index.php/Threat_Risk_Modeling
- Daniel P.F. «Análisis y Modelado de Amenazas» [in Spanish], Version 1.0, 18 December 2006. Accessed 25 January 2007 at: http://metal.hacktimes.com/files/Analisis-y-Modelado-de-Amenazas.pdf

4.5.1.4 Other modeling techniques

The following are some additional attack modeling techniques that have been promoted for aiding in specification of secure software.

4.5.1.4.1 Attack trees and attack graphs

An attack tree (sometimes referred to as a "threat tree") is, in essence, a fault tree that concentrates on faults with security impacts. In an attack tree, the attacker's goal is placed at the top of the tree, then the analyst documents possible alternative ways in which that attacker goal could be achieved. For each alternative, the analyst may recursively add precursor alternatives for achieving various sub-goals that collectively achieve the main attacker goal.

The attack tree analysis process is repeated for each attacker goal. By examining the lowest-level nodes of the resulting attack tree, the analyst can identify all possible techniques that might be used by an attacker to compromise the system's security, and can then specify the means of preventing or avoiding those attack techniques as security requirements for the system.

Compared with the misuse/abuse case, the attack tree captures a greater level of detail, and thus may be more helpful in developing a detailed design. Tree and graph-based attack models:

- Capture the steps of a successful attacks;
- Range from simple tree models to formal Petri nets;
- Can capture both general and system specific attack methods, and system properties and other preconditions that make a successful attack possible;
- Focus on causes of vulnerabilities, but do not identify countermeasures;
- Should be complemented by secure coding guidelines, security patterns, *etc.*

Attack trees are represented both graphically and textually. A graphical representation is usually built with the root node, or goal, on the top. The tree then descends branches and sub-goals until the leaves are finally reached at the bottom level.

The textual representation of an attack tree follows a numeric outline structure. The root node, or goal, is represented at the first level with no indentation. Each sub-goal is then numbered accordingly and indented one unit per level of decomposition. The representation below presents the textual view using the same example content found in Figure 4-2.

Goal: Fake Reservation

1. Convince employee to add reservation
    1.1 Blackmail employee
    1.2 Threaten employee
2. Access and modify flight database
    2.1 SQL injection from Web page
    2.2 Log into database
        2.2.1 Guess password
        2.2.2 Sniff password
        2.2.3 Steal password from server
            2.2.3.1 Get account on server (AND)
                2.2.3.1.1 Exploit buffer overflow
                2.2.3.1.2 Get access to employee account
            2.2.3.2 Exploit race condition to access protected profile

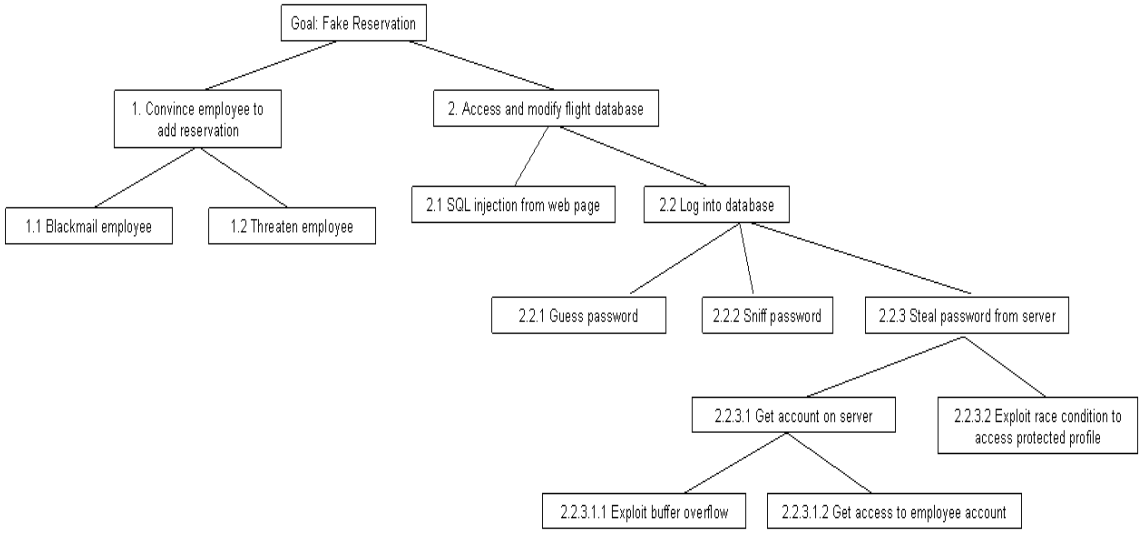


Figure 4-2. Conceptual view of a graphical attack tree

Used in combination with attack patterns, attack trees can capture how the attack patterns are likely to be combined and sequenced, so that appropriate multi-action responses can be designed in to the software.

An attack graph is a concise representation of the attack paths through the system that attack patterns may follow in order to achieve the end-state of the attacker's ultimate goal. The attack

graph complements the attack tree by modeling all of the potential attack paths throughout the system that could be associated with a specific attack, while the attack tree focuses on the "stopping points" at which each attack goal and sub-goal is attained.

Failure Modes and Effects Analysis (FMEA) is a bottom-up approach that can complement attack tree and attack graph analysis. In FMEA, the analyst examines all potential failures of the existing or planned security protections, mitigations, and countermeasures and models and documents the anticipated consequences of each failure, tracing that consequence to extrapolate how it could inhibit the ability of the system to achieve its mission or operational objectives.

Manual attack tree and graph generation and FMEA are time-consuming and tend to be error-prone when used to model and analyze large software systems. Researchers are pursing the use of formal modeling techniques and model-checking technology to automate the generation and analysis of attack trees and graphs.

SUGGESTED RESOURCES

- Edge, Kenneth S. *A Framework for Analyzing and Mitigating the Vulnerabilities of Complex Systems via Attack and Protection Trees.* Air Force Institute of Technology doctoral thesis, July 2007. Accessed 21 January 2008 at: http://handle.dtic.mil/100.2/ADA472310
- Schneier, Bruce. "Attack trees: Modeling security threats". *Dr. Dobbs Journal*, December 1999. Accessed 21 January 2008: http://www.schneier.com/paper-attacktrees-ddj-ft.html —*and*— http://www.ddj.com/184411129
- Kortti, Heikki. "Input Attack Trees". Presented at Black Hat Japan, 5-6 October 2006. Accessed 21 January 2008 at: http://www.blackhat.com/presentations/bh-jp-06/BH-JP-06-Kortti-up.pdf
- Gupta, Suvajit and Joel Winstead. "Using Attack Graphs to Design Systems". *IEEE Security and Privacy*, July/August 2007, pages 80-83.
- Carnegie Mellon University. Scenario and Attack Graphs project Webpage. Accessed 26 January 2008 at: http://www.cs.cmu.edu/~scenariograph/

4.5.1.4.2 Anti-models

As described by Axel van Lamsweerde, the anti-models are a type of formal model, *i.e.,* they are captured in mathematical notation. Specification requires temporal logic that is first-order, realtime, and linear. Mathematically-notated semantics are used to define how to evaluate truth of a model. What are modeled are the goals of each component (or agent), along with the component's role (what it does to achieve its goal), and properties of the domain (execution environment). Also captured are the "anti-goals", *i.e.,* the security goals of each component (or agent); along with these are modeled the environment, attackers, threats posed by the attackers, and assets threatened. Goals are refined to the level at which a single agent can realize a single goal.

Then specific benefits of anti-goals are:

- They elicit answers to questions such as "Which attacks are likely threaten this asset?"
- They capture the high level goals of such attacks;
- They enable the refinement of attacker goals by expressing the negations of specified security requirements, then constructing *AND/OR* trees from the resulting anti-goal statements;
- They can be analyzed with the assistance of an automated tool such as *Objectiver*[23] to determine whether they can be realized.

SUGGESTED RESOURCES

- van Lamsweerde "A. Elaborating Security Requirements by Construction of Intentional Anti-Models". *Proceedings of the 26th International Conference on Software Engineering*, Edinburgh, Scotland, May 2004, pages 148-157. Accessed 19 January 2008 at: http://www.info.ucl.ac.be/Research/Publication/2004/avl-Icse04-AntiGoals.pdf
- van Lamsweerde, Axel, Simon Brohez, Renaud De Landtsheer, David Janssens. "From System Goals to Intruder Anti-Goals: Attack Generation and Resolution for Security Requirements Engineering". *Proceedings of the 2003 Workshop on Requirements for High Assurance Systems*, Monterey, California, September 2003, pages 49-56. Accessed 17 January 2008 at: http://www.cs.toronto.edu/~jm/2507S/Readings/avl-RHAS03.pdf —*and*— http://www.cs.ndsu.nodak.edu/~vgoel/Security-Engineering/avl-RHAS03.pdf —*and*— http://www.info.ucl.ac.be/Research/Publication/2003/avl-RHAS03.pdf

4.5.1.4.3 State transition diagrams

In their paper "Building Security Requirements Using State Transition Diagram at Security Threat Location",[24] Seong Chae Seo, *et al.,* describe an approach for threat modeling using state-transition diagrams.

23 This tools can be downloaded from: http://www.objectiver.com

24 Seo, Seong Chae, Jin Ho You, Young Dae Kim, Jun Yong Choi, Sang Jun Lee, and Byung Ki Kim. "Building Security Requirements Using State Transition Diagram at Security Threat Location". In *Lecture Notes in Computer Science: Computational Intelligence and Security*, Volume 3802/2005, pages 451-456. Heidelberg, Germany: Springer-Verlag, 2005.

5 SECURE DESIGN PRINCIPLES AND PRACTICES

> *"Many systems fail because their designers protect the wrong things, or protect the right things in the wrong way".* [25]

5.1 SECURE ARCHITECTURE CONSIDERATIONS

At the architectural level, the software's execution environment can be "provisioned" with a variety of security services and protections that will reduce the possibility of malicious input reaching the software, minimize the exposure of the software's own vulnerabilities to the outside world, minimize the external visibility of trusted and high-consequence components to reduce their exposure to threats, and isolate untrustworthy components so that their execution is constrained and their misbehaviors do not threaten the dependable operation of other components. Such security services/protections may include:

- Application-level firewalls and intrusion prevention systems to block known malicious and problematic input before it reaches the software;
- Virtual "sandboxes" that provide an isolated environment in which untrustworthy components can be executed to prevent their potential misbehaviors from affecting trustworthy components;
- Code signature validators (code signatures are digital signatures applied to executable code for validation either at time of delivery (mobile code) or installation, or at runtime, to determine one or both of the following:
    - Whether the code originated from a trusted source;
    - Whether the code's integrity has been compromised since it was generated.)

In the development of commercial software, it is often not possible for designers to make any reliable assumptions about the security services/protections that will be present or active in the execution environment. For this reason, engineers of commercial software need to be extra diligent in designing software that is capable of "looking after" its own dependability, trustworthiness, and survivability since the assurance of these properties cannot be depended upon from external sources.

If the software is to be deployed in a variety of platforms, the design should include an interface abstraction layer that minimizes the software's need to accommodate the differences between the platforms.

---

25 Anderson, Ross. J. *Security Engineering: A Guide to Building Dependable Distributed Systems,* Second Edition. New York, NY: John Wiley & Sons Inc., 2008.

NOTE: A Virtual Machine Monitor (VMM) is an increasingly popular implementation of an interface abstraction layer.[26]

SUGGESTED RESOURCES

- *Software Security Assurance*, Section 5.3.
- Howard, Michael and David LeBlanc. *Designing Secure Software.* San Francisco, California: McGraw-Hill Osborne Media, 2007.
- Aaby, Anthony A. "Security and the Design of Secure Software". In *Software: A Fine Art*, Draft Version 1.0, 9 February 2007. Accessed 19 January 2008 at: http://cs.wwc.edu/~aabyan/FAS/book/node5.html#SECTION05500000000000000000 - *and* - http://moonbase.wwc.edu/~aabyan/FAS/book.pdf —*and*— http://cs.wwc.edu/~aabyan/FAS/book.pdf

5.2 SECURE SOFTWARE DESIGN PRINCIPLES AND PRACTICES

The first three general secure software principles in Section 3.1—*(1) Minimize the number of high-consequence targets; (2) Don't expose vulnerable or high-consequence components;* and *(3) Deny attackers the means to compromise*—provide a framework by which to categorize the secure design principles discussed below.

One general design principle falls outside these three categories:

- **Make sure that the design specification is easily comprehensible and traceable.** Comprehensibility will make the design specification easier to analyze to reveal possible vulnerabilities and weaknesses. A specification that is fully traceable will make it easy to determine whether the design satisfies all of its requirements, including its security-relevant requirements. This traceability should be backward and forward, *i.e.*, it should be possible to trace forward from a requirement to its manifestation in the design, and backward from a point in the design to derive the requirement(s) satisfied at that point. It should also be possible to trace forward from any point in the design to its manifestation in the implemented code, and backward from the code to the part of the design realized by that code.

---

26 The VMM hides the details of the underlying platform from the software running within the virtual machines provided by the VMM. Each virtual machine presents the software with an unchanging set of execution environment interfaces, with the VMM acting as a kind of "translator" between those interfaces and the actual interfaces of underlying platform (Windows, Linux, Mac OS X). As long as the VMM runs on a given platform, the software can be installed and executed on that platform without any modification. This portability extends to the software's expectations of the environment-level security services/protections it will receive. Problems may arise, however, if a certain platform does not provide a certain expected service/protection. The VMM as conduit to the underlying platform cannot hide the fact that the relied-on service/protection is not there. It may, however, make it easier to add a third-party service/protection to the platform without having to change the software running in the virtual machine—as long as the VMM's platform-level interfaces can be extended to address that third-party package.

NOTE: In practice, conformance to secure design principles presents a challenge because developers typically do not have the deep, detailed understanding of the target execution environment in which their software will run. This understanding is critical to the ability to define a detailed design of software that will conform to these principles. For example, the ability to design software that conforms to the principle of least privilege requires the developer to fully understand the operation of the privilege granting and revoking mechanisms in the underlying operating system on which the software will be hosted. In any case, the designer is always required to make assumptions about the execution environment. The likelihood of the designer's security assumptions being correct will be greatly increased through consultation with the security expert assigned to the development team; this challenge is, in fact, one of the strong arguments supporting the need of such an expert.

5.2.1 General Principle 1: Minimize the number of high-consequence targets

The following principles and practices contribute to the design's ability to conform to this general principle.

5.2.1.1 Principle of least privilege

According to the Software Assurance CBK:

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

Least privilege supports General Principle 1 by minimizing the number of actors in the system granted high levels of privilege, and the amount of time any actor holds onto its privileges.

The essence of least privilege is "That which is not expressly permitted is forbidden." Least privilege determines how access (read, write, delete, execute) privileges are granted to actors (users, software entities initiated by users, autonomous software entities). In many systems, privileges assigned to software entities initiated (and thus acting on behalf of) users are identical to those assigned to the users that initiate them; this is usually done because it is the most convenient way to design the system.

However, the principle of least privilege requires privileges to be assigned based on the real need of the actor who needs the privilege to perform a task. The requirements of the task should determine the level of privilege assigned to the actor that will perform that task. If the task can only be performed by a software entity, and not by a human user, the privilege assigned to the software entity should differ from that assigned to the human user who initiated the software entity.

The principle of least privilege requires the designer to think about the actual privilege required for each task a software entity may perform, rather than simply assigning a general

(user) set of privileges to that entity that will enable it to perform all tasks, including tasks it doesn't ever need to perform. Software entities specified in the design should never request or receive privileges that exceed the minimum level of privilege needed to accomplish a given task.

Nor should actors retain privileges indefinitely. In a truly secure design, each entity would retain the privilege it required to perform a given task only long enough to complete that task. The entity would then relinquish that privilege and, if it needed to perform the same task again later, it would be reissued the necessary privilege at that time. In practice, this is unlikely to be feasible, and for most low-privilege, "untrusted" tasks, it will be acceptable to assign the privilege to the software actor for some fixed duration (*e.g.,* the length of a "session"), before requiring the entity to relinquish the privilege then provide whatever credentials it needs to request the privilege again. The more sensitive or critical a task, however, the less time the software entity that performs that task should be allowed to retain the privilege required to perform it. In the case of tasks associated with trusted functions (*e.g.,* cryptographic and policy enforcement functions, program control/configuration update functions), the rule of "use then relinquish" should be enforced.

Least privilege is made easier to enforce in a design in which the number of "sensitive" and trusted (*i.e.,* highly privileged) functions is kept to a bare minimum. In a software system in which multiple functions will require different privileges, the functions should be designed and implemented as multiple discrete, small, simple (ideally single-purpose) executables that call each other, and not as a single, large, complex multifunction executable that requires a variety of privileges.

Modularizing functions into separate simple executables minimizes the number of privilege changes that must occur during the software's execution. It should be possible to decompose complex functions that on first glance appear to require a high level of privilege into multiple, simpler functions (or tasks), most of which will not require high level privileges. For example, a cross-domain solution that is used to review, reclassify, and release documents from a classified network to an unclassified network can be decomposed into a series of tasks that will collectively accomplish the file "downgrade" and "release":

Parsing of file text content;

1. Determining whether parsed text is unclassified or classified;
2. Changing the sensitivity label on the file if unclassified;
3. Routing the downgraded file to its intended destination/recipient on the unclassified network.

Only one of these tasks actually needs a high level of privilege that will enable it to violate the mandatory access control policy of the file system in which the file to be "downgraded" resides: the label-changing function. While it is crucial to protect the integrity of the other functions to ensure their trustworthiness, these functions do not require high levels of privilege.

Software that is comprised of small, simple functions (implemented by small, simple executables) will be easier to analyze and troubleshoot, and its security will be much easier to assure.

Least privilege alone will not guarantee secure software. Other principles of secure design must also be followed, including avoiding single points of failure and not presenting high-value targets to attackers. For this reason, highly privileged functions should never be centralized in a single component or module.

The privileges a component/process requires should never conflict with known security policy and configuration constraints mandated for the organization and environment in which the software will operate. For example, the DoD Security Technical Implementation Guides (STIGs) define constraints that must be configured for various application frameworks/platforms and operating systems, and the architect and designer of software that is custom-developed for DoD should make himself/herself aware of these mandatory constraints to ensure that his/her assumptions about the software's execution environment do not conflict with what will actually be provided in the "STIGed" execution environment. For example, if an operating system STIG prohibits the assignment of "root" or "superuser" privileges to untrusted software programs, the design should not require the software to be granted such privileges. Responsible architects and designers should always ask or research to determine whether mandatory constraints similar to those defined in DoD's STIGs will apply in the environment/organization for which their software is being custom-developed.

5.2.1.2 Principle of separation of privileges, duties, and roles

These essence of separation of privileges is "No single entity (human or software) should have all the privileges required to modify, overwrite, delete, or destroy the system as a whole, or all components and resources that comprise the system." Separation of privileges and duties/roles supports General Principle 1 by helping minimize the number of different privileges required by any single actor in the system.

The way to make separation of privilege easier is to also enforce separation of duties or roles. What this means is that instead of all entities being able to access all parts or perform all functions, the entities are assigned roles or duties that require them to perform only a subset of the overall functions provided by the system.

In terms of how software is designed, separation of roles and privileges is also consistent with simplicity: *i.e.,* instead of designing one large, complex multifunction entity that requires "superuser" privileges in order to perform all of its functions, the system should include multiple simple, single-function entities that require only the privileges needed to accomplish their function.

In a Web service/service oriented architecture (SOA) context, software entities (services) also take on different roles: there are consumers, providers, and intermediaries. The duties associated with each role are different:

- A consumer service needs to be able to generate and transmit Universal Description, Discover, and Integration (UDDI) and SOAP (formerly Simple Object Access Protocol) request messages and to receive and parse the content of Web Service Definition Language (WSDL) and SOAP response messages;

- A provider service needs to be able to receive and parse the content of SOAP request messages, and to generate and transmit SOAP response messages;

- An intermediary service acts as a message router, and therefore needs to be able to generate and transmit UDDI and SOAP messages and to receive and parse the content of WSDL response messages, but does not need to be able to parse the content of SOAP request or response messages.

Similarly, the expectations for software systems' users should also be governed by separation of privileges, duties, and roles (and by least privilege): end users of server-side systems will not require the same privileges or functions as those in administrator, Webmaster, and other such roles. A simple example of this is provided by how duties are separated and the associated privileges granted for a traditional Web portal application. The end user requires only the ability to read posted content and to enter data into HTML forms. The Webmaster, by contrast, needs to be able to read, write, and delete content and to modify the HTML forms' software code. The privileges required by the user role and its duties, therefore, are significantly different from those required by the Webmaster role and its duties.

5.2.1.3 Principle of separation of domains

Separation of domains is a principle that supports General Principle 1 by making separation of roles and privileges easier to implement. It also supports General Principle 2 by reducing the exposure of different actors and objects in the system to each other, which minimizes the likelihood that non-malicious actors will interact with malicious actors, or that a malicious actor will be able to easily gain access to any memory locations or data objects on the system. Together, these two controls ensure that users and processes are able to perform only tasks that are absolutely required, and to perform them on only the data, in only memory space, and using only the functions that they absolutely must access to accomplish those tasks. In practical terms, this is achieved through compartmentalization of users, processes, and data. This compartmentalization also helps contain the impact of faults and failures.

In Windows, Linux, and Unix (and its derivatives), access controls alone cannot isolate intentionally cooperating programs. If cooperation of malicious programs is a concern, the software should be implemented on a more secure platform, such as a trusted operating system or in a virtual machine reinforced by hardware isolation.

5.2.2 General Principle 2: Don't expose vulnerable or high-consequence components

The following practices contribute to the design's ability to conform to this general principle.

5.2.2.1 Keep program data, executables, and program control/configuration data separated

This practice supports General Principle 2 by reducing the likelihood that an attacker who gains access to program data will easily locate and gain access to program executables and/or control/configuration data.

The majority of techniques for separating program data from control data, and from executables at all levels of the system (processor up to application) are execution environment-, and more specifically file system-, level techniques such as:

- If possible, host the software only on platforms that implement the Harvard architecture, which ensures that program data and control data are stored in two physically separate memory segments.
- Set permissions on program data and its associated metadata to be readable and writable only by the program that creates the data/metadata unless there is an explicit need for other programs/entities to be able to read and/or write that data/metadata. The exception to this rule is the data's access permissions themselves (which could be seen as a type of metadata): these should be writable only by the program and the administrator, but readable by all users (human and software) unless the data is hidden, in which case its metadata and permissions should also be hidden.
- A program's control/configuration data should only be readable by that program, and should only be writable by the administrator. The exception is client application or browser configuration/preferences data that is expressly intended to be configurable by the user. In this case, the user should be allowed to read/write such data only *via* a purpose-built configuration or preferences interface.
- In a Web server application, unless there is an explicit need for users to directly view data used by a script, all such data should be placed outside the Web server's document tree.
- Prohibit programs and scripts from writing files to world-writable directories such as the Unix */tmp* directory. All directories to which programs write should be configured to be writable only by those programs.
- Store data files, configuration files, and executable programs in file system directories that are separate from each other. An executable program or script should not be writable by anyone except the administrator (and the installer if that is a separate role). The deployed (operational/production) executable should not be readable by anyone; the program's users should be granted only execute-only privileges to the executable.

- Programs and scripts that are configured to run as a Web server's "nobody" user should be modified to run under a specific username, and the "nobody" account should be deleted.

- If possible, encrypt all executable files, and implement a trusted decryption module that executes as part of program initiation to decrypt the executable so it can run.

If file system access controls alone are not strong enough to isolate software's control/configuration data from tampering and deletion/destruction, additional measures such as file encryption and digital signature should be implemented. These would require the software to include cryptographic logic to decrypt and validate the signature on the control/configuration file at program startup.

If the software's own host's access controls are not considered adequately robust, it may be desirable to store the software's control/configuration data on a remote trusted server (*e.g.*, the server that hosts the single sign-on service used by the system, the Lightweight Directory Access Protocol directory used by the system's public key infrastructure, or another such trusted system service. Each request to access the remote server should be transmitted over an encrypted (*e.g.*, by Secure Socket Layer/Transport Layer Security [SSL/TLS]) connection to prevent the clear-text data from being "sniffed" in transit (*e.g.*, as part of a reconnaissance attack, or in preparation for tampering with the configuration). If the software requires read-back verification of the configuration data it receives, the connection over which verification is received should also be encrypted.

For software components of a system that is likely to be cloned,[27] when a remote server is used to store the software's control/configuration data, if that data can be changed *via* the software, the changes should not be sent over the same communication channel by which the software earlier retrieved the configuration data from the server; instead, the software should send the changed data back to the remote server *via* a separate encrypted channel.

Section 5.5 describes the wide variety of environment-level mechanisms that can be used to provide a constrained execution environment.

5.2.2.2 Segregate trusted entities from untrusted entities

This practice supports General Principle 2 by reducing the exposure of the software's high-consequence functions from its high-risk (vulnerable and untrustworthy) functions, which are

---

27 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.

susceptible to delivery of malicious code or corruption by attackers so that their execution threatens the dependable, trustworthy operation of the software as a whole.

> *NOTE: High-consequence functions are those whose failure would have a high negative impact on the ability of the software's users to accomplish their objectives or mission.*

An untrusted entity (component, agent, process) is one that is considered untrustworthy based on its inability to satisfy some predefined criterion/criteria for determining trustworthiness. Determination of trustworthiness (and thus "trust") may be made at any point in the software's development, or it may be determined during its operation. For example, a COTS component whose source code could not be reviewed during the system's development may be deemed "untrusted" because its trustworthiness could not be adequately assessed before its deployment. By the same token, a mobile agent that cannot be authenticated during the system's operation may be designated "untrusted", as could a Java applet that is not digitally signed or whose code signature cannot be validated.

By contrast, a trusted entity is able to satisfy the criteria by which trustworthiness is determined. Trusted entities are most often used to perform high-consequence functions, including those that involve security decisions or control/configuration transformations.

All entities should be considered untrusted until expressly verified to be trustworthy (and thus trusted). An untrusted entity should never be granted privileges higher than those assigned to the end user that invoked the entity and/or on whose behalf the object is operating. This is true whether the object was invoked by direct execution or indirectly *via* a chain of directory or Web service requests that can ultimately be traced back to the user.

Isolate trusted entities in their own execution area (with resources dedicated to that execution area) to minimize their exposure to untrusted entities and the software's external interfaces (through which attack-patterned input, delivered malicious code, *etc.,* may be delivered).

The software's high-risk untrusted entities should also be isolated to limit the potential for propagation of the impact and to minimize the damage that results from the execution of any malicious or attack-compromised logic embedded within those entities. In particular, remotely-sourced downloaded software, such as mobile code and mobile agents, and software that processes files/documents (*e.g.,* word processing documents) containing embedded macros should (1) always be considered untrustworthy (and thus "untrusted"), and (2) isolated before execution.

Many execution environments provide mechanisms for configuring restrictive "isolation areas" for this purpose. As with isolation of high-consequence functions, isolation of high-risk functions will prevent those functions from accessing other areas of the software and of its execution environment, including the portions of the file system that contain the rest of the software's executable image(s), data files, and control/configuration file(s).

The Java and Perl (Practical extraction and report language) security architectures include sandboxing functions. .NET includes a Code Access Security mechanism in its Common

Language Runtime (CLR). At runtime, the sandbox or CLR assigns a level privilege to the executable(s) contained within it. This privilege level should be the minimal needed by the function(s) the code is *expected* to perform during its normal, correct operation. If the executable operates in an unexpected way, *i.e.,* performing any unexpected function, the sandbox/CLR will generate an exception, and the exception handler will prevent the executable from performing that unexpected operation and from accessing any resources outside the sandbox. On Unix systems, *chroot* jails can be configured to provide sandbox-like isolation.

More robust isolation mechanisms include virtual machines, trusted operating systems, and trusted processor modules. These and other execution environments that provide software protection and isolation mechanisms are discussed in Section 5.5.

Other than constrained execution environments, some mechanisms that can be used to protect and constrain the execution of software include:

- **Hardware initialization:** Initialization of hardware memory to a bit pattern that will revert to a safe state if, for any reason, instructions start being read from random memory;

- **Program shepherding:** A technique for monitoring 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:** These are 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:** These can be used 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.[28]

Note that successful isolation and containment depend on the ability to establish and sustain the isolation/containment mechanism in a secure state, and to receive warnings of imminent failures in sufficient time to minimize damage and ensure that the failure will not endanger the protected/constrained software. Related secure design principles are described in Section 5.3.3.

SUGGESTED RESOURCES

- Shi, Weidong, Hsien-Hsin S. Lee, Chenghuai Lu, and Mrinmoy Ghosh. "Towards the Issues in Architectural Support for Protection of Software Execution". Georgia Institute of Technology Report Number CERCS;GIT-CERCS-04-29, 2004. Accessed 31 December 2007 at: http://smartech.gatech.edu/bitstream/1853/4949/1/git-cercs-04-29.pdf

5.2.2.3 Minimize the number of entry and exit points into and out of any entity

Strive for one entry point into any software entity (function, process, module, component) and ideally one, or at most very few, exit points. This principle supports General Principle 2 by reducing the number of software access points exposed to attackers. It also makes the resulting software easier to analyze, and when implemented at the component level, it makes substitution and replacement of components easier.

5.2.2.4 Assume environment data is not trustworthy

The designer should assume that all components of the execution environment are neither dependable nor trustworthy unless and until this assumption is proved wrong. This principle supports General Principle 2 by reducing the exposure of the software to potentially malicious execution environment components or attacker-intercepted and modified environment data.

> *NOTE: This principle is not limited to environment components. The designer should assume that* all *entities external to the software are untrustworthy, and should accordingly validate all data received from those entities.*

---

28 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".

For this reason, the software should be designed with minimal dependency on data provided by its execution environment, and should validate all environment data it does receive before using that data.

Some application frameworks are verified to provide trustworthy environment data to the applications hosted within those frameworks. 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 trustworthy environment data at runtime to Java programs.

5.2.2.5 Use only safe interfaces to environment resources

This practice supports General Principle 2 by reducing the exposure of the data passed between the software and its environment.

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* **unless** controls are imposed that are adequate to prevent any user, attacker, or malicious program from gaining control of the calling program and exploiting its direct calling mechanism(s). Not only does each application call to a system-level function create a potential target for attack, whenever the software issues a system call, 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).

Filtering of system calls, whereby a monitoring program must inspect and approve system calls invoked by an untrusted program before the program's processing is 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.

5.2.3 General Principle 3: Deny attackers the means to compromise

The following principles and practices contribute to the design's ability to conform to this general principle.

5.2.3.1 Simplify the design

By keeping the design as simple as possible, the designer will be less likely to include weaknesses and vulnerabilities, especially hard to detect weaknesses/vulnerabilities, or to introduce complexities that make the design and its security implications difficult to analyze and understand. This principle supports General Principle 3 by minimizing the number of attacker-exploitable vulnerabilities and weaknesses in the system. This principle will also make the design, and the implemented software, easier to analyze and test.

Some specific design choices that will simplify the software's design are:

1. Limit the number of states possible in the software;

2. Favor deterministic processes over non-deterministic processes;

3. Use single-tasking rather than multitasking whenever practical;

4. Use polling rather than interrupts;

5. Include minimal feature sets and capabilities in components; these should be only those features/capabilities the components require to perform their job in the program/system. The architectural decomposition of a program should match its functional decomposition, enabling a one-to-one mapping of program segments to their intended purposes;

6. Decouple components and processes to minimize interdependencies among them. Minimizing interdependencies will prevent a failure or anomaly in one component/process from rapidly affecting the states of other components/processes. Note that vulnerabilities frequently arise when environment and hardware components that a software component depends on fail. Decoupling can be best achieved by:

    a. Modularizing the program's functionality into discrete, autonomous processing units. This includes dividing individual functions' processing sequences into multiple series of small, single-purpose increments, rather than implementing the function as a single complex step;

b. Establishing barriers to prevent communications between components that are not intended to interact;

c. Allocating only read-only or restricted-write memory space for processes to use, to prevent all but explicitly-authorized components from changing data values;

d. Favoring loose coupling of functions over tight coupling;

e. Avoiding time-dependent processes (*i.e.,* processes that must execute within a given timeframe so they cannot wait for other conditions to occur before they execute). This will increase the program's threshold of tolerance for unanticipated events or interactions;

f. Avoiding invariant processing sequences that allow only one way for the program to reach its goal. Such sequences can be exploited by an attacker to cause unexpected events and interactions (other than those defined in the invariant processing sequence);

7. Leave out unnecessary features. If the design includes COTS or OSS components in which dormant code, dead code, unnecessary functions ("features"), or undocumented functions are present, the design should also include wrappers to isolate those unused code segments to prevent them from being inadvertently or intentionally accessed or triggered during the software's execution.

5.2.3.2. Hold *all* actors accountable, not just human users

This practice supports General Principle 3 by ensuring that all attacker actions are observed and recorded, contributing to the ability to recognize and isolate/block the source of attack patterns and thereby prevent attacks from succeeding.

The traditional means of enforcing accountability has been a combination of auditing and non-repudiation measures. Auditing amounts to security-focused event logging to record all security-relevant actions performed by actors while interacting with the system. What distinguishes auditing from standard event logging are (1) the type of information captured in the audit record; (2) the level of integrity protection applied to the audit records to prevent them from being intentionally or inadvertently deleted, corrupted, or tampered with.

Non-repudiation measures are applied to any data objects created or manipulated as a result of an actor's interaction with the system. Such data objects can range from electronic documents to email messages to database or form field entries to interprocess communications (*e.g.,* SOAP messages between Web services). The non-repudiation measure, most often a digital signature, binds proof of the identity of the actor responsible either for the creation or manipulation (modification, transmission, receipt) of the data object, so that the actor cannot later deny responsibility for that act.

A prerequisite for auditing and non-repudiation is the ability to bind the authenticated identity of the actor to the act/event or data object for which it is to be held accountable. This

presumes that authentication, ideally strong authentication, of the actor will occur, and that some mechanism, such as a digital identity certificate, will be irrevocably associated with that actor as a result of the authentication of its identity.

In a software-intensive system, auditing and non-repudiation measures need to extend beyond just the human users to include semi- and fully-autonomous software entities, such as agents and Web and grid services that operate without human intervention, and in some cases without human knowledge. The accurate attribution of all software actors is the only way to establish a "paper trail" by which responsibility for security violations or compromises can be traced back to whatever human agent is responsible for the software entity (or entities) that caused the violation/compromise.

For Web services in particular, SAML certificates provide a mechanism for service-to-service authentication; SAML can be used in other types of applications as well, or secure RPC may be used. It is not yet clear that a comparable approach for establishing software process identity and accountability is workable within non-distributed software systems.

5.2.3.3 Avoid timing, synchronization, and sequencing issues

This practice supports General Principle 3 by reducing the likelihood of race conditions, order dependencies, synchronization problems, and deadlocks. The developer should be careful to understand and apply effective techniques and measures to avoid such issues and to ensure asynchronous consistency within any multitasked and multithreaded programs.

Many timing and sequencing issues are caused by the sharing of state information (particularly realtime and sequence-order information) across disjoint program abstractions such as unrelated or conflicting classes in object-oriented programs.

> *NOTE: An example of a disjoint abstraction is a class called* CustomerTable *in an object-oriented database application. The class is disjoint because the objects "customer" and "table" have nothing in common; by contrast, "circle" and "ellipse" are related abstractions – both are geometric shapes defined by continuous curved lines; therefore, the class* CircleEllipse *in an object-oriented drawing program would not be disjoint.*

The dissimilarity of objects in a disjoint abstraction can result in a conflict if the two attempt to interact. For example, consider the following situation:

Object A is designed to request a validation from Object B then, if it doesn't receive that validation within five seconds, to terminate. Object B is designed to give precedence to validation requests it receives from Object C; if such a request is received while Object B is processing one of Object A's validation requests, Object C is designed to interrupt that processing to handle Object C's request. This interrupt may exceed Object A's five-second wait threshold, thus causing Object A to terminate.

Timing, synchronization and sequencing errors may be exploitable by attackers. In the example above, an attacker could use man-in-the-middle spoofing attack to pose as Object C in order to issue a validation request that it knows will cause Object B to interrupt all processing

in order to handle the ostensible Object C request. If the interrupt exceeds five seconds, Object A will terminate: so, the attacker has, in essence, caused a failure in the program.

To avoid such timing, synchronization, and sequencing issues in software, developers should:

- Make all individual transactions atomic (non-interdependent);
- Use multiphase commits for data "writes";
- Use hierarchical locking to prevent simultaneous execution of processes;
- Set processes' "wait" thresholds (for responses from other processes) as high as possible, given the program's performance requirements;
- Reduce time pressures on system processing whenever possible, *e.g.,* by slowing processing rates.

Multitasking and/or multithreading in programs that run on operating systems that support those features can improve program performance. However, they also increase the programs' complexity, making it harder to analyze and verify software correctness and security.

Multitasking and multithreading also increase the likelihood of deadlocks. Deadlocks occur when two tasks or threads both stop executing at the same time because each is waiting for the other to release a resource or to terminate (this is true even when only a single thread is used, if two processes communicate with one another and share resources such as memory and disk addresses).

If the program is designed to multitask or multithread, the designer should ensure that tasks/threads that are intended to execute simultaneously have been synchronized to avoid conflicts among their system resource usage attempts.

5.2.3.4 Make secure states easy to enter and vulnerable states difficult to enter

This practice supports General Principle 3 by reducing the likelihood that the software will be allowed to inadvertently enter a vulnerable (exploitable) state.

Software should always begin and end its execution in a secure state. State changes should always be deliberate and never inadvertent; this is particularly important for changes from secure to vulnerable states.

If multiple secure states are possible, as a result of different processing conditions, the software should include a decision-making capability that enables it to choose the most appropriate secure state to enter.

5.2.3.5 Design for controllability

Include logic for the control of the software's execution to increase the resilience of the software. This principle supports General Principle 3 by making it easier to detect attack paths, and disengage the software from its interactions with attackers. Some specific design features that will increase the software's controllability include:

- Ability to self-monitor and limit resource usage;
- Providing feedback that enables all assumptions and models upon which the program makes decisions to be validated before those decisions are taken. This feedback should include intermediate states and partial results of processing events;
- Exception, error, and anomaly handling and recovery that:
    - Place critical flags and conditions as close as possible to the code they protect.
    - Interpret the absence of a protected condition as indicating the software process is not protected;
    - Use interlocks (batons, critical sections, synchronization mechanisms) to enforce sequences of actions or events so that no event can occur inadvertently, or when an undesirable condition exists, or out of sequence. For example, lockouts can be used to block access to a vulnerable process, or to preserve secure state and protect the software against using invalid data or valid data received in the wrong order or at an unexpected time or speed.

5.2.3.6 Design for secure failure

This practice supports General Principle 3 by reducing the likelihood that a failure in the software will leave it vulnerable to attack. Some specific design features that will increase the likelihood that software will fail securely include:

Implement watchdog timers that check for "I'm alive" signals from processes. Each watchdog timer should be set by software other than that which it is responsible for observing.

The exception handling logic should always attempts to take corrective action before a failure can occur, and to allow thresholds to be set to indicate "points of no return" beyond which recovery from a fault, vulnerable state, or encroaching failure is recognized to be unlikely or infeasible. Upon reaching this threshold, the exception handler should allow the software to enter a secure failure state (*i.e.*, a failure state in which none of the software's program and control data and not other sensitive data or resources controlled by the software are suddenly exposed, and in which damage resulting from the failure is minimized).

5.2.3.7 Design for survivability

This practice supports General Principle 3 by minimizing the amount of time a faulty or failed software component remains unable to protect itself from attack.

The software design should enable the software to take advantage of any redundancy and rapid recovery features at the system level. For example, if the system will support automatic backups and hot sparing of high-consequence components with automatic swap-over, the software system's design should be modularized in such a way that its high-consequence components can be decoupled and replicated on the "hot spare" platforms.

The software's own error, anomaly, and exception, handling and recovery should support both backward and forward recovery. Backward recovery enables the software to detect every anomaly and error before it is able to create an exploitable vulnerability or escalate to a failure. If a failure does occur, the exception handler should return that software to a known good state that is more secure than the failure state. The biggest challenge will be detecting and recognizing anomalous and erroneous states in the first place.

Forward recovery measures include the use of robust data structures, the dynamic alteration of flow controls, and the tolerance (*i.e.,* ignoring) of single-cycle errors that do not persist beyond one cycle.

Error handling in the software should recognize and tolerate errors likely to originate with human mistakes, such as input mistakes. The designer needs to:

- Allow enough fault tolerance in the software to enable it to continue operating dependably in the presence of a fairly large number of user input mistakes;
- Determine just how much information to provide in error messages by weighing the benefit of helping human users correct their own mistakes against the threat of reconnaissance attackers being able to leverage the knowledge they gain from overly informative error messages.

5.2.3.8 Server functions should never rely on clients to perform high-consequence functions or trust client-originated data

This principle supports General Principle 1 by reducing the consequence of the client (and thus eliminating a high-consequence target) and General Principle 3 by eliminating an attack path by frequently used by attackers to target server applications. Browsers and other clients should never be trusted to perform security-critical or other high-consequence functions. Reliance on client-originated data makes a server application 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.

The designer should always assume that clients will run in the most hostile execution environments possible.[29] Server applications, portals, and proxy agents that interact with

29 There are exceptions, but these are "trusted clients" that have been expressly engineered as components of high-confidence systems, and that run on high-assurance platforms, such as MILS (Multiple Independent Layers of Security) platforms.

clients should be designed to protect themselves against attacks originating from clients that have been subverted, hijacked, or spoofed. ***Server-side software should always validate all data originating from a client, even if the client validated that data first.*** While client-side input validation may be useful as a filtering mechanism to eliminate some unacceptable data before it can be sent to the server, the server should never count on the client for filtering out bad data. It should always perform its own input validation.

Principles of good input validation include:

- Centralize input validation logic;
- Ensure that input validation cannot be bypassed;
- Rely on positive "white list" validation, not negative "black list" filtering. Use "black lists" only for preliminary filtering to reduce the amount of data that must undergo white list validation;
- Validate all user input, including input from software proxies and agents acting on behalf of human users. Validation should check for correct, allowable length, format, and syntax;
- Reject all executable content in input from sources not explicitly authorized to submit executable content (*e.g.,* sources of mobile code downloads);
- Verify that programs that request actions or call processes are entitled (by policy) to issue those requests/calls;
- Define meaningful reactions to input validation failures. Rejection of failed input and sanitization (according to pre-defined rules) of failed input are the most frequent reactions.

Section 6.4 describes how to implement server-side input validation.

SUGGESTED RESOURCES

- BuildSecurityIn Design Principles resources. Accessed 21 January 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/principles.html
- BuildSecurityIn Design Guidelines resources. Accessed 21 January 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi-rules/home.html
- *Software Assurance CBK*, Section 3.4.

5.3 MODELING AND RISK ANALYSIS FOR ARCHITECTURE AND DESIGN

During architecture and design, decisions must be made about how the software will be structured, how the various components will integrate and interact, which technologies will be

leveraged, and how the requirements defining how the software will function will be interpreted. Careful consideration is necessary during this activity, as up to 50 percent of software defects leading to security problems are design flaws.

Security models are abstractions that capture, clearly and simply, the security properties and risk areas of the software. Models enable the developer to experiment with different variations and combinations of components at the architectural level, and of individual functions and constraints at the design level, to assess their comparative effectiveness in minimizing risk. By organizing and decomposing the whole software system into manageable, comprehensible parts, security modeling can also help the developer pinpoint and specify design patterns for software's security functions.

Modeling frameworks can help the designer capture the architectural patterns that specify the structure and security-relevant behaviors of the software system. Collaboration frameworks are similarly helpful for capturing design patterns that specify sets of abstractions that work together to produce common security-relevant behaviors.

Specifically, software security modeling can help the designer:

1. Anticipate the security-relevant behaviors of individual software components, behaviors related to the interactions among components, and behaviors of the software as it interacts with environment components and other external entities;

2. Identify functions that may need to be modified or constrained, and inter-component interfaces that may need to be filtered or protected;

3. Detect and correct errors in and omissions from the assumptions that informed the software requirements specification, and in specifying the detailed design, the software architecture;

4. Identify known vulnerabilities and failure modes in the architectural and design-level countermeasures to ensure that the software's most vulnerable components and functions are not exposed to exploitation and compromise;

5. Identify conflicts between any component's assumptions about the behaviors, security constraints, or security functions of any other component, and experiment with alternative integration/assembly options to eliminate those conflicts or to identify countermeasures to minimize their impact (if they can't be eliminated). Architectural modeling in particular should reveal all security dependencies between different parts of the system;

6. Identify conflicts between the components' and whole system's assumptions about the security protections and services provided by the execution environment, again enabling experimentation with alternative integration/assembly options that eliminate those conflicts and identification of countermeasures to minimize the impacts of irresolvable conflicts. Architectural modeling should also reveal all security

dependencies between the system (and its individual components) and its execution environment;

7. Analyze the security impact of any new or changed requirements or component additions, deletions, or substitutions;

8. Reveal any conflicts between the architecture and the design, or the design and the implementation, so as to document in the appropriate development artifacts any additional, unanticipated risks posed by those conflicts.

The software's architectural and design models should be revalidated at each phase of the life cycle to ensure that changes made in each subsequent SDLC phase are iterated back the earlier phase's artifact(s). For example, if during implementation the developer decides that to improve performance, he will code input validation logic into a module instead calling out to a separate input validation engine, this change in approach would have to be iterated back into the design for that module.

As with requirements, if formal specification language is used in documenting the design, proof checkers and other formal tools can be used to validate its completeness, correctness, and internal consistency (with the same caveat about aiding in predicting the security-effectiveness of the design).

SUGGESTED RESOURCES

- BuildSecurityIn Architectural Risk Analysis resources. Accessed 21 January 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/architecture.html
- BuildSecurityIn Modeling tools resources. Accessed 21 January 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/tools/modeling.html
- Xu, Dianxiang and Joshua J. Pauli. "Threat-Driven Design and Analysis of Secure Software Architectures". *Journal of Information Assurance,* Volume 1 Issue 3, 2006. Accessed 17 December 2007 at: http://www.homepages.dsu.edu/paulij/pubs/xu-pauli-JIAS.pdf
- Pauli, Joshua and Dianxiang Xu. "Threat-Driven Architectural Design of Secure Information Systems". *Proceedings of the Seventh International Conference on Enterprise Information Systems,* Miami, Florida, 24-28 May 2005. Accessed 19 December 2007 at: http://cs.ndsu.edu/%7Edxu/publications/pauli-xu-ICEIS05.pdf
- Shina, Michael E. and Hassan Gomaab. "Software requirements and architecture modeling for evolving non-secure applications into secure applications". *Science of Computer Programming,* Volume 66, Issue 1, 15 April 2007, pages 60-70.

5.3.1 Leveraging misuse and abuse cases in architecture

Misuse and abuse cases can form the basis of scenarios that can be used to analyze the architecture and, for new software products, to adjust that architecture. Questions that should be answered during this analysis include:

- Which areas and components of the architecture does each misuse and abuse scenario affect?
- How do the effects of a given scenario propagate through the architecture? Which other areas/components are affected as a result?
- What is the nature of the damage that results from a particular misuse/abuse scenario?
- Do the security constraints and protections in the architecture prevent the abuse/misuse scenario from succeeding?
- What additional security constraints/protections can be applied, and where, to prevent an abuse/misuse from succeeding, or barring prevention, to minimize the resulting propagation and damage?

In component-based development, the scenarios should be used to analyze different candidate component assembly architectures to determine and select the architecture that is most effective in its ability to prevent the greatest number of abuse/misuse scenarios, and to

minimize the propagation and damage from those scenarios whose success cannot be prevented.

After identifying shortfalls, the architecture can be adjusted by adding new constraints and protections, replacing vulnerable components with more robust components, changing order of processing flows, *etc.* The new architecture model that results should document these changes along with their rationales. The new architecture should then be evaluated against the misuse/abuse scenarios, with the refinement process reiterated, and another new architecture model generated. This iterative architectural engineering should be repeated until, ultimately, an acceptable architecture is defined.

These scenarios should also be added to the software's test plan so they can later be used to verify the effectiveness of the anti-abuse/misuse countermeasures as implemented in the built software.

SUGGESTED RESOURCES

- Damodaran, Meledath. "Secure Software Development Using use cases and Misuse Cases". *Issues in Information Systems,* Volume VII, Number 1, 2006, pages 150-154. Accessed 13 December 2007 at: http://www.iacis.org/iis/2006_iis/PDFs/Damodaran.pdf
- Pauli Joshua J. and Dianxiang Xu. "Misuse Case-Based Design and Analysis of Secure Software Architecture". *Proceedings of the International Conference on Information Technology Coding and Computing,* Las Vegas, Nevada, April 2005. Accessed 17 December 2007 at: http://www.homepages.dsu.edu/paulij/pubs/pauli-xu-ITCC05.pdf
- Pauli, Joshua J. and Dianxiang Xu. "Trade-off Analysis of Misuse Case-based Secure Software Architectures: A Case Study". *Proceedings of the 3rd International Workshop on Modeling, Simulation, Verification and Validation of Enterprise Information Systems,* Miami, Florida, May 2005, pages 89-95. Accessed 13 December 2007 at: http://cs.ndsu.edu/~dxu/publications/pauli-xu-MSVVEIS05.pdf
- Mouratidis, Haralambos, Paolo Giorgini, and Gordon Manson. "Using Security Attack Scenarios to Analyse Security During Information Systems Design". *Proceedings of the International Conference on Enterprise Information Systems,* Porto, Portugal, April 2004. Accessed 17 December 2007 at: http://homepages.uel.ac.uk/H.Mouratidis/Paper91_CR.pdf

5.3.2 Leveraging attack patterns in architecture and design

Attack patterns at all levels can provide a useful context for understanding the threats that software is likely to face, and determining which architectural and design features to avoid and which to specifically incorporate. Attack patterns can be particularly useful as "building blocks" for defining some of the abuse and misuse scenarios described in Section 5.2.1 (*i.e.,* scenarios that incorporate known attacks).

Some attack patterns describe attacks that directly exploit architectural and design flaws in software. And example of such an attack pattern is provided below.

Make the Client Invisible

This attack pattern exploits client-side trust issues apparent in the software architecture. The pattern indicates that absolutely nothing returned by the client to the server should be trusted, regardless of what security mechanisms are used to secure the communications path between them (*e.g.*, SSL/TLS).

Because the client is untrusted, the attacker could exploit it to return literally any data. For this reason, the server should perform all input validation, authorization checks, *etc.* on which trust is based (client-side input validation may be relied on only as a filtering mechanism to filter out bad data erroneously submitted by non-malicious users, while server-side input validation filters out bad input data intentionally submitted by malicious users or attackers.

Any authorization checks by the client should be seen as mere convenience measures: they may be used to determine whether the user has the authority to change client-side configuration parameters, such as the aesthetic and ergonomic aspects of how content is presented by the browser, or to reduce the amount of content displayed in accordance with the user's personal preferences. Client-side authorizations should not, however, be used as the basis for access control, *i.e.*, to prevent content from being returned from the server to a client that is not authorized to see that content.

All content sent from server to client should always be handled as if it were visible by the client. The server should never rely on the client to enforce "hidden data" or other data tagging that is intended to prevent the client from displaying certain content. Content that is not meant to be displayed by the client should simply not be sent to the client. Security-through-obscurity should never be relied on to enforce confidentiality or privacy requirements. If relied on at all, it should only be used as an "inconvenience measure", *i.e.*, to make it harder for the average non-malicious user to discover content by accident. Security-through-obscurity should never be relied on to prevent a determined attacker from discovering the obscured or hidden content, or even to prevent a non-malicious user from accidentally "stumbling across" (revealing) that content.

Thus, an architect who considers the Make the Client Invisible attack pattern should recognize the need to design the system so that no critical or trusted business logic is ever performed on the client side. Client-side business logic can only be seen as a convenience measure for performing untrusted, non-critical functions, such as tailoring the way content is visually presented to the end user the subversion of which would in no way threaten the security of the trusted and critical functions of the software system or the information it handles.

The designer should document the attack patterns that have been used in the specification of abuse/misuse scenarios and in other analyses performed as part of specifying the software design. The scenarios that incorporate these attack patterns can be added to the software's test plan. Such testing will validate the effectiveness of measures designed into the software to counter those attack patterns.

SUGGESTED RESOURCES

- Barnum, Sean and Amit Sethi. "Attack Patterns as a Knowledge Resource for Building Secure Software" (white paper). Ashburn, Virginia: Cigital, Inc., 2007. Accessed 26 December 2007 at: http://capec.mitre.org/documents/Attack_Patterns-Knowing_Your_Enemies_in_Order_to_Defeat_Them-Paper.pdf

- Fernandez, Eduoardo B., J.C. Pelaez, and M.M. Larrondo-Petrie. "Attack patterns: A new forensic and design tool". *Proceedings of the Third Annual IFIP WG 11.9 International Conference on Digital Forensics,* Orlando, Florida, 29-31 January 2007. Accessed 11 September 2008 at: http://www.springerlink.com/content/mv0541345hx15345/

- Gegick, Michael and Laurie Williams. "Matching attack patterns to security vulnerabilities in software-intensive system designs". *Proceedings of the Workshop on Software Engineering for Secure Systems,* St. Louis, Missouri, 15-16 May 2005.

- Gegick, Michael and Laurie Williams. "On the design of more secure software-intensive systems by use of attack patterns". *Information and Software Technology,* Volume 49, Issue 4, April 2007, pages 381-397.

5.4 RELATIONSHIP OF SECURITY PATTERNS TO SECURE SOFTWARE

Security patterns are means by which expert knowledge about characteristics of secure designs are captured. Security patterns are problem-oriented and intended to be comprehensible by non-experts. They primarily address questions at higher levels of abstraction than source code, *i.e.,* at the levels of architecture, design, algorithms, protocols, security policy, network structure, *etc.* In practical terms, security patterns are intended as reusable "building blocks" of language defining commonly needed design features; this language can be "dropped into" a new design specification, with modifications if needed, instead of the designer having to invent the language from scratch.

The best way to determine whether a given security pattern will help in the definition of a design that achieves software assurance objectives is to determine whether the pattern upholds one or more of the secure design principles described in Section 5.2. Examples of software assurance-relevant security patterns are provided in Table 5-1.

Table 5-1. Security Pattern Examples

Pattern	Corresponding Secure Design Principle
Boundary protection to eliminate risks from zero day attacks through realtime detection and prevention of known attack patterns and circumvention of unknown, abnormal behavior	Design for resilience
Exception handling to enable the system, if compromised by any type of malicious cyber attack, to handle the results of the compromise, and at worst to fail into a known, secure state	Design for resilience, Design for secure failure
Orderly quarantine using pre-defined fallback configurations that increase isolation and protection in response to attack patterns or perceived threats	Segregate trusted entities from untrusted entities
Informed recovery by trustworthy automated recovery mechanisms that assist the software system in the timely isolation and correction of its vulnerabilities	Design for resilience

An increasing number of security patterns have been defined that are directly relevant to software assurance concerns (by contrast with information or network security functionality). For example, in their *Secure Programming Cookbook for C and C++*,[30] authors John Viega and Matt Messier describe a number of what are essentially secure design patterns, along with code examples in C and/or C++ for implementing those patterns. While the majority of patterns they describe pertain to implementation of security functions such as authentication, access control, and cryptographic functions, there are some key chapters devoted to patterns directly pertinent to achieving software dependability, trustworthiness, and resilience. These are: Chapter 1, Safe Initialization; Chapter 3, Input Validation; Chapter 12, Anti-Tampering; and Chapter 13, Other Topics, which include secure error handling, secure memory management, correct use of variables, and secure management of threads, sockets, and resources.

Another source of software assurance-relevant security patterns is the Microsoft Patternshare repository,[31] which includes 30 such patterns: Compartmentalization, Comparator Checked Fault Tolerant System, Checkpointed System, Container Managed Security, Trust Partitioning,

---

30 Sebastopol, California: O'Reilly & Associates, Inc., 2003.

31 Patternshare Repository of Security Patterns. Accessed 17 December 2007 at: https://netfiles.uiuc.edu/mhafiz/www/ResearchandPublications/Patternshare_Security_Patterns.htm. The Patternshare repository is no longer maintained by Microsoft. However, Munawar Hafiz, a researcher at University of Illinois at Urbana-Champaign has continued to maintain the repository, so that developers can still benefit from its contents. Hafiz also extracted what he felt were the most important patterns in the repository into a conference tutorial: Hafiz, Munawar. "Security Patterns and Secure Software Architecture." Tutorial presented at ACM Special Interest Group on Programming Languages International Conference on Object-Oriented Programming, Systems, Languages, and Applications, Portland, Oregon, 22-26 October 2006. Accessed 23 January 2008 at: https://netfiles.uiuc.edu/mhafiz/www/ResearchandPublications/Security%20Patterns%20Talk.ppt; this tutorial includes the 30 security patterns listed here.

Controlled Virtual Address Space, Controlled Process Creator, Controlled Object Creator, Defense in Depth, Denial of Service (DoS) Safety, Dynamic Service Management, Exception Shielding, Execution Domain, *chroot* Jail, Hidden Implementation, Minefield. Intercepting Validator, Trusted proxy, Low-Hanging Fruit, Replicated System, Standby, Safe Data Structure, Secure Pre-forking, Security Context, Server Sandbox, Single Access Point, Single Threaded Façade, Small Processes, Unique Location for Each Write Request, White Hats Hack Thyself.

IBM has developed a rich set of design patterns for WebSphere-based Web service applications. Among these are a set of Non-Functional Requirements High Availability Runtime patterns[32] that are somewhat relevant to software resilience concerns; these include: Single load balancer, Load balancer hot standby, Mutual high availability, Wide area load balancing, and Caching proxies with security plug-in.

Other software assurance-relevant security patterns can be found in the books listed among the Suggested Resources below.

SUGGESTED RESOURCES

- Schumacher, Markus, Eduardo Fernandez-Buglioni, Duane Hybertson, Frank Buschmann, and Peter Sommerlad. *Security Patterns: Integrating Security and Systems Engineering.* New York, New York: John Wiley & Sons; 2005.
- Blakley, Bob and Craig Heath, Craig. *Technical Guide to Security Design Patterns.* San Francisco, California: The Open Group, 2004.
- SecurityPatterns.org Website. Accessed 19 December 2007 at: http://www.securitypatterns.org/index.html
- Steel, Chris, Ramesh Nagappan, and Ray Lai. *Core Security Patterns.* Indianapolis, Indiana: Prentice-Hall Professional, 2005.
- Viega, John and Matt Messier. *Secure Programming Cookbook for C and C++.* Sebastopol, California: O'Reilly, 2003.
- Kienzle, Darrell M., Matthew C. Elder, David Tyree, and James Edwards-Hewitt. *Security Patterns Repository,* Version 1.0 —*and*— "Security Patterns for Web Application Development: Final Technical Report", 4 November 2003. Accessed 21 January 2008 at: http://www.modsecurity.org/archive/securitypatterns/ - *and* - http://www.scrypt.net/~celer/securitypatterns/

32 For more information, see: http://www.ibm.com/developerworks/patterns/edge/at1-runtime.html

- Halkidis, Spyros T., Alexander Chatzigeorgiou, and George Stephanides. "A practical evaluation of security patterns". *Proceedings of the Sixth International Conference on Artificial Intelligence and Digital Communications,* Thessaloniki, Greece, 18-20 August 2006. Accessed 21 January 2008: http://www.inf.ucv.ro/~aidc/proceedings/2006/5%20shalkidis.pdf

- Mouratidis, Haralambos, Michael Weiss, and Paolo Giorgini. "Modelling Secure Systems Using an Agent-Oriented Approach and Security Patterns". *International Journal of Software Engineering and Knowledge Engineering,* Volume 16 Number 3, 2006, pages 471-498. Accessed 21 January 2008 at: http://www.scs.carleton.ca/~weiss/papers/ijseke06.pdf

5.5 EXECUTION ENVIRONMENT SECURITY CONSTRAINTS, PROTECTIONS, AND SERVICES FOR SOFTWARE

Environment-level components are often relied on to provide security constraints and protections, and related services, for the software hosted in the environment. These usually include cryptographic services to enable code signature validation, virtual machine constraints/sandboxing, input validation filtering, *etc.*

Because environment-level components are often implemented as software, they are subject to the same security issues as the software hosted in the environment. When such environment-level measures are used, the software itself should be designed and implemented so that a failure in any of the environment components will not threaten the software. All software needs to include:

- Input validation logic that enables the software itself to recognize and reject all malicious and unexpected (anomalous) input, in case environment-level blocking of such input fails;

- Error and exception handling logic that provides the software with a high level of fault tolerance, and ensures that when the software cannot avoid failing, the failure never leaves the software, its data or resources in an insecure or vulnerable state.

The types and strength of security services and protections software may need from its execution environment is based on three factors:

- The type, characteristics, and purpose of the software to be protected;

- The types and strength of security protections and services available in the execution environment;

- The anticipated threats to the software in its expected operational context(s).

For example, embedded software in a military weapons system has a different purpose and set of characteristics, different available environment protections and services, and a different set of anticipated threats than embedded software in a game console. Software that implements an

operating system will differ in purpose and characteristics from, and expect a different set of protections and services (exclusively in firmware and hardware) from its environment than, a distributed Web application (for which the operating system will be a key provider of such protections and services). While at a high level both the operating system and application may appear to be subject to some of the same threats, it would be more accurate to say that the system, of which both the application and operating system are two components, is subject to those threats; when considered at the attack-pattern level, the differences between the threats to the operating system and the threats to the application become clear.

Software's dependencies on and interfaces to environment-level components and services should not be hard-coded because such hard-coding increases the likelihood that, if an environment component is upgraded or replaced, or the software is moved/ported to another environment, the software will become vulnerable and will no longer operate securely. Developers should implement only standard interfaces to environment components, with these interfaces configurable, modifiable, and substitutable at compile-time or run-time. This flexibility not only will minimize the risks component changes pose to software, but also avoids the need to re-implement the software to accommodate the environment changes.

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 when the tool that executes them provides a user-controllable interface, will allow the developer to explore his/her assumptions about the execution environment, and reveal hitherto unrecognized requirements for software/environment interfaces.

Several organizations, including NIST, the National Security Agency (NSA), and the Defense Information Systems Agency (DISA), have published secure configuration guides and scripts for popular and/or approved COTS products. Other agencies have mandated use of specific vendor or third-party configuration guides. These should be consulted by custom developers and integrators, who should strive to ensure that their 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. This may require adjusting the assumptions under which the developers of COTS and OSS software to be installed on securely-configured platforms had about their software's execution environment, especially when those assumptions conflict with the mandated "locked down" environment. There will be cases in which such conflicts cannot be overcome, and these need to be carefully documented to provide justification for waiving those secure configuration requirements that will prevent the correct operation of the installed COTS/OSS software.

SUGGESTED RESOURCES

- National Defense Industrial Association System Assurance Committee. *Engineering for System Assurance*, Version 0.90, 22 April 2008. Accessed 30 May 2008 at: http://www.acq.osd.mil/sse/ssa/docs/SA+guidebook+v905-22Apr08.pdf
- Dean, J. and L. Li. "Issues in Developing Security Wrapper Technology for COTS Software Products". *Proceedings of the First International Conference on COTS-Based Software Systems*, Orlando, Florida, 4-6 February 2002, pages 76-85. Accessed 19 December 2007 at: http://iit-iti.nrc-cnrc.gc.ca/iit-publications-iti/docs/NRC-44924.pdf
- Johansson, Jesper M. and E. Eugene Schultz. "Dealing with contextual vulnerabilities in code: distinguishing between solutions and pseudosolutions". *Computers and Security*, Volume 22 Number 2, 2003, pages 152-159.
- Fiskiran, A. Murat and Ruby B. Lee. "Runtime execution monitoring (REM) to detect and prevent malicious code execution". *Proceedings of the 22nd IEEE International Conference on Computer Design*, San Jose, California, 11-13 October 2004, pages 452-457. Accessed 21 January 2008 at: http://palms.ee.princeton.edu/PALMSopen/fiskiran04runtime.pdf

5.5.1 Environment-level compartmentalization: constraint and isolation mechanisms

The environment-level mechanisms that can be used to constrain and protect software execution range from standard operating access control mechanisms to hardware trusted processor modules.

5.5.1.1 Standard operating system access controls

On Unix or Linux systems, the *chroot* "jail" feature of the standard operating system access controls can be configured to create an isolated execution area for software, thus serving the same purpose as a Java or Perl "sandbox".

5.5.1.2 Trusted operating systems

A trusted operating system includes a file system that enforces a confidentiality-based mandatory access control policy (or policies). In most trusted operating systems, including Security-Enhanced Linux and Trusted Solaris, this confidentiality-based mandatory access control (MAC) policy conforms to the Bell-LaPadula hierarchical confidentiality model. Because its concern is with protecting information from disclosure, Bell-LaPadula-based MAC provides no real benefit in terms of protection software, where the main threat is to the integrity of executable programs and control files, *i.e.,* the need to prevent tampering or corruption. Therefore, trusted operating systems based solely on hierarchical confidentiality are of little value and an unnecessary expense if the only concern is software rather than data protection.

At least one trusted operating system—BAE Systems' Secure Trusted Operating Program—enforces an integrity-based MAC policy conformant with the K.J. Biba hierarchical integrity

model. Unlike a mandatory confidentiality policy, a mandatory integrity policy is particularly useful for preventing active entities within a system (*i.e.,* users, software processes) from modifying or deleting passive entities (*i.e.,* files, devices, including software executable images, control files, and configuration files). In practical terms, this means:

1. The program's executable image can be set at a higher integrity level than the privileges assigned to the users, thus preventing them from modifying or deleting that executable;

2. The program's control files can be set at a higher integrity level than the executing program's privileges, thus enabling the program to read those files but not to modify or delete them.

3. Write/delete privileges for high-integrity files can be restricted only to the administrator.

One non-hierarchical MAC model is Domain and Type Enforcement (DTE). In DTE, active entities are assigned "*domain* attributes" while passive entities are assigned "*type* attributes". A table is then used to list rows of all domains and columns of all types supported by the access control system. At each intersection of domain and type, there are indicators of all access modes (*e.g.,* read, write, execute, traverse) that active entities in the domain are allowed to perform on passive entities of the intersecting type.

DTE provides the basis for Role-Based Access Control, Attribute-Based Access Control, and Risk Adaptive Access Control; it is also useful for implementing compartmentalization. Compartmentalization is a form of isolation in which interactions between passive and active entities are mediated not based on non-hierarchical considerations rather than hierarchical levels of access *vs.* privilege. Such non-hierarchical considerations include an active subject's need-to-read, need-to-write, or need-to-execute a given passive object.

5.5.1.3 Security-enhanced operating systems

Operating system security-enhancements range from add-on security functions, such as mobile code signature validation, input/output filtering, and disk encryption, to middleware such as pluggable authentication modules, to a full mandatory access control capability including reference monitor. In some cases, security features are either built in to a "secure version" of a common operating system by its vendor or a third party, or packaged as a third-party "add on" product.

5.5.1.4 Hardened and security-enhanced operating systems

A number of vendors offer secured or "locked down" versions of operating systems. Many of these are application-specific, *i.e.,* they are used for hosting specific types of high-consequence applications, such as firewalls, intrusion detection systems, virtual private network servers, *etc.* The operating systems have been stripped of all features (services and resources) that are not expressly used by the application, and are pre-configured with the most restrictive access control and networking settings possible, thus minimizing the "attack surface" of the resulting

operating system. The intent of these operating systems is similar to that of the minimized kernels described below.

A few vendors offer secure operating system enhancement packages, such as Argus' PitBull and Hewlett-Packard's VirtualVault, that integrate full-blown DTE or mandatory access controls to commodity operating systems.

5.5.1.5 Minimized kernels and microkernels

Minimized kernels and microkernels are 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".

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 virtual machine (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.

There are some secure microkernels emerging that are intended to be system-independent and thus can be more widely adopted. The now-obsolete Trusted Mach operating system was among the first to implement a secure microkernel. Examples of secure microkernel

implementations include the Multiple Independent Layers of Security (MILS) Partitioning Kernel[33] (developed by the NSA and the U.S. Air Force Research Laboratory), the Flux Advanced Security Kernel (sponsored by NSA and embedded in SE Linux), Safe Language Kernels, such as J-Kernel (developed by Cornell University and sponsored by the Defense Advance Research Projects Agency), as well as a number of other security-focused microkernels developed in the U.S. and abroad.[34]

5.5.1.6 Virtual machines

Isolation is the aspect of VMs most frequently cited as improving the reliability and security of software running within those VMs. Isolation means that software within the VM obtains the specific resources (memory, hard drive space, virtual network interface, *etc.*) it needs to operate—the VM directly accesses these resources on the software's behalf, in essence acting as the software's host while isolating it from the actual underlying platform (operating system and/or hardware, depending on the type of VM), while preventing the software from affecting any programs and resources that reside outside the VM. Other VM features that aid in improving dependability, trustworthiness, and/or resilience are load balancing, support for image restoration, and introspection. VM implementations range from virtual application programmatic interface (API) layers (such as those provided by the Java Virtual Machine [JVM] and the .NET CLR) to operating system virtualization, to full system (hardware and software) virtualization.

VMs, especially those that implement full system or operating system virtualization, are large an complex, and unsurprisingly have been revealed to include their vulnerabilities that enables their isolation feature to be bypassed. It has been suggested that use of "ultrathin" (*i.e.*, very small functionality-limited) VMs can increase the VM's robustness and reduce the likelihood of VM vulnerabilities, in the same way that smallness and simplicity reduces likelihood of vulnerabilities in other software.

---

33 MILS-compliant microkernel implementations are found in the DO-178B/Aeronautical Radio Incorporated (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 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 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. 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 Common Criteria Evaluation Assurance Level 4.

34 Examples include the L4 Secure Microkernel (seL4) for use with the Embedded Real Time Operating System; VFiasco, a formally-verified version of the Fiasco µ-kernel; µSINA, used in the Nizza security architecture; kaneton; and Coyotos, a refinement of EROS.

5.5.1.7 Trusted processor modules

Similar in intent to virtual machines (VMs) and sandboxes, trusted processor modules (TPMs) use hardware to enforce VM-like isolation of processes, in terms of their interactions with other processes and their access to data and resources. In addition to the commercial TPMs now available, the DoD's Anti-Tamper/Software Protection Initiative is working to produce trustworthy, tamperproof hardware modules for hosting high-consequence software.

5.5.1.8 Tamper-resistant processors

Often use hardware-based cryptographic solutions, such as copy-protection dongles, tamper-resistant processors provide software license enforcement and intellectual property protection by deterring reverse-engineering and illegal copying of software hosted on those processors. Tamper-resistant processors also deter unauthorized modification of that software, and thus provide a lower-assurance alternative to TPMs for protecting the integrity of hosted software.

SUGGESTED RESOURCES

- Li, Ninghui, Ziqing Mao, and Hong Chen. "Usable Mandatory Integrity Protection for Operating Systems". *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, Oakland, California, 20-23 May 2007. Accessed 20 March 2008 at: http://www.cs.purdue.edu/homes/ninghui/papers/umip_oakland07.pdf

- Walker, Kenneth M., Daniel F. Sterne, M. Lee Badger, Michael J. Petkac, David L. Shermann, and Karen A. Oostendorp. "Confining Root Programs with Domain and Type Enforcement (DTE)". *Proceedings of the 6th USENIX Unix Security Symposium*, San Jose, California, July 1996. Accessed 20 March 2008 at: http://www.usenix.org/publications/library/proceedings/sec96/full_papers/walker/walker.ps

5.5.2 Application frameworks

Application frameworks are increasingly being used to provide application software with a standard set of interfaces to middleware-level and environment-level services. In addition, application frameworks also provide middleware-like services, software libraries, prepackaged applications, and other resources to help the application developer. The services provided by the framework constitute standard implementations of functions (including security functions) that can be invoked by applications implemented within the framework. The security services (encapsulated security functions with APIs) application frameworks commonly provide include encryption/decryption, user authentication and authorization, hashing and code-signing. By providing standard services, the framework eliminates the need for the developer to write those functions from scratch.

The security functions provided by application frameworks that directly contribute to the security of the application software hosted within the framework include code signing and code signature validation, and the likelihood that applications whose components have been

integrated *via* the framework will contain fewer vulnerabilities and weaknesses originating in the inter-component interfaces and interactions.

Application frameworks may be low-level or high-level. Low-level frameworks are those included in application platforms such as Java and .NET. They usually provide a pluggable mechanism framework that supports a common front-end API for applications to use to request a variety of security functionality, and a backend service provider interface that can access multiple different security mechanisms and interfaces, such as pluggable authentication modules, Java Authentication and Authorization Service, Generic Security Service Application Programming Interface, Integrated Windows Authentication, Microsoft Security Support Provider Interface, IBM System Authorization Facility, *etc.*

Low-level frameworks are both flexible and complex, making them difficult for non-experts to use effectively. They are most useful for developing applications for which there is not a widely available set of "pre-packaged" standard APIs, protocols, and technologies, *i.e.,* traditional client/server applications based on distributed object technologies such as Common Object Request Broker Architecture, Microsoft .NET Remoting or Distributed Component Object Model, or Java Remote Method Invocation. Low-level frameworks are also useful for extending or customizing the capabilities of higher-level security frameworks.

High-level application frameworks, including Web server and Web service frameworks, are provided by application servers and Web servers, such as Microsoft's Internet Information Services and Active Server Pages for .NET (ASP.NET), Java EE servers, and Java servlet engines (*e.g.,* Apache Tomcat). The main functions provided by such frameworks are configurable authentication, authorization, and access controls (*e.g.,* Java containers) for the applications they host, as well as logging for a rudimentary audit capability. Web service provider frameworks also provide Web-service unique infrastructure components such as registries, as well as Web service management capabilities.

In addition to the two popular COTS application frameworks (Java EE and .NET, both of which use Java as a "managed code architecture" to control the behavior of client and server application code running within the framework), there are commercial frameworks such as Oracle SOA Suite as well as several popular OSS application frameworks, including Eclipse, Ruby-on-Rails (a framework for applications developed in the Ruby language), Jakarta Struts, Spring, and Hibernate.

When an application framework such as .NET, Java EE, Eclipse, *etc.,* does not provide all of the services needed to satisfy a hosted application's security requirements, the developer will need to programmatically extend or override the framework's built-in capabilities. Each application framework supports programmatic security extensions *via* its own programming model.

From a software assurance viewpoint, reliance on application frameworks to provide high-consequence services, such as security services, is a two-edged sword. Few COTS and OSS frameworks have been developed in accordance with secure design, code, testing, and configuration management principles and practices, although some have undergone (or are undergoing) Common Criteria evaluations. In the absence of assurance that such security

considerations guided the framework's own development, the developer of applications that will run in the framework has no independently established basis for assurance that the security functions and other high-consequence services on which his/her application will rely are dependable, trustworthy, or survivable.

For applications that need a medium to high level of software assurance, it may make more sense to implement a custom framework to provide high-consequence services, and to rely on a COTS or OSS framework only for services for which a low-level of software assurance is acceptable. In this way, the custom framework may be implemented as an extension to or overlay on top of a COTS or OSS framework, and the services it provides can, in fact, include services expressly designed to mitigate the security risks posed by the COTS/OSS framework. For example, the custom framework could include logic to rigorously validate all parameters sent to and all return values received from the COTS/OSS framework APIs.

SUGGESTED RESOURCES

- Niski, Joe. "Application Security Frameworks". Burton Group Application Platform Strategies Reference Architecture Technical Position Paper, January 2008. Accessed 2 February 2008 at: http://www.burtongroup.com/Research/PublicDocument.aspx?cid=21

5.5.3 Benign software on a malicious host

The possibility that the host on which software will operate may be malicious presents a particular concern since such software is made vulnerable, by the host, to a variety of attacks, including:

- Reverse engineering attacks;
- Fake library attacks, which attempt to fool software into believing it's communicating with valid library functions when it is actually communicating with malicious functions that emulate valid functions;
- Memory tampering attacks, which attempt to alter memory content of software protection mechanisms to circumvent their protection;
- Kernel-level emulator attacks, which record and replay all communications between software and its environment-level protections. (Note that Microsoft claims that its current operating systems provide security measures that make this type of attack infeasible or impossible.)

The need to minimize the risk to software from a malicious host highlights the importance of verifying the trustworthiness of the host(s) on which the software is expected to be installed. More importantly, it highlights the need to design and implement the software to have as few dependencies on and interactions as possible with the untrusted components of its execution environment.

5.6 SECURE ARCHITECTURE AND DESIGN METHODOLOGIES

Three of the security-enhanced requirements engineering methodologies listed in Table 4-2—TRIAD, AEGIS, and AOM—also support secure software architecture and design modeling. In addition, MDA can be adapted to support the concepts, and framework and tool augmentations needed to add constructs for generating executable secure software models.

SUGGESTED RESOURCES

- Fléchais, Ivan. *Designing Secure and Usable Systems.* University of London doctoral thesis, February 2005. Accessed 19 December 2007 at: http://www.softeng.ox.ac.uk/personal/Ivan.Flechais/downloads/thesis.pdf

6 SECURE COMPONENT-BASED SOFTWARE ENGINEERING

Under pressure from their customers to produce software more quickly, software engineers frequently turn to existing (COTS, OSS, reusable legacy) components about whose security properties and assurance they have little or no knowledge. This lack of knowledge makes it difficult to determine what security assumptions those components harbor about the environment in which they will operate and the other components with which they will interact.

Component-based software engineering also makes it difficult to validate the requirements, including security requirements, for the whole software system in which the COTS/OSS components will be used as it will be difficult to determine whether those components do, in fact, satisfy the software assurance-related requirements for those individual components (given their role in the system) and for the system as a whole, such as requirements for a given component to exhibit only safe behaviors and constrain all unsafe behaviors.

Once there has been an investment in and familiarity gained about a given component, there will be a strong temptation to reuse that component across multiple systems operating in a variety of environments. In each system, the same component may play a different role. Its behaviors and state changes—both those externally observable and those not—are unlikely to be equally satisfactory in all of these systems and environments. The component that proves adequately secure when combined with the other components of System A in System A's environment, but may prove *in*adequate when combined with the different components of System B in System B's different environment.

The COTS and OSS components most often used in software-intensive systems include:

- **Environment components,** such as: firmware, device drivers and Basic Input/Output System (BIOS), operating systems, and networking and network security (*e.g.,* Internet Protocol Version 6) software;

- **Middleware components,** such as: software libraries, application frameworks, expert systems and other artificial intelligence "engines", financial analysis kits, database management systems, Web and application servers, grid computing services, and security components such as public key infrastructure components, single sign-on services, pluggable authentication modules, and Kerberos components;

- **Application-level components,** such as: directory services; email/messaging services; Web service discovery services; spreadsheet packages; browsers.

In addition, on most software development projects, many if not all of the tools used by the developers, including specification and design tools, coding and compilation tools, integration/assembly frameworks, testing tools, code signing tools, *etc.*, are COTS or OSS.

The security properties, assumptions, and vulnerabilities and weaknesses of these components all need to be considered when (1) determining which specific product or implementation to use; (2) architecting, designing, and assembling the system to minimize the exposure of existing vulnerabilities and the introduction of new vulnerabilities as a result of mismatched assumptions, inadequate constraints on behaviors, *etc.,* as well as implementing any custom wrappers, filtering interfaces, *etc.,* to mitigate security issues introduced by COTS/OSS components.

Note that individual component security evaluations focus on the component in isolation.[35] Examining a component in isolation will not reveal the security assumption and interaction conflicts that will arise when the component is assembled/integrated together with other components. Such conflicts, often referred to as security mismatches, usually originate in an inconsistency in the security assumptions one component has about another's security properties, functionality, policy rules, constraints, *etc.*, and those that the second component actually exhibits. The problem is complicated by the need to periodically add or change the functionality of individual components or the system as a whole, often necessitating changes to the assembly's design, as well as to the individual components.

Even if the security of all of the system's individual components could be established, this would not be sufficient to predict whether their secure behavior would continue to be exhibited when they interacted with other components in the larger component-based system. Nor will it help predict the overall security of the system assembled from those components.

The security claim for a single component (*e.g.,* in an assurance case) in a system assembled from multiple components is of little help in determining the assurance of the whole system. For this reason, the component security evaluations should not be performed in isolation. An assembly architecture framework should be used to model, test, and evaluate the security of behaviors among pairs and larger combinations of interacting components. The outcomes of these pairings/combinations should help narrow down the acceptable combinations components and thereby point to the set of components that collectively operate the most securely. It is this set of components that should, if possible (given other considerations) be selected for use in the system.

This said, the component evaluator needs to recognize that changes in future versions/releases of the components may change the nature of their interactions/ behaviors when combined with the system's other components. Any component evaluation reflects a single point-in-time understanding of the components under consideration. For this reason, some trends analysis of how secure previous versions of the components have been, and whether there has been a general trend towards more rather than less security may also provide a helpful data point in determining which components are most likely to continue as secure through future updates.

35 This is true of Common Criteria evaluations which, moreover, focus almost exclusively on the correctness of the component's security functionality rather than its dependability, trustworthiness, and resilience.

SUGGESTED RESOURCES

- Haley, Charles B., Robin C. Laney, Jonathan D. Moffett, and Bashar Nuseibeh. "Using Trust Assumptions with Security Requirements". *Requirements Engineering Journal*, Volume 11 Number 2, April 2006, pages 138-151. Accessed 11 September 2008 at: http://www.the-haleys.com/chaley/papers/REJ06.pdf

6.1 ARCHITECTURE AND DESIGN CONSIDERATIONS FOR COMPONENT-BASED SOFTWARE SYSTEMS

Because there is likely to be more than one component that can satisfy the purely functional requirements of the software system, modeling of the software architecture should focus equally on developing alternative models to accommodate the evaluation of functionality in "competing" candidate components, and determination of which component combinations will result in the most secure collective behavior and the least number and exposure of vulnerabilities.

A component assembly modeling framework can be extremely helpful in generating different architecture models (or "assembly options") that can then be used to guide integration when evaluating candidate components, as well as when building the system from the components that are ultimately selected.

The architecture/framework in which the components will be integrated/assembled should minimize the exposure of each component's vulnerabilities, and constrain the potential for a component's insecure behavior, state change, or failure to negatively impact other components in the system. The least trustworthy, most vulnerable components should be located in the architecture in the least exposed positions in terms of access to and by entities external to the system, and other untrustworthy components within the system. They should never be trusted to perform high-consequence functions.

Component-based development influences architecture and design in the following ways.

- The different component architecture models must be based on both explicit and implicit assumptions about how each component will interact with the other components (*i.e.,* will the component play a service-providing or protective role, or a dependent role?).

- The suppliers of COTS components (as well as those of other binary components, *e.g.,* GOTS, shareware, freeware [non-open source], and often legacy) virtually always retain the intellectual property rights to their components' source code. Most suppliers of binary components intend for those components to be used without modification. A binary component is, therefore, a "black box" whose functions, interfaces, and constraints can only be changed through reconfiguration *via* an internal interface, and then only to the extent supported by the component. Otherwise, external means such as filters, application firewalls, XML gateways, and

security wrappers must be used to counteract any unacceptable security behaviors and to mask as many vulnerabilities as possible in the component itself.

- When a particular component is selected, the ongoing security posture of the component, and thus of the whole software system depends, at least in part, on the priorities of the component's supplier in terms of how discovered vulnerabilities are addressed, how often patches are released, the security impact of changes in new versions, *etc.* 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. For this reason, the system design needs to be able to easily accommodate:

    1. Replacement of components with new versions or with substitute components from different suppliers;

    2. Reconfigurations of components;

    3. Insertion of countermeasures (such as wrappers) to mitigate security vulnerabilities discovered after the system has been integrated (and not yet patched), including those resulting from new versions or substitutions.

The ideal component-based architecture and design should be as generic as possible. It should reflect the *roles* of the components but not the specific implementation details of any specific COTS or OSS package. This means, of course, relying to the absolute greatest extent possible on standard interfaces and avoiding proprietary interfaces.

SUGGESTED RESOURCES

- BuildSecurityIn Assembly, Integration, and Evolution resources. Accessed 21 January 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/assembly.html

- *Software Security Assurance*, Sections 5.1.1.1-5.1.1.2.2.

- Dobson, John E. and Brian Randell. "Building Reliable Secure Computing Systems out of Unreliable Insecure Components". *Proceedings of the 17th Annual Computer Security Applications Conference (ACSAC)*, New Orleans, Louisiana, 10-14 December 2001. Accessed 19 December 2007 at: http://www.cs.ncl.ac.uk/research/pubs/inproceedings/papers/355.pdf

- Jeong, Gu-Beom and Guk-Boh Kim. "A Framework for Security Assurance in Component Based Development". *Proceedings of Workshop on Approaches or Methods of Security Engineering*, Singapore, China, 9-12 May 2005, pages 42-51.

- Neumann, Peter G.. "Principled assuredly trustworthy composable architectures". CDRL A001 Final Report, 28 December 2004. Accessed 21 January 2008 at: http://www.csl.sri.com/users/neumann/chats4.html

- Carnegie Mellon University Software Engineering Institute. Predictable Assembly from Certifiable Components Webpage. Accessed 21 January 2008 at: http://www.sei.cmu.edu/pacc/
- Minkiewicz, Arlene F. "Security in a COTS-Based Software System". *CrossTalk: The Journal of Defense Software Engineering*, November 2005. Accessed 17 December 2007 at: http://www.stsc.hill.af.mil/crosstalk/2005/11/0511Minkiewicz.html
- Jarzombek, Joe and Karen Mercedes Goertzel. "Security Considerations in the Use of Open Source Software". Presented at the Technology Training Corporation Military Open Source Software Conference, Washington, D.C., 21 April 2008.

6.2. SECURITY ISSUES ASSOCIATED WITH COTS AND OSS COMPONENTS

Properties and capabilities delivered in COTS/OSS components do not always map directly to the requirements of the system in which the components will be used. Many commercial developers admit that security is not a major requirement of most of their customers, so it is not a major consideration in the development of the commercial software product. Specific security issues commonly found in COTS and OSS components are discussed below.

6.2.1 Lack of visibility (the "black box" problem)

COTS and other reused binary components (*i.e.,* GOTS, shareware, freeware, reusable legacy) present a challenge because the lack of source code and detailed design specification means the analyst cannot determine what the internal behaviors and state changes actually are, except in so far as those behaviors/state changes are externally observable. In short, the only practical analyses possible for most binary components are those that treat the components as "black boxes", with observations limited to how the component interacts with external entities (humans, execution environment, other software, *etc.*). What goes on within the component is unknowable.

Even middleware components and development tools, such as software libraries, that one would expect to be documented fully and in detail to aid the developers who are their intended users, often have only their interfaces well-documented. This is because the vendors of these components and tools are motivated by the imperative to protect their intellectual property (*e.g.,* algorithms, data, parameters), and thus are reluctant to reveal how the internals of those products actually work. Similarly, hardware-level components often contain a great deal of information about the underlying hardware and architecture design. This is information that device/platform developers often wish to hide from software developers and system users, to deter competitors and hackers from studying and reverse engineering their products.

To a great extent, the analysis and testing of black box components is an exercise in discovery of component's (one hopes *strictly* defined) inputs and outputs, assumptions (*i.e.,* about the services, protections, *etc.* the component expects to obtain from external components at various levels), and pedigree/provenance (indicating something about the trustworthiness of the

component's developers and the likelihood that they used a disciplined engineering process to produce the component).

In-depth security testing will require that each individual component be tested as thoroughly as possible, not just for its interactions and externally observable behaviors under expected operating conditions, but for its interactions/behaviors under unexpected and even seemingly impossible conditions. The expected conditions will be those manifested when the component is combined, as intended, with the other components of the system in which it will be used. The unexpected conditions will be those that arise when any of those other components behaves anomalously or maliciously, or fails to operate at all, leaving the component being tested exposed to direct interactions with entities it was not intended to interact with.

To increase the inherent "knowability" of components, and particularly binary components, use of open standards for the component interfaces should be a component selection criterion that cannot be waived. The ability to study the standards documents to understand the details of the standards-based interface (including known vulnerabilities and available countermeasures for those vulnerabilities) can compensate, to some extent, for the lack of detailed specifications of how the interface was implemented in the specific component under review.

Using open standards also enables a more "apples to apples" comparison of how multiple candidate components operate: they are all expected to communicate externally in the same way, so the comparison can focus on deviations and revealed vulnerabilities in those interfaces, rather than on comparing two different types of interfaces used for the same purpose. Once a component has been integrated into the software system, if it later proves inadequate, its open standards-based interfaces will make it easier to replace with a more satisfactory component that supports the same standards-based interface.

6.2.2 Ignorance of pedigree and provenance

Pedigree of software refers to information that identifies the software's:

- **Development practices:** How was the software built? What methodologies, practices, tools were used?
- **Developers:** Did the development process include checks and controls to establish trustworthiness of developers? Is this possible on large, distributed community development (OSS) projects in which many developers use aliases?
- **Requirements:** Is the specification available? If so, were security requirements included? Are those requirements relevant to the security goals of the system in which the component will be used?
- **Review and testing regime:** Unless there is explicit evidence to the contrary, it should be assumed that security was not considered during a component's reviews and tests.

Provenance of software refers to information that identifies the influences on the software after its initial creation. These exposures include:

- **Distribution practices** such as bytecode obfuscation, code signing, or any other intentional or unintentional code modifications that occur during distribution;
- **Installation practices and runtime configuration** of the software and its execution environment;
- **Operational exposures,** including changes to its execution environment, user interactions, and interactions by other software (malicious and non-malicious);
- **Intentional and unintentional runtime modifications,** *e.g.,* through runtime interpretation, dynamic linking, installation of patches/updates, malicious code insertion, *etc.*

Reliable knowledge about pedigree and provenance allows the evaluator of COTS and OSS components to makes some fairly reliable assumptions about how the component was built and maintained, and by whom; the more is known about how the component was created and the influences to which it has been exposed since its creation, the more reliable the assumptions.

There are difficulties in discovering reliable pedigree/provenance information, however. COTS components are often the product of multiple development teams, some of which are contractors to the supplier-of-record; these teams may be located offshore. OSS components are often the product of collaboration by a "community" of developers in dispersed geographic locations, often working under aliases. While some well-known OSS communities, such as the Linux and Apache communities, follow extremely disciplined SDLC and version control practices, many other smaller OSS projects do not subject themselves to the same rigor.

In both COTS and OSS development, unless the development process includes strict checks and controls that establish developer trustworthiness based not just on an initial background check, but based on observation of the developer's behavior and performance over a relatively long period, the user of the COTS/OSS component will be at the mercy of the developers' unknown and undiscoverable (and even non-inferable) national and political affiliations, ideological tendencies, malicious proclivities (and skills to act upon them in their code through surreptitious, virtually undetectable sabotage *via* embedded malicious logic).

The tools and services provided by Palamida and Black Duck Software represent the current state-of-the-art in pedigree and provenance discovery. These tools able to discover pedigree "hallmarks" in source code in repositories, COTS products, and large software systems. Initially, their focus was license enforcement, but Palamida now offers service that also attempt to verify security based on pedigree-related evidence. These service providers only make their discovered pedigree data available to paying subscribers. A less sophisticated approach that is widely available is the use of open source pedigree discovery tools for finding

common OSS code segments in large source trees, *e.g., comparator* and *filterator*.[36] These tools are limited to flagging OSS content; they provide no helpful metadata about that content.

Given the limitations on the types and reliability of pedigree/provenance information is most likely to be discoverable, the component evaluator's best use of such information is as input to the decision on whether and how to proceed with the security evaluation of the component. For example, if there is no pedigree information available at all, this lack may be used to justify rejection of the component, particularly if it was being considered for implementing a trusted or high-consequence function, and/or as a component in a national security system in which 100% U.S. (or other nationality)-originated content is required. At a minimum, inadequate pedigree and provenance information should trigger the following:

- Deeper security analysis of the component than would have been required had reliable evidence of its secure development process been found;
- Use of environment-level controls to isolate the component from other, more trusted components and to constrain the component's execution to minimize potential damage from its non-secure behavior or compromise.

6.2.3 Questionable validity of security assumptions

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.

Each software component has its own set of security assumptions and requirements regarding the other components and external entities with which the component expects to interact, and the environment services and protections it expects to receive. Lack of visibility into COTS and other black box software makes it particularly difficult to discover what its security assumptions and requirements are, and whether or not they are consistent or conflict with the security assumptions of the "generic component" in the system architecture whose role they are intended to fulfill in the software system.

6.2.4 Presence of unused and unexpected functions: dormant, dead, and malicious code

Lack of visibility into the source code of COTS and other acquired-as-binary components, and lack of time and resources to thoroughly analyze the source code of OSS and other source-

36 Available at: http://catb.org/~esr/comparator/

code-available components, means that it is not possible for the developer to know exactly what the component's code base consists of—not just in terms of the nature of the code that implements the functions for which that component is being used in the software system, but in terms of code for functions not used, and code that simply remains in the component's code base because its removal was considered unimportant or risky. There are three kinds of "unexpected" code possible in COTS/OSS components:

- **Dormant code:** This is the code that implements the features that will not be used when those components are assembled into a the software system being built. These features and their interfaces are fully functional; they simply will not be invoked in the normal course of executing the component within the software system. However, they can easily be invoked unintentionally, with unexpected and potentially dangerous results. Also, because the dormant features are not used in the software system, they are unlikely to undergo thorough analysis and testing unless such analyses/tests are expressly added to the system's security review and test plans;

- **Dead code:** Because many COTS and OSS software components are the result of multiple evolutionary "builds" over time in which new features are often added to replace older obsolete features. In many cases, the older code is simply "cut" or "blocked" off by eliminating its external and call-level interfaces, rather than actually being removed; this is because the implications of removing old code, particularly code that has been part of the component through many previous version, are not well understood by the developers who build the later versions. It is very likely, therefore, that larger and more complex COTS/OSS components that have a long history of versions (*e.g.,* operating systems, database management systems, popular applications) contain "dead code", *i.e.,* code that has been "blocked off" but not removed from the component's code base. While the elimination of expected interfaces to the dead code reduces the risk that such code will be inadvertently executed, there are still conditions under which such execution might occur, particularly if the code is suspected or known by an attacker to be present, and the attacker therefore crafts and exploit specifically to reach that code and trigger its execution. The problem, as with dormant code, is that the results of such an execution are unpredictable and potentially dangerous. Even more than dormant code, dead code is unlikely to undergo reviews, and crafting tests to trigger its execution to observe what results can be a challenge;

- **Malicious code:** Embedded malicious logic such as logic bombs, time bombs, Trojan horses, and malicious bots, is the third type of unexpected code found in COTS, OSS, and other acquired and reused components. Unlike dormant and dead code, malicious code is intentionally built to cause the component to behave non-securely and to compromise or damage the system in which the component resides, the environment in which it operates, and/or the external entities, resources, and data with it interacts. Like dead code, malicious code is not expected to be present, and thus finding it can prove particularly challenging.

Presence of unused code can make analysis of what is used more difficult. Because the code is not expected to be executed, testers may not think about executing the unused portions of the program to ensure they cannot be used to compromise the rest.

Secure assembly/integration requires the maximum possible isolation and constraint of code portions/functions not used in the software system. Isolation and constraint should focus on allowing access only to functions that are intended to be accessed. In essence, these functions should be isolated from the external access (human or process) to the unused functions, so as to prevent the inadvertent execution of the unused functions, and to limit the impact of any inadvertent executions that cannot be prevented so that those executions do not threaten the secure operation of the system as a whole.

6.3 SECURITY EVALUATION AND SELECTION OF COMPONENTS

The developer's understanding of a component's security properties must not be based wholly or even predominantly on the supplier's claims about that component. Only an objective, thorough, and detailed security evaluation of the components that are being considered for use in the software system can provide reliable data on the security assumptions of the component, and the presence of insecure behaviors and state changes, and of vulnerabilities and weaknesses in it.

Much of the selection process for acquired and reused components should focus on determining and mitigating the impact of conflicts between assumptions and requirements. These include conflicts between supplier and integrator assumptions and requirements for how the component will be used and in what environment. They also include conflicts between the security assumptions (*e.g.,* about environment services that will be available) that the architect has made on behalf of the notional components that "populate" his/her architecture, and the security assumptions under which the components selected to fulfill the notional components' roles were developed.

The ability to determine assumptions made by component suppliers in and the components they supply will be particularly difficult in the absence of source code and/or high-quality detailed technical documentation for those components. For this reason, the availability of source code and documentation should be a key criterion in the selection of components that will perform high-consequence functions in the assembled system.

Developers should never assume that a black box (binary) component's calls to external functions (*e.g.,* in execution environment components) will always succeed. Instead, the developer should use a technique such as black box debugging (see Section 8.2.2.5) to observe all call data passed from the component and any external components, and check the return values of all function calls to those external components. This will enable the developer to

pinpoint flaws in those external components that threaten the dependable operation of the software.

All of the considerations discussed in section 6.1 and 6.2 need to be taken into account in the evaluation and selection of components.

SUGGESTED RESOURCES

- Yin, Jian, Chunqiang Tang, Xiaolan Zhang, and Michael McIntosh. "On Estimating the Security Risks of Composite Software Services". *Proceedings of the Program Analysis for Security and Safety Workshop Discussion,* Nantes, France, 4 July 2006. Accessed 23 January 2008 at: http://research.ihost.com/password/papers/Yin.pdf

- Li, Zhenmin, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. "Have things changed now? An empirical study of bug characteristics in modern open source software". *Proceedings of Workshop on Architectural and System Support for Improving Software Dependability,* New York, New York, October 2006, pages 25-33. Accessed 11 September 2008 at: http://opera.cs.uiuc.edu/~lintan2/publications/bugchar_asid06.pdf —*and*— Presentation slides: http://opera.cs.uiuc.edu/~lintan2/publications/bugchar_asid06_slides.pdf

- Balzarotti, Davide, Mattia Monga, and Sabrina Sicari, Università di Catania. "Assessing the Risk of Using Vulnerable Components". *Proceedings of the First Workshop on Quality of Protection,* Milan, Italy, 15 September 2005. Accessed 19 December 2007 at: http://dabalza.net/publications/download/risk-qop05.pdf - *and* - http://homes.dico.unimi.it/~monga/lib/qop.pdf

- Neuhaus, Stephan, Thomas Zimmermann, Christian Holler, and Andreas Zeller. "Predicting vulnerable software components". *Proceedings of the 14th ACM Conference on Computer and Communications Security,* Alexandria, Virginia, 29 October-2 November 2007, pages 529-540. Accessed 11 September 2008 at: http://www.st.cs.uni-sb.de/publications/files/neuhaus-ccs-2007.pdf

- Wilson, David L. *Risk Perception and Trusted Computer Systems: Is Open Source Software Really More Secure than Proprietary Software?* Purdue University master's thesis, CERIAS TR 2004-07, 2004. Accessed 19 December 2007 at: https://www.cerias.purdue.edu/tools_and_resources/bibtex_archive/view_entry.php?bibtex_id=2686

6.3.1 Steps in a component security evaluation

A component's security evaluation should include the following steps:

1. **Availability of components that satisfy requirements:** Determine the availability of COTS or OSS components that provide the needed functionality, and conform (at least to some extent) with the security assumptions made by the developers about and by the generic architecture components whose roles will be fulfilled by the actual COTS/OSS components.

2. **Security assumptions:** Establish the set of security assumptions made by each generic component in the system design; the component security evaluation will compare these assumptions against the set of assumptions that can be derived from the actual components being evaluated to fulfill the generic component's role in the system.

3. **Interface definitions:** Establish the set of inter-component and extra-component interfaces (*e.g.,* RPCs, APIs) that must be provided by each generic component,

along with the security protection requirements for each interface; the interfaces of the actual components can be compared against these interface and interface-security requirements.

4. **Assembly architecture-framework:** Define and implement (or acquire) a component assembly architecture-framework in which to test combinations of candidate components that are intended to fulfill the roles of the generic components in the system architecture.

5. **Design trade-off methodology:** Define or select a methodology, set of metrics by which design trade-offs that must be made in order to accommodate use of the different actual components can be quantified and a measurement of each component's fitness for secure use can be computed;

6. **Evidence on which to base evaluation:** Gather as much evidence as possible of a variety of types and from a variety of sources, to ensure that there is sufficient information on which to base the security evaluation. Direct evidence is preferable to indirect evidence; this said, there are likely questions that cannot be answered by direct evidence alone - particularly when it comes to COTS and legacy components for which no source code or technical (*vs.* user/administrator) documentation is available - and indirect evidence can be helpful in "filling in gaps" in knowledge that cannot be gained through analysis of direct evidence alone.

   Direct evidence includes source code and detailed design specifications - *i.e.,* it is evidence that enables the direct analysis of the content of the software. The security analysis of direct evidence will necessarily be limited by the resources and time available to analyze it; the larger and more complex the code, the less realistic it will be to attempt an analysis of more than a small proportion of the total code base or design spec. But no matter how large/complex, at a minimum, the analysis should include the software's high-consequence functions and external interfaces.

   Indirect evidence includes the following:

   - Software developer, "black hat", and "white hat" Websites, newsgroups, listservs, *etc.*;
   - Vulnerability scan results;
   - Higher level technical documentation (requirements specification, architecture, pre-existing test results), and documentation of standards and technologies used in the component;
   - Independent security incident, error, and vulnerability reports (supplier reports as well as independent reports from Computer Emergency Response Teams [CERTs], Computer Security Incident Response Teams [CSIRTs], IA Vulnerability Alerts [IAVAs], Bugtraq, National Vulnerability Database, *etc.*);

these are of greatest value in supplier assurance-related analysis of the supplier's track records in patch issuance and product vulnerability/security improvement over time;

- Published reviews, case studies, lessons learned, *etc.*, indicating an established track record of secure use of the component by other organizations in software systems/applications operating under environmental conditions and risk factors similar to those for which the component is currently under consideration;

- Reliable pedigree and provenance information.

7. **Component evaluation models:** Evaluate components using the following models of the desired component-based system:

   - *Software security model* describing the acceptable (manageable) distribution of security vulnerabilities in various individual components. This model will be used to compare the vulnerability distribution in the actual components against what is expected in the system-as-designed;

   - *Service composition model* describing the interactions of generic components in the system architecture, and the anticipated behaviors of those components if subjected to malicious and anomalous interactions with external entities. This model will be used to compare the behaviors of actual components (in generic component roles) when they are subjected to the same malicious/anomalous interactions;

   - *Attack exposure model* estimating the amount of knowledge a low-skilled, medium-skilled, and highly skilled attacker is expected to have about vulnerabilities and weaknesses in COTS and OSS components. This model will help determine the probability that a known vulnerability in a given component will be exploited when that component has been assembled into the software system.

8. **Component combinations:** Using the assembly framework, test various combinations of candidate components to determine:

   - Which components exhibit the most consistently secure behaviors and state changes in response to inputs/messages from other components, the environment, and humans;

   - Which components output the most consistently acceptable data;

   - Which combinations of components are the most difficult to force into insecure states/behaviors/failures.

By establishing the combination of components that proves the most secure in terms of state changes, behaviors, and interfaces both the individual components that should be selected will be revealed, and the best architecture for integrating/assembling those components will also emerge.

9. **Architecture adjustment:** The security elements of the system architecture should also be adjusted as an outcome of the component evaluation and selection process. A given set of components assembled in a certain way may still exhibit residual risks that must be further mitigated by the application of additional security constraints, wrappers, filters, isolation mechanisms, *etc.* As part of the component selection process, the cost (in terms of time and resources) needed to implement and/or acquire and configure these additional security measures should be weighed against the cost of using a functionally less acceptable component, or the cost of custom developing a component instead of acquiring or reusing one.

6.3.2 Questions to ask about components under evaluation

Key questions to ask as part of the component evaluation include:

1. How "knowable" and "modifiable" is the component?

2. Does the component's developer appear to have adhered to secure design principles, and followed secure coding practices and standards? (more easily determined for white box components);

3. If the component is a black box:

    a. Is it configurable so that undesired functions can be "turned off" and unused interfaces can be disengaged?

    b. Has it been obfuscated or otherwise protected to prevent reverse engineering? If the component is being considered to perform a high-consequence function, the ability to reverse engineer for code review may be required.

4. If the component is inadequate in terms of exposed vulnerabilities or non-secure behaviors:

    a. Is there is an alternative component that provides the same functionality? This determination naturally requires the security evaluation of the identified alternative component.

    b. Would it be less costly in the long run to custom develop the functionality? Determining this requires a comparative analysis of whole-life cycle costs for custom development *vs.* cost of security measures, constraints, *etc.,* plus the risk that each new version of the inadequate component might be incompatible with the security measures developed for the previous version,

thus requiring additional countermeasure or constraint development later on. For high-consequence functions as well as simple functions, it may well be more cost effective to custom-build.

6.4 IMPLEMENTING SECURE COMPONENT-BASED SOFTWARE

Preparation of components for assembly/integration entails:

1. Tailoring the components to fulfill the requirements of their roles in the component-based system. Security measures implemented to mitigate component-based risks (*e.g.*, input validation filters, isolated execution areas, *etc.*) should be captured as part of the requirements for the system;

2. Modifying component software (especially OSS, if possible and desirable), to mitigate security vulnerabilities and non-secure behaviors that cannot be corrected using wrappers, filters, virtual machine isolation, *etc.;*

3. Designing, coding, and testing of "glue code", wrappers, filters, *etc.*

In addition to wrapping, filtering, and constraining, an additional countermeasure for addressing vulnerabilities and non-secure behaviors in components is software dynamic translation, a tool for instrumenting programs by performing substantial rewriting of their code at runtime. Originally conceived to reduce software's execution time, dynamic translation is increasingly been used to add security policy enforcement to software, to prevent code and command injections, and to otherwise compensate for security shortcomings in existing software. Resources describing security-specific use of software dynamic translation are provided at the end of this section.

The actual assembly/integration process entails integrating and testing the components in combination with other components. The system's security requirements will not be fully validated until all of the components of the system have been established to be working securely in combination. Note that certain combinations of components should be expected to introduce or expose security vulnerabilities that did not appear when the components were examined individually.

SUGGESTED RESOURCES

Those cited at the end of the previous section, plus:

- Minkiewicz, Arlene F. "Security in a COTS-Based Software System". *CrossTalk: The Journal of Defense Software Engineering*, November 2005. Accessed 4 February 2008 at: http://www.stsc.hill.af.mil/crosstalk/2005/11/0511Minkiewicz.html

- Neumann, Peter G.. "Principled assuredly trustworthy composable architectures". CDRL A001 Final Report, 28 December 2004. Accessed 21 January 2008 at: http://www.csl.sri.com/users/neumann/chats4.html

- Neumann, Peter G. and Richard J. Feiertag. "PSOS (Provably Secure Operating System) Revisited". *Proceedings of the 19th Annual Computer Security Applications Conference*, Las Vegas, Nevada, 8-12 December 2003. Accessed 4 February 2008 at: http://www.csl.sri.com/users/neumann/psos03.pdf

- Ellison, Robert J. "Trustworthy Composition: The System is Not Always the Sum of Its Parts". September 2005. Accessed 4 February 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/50.html?branch=1&language=1

- Hu, Wei, Jason Hiser, Dan Williams, Adrian Filipi, Jack W. Davidson, David Evans, John C. Knight, Anh Nguyen-Tuong, and Jonathan Rowanhill. "Secure and Practical Defense Against Code-Injection Attacks Using Software Dynamic Translation". *Proceedings of the 2006 ACM Virtual Execution Environments Conference*, Ottawa, Ontario, Canada, 10-13 June 2006. Accessed 31 December 2007 at: http://dependability.cs.virginia.edu/publications/2006/strata-isr.vee2006.pdf

- Kumar, Naveen and Bruce Childers. Flexible Instrumentation for Software Dynamic Translation. *Proceedings of the Workshop on Exploring the Trace Space for Dynamic Optimization Techniques*, at the ACM International Conference on Supercomputing, New York, New York, June 2003. Accessed 31 December 2007 at: http://www.cs.pitt.edu/coco/papers/traces-kumar.pdf

6.5 SECURE SUSTAINMENT OF COMPONENT-BASED SOFTWARE

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.

Sustainment will include the evaluation to determine which patches, upgrades, *etc.*, can be safely adopted. Note that applying upgrades/patches may also entail modifications to wrappers and filters to accommodate new or changed interfaces. Upgrades may also require recertification and some level of integration and testing (whole system level) to ensure that security constraints will still be satisfied in the system after the upgrades occur.

Vendors' contractual obligations regarding security and quality must also be determined to be satisfied in all new product versions over time. License or maintenance fees should to be paid

to ensure that all updates, upgrades, and ongoing vendor support contracts are also maintained.

Developers/maintainers should anticipate ongoing changes to system requirements, and ongoing need to develop bug fixes and security patches to custom-developed components, as well as new wrappers and other countermeasures to address vulnerabilities in replacement and upgraded acquired components.

Over time, refactoring and/or reengineering of reused legacy components and even of OSS components may be indicated in order to maintain an acceptable security posture of the system as a whole.

SUGGESTED RESOURCES

- BuildSecurityIn Legacy Systems resources. Accessed 21 January 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/legacy.html
- *Software Security Assurance*, Section 5.1.1.4.
- Lieberman, Danny. "Software security assessment of production systems". 2006. Accessed 9 April 2008 at: http://www.software.co.il/content/view/195/41/. Also published by the Control Policy Group as "Practical Software Security Assessment". 2007. Accessed 9 April 2008 at: http://www.controlpolicy.com/practicalsoftwaresecurityassessment
- Kolb, Ronny, Dirk Muthig, Thomas Patzke, and Kazuyuki Yamauchi. "Refactoring a legacy component for reuse in a software product line: a case study: Practice Articles". Presented at the 2005 IEEE International Conference on Software Maintenance. Published in *Journal of Software Maintenance and Evolution: Research and Practice*, Volume 18 Issue 2, March 2006, pages 109-132.
- Laney, Robin C., Janet van der Linden, and Pete Thomas. "Evolving Legacy System Security Concerns Using Aspects". Open University Technical Report Number 2003/13, 11 November 2003. Accessed 9 April 2008 at: http://computing-reports.open.ac.uk/index.php/content/download/82/322/file/2003_13.pdf

7 SECURE CODING

Secure coding is a prerequisite of defensive programming. Defensive programming is intended to produce robustly secure software. It 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. Defensively programmed software:

- Does not rely on any parameters that are not self-generated;
- Assumes that attempts will be made to subvert its behavior, directly, indirectly, or through manipulation of the software to violate a security policy.

This section introduces secure coding principles and practices that will contribute to defensive programming and software security.

NOTE: In addition to the Suggested Resources below, consult the Suggested Resources for Section 7.1.3, and for the individual programming languages discussed in Appendix C:C.4.

SUGGESTED RESOURCES

- BuildSecurityIn Coding Rules resources. Accessed 21 January 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi-rules/home.html
- BuildSecurityIn Coding Practices resources. Accessed 21 January 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/coding.html
- *Software Security Assurance*, Sections 5.1.1.3 and 5.4.
- Carnegie Mellon University Software Engineering Institute/Computer Emergency Response Team Secure Coding Standards Webpage. Accessed 21 January 2008 at: https://www.securecoding.cert.org/
- Microsoft Security Developer Center. Writing Secure Code Webpage. Accessed 12 December 2007 at: http://msdn2.microsoft.com/en-us/security/aa570401.aspx
- Apple Computer. *Secure Coding Guide.* 23 May 2003. Accessed 12 December 2007 at: http://developer.apple.com/documentation/Security/Conceptual/SecureCodingGuide/SecureCodingGuide.pdf

7.1 SECURE CODING PRINCIPLES AND PRACTICES

As they write code, programmers should conform to the following principles.

7.1.1 Keep code small and simple

The smaller and simpler the code base is, the easier it will be to verify the security of the software. Programmers should implement functions in the smallest number of lines of code possible. Only required (by specification) functions should be included in the software.

Software should never contain unnecessary (*i.e.,* not specified) functions. The number of flaws in the code that implements high-consequence functions can be significantly reduced by reducing the size of the source code 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. This 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.

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.

Object inheritance, encapsulation, and polymorphism are additional techniques that can help simplify code.

7.1.2 Use a consistent coding style

A consistent coding style should be used throughout the system's from-scratch/custom code base (*i.e.,* code written rather than obtained commercially or open source), regardless of how many programmers participate in writing that code.

The coding style defines the physical appearance of the code listing with regard to indentation, line spacing, *etc.*, and should emphasize comprehensibility by code reviewers and maintainers who were not involved in writing the code. All programmers on the project should follow the same coding style guide.

Coding style should also be considered as an evaluation criterion for open source software. This is particularly true for code of software that will be used to implement high-consequence/trusted functions. As with from-scratch code, the acquired code's clear, comprehensible coding style will make the security analysis of that code easier.

7.1.3 Follow secure coding standards and/or guidelines

Secure coding standards and guidelines identify safe coding practices and constructs and identify common coding flaws and constructs that can manifest as vulnerabilities, along with secure alternatives to those problematic flaws/constructs. These standards and guidelines, which should cover both what should be done and what should not be done when coding in all languages that will be used in the programming of the software, can be "home grown"—as is most likely to be the case for commercial software producers, or existing externally-sourced standards and guidelines may be used—which may prove helpful for government and

private-sector contractors who develop custom software. For example, the CMU Software Engineering Institute published secure coding standards for C and C++; these are included in the Suggested Resources at the end of Appendix C:C.4.1. While no comparably detailed secure coding standards have yet been published for other languages, there are some higher-level secure coding guidelines available for Java, Perl, *etc.* These are listed in the Suggested Resources sections for each programming language in Appendix C:C.4.

7.1.4 Make code forward and backward traceable

Structure the program code so that it is both backward and forward traceable. It should be possible to easily trace each requirement from the specification to its manifestation in the code, and each design element from its manifestation in the detailed design to its manifestation in the code. It should also be possible to derive each requirement and design element from its manifestation in the code.

7.1.5 Code for reuse and maintainability

The features that make code elegant and secure—simplicity, comprehensibility, traceability—also contribute to its reusability and maintainability. The programming team should start the coding process by writing a code specification that is clear, understandable, and comprehensive enough to guide other programmers in writing the specified code. This way, even if mismatches in design or architecture prevent later reuse of the code, that code will be easily maintainable.

The programmer should never assume that his/her source code will be self-explanatory. Extensive commenting and documentation, including the results of reviews and tests, will help other programmers and maintainers gain the complete and accurate understanding of the code that they need to reuse or modify it without introducing vulnerabilities.

7.1.6 Allocate memory and other resources carefully

Minimize the computer resources made available to each process. For example, for software that will run on Unix, use *ulimit(), getrlimit(), setrlimit(), getrusage(), sysconf(), quota(), quotactl(),* and *quotaon()* (also *pam_limits* for pluggable authentication module processes) to limit the potential damage that results when a particular process fails or is compromised and to help prevent DoS attacks on the software.

If the software is a Web server application or Web service, set up a separate process to handle each session, and limit the amount of central processing unit (CPU) time that each session in each process is allowed to use. This will prevent any attacker request that hogs memory or CPU cycles from interfering with tasks beyond its own session.

> *NOTE: Designing all sessions to be atomic and resource-limited will make it difficult to create denials of service by spawning multiple sessions.*

Memory locations for cache buffers should not be contiguous with executable stack/heap. Whenever possible, set the stack to be non-executable. See Section 6.5 for a discussion of how to implement secure memory/cache management.

7.1.7 Minimize retention of state information

The software should retain only the bare minimum of needed state information, and should frequently purge data written in cache memory and temporary files on disk. These measures will minimize the likelihood of undesired disclosure of sensitive information, including information about the software itself that can be leveraged by attackers, in case of the software's failure.

7.1.8 Leverage security through obscurity only as an additional deterrence measure

Security through obscurity measures, such as code obfuscation, use of hidden files, *etc.*, at best provide a weak deterrence against reconnaissance attacks and reverse engineering. Such measures should only ever be used in addition to a robust set of true security measures, to provide enough of an inconvenience factor to possibly deter unsophisticated and casual attackers.

7.1.9 Avoid unauthorized privilege escalation

The programmer should not write logic that enables users or processes to perform unexpected or unintended privilege escalations. Attackers can observe a process that attempts to reference another process that has higher privileges than its own, or that attempts to exploit race conditions in that second process. The attackers will interpret such actions on the first process' part as indicating laxness in privilege enforcement and authentication validation by the software system. Processes with higher privileges than the attacker should not be visible to the attacker; if the attacker can see the higher privileged process, he/she can exploit it to escalate his/her own privileges.

7.1.10 Use consistent naming

A common cause of security flaws in implemented software is incorrect developer use of aliases, pointers, links, caches, and dynamic changes without re-linking. To reduce the likelihood of such problems, developers should:

- **Treat aliases symmetrically:** Every alias should be unique, and should point to only one resource;
- **Be cautious when using dynamic linking:** This will avoid unpredictable behaviors that result from runtime introduction of components. For example, Java-extensible Web browsers rely on static type systems with link checks to enforce a wide class of important safety properties—properties that could be compromised through dynamic linking. Several approaches have been proposed to address the need for safe dynamic

linking, including: runtime and compile-time type-based analyses and procedural analyses, to ensure type safety of dynamically-linked code and to identify the code's calling context and ensure that only the names needed in that context are linked, as well as use of proof-carrying code techniques for link-time validation of native code. These countermeasures have been found to address security issues, such as the introduction of unexpected behaviors and the unacceptable expansion, through dynamic linking, of a minimal trusted computing base. In the absence of such countermeasures, at a minimum, dynamic linking should be used only with code whose security and other critical properties are not put at risk by runtime additions, modifications, or replacements.

- **Minimize use of global variables:** When such variables are necessary, give the variables globally-unique names;
- **Clear caches frequently**;
- **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;
- **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.

7.1.11 Use encapsulation cautiously

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.

7.1.12 Leverage attack patterns

To use attack patterns to identify specific coding flaws (*i.e.,* bugs, valid constructs with negative security implications) targeted by relevant attacks, and ensure that these flaws do not occur in his/her code, the programmer should:

1. Determine which attack patterns are applicable: *i.e.,* which subset of available attack patterns are relevant given the software's architecture and execution environment, and the technologies used to implement the software. For example, the *Buffer Overflow* attack pattern would be relevant for a C or C++ program running on native Linux, but not for a C# program running on .NET.

2. Determine what constructs, *etc.*, should not appear in their code, based on what attack patterns need to be avoided.

The following example illustrates how a programmer can leverage an attack pattern:

> **Attack pattern:** Simple Script Injection
>
> **Use to:** Avoid cross-site scripting vulnerabilities.
>
> **Areas of code which this pattern is likely to target:** Areas from which output data is sent to the user from an untrusted source.
>
> How to protect code against this attack pattern: If no countermeasure has already been implemented (*i.e.*, based on an architectural decision to include a self-contained input validator/output filter at the juncture between the server and the client), implement a programmatic countermeasure such as:
>
> 1. Convert potentially dangerous characters into their HTML equivalents to prevent the client from displaying untrusted input that might contain malicious data or artifacts, such as <*script*> tags inserted by an attacker. Examples of such conversions: "<" *becomes* "<" | ">" becomes ">"). There are third-party Java libraries that automatically perform such conversions; JavaScript's *escape()* function also performs similar conversions. Note that such conversions need to be managed carefully to avoid potential unintended buffer overflow vulnerabilities that may result from routinely replacing single characters with longer character strings.
>
> 2. Implement an input validation filter that filters input based on a white list of allowable characters.

7.1.13 Input encoding and validation

Injection attacks are commonly performed against applications. These can take the form of format string attacks in C or cross-site scripting attacks in Web scripting languages. An application that accepts user input and forwards it on to the output or some trusted function may be targeted for attack. As such, it is important to properly validate input to ensure that it meets the application's expectations (*e.g.,* by verifying that the input has a certain length and contains no "special" HTML characters), and by securely handling any invalid input. In cases where input validation may allow potentially malicious characters (*e.g.,* "<" in Web applications), applications should encode these characters so that they will not be mistaken by other functions or relying applications.

7.1.13.1 Implementing 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. 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.

Input validation should verify:

- The conformance of input to specified parameters for that input. These parameters include:
    - Length,
    - Range,
    - Format,
    - Type.
- The absence from input of any constructs that are not explicitly allowed and expected. Such constructs include:
    - Query strings,
    - Cookies,
    - File paths,
    - URL paths.

The length of every input element should be checked (*i.e.*, bounds checking), and the acceptable length should be restricted to the shortest possible value.

Each component should do its own input validation unless the design ensures that trustworthy validation of the input has been successfully achieved before the component receives that input (example of design by contract). In many cases the application framework or integrated development environment being used will provide reusable code samples for input validation; for example, ASP.NET Validator Controls and JavaScript filtering.

As noted earlier, white listing is the preferred approach to input validation. White listing verifies that input conforms to defined acceptable parameters, and rejects all input that does not conform. White list validation has proved effective in filtering out input patterns associated with unknown attacks. By contrast, the other approach, black listing, verifies that input does not conform to defined unacceptable parameters associated with suspected malicious input. Black listing accepts all input that is not explicitly identified as "bad". It is useless in protecting against unknown attacks. It can, however, be useful in weeding out some

bad input prior to white list filtering. It can also be used as a stopgap for filtering out attack input associated with zero-day vulnerabilities until a patch is available. Ideally, white listing and black listing will be used in combination, as indicated, with black listing preceding white listing.

Once bad input is detected, it needs to be handled. The best approach is to simply reject bad input, rather than attempting to sanitize it. However, there will be times when sanitization is necessary. If this is the case, all data that is sanitized should be resubmitted for white list validation to ensure the sanitization was effective.

7.1.13.1.1 Preliminary client-side validation

As a first line of defense against malicious users, it may make sense to include logic for preliminary validation of input within the client (*e.g.,* as a browser plug-in implemented in JavaScript) before the accepted or sanitized input is forwarded to the server, where it will undergo more robust mandatory validation.

Validation on the client can help reduce requirements for bandwidth and server CPU cycles, because less unacceptable data will be forwarded from client to server, and more client-originated data will be valid and thus will not require sanitization by the server.

SUGGESTED RESOURCES

- Viega, John and Matt Messier. "Input Validation in C and C++". Chapter excerpt from *Secure Programming Cookbook for C and C++,* posted on O'Reilly Network Website, 20 May 2003. Accessed 19 December 2007 at: http://www.oreillynet.com/pub/a/network/2003/05/20/secureprogckbk.html

- secologic. "A Short Guide to Input Validation". Version 1.0, 25 April 2007. Accessed 14 December 2007 at: http://www.secologic.org/downloads/Web/070509_secologic-short-guide-to-input-validation.pdf

- Norton, Francis. "Implementing Real world Data Input Validation Using Regular Expressions" (for .NET). *Simple-Talk,* 14 May 2007. Accessed 14 December 2007 at: http://www.simple-talk.com/dotnet/.net-framework/implementing-real world-data-input-validation-using-regular-expressions/

- "Reg Ex Input Val Code—Validate User Input with Regular Expressions" [C#]. Patterns and Practices Guidance Library Wiki. Accessed 14 December 2007 at: http://www.guidancelibrary.com/default.aspx/Home.RegExInputValCode

- Kurz, John. "Dynamic Client-Side Input Validation". *ColdFusion Developer's Journal,* 1 May 2003. Accessed 14 December 2007 at: http://coldfusion.sys-con.com/read/41599.htm

- Grossman, Jeremiah. "Input validation or output filtering, which is better?" On his Weblog, 30 January 2007. Accessed 14 December 2007 at:

http://jeremiahgrossman.blogspot.com/2007/01/input-validation-or-output-filtering.html

7.1.13.1.2 XML schema and input validation

The rationale for validating input for Web services is no different than it is for any other type of software program. The first line of defense should be to perform XML schema validation to ensure the XML message is well formed and meets the data accuracy, completeness, and validity constraints. Validating using an XML Schema Design (XSD) helps prevent parameter tampering. XSDs also allow the use of regular expressions to define restrictions. For example, the following regular expression in XSD restricts the speed field to three integers ranging from 0-9:

```
...
<xs:simpleType name="speed">
 <xs:restriction base="xs:string">
 <xs:pattern value="[0-9]{3}"/>
 </xs:restriction>
</xs:simpleType>
...
```

Although well-formedness and validation can block a substantial proportion of Web service attacks, there are still vulnerabilities specific to XML that may not be caught using an XSD validator. The majority of these attacks are aimed at overloading the parser. When writing in Java, such attacks can be addressed, and potential for overflows and denial of service minimized, by using the secure processing feature of Simple API for XML (SAX)—*XMLConstants.FEATURE_SECURE_PROCESSING*—in Java API for XML Processing Version 1.3 (*i.e.,* bundled with Java 1.5 and available as an option in earlier Java versions). Secure SAX processing will cause excessively long constructs to be flagged as well-formed-ness errors, regardless of whether the excessive length is due to inclusion of too many attributes in an element or too many characters in an element name. After obtaining an instance of a SAX parser, configure it to use secure processing, as in the example provided in the Codase source code referenced at the end of this section.

SUGGESTED RESOURCES

1. Harold, Elliotte. "Configure SAX parsers for secure processing (Prevent entity resolution vulnerabilities and overflow attacks)". IBM DeveloperWorks, 27 May 2005. Accessed 9 April 2008 at: http://www-128.ibm.com/developerworks/xml/library/x-tipcfsx.html

2. Codase BETA Java Source Code Search Engine. Entry for javax.xml Class XMLConstants#FEATURE_SECURE_PROCESSING. Accessed 8 April 2008 at: http://www.codase.com/java/javax/xml/XMLConstants.html#FEATURE_SECURE_PROCESSING

7.1.13.2 Testing input validation logic

Testing of input validation logic should verify the correctness of the input validation routines. The test scenarios should include submission of both valid and invalid data, and the tester should look for false positives (valid data rejected by input validation) and false negatives (invalid data accepted by input validation). If client-side validation is implemented as a first-line-of-filtering, it should also be tested with and without JavaScript (JavaScript interprets regular expressions slightly differently than server validation engines do; this can generate false positives client side, and false negatives on the server side). Finally, the tester should also run the application with validation turned off both server-side and client-side. Doing so will reveal how receipt of invalid data affects other application tiers, *e.g.,* by exposing vulnerabilities. In architectures in which input validation is centralized, it will also reveal whether additional validation may be needed at other points in the system.

7.1.14 Output filtering and sanitization

Processes that generate or pass output to external entities should be implemented so that their output is checked to ensure it conforms to parameters for allowable output, and that it contains no content that is unallowable. In essence, the same types of checks used for validating input are applied to output before that output is released to its intended external recipient.

7.1.15 Avoid security conflicts arising between native and non-native, passive and dynamic code

Increasingly, applications are relying on code that is written in another programming language—or code that may not have been written when the application was originally developed. For example, some Java applications may rely on C code to interface directly with hardware, or an Asynchronous Java And XML (eXtensible Markup Language) (AJAX) application may supply dynamically generated JavaScript to the Web browser to perform an operation. In both cases, it is important to understand the security implications involved. It is imperative that applications treat native and dynamic code as potentially untrusted entities.

In the Java example, an attacker may be able to perform a buffer overflow attack against the native code. In the dynamic code example, there is the possibility that the dynamically generated code may not meet the expectations of the application. In the AJAX example, the dynamically generated code may have been developed after the original application, making different assumptions about the environment and state of the application—leading to potentially invalid input to the AJAX application. As such, it is imperative that developers perform validation on data going to the untrusted code as well as data received.

7.1.16 Review code during and after coding

Programmers should review code as they write it, to locate flaws within individual units/modules before checking those units/modules into the SCM system. Programmers should also look for flaws in interfaces between units/modules before submitting those units/modules for compilation and linking.

Before compilation, 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).

Static analysis, also referred to as code review, is any analysis that examines the software without executing it. In most cases, this means analyzing the program's source code, although a number of tools are emerging to enable static analysis of binary executables. Because static analysis does not require a fully integrated or installed version of the software, it can be performed iteratively throughout the software's implementation. This said, code review has inherent shortcomings in terms of its impracticality for examining sections of code larger than

individual code units. For this reason, static analysis should always be combined with other software security testing techniques.

The main objective of code reviews is to discover security flaws and to identify their potential fixes. The test report should provide enough detailed information about software's possible failure points to enable its developer to classify and prioritize the software's vulnerabilities based on the level of risk they pose to the system (*e.g.,* the vulnerabilities that hackers are most likely to be able to exploit successfully).

Source code analysis and white box testing should be performed as early and as often in the life cycle as possible. The most effective white box tests are performed on granularly small code units—individual modules or functional-process units—which can be corrected relatively easily and quickly before they are added into the larger code base. Iteration of reviews and tests ensures that flaws within smaller units will be dealt with before the whole-system code review, which can then focus on the "seams" between code units, which represent the relationships among and interfaces between components.

Static analysis can be wholly manual or tool-assisted. In a manual static analysis, the reviewer inspects all code without the assistance of automated tools. Manual code review is highly labor-intensive, but can, when reviewers with appropriate levels of experience perform the review, produce the most complete, accurate results early in the review process, before reviewer fatigue sets in. It is common for the reviewer to begin by very meticulously checking every line of code, then to gradually skip larger and larger portions of code, so that by the end of the review, the inconsistent and decreasing amount of "code coverage" is inadequate to determine the true nature of the software. It is important to note, that as the size of the code-base increases it comes less feasible to perform a complete manual review. In such cases, it may be beneficial to perform a semi-automated review, with manual reviews being performed on critical subsets of the code base.

The fully automated code review relies on a tool or tools to perform the entire code inspection. The reviewer's job limited to running the tool and interpreting its results. While automated tools can scan very large code bases in a short time with consistent results and metrics, their findings will necessarily be only as complete as the list of patterns (*e.g.,* common unsafe coding constructs) they are programmed to scan for. Automated code review tools are not sophisticated enough to detect anomalies that a human reviewer would notice, but which are not included in the tool's pre-programmed list of patterns. Nor can such tools identify vulnerabilities in the "seams"/relationships between different sections of code (though there are tools emerging that can identify such relationships), or vulnerabilities arising from the interactions between non-contiguous segments of code. Fully automated tools can provide additional benefits, by allowing developers to run scans as they are developing—addressing potential security vulnerabilities early in the process. Similarly, the level of expertise required for an automated review might be less than that required for a manual review. In many cases, the automated tool will provide detailed information about the vulnerability found, including suggestions for mitigation.

One of the primary concerns with code review tools concerns assessing their accuracy. In most situations, the accuracy of any analysis tool revolves around its false negative (*i.e.,* vulnerabilities overlooked by the tool) and false positive (*i.e.,* false alarms produced by the tool) rates. Most of the algorithms used by these tools can be tailored to reduce the false positive rate at the expense of increasing the false negative rate or vice-versa. When relying on these tools to generate evidence to support assurance case, the reviewer must provide strong evidence that the tool was used correctly and in a way that ensured that the number of false negatives and false positives were minimized, and the maximum number of *true* results was discovered. In particular, it is difficult for testers to rely solely on the information provided by vendors. Vendors have a financial incentive to ensure that their respective tools have the lowest false positive and false negative rates, it is very difficult for an independent observer to truly judge a tool's false negative rate.

A number of resources are available to aid in this regard. The NIST Software Assurance Metrics And Tool Evaluation (SAMATE) project has defined the draft NIST SP 500-268, *Source Code Security Analysis Tool Functional Specification*, which identifies a definitive set of requirements against which source code analysis tools can be measured. Additionally, SAMATE has released the draft NIST SP 500-270, *Source Code Security Analysis Tool Test Plan*, which provides insight into using the SAMATE Reference Dataset (SRD), which is a compilation of insecure code examples, against which users can run multiple source code analysis tools and compare their results to get a better understanding of their comparative strengths and weaknesses.[37]

The semi-automated code review involves the human reviewer leveraging automated tools to assist in the otherwise manual inspection of the code. The tool is used to locate portions of code that contain known problem patterns as a "jumping off point" for the reviewer's further analysis. In this way, the reviewer is "guided" towards problem areas in the code, but does not rely on the tool alone to locate any additional problems, anomalies, *etc.,* in those areas.

Whether manual, semi-automated, or fully automated, a simple static analysis entails searching for strings, identifying user input vectors, tracing the flow of data through the application, mapping execution paths, *etc.*

A more thorough analysis will examine the structure of the source code to reveal the software's intended behaviors, data flows, function calls, and loops-and-branches.

The most resource-intensive form of static analysis is direct code analysis, which focuses on verifying the software's satisfaction of required security-related properties, such as non-interference, non-inference, separability, *persistent_BNDC,* forward-correctability, non-deducibility of outputs, *etc.* Because of the time and resources required for direct code analysis,

37 Due to the rapid development and improvement of source code analysis tools, along with restrictions in place in many tools' end user license agreements, it is impractical to publish comparisons of source code analysis tools. As such, organizations should perform their own analysis against the SRD as well as against their own codebases.

it is only practical for examining relatively small portions of code, and will probably be limited to code that implements high-consequence functions and external interfaces.

Code reviews and other white box tests are also useful for detecting indicators of the presence of malicious code. For example, if the code is written in C, the reviewer might seek out comments that indicate exploit features, and/or portions of code that are complex and hard-to-follow,[38] or that contain embedded assembler code (*e.g.,* the *_asm_* feature, strings of hexadecimal or octal characters/values).

Other things to look for during static code analysis include:

- Presence of developer backdoors and implicit and explicit debug commands (implicit debug commands are seemingly innocuous elements added to the source code to make it easier for the developer to alter the software's state while testing; if these commands are left in the software's comment lines they may be exploitable after the software has been compiled and deployed);

- Unused calls that don't accomplish anything during system execution, such as calls invoking environment-level or middleware-level processes or library routines that are not expected to be present in the installed target environment.

Static analysis has inherent limitations in terms of the types of vulnerabilities and flaws it can detect. These limitations and shortcomings include:

1. Inability to find vulnerabilities introduced or exacerbated by the execution environment;

2. Inability to support novice code reviewers, *i.e.,* those without profound understanding of the security implications of software coding constructs;

3. Lack of support for inter-procedural analysis spanning multiple source files;

4. Difficulty analyzing files that have been preprocessed with build configuration dependencies; for example, lack of ability to parse into a single abstract syntax tree C/C++ programs that include preprocessor directives combined with inability to check every possible build of a program because the number of potential combinations of preprocessor directives increases exponentially with each build;

5. Inability to determine whether code that contains no "dangerous" constructs or security flaws will not manifest vulnerabilities or insecure behaviors after it has been compiled

38 Submissions to the International Obfuscated C Code Contest and the Underhanded C Contest at 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.

and executed, *e.g.,* due to interactions with the execution environment, users, other systems, *etc.* that cannot be predicted by a code review.

Tools that support manual, semi-automated, and fully automated static analyses include code review assistants, source code scanners, and buffer overrun detectors.

SUGGESTED RESOURCES

- BuildSecurityIn White Box Testing resources. Accessed 21 January 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/white box.html
- BuildSecurityIn Code Analysis resources. Accessed 21 January 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/code.html
- Secure Software, Inc. "Risk in the Balance: How the Right Mix of Static Analysis and Dynamic Analysis Technologies Can Strengthen Application Security". 2004. Accessed 3 January 2008 at: http://secureitalliance.org/blogs/files/164/1137/Risk%20in%20the%20Bal_wp.pdf
- OWASP Code Review Project page. Accessed 14 December 2007 at: http://www.owasp.org/index.php/Category:OWASP_Code_Review_Project
- Chess, Brian and Jacob West. *Secure Programming with Static Analysis.* Indianapolis, Indiana: Addison-Wesley, 2007.
- Hsu, Francis. "Input validation of client-server Web applications through static analysis". Presented at Web 2.0 Security and Privacy 2007, Oakland, California, 24 May 2007. Accessed 14 December 2007 at: http://seclab.cs.rice.edu/w2sp/2007/papers/paper-210-z_9464.pdfSecure
- Wilander, John and Pia Fåk. "Pattern Matching Security Properties of Code using Dependence Graphs". *Proceedings of the First International Workshop on Code Based Software Security Assessments,* Pittsburgh, Pennsylvania, 7 November 2005, pages 5-8. Accessed 3 January 2008 at: http://www.ida.liu.se/~johwi/research_publications/paper_cobassa2005_wilander_fak.pdf
- NIST SAMATE Home Page. Accessed 11 September 2008 at: https://samate.nist.gov/
- Howard, Michael. "A Process for Performing Security Code Reviews." *IEEE Security and Privacy*, Volume 4 Number 4, July/August 2006, pages 74-79. Accessed 11 September 2008 at: http://doi.ieeecomputersociety.org/10.1109/MSP.2006.84
- Shostack, Adam. "Security Code Review Guidelines." Accessed 11 September 2008 at: http://www.homeport.org/~adam/review.html

7.2 SURVIVABILITY THROUGH ERROR, ANOMALY, AND EXCEPTION HANDLING

Survivable software contains more error- and exception-handling functionality than program functionality. Error and exception handling can be considered to aid in survivability 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.

The exception handling block should include a logger to record when and why an exception was thrown; auditors can later review this log. The exception handling block should also be written to include messaging code containing that will automatically send an email alert to the system administrator when an exception requires human intervention.

Table 7-1 lists several common software errors, and suggests remediations for those errors at the design and implementation levels.

Table 7-1. Software errors and suggested remediations

Expected Problem	How Software Should Handle the Problem
Input received by the software contains anomalous content.	IMPLEMENTATION: Validate all input.
The execution environment differs significantly from the environment for which the software was designed.	DESIGN: Recognize all explicit and implicit assumptions COTS and OSS components have about their environment. Design to mitigate vulnerabilities created by mismatch between component assumptions and actual environment. Design also to minimize external exposure of component/environment interfaces.
There are errors in the results returned by called functions.	DESIGN: Design for resilience. IMPLEMENTATION: Anticipate all likely errors and exceptions, and implement error and exception handling to explicitly address those errors/exceptions.
The software contains vulnerabilities that were not mitigated or detected before the software was deployed.	DESIGN: Include measures that isolate untrusted, suspicious, and compromised components, and to constrain and recover from damage. IMPLEMENTATION: Measures that reduce exposure of untrusted and vulnerable components to externally-sourced attack patterns, *e.g.*, using wrappers, input filters, *etc.*

Practices for implementing effective error, anomaly, and exception handling are described below.

7.3.1 Anomaly awareness

In most distributed software 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.

7.3.2 Event monitors

Effectiveness of event—error or anomaly—monitors depends on the correctness of assumptions about:

1. The structure of the program being monitored;
2. The anomalies and errors that are considered possible, and those that are considered unlikely.

As with all assumptions, these may be invalidated under certain conditions.

An anomaly or error should be detected as near in time to the causal event as possible. This will enable isolation and diagnosis *before* erroneous data is able to propagate to other components. The number of program self-checks that can be implemented is usually limited by the availability of time and memory. At a minimum, there should be checks for all security-critical states. Use risk analysis to establish the basis for defining the optimal contents and locations of each check.

Monitor checks should be non-intrusive, *i.e.,* they should not corrupt the process or data being checked. In addition, the developer should take particular care when coding logic for monitoring and checking to avoid including exploitable flaws.

7.3.3 Security error and failure handling

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 that 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.

7.3.4 Core dump prevention

Core dumps are only acceptable as a diagnostic tool during testing. Programs should be implemented to be configurable upon deployment to turn off their ability to generate core dumps when they fail during operational use. Instead of dumping core when the program fails, the program's exception handler should log the appropriate problem before the program exits. In addition, if possible, configure the size of the core file to be 0 (zero) (*e.g.,* using *setrlimit* or *ulimit* in Unix); this will further prevent the creation of core files.

7.4 SECURE MEMORY AND CACHE MANAGEMENT

Much of today's software is written to maximize performance through extensive use of persistent memory caching. The problem with persistent memory is that the longer data remains in memory, the more opportunity there is for that data to be inadvertently or intentionally disclosed if a failure causes the content of memory to core dump or otherwise become directly accessible. The problem arises because memory is not subject to the access control protections of the operating system level file system, or in a database application, the database management system.

For example, when using entity beans on a Java Platform, Enterprise Edition (Java EE) server, data can be stored on the server with either container-level persistence or the bean-level

persistence. In either case, the program that processes and caches the data need to provide secure cache management capabilities that can accommodate all of the simultaneous processes the program has to multitask (*e.g.,* receipt and handling of requests, management of sessions, reading of data from database or file system). The developer can force a program that uses Enterprise Java Beans to write data to persistent non-volatile storage after each transaction instead of making the data persistent in memory; the tradeoff is one of performance (fewer database accesses = better performance) *vs.* security (less memory persistence = less opportunity for attackers to access sensitive data not protected by file system or database access controls). In multi-user programs, the amount of overhead required to securely manage persistent cache is also affected by the number of users who simultaneously access the program: the more data that must be cached, the less memory available to other processes.

If writing a program in which memory will be persistent, the developer should ensure that the length of persistence is configurable, so that it will be purged as frequently as the administrator or user desires. Ideally, the program will also provide a command that allows the administrator or user to purge the memory at will. When using a COTS or OSS component that makes memory persistent, if the data the component will store in memory is likely to be sensitive, consider hosting the component on a TPM to isolate its persistent memory from the rest of the system. Regardless of the length of memory persistence when the program is running, data should always be purged from memory as soon as the program shuts down. For high-consequence and trusted processes, the program should not retain the memory beyond the completion of the process; it should be purged when the process completes.

Ideally, extremely sensitive data, such as authentication tokens and encryption keys will never be held in persistent memory. However, persistent memory cannot be avoided in COTS and OSS components. In these cases, if the component is likely to store sensitive data in persistent memory, the developer should leverage the cache management and object reuse capabilities of the operating system and, if there is one, the database management system to overwrite each persistent memory location with random bits seven times (the number of times considered sufficient for object reuse).

7.4.1 Safe creation and deletion of temporary files

Temporary files are created by some programs to hold intermittent state information about the transaction or operation in progress. As with data cache, temporary files may contain sensitive information, making them a target of interest to attackers. Security attacks can be realized due to:

- **Insecure temporary file management:** The attacker creates a file with the same name as an existing file in the *temp* file directory. The attacker then copies the bogus *temp* file into the *temp* file directory, thereby overwriting the real *temp* file.
- **Symbolic link vulnerability:** If the attacker knows where the application creates its temporary files and can guess the name of the next temporary file, the attacker can place a symbolic link at the temporary file location, then link that symbolic link to a

privileged file. As a result, the application will unknowingly write its temp data to the privileged file instead of the temp directory.

On his application security Weblog, Richard Lewis describes a safe approach to temporary file creation and handling by software applications.[39]

Ideally, programs would not create temporary files or file copies in the first place. If use of temp files cannot be avoided (*e.g.,* due to performance requirements), ensure that the program always deletes all temporary files at a minimum at the time the program's execution is terminated. As with cache purging, the temp file deletion should completely erase the file from disk or overwrite its disk location seven times with random bits. The program should also allow for temp file deletion-on-command by the user or administrator (without having to terminate the program), and should ideally also support configuration of greater frequency of automatic temp file deletions.

7.5 INTERPROCESS AUTHENTICATION

The primary purpose of interprocess authentication is to link the identity and privileges of a human user with those of the application processes that operate on his/her behalf.

Identification and authentication, whether of human users or software entities (processes, services, components), provides the basis for two associations critical to assuring the secure operation of the software system:

- Association with security-relevant attributes upon which security decisions related to the user or software entity will be based, such as whether or not to grant the entity to access certain data or resources;

- Association of a software process' actions with the user's or software entity's authenticated identity, for purposes of accountability. For accountability of software entities to be useful, there also needs to be a means to reliably and irrevocably associate the software entity with a human user, such as the association between a human user and the requestor Web service that is spawned on his/her behalf. Authentication using Kerberos tickets, Security Assertion Markup Language assertions, SSL/TLS with X.509 certificates or one-time encrypted cookies, secure remote procedure call (RPC) protocols, *etc.,* enables the binding of a human identity with a software process that will act on that human's behalf.

However, new computing models require the ability of autonomous processes to dynamically invoke other autonomous software processes or agents which in turn can dynamically invoke yet more autonomous processes/agents, all of which may be widely distributed throughout

39 Lewis, Richard. "Temporary files security in-depth". Posted on his Application Security blog, 12 October 2006. Accessed 31 December 2007 at: http://secureapps.blogspot.com/2006/10/temporary-files-security-in-depth.html

the software-intensive system, or even belong to other software systems—resulting in a system of systems. Under such conditions, it becomes extremely difficult if not impossible to extend the association of a human with each process or agent subsequently invoked by the first process/agent with which the user is associated. Accountability becomes a matter of keeping track of all the downstream processes/agents that are invoked as part of the need to respond to the original user request or action that originally associated that user with the first process/agent in the chain.

In Web services-based SOAs, this is the problem that large-scale trust management models such as WS-Security and the Liberty Alliance framework are meant to solve. In the longer term, however, it is likely that accountability of software entities will become a goal in itself. Such accountability will have the benefit of providing a basis for dealing with badly behaved software processes/agents by denying them access to resources, isolating them in sandboxes, and even removing from the system.

Grid computing initiatives, such as the Globus Grid Security Infrastructure, are defining solutions such as "run-anywhere" single-sign on authentication of grid agents using SSL/TLS and X.509 certificates. Moreover, the emergence of agent-based systems is driving definition of standards and technical solutions to provide more robust inter-agent authentication and accountability without reference back to a human user.[40]

7.5.1 Secure RPC

RPC is designed to implement secure communications between the processes within distributed applications. The RPC runtime library provides a standardized interface to authentication services for both client and server processes. The authentication services on the server host system provide RPC authentication. Applications use authenticated remote procedure calls to ensure that all calls come from authorized clients. They can also help ensure that all server replies come from authenticated servers.

RPC specifications are limited to relatively narrow applications that are confined within a single administrative domain. Since many Web applications need to operate across domain boundaries, RPC for Web applications needs a comprehensive security infrastructure beyond what is possible by simply layering the RPC mechanism over SSL/TLS.

The OpenGroup's Distributed Computing Environment (DCE) included a specification for use of authenticated RPCs between clients and servers. Authenticated RPC works with the authentication and authorization services provided by the DCE Security Service, specified in the RPC runtime library for the particular server application for which authenticated RPC is

---

40 Work is being done to address the problem of accountability of software entities, especially grid agents. Some of these approaches are described in Yumerefendi, Aydan R. and Jeffrey S. Chase. "Trust but Verify: Accountability for Network Services". *Proceedings of the Eleventh ACM SIGOPS European Workshop,* Leuven, Belgium, 19-22 September 2004. Accessed 7 July 2008 at: http://issg.cs.duke.edu/publications/trust-ew04.pdf

being enabled. DCE specifies a number of authenticated RPC routines that can be used by client-server application programmers in this context.

7.6 SECURE SOFTWARE LOCALIZATION

Creation of variants localized for different countries' use can result in availability problems if caution is not used in writing source code that is suitable for both global and localized usage.

When abuse/misuse cases and attack models are developed (*i.e.,* as a basis for defining security requirements), there should be cases and models for each localized software variant.

The main problems associated with local variants relate to potential for creating buffer overflow vulnerabilities, due to the longer lengths of resources strings in some languages, the expression of string lengths in bytes *vs.* characters in various character encodings, and the increase in required buffer size that can result from conversions from one encoding to another.

SUGGESTED RESOURCES

- Elgazzar, Mohamed. "Security in Software Localization". Microsoft Global Development and Computing portal, no date. Accessed 19 December 2007 at: http://www.microsoft.com/globaldev/handson/dev/secSwLoc.mspx

7.7 LANGUAGE-SPECIFIC SECURITY CONSIDERATIONS

NOTE: The security benefits and concerns associated with individual programming languages are discussed in Appendix C:C.4.

The choice of programming language is an important factor in writing secure code. While in many cases, existing libraries or requirements may require the use of one programming language over another, there are many situations where the choice of language can directly affect the security of the system. The most prominent example is the effect of array bounds checking in Java *vs.* C (though most modern C compilers support run-time bounds checking). This feature (or lack thereof) has led to a rash of buffer overflow based vulnerabilities in Web servers, operating systems, and applications for decades. Similarly, in environments where performance and footprint are extremely important (such as embedded devices and smart cards), a Java virtual machine may introduce too much strain on the system, resulting in potential denials of service.

Type-safe languages such as Java, Scheme, ML (MetaLanguage), F#, and Ada 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.

Some key questions should be answered in evaluating and selecting the programming language(s) to be used in writing new code. These are:

- Is the language simple and straightforward? Will it encourage writing simple, understandable programs?
- Does the language include any features that will aid in writing secure software? *Example:* taint mode, own virtual machine environment
- Does the language include any features that will make writing secure software more difficult? *Example:* lack of type safety
- If a non-secure language was initially considered, is there a more secure alternative that is suitable for writing the code? *Example:* Java instead of C++
- Are there secure coding standards that can be followed to avoid non-secure coding constructs?
- Are there tools available to support secure compilation, security-oriented debugging, in the language?
- Are there secure alternatives to standard library routines in the language?

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 re-implementing the program in a language that includes its own security model and self-protecting features (*e.g.,* Java, Scheme, Categorical Abstract Machine Language) rather than C or C++.

A programming language that supports good coding practices and has few inherent vulnerabilities is more likely to be used securely than a language that has critical security flaws 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# CLR). Ada is a language that its proponents actively promote due to its inherent support for producing code that is reliable, predictable, and analyzable. Security-enhancing features of Ada are described in Appendix C:C.4.4.

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 important to note that operating system (OS)- and compiler-level buffer overflow protections are becoming increasingly commonplace). 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.

Since the majority of programs use system calls to transfer data, open files, or modify file system objects (two noteworthy exceptions are embedded programs and safety-critical software systems, which are often implemented without any use of operating system calls or third-party libraries; see Appendix C:C.2 for more information on the unique security aspects of such software), 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.

- Tainting (*e.g.,* Perl);
- Code security (language or environment-based);
- Tightly-coupled execution environments (*e.g.,* JVM);
- Secure language derivatives (see Appendix C:C.4.10).

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.

7.8 TOOLS THAT ASSIST IN SECURE CODING AND COMPILATION

NOTE: Tools that support earlier life cycle activities (requirements specification, architectural modeling, design) were discussed at appropriate points in Sections 3, 4, and 5.

Developers who have a true and profound understanding of the security implications of their development choices will be more likely to produce secure software using general-purpose development tools that enforce good software engineering choices than developers who use

only "safe" languages and "secure" development tools but who are insufficiently knowledgeable in secure development principles and practices to understand how to best leverage the security features and assistance provided by those tools.

This said, a security-aware developer will find that tools that actively encourage secure specification, design, and implementation will greatly assist in the:

- Reduction of exploitable flaws and weaknesses;
- Reduction of exposure of residual exploitable flaws and weaknesses;
- Implementation of software security constraints, protections, and services;
- Minimization and constraint of propagation, extent, and intensity of damage caused by insecure software behaviors.

The remainder of this section describes several categories of helpful security-oriented development tools.

7.8.1 Compiler security checking and enforcement

Compile-time detection and runtime detection rely on the compiler to ensure that correct language usage rules have been adhered to, and to detect and flag, or in some cases eliminate, faults and dangerous constructs in the source code that were not detected during code review and that could make the compiled software vulnerable to compromises, *e.g.,* buffer overflow-prone calls in C and C++.

A simple version of compile-time detection occurs in all basic compilers: type checking and related program analysis. The level of type checking 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 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.

Security problems specific to non-typed languages may be addressed by more robust type-checking compilers that flag and eliminate code constructs and flaws associated with insecure typing (*e.g.,* pointer and array access semantics that could generate memory access errors). These compilers also perform bounds checking of memory references to detect and prevent

buffer overflow vulnerabilities in stacks and (sometimes) heaps. Two examples of open source type-checking compilers are Fail-Safe C and the Memory Safe C Compiler.[41] As their names imply, both are intended to compile C programs in ways that eliminate buffer overflow vulnerabilities.

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).

In addition, compilers may be modified to detect a maliciously modified stack or data area. A simple form of this protection is the stack canary (a measure first introduced in StackGuard), which is placed 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, the exit code terminates the program with an error.

Many C/C++ compilers can detect inaccurate format strings. For example, the Gnu Compiler Collection 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. Note that the Ada community has spent many years in developing compilers with special compilation modes for high assurance software; these include modes that automate checking of language subsets, perform extended runtime checking, and support Design-by-Contract. (One example is the compiler included the AdaCore GNAT Pro High-Integrity Edition tool suite.)[42]

While type and format string checks are useful for detecting simple faults, they are not extensive or sophisticated enough to detect more complex vulnerabilities. There are compile-time tools that 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. An example is Flayer, an open source taint analysis logic and wrapper.[43] Other compilers include more extensive logic to perform full program verification to prove complex security properties based on formal specifications generated prior to compilation. Program verification compilers are most often used to detect flaws and "dangerous" constructs in C and C++ programs and

---

41 For more information, see: National Institute of Advanced Industrial Science (Tokyo, Japan) Technology Research Center for Information Security. "Fail-Safe C: a memory-safe compile for the C language". Accessed 21 January 2008 at: http://www.rcis.aist.go.jp/project/FailSafeC-en.html - *and* - the Fail-Safe C version 1.0 Webpage. Accessed 21 January 2008 at: http://homepage.mac.com/t.sekiguchi/fsc/index.html - *and* - the Memory Safe C Compiler Webpage. Accessed 21 January 2008 at: http://www.seclab.cs.sunysb.edu/mscc/

42 For information, see: http://www.adacore.com/home/gnatpro/development_solutions/safety-critical/

43 For more information see the Flayer Web page. Accessed 21 January 2008 at: http://code.google.com/p/flayer/

libraries, including constructs that leave the program vulnerable to format string attacks and buffer overflows.

Additional protections that be implemented at compile time are:

- Compiler randomization of variables and code positions in memory, particularly the randomization of the location of loaded libraries.
- Assembler preprocessors to reduce C and C++ program susceptibility to stack overflows.

  NOTE: An effective countermeasure to heap overflows is the malloc() *debugger.*

7.8.2 Safe software libraries

Libraries of "safe" and "secure" library routines generally work by detecting 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), which are then replaced with safe versions or alternatives of the called functions.

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. Two open source examples of "safe" libraries are the Safe C String Library and Libsafe.[44]

7.8.3 Runtime error checking and safety enforcement

Runtime protections can be applied to prevent buffer overflows in binaries executed under a particular operating system, or to dynamic runtime security analyses of compiled binaries.

Security wrappers and content validation filters can be applied to OSS code and binary executables to minimize the exposure of their vulnerabilities. By and large, security wrappers and validation filters are used to add content (input or output) filtering logic to programs that don't have that logic "built in". The wrappers intercept and analyze input to or output from the "wrapped" program, detect, and then remove, transform, or isolate content that is suspected of being malicious (*e.g.,* malicious code), or which contains unsafe constructs, such as very long data strings (associated with buffer overflows) and command strings associated with escalation of privilege. By and large, security wrappers and content validation filters must be custom-developed.

---

44 For more information see: Safe C String Library v1.0.3 Web page. Accessed 21 January 2008 at: http://www.zork.org/safestr/ - *and* - Avaya Labs' Libsafe research Webpage. Accessed 21 January 2008 at: http://www.research.avayalabs.com/gcm/usa/en-us/initiatives/all/nsr.htm&Filter=ProjectTitle:Libsafe&Wrapper=LabsProjectDetails&View=LabsProjectDetails

7.8.4 Code obfuscation

A deception measure to counter reconnaissance attacks and intellectual property violations, code obfuscators protect intermediate code, such as Java byte code, and runtime-interpreted source code, such as scripting code in Perl, PHP,[45] Python, JavaScript, AJAX, *etc.*, against decompilation, disassembly, and forms of reverse engineering. Obfuscation may also be used to protect intellectual property by preventing source code from being viewed and/or copied.

45 PHP is a recursive acronym for PHP Hypertext Processor; PHP originally stood for Personal Home Page.

8 RISK-BASED SOFTWARE SECURITY TESTING

Software security testing is not the same as testing the correctness and adequacy of security functions implemented by software, which are most often verified through requirements-based testing. While such tests are important, they reveal only a small piece of the picture needed to verify the security of the software.

Requirements-based testing is particularly inadequate when the software's specification does not adequately capture requirements directly related to dependability, trustworthiness, and survivability. If such requirements have been specified, requirements-based security testing can help verify the correctness and predictability of the software's secure behavior, which are prerequisites of dependability. The focus of correctness and predictability tests should be to demonstrate that the software can be shown to perform its specified functions—and only those functions—under both "normal" and abnormal (anomalous and hostile) conditions without compromising the secure behavior of the software itself, nor the security of its environment, data, and resources. Specific things to look for in such tests are whether dormant functions can be triggered, either inadvertently during normal execution or intentionally (*e.g.,* by submitting malicious input/attack patterns). "Consistently secure behavior" should be an unwaivable criterion for verifying correctness.

Requirements-based testing should always include tests that specifically determine the implemented software's conformance with its specified requirements for security constraints and protections such as sandboxing, code signature validation, input validation, output filtering, exception handling for intentionally-induced failures, *etc.*

Finally, requirements-based testing should verify that the implemented software conforms to its design, again with a focus on verifying the correct interpretation and implementation of designed security constraints and protections.

Note that a commitment to use formal methods in the specification of the software will enable a very high level of assurance in the validation of that software's conformance to its formal specification. This is obviously something that needs to be determined very early in the SDLC, and it will be driven in large part by the criticality/high consequence of the software, as well as its anticipated size and complexity. As noted in Section 3.2.5.2, formal methods are labor-intensive, and thus practical only for the most critical, high-consequence components of a software-intensive system.

Unfortunately, no amount of requirements-based testing can fully demonstrate that software does not contain vulnerabilities. Nor is requirements-based testing the best approach to determining how software will behave under anomalous and hostile conditions. This is because even the most robustly-specified security requirements are unlikely to address all possible conditions in which the software may be forced to operate in the real world. First, at least some of the assumptions under which the requirements were originally specified are very likely to be obsolete by the time the software is ready for testing. This is due, in part, to the changing nature of the threats to the software, and the new attack strategies and assistive

technologies that have emerged with the potential to target its vulnerabilities. These factors change often, and always much more quickly than any specification can keep up with. Moreover, if the software contains acquired components, the versions actually included in the implemented system may be different than those imagined when the software was architected. The new versions may contain more, fewer, and/or different vulnerabilities than those that shaped the assumptions under which the software was developed. For all these reasons, requirements-based testing should always be augmented with risk-based security testing.

The objectives of risk-based security testing are threefold:

1. To verify that the software's dependable operation continues even under hostile conditions, such as receipt of attack-patterned input, and intentional (attack-induced) failures in environment components;

2. To verify the software's trustworthiness, in terms of its consistently safe behavior and state changes, and its lack of exploitable flaws and weaknesses;

3. To verify the software's survivability, by verifying that its anomaly, error, and exception handling can recognize and safely handle all anticipated security-relevant exceptions and failures, errors, and anomalies; this means minimizing the extent and impact of damage that may result from intentional (attack-induced) failures in the software itself, and preventing the emergence of new vulnerabilities, unsafe state changes, *etc.* Secure software should not react to anomalies or intentional faults by throwing exceptions that leave it in an unsafe (vulnerable) state. Vulnerabilities are most likely to arise during the critical processing state changes that occur during start up, shutdown, and when the software is subjected to errors and anomalies.

Risk-based testing is predicated on the notion of "tester-as-attacker". The test scenarios themselves should be based on misuse and abuse cases, and should incorporate known attack patterns as well as anomalous interactions that seek to invalidate assumptions made by and about the software and its environment. In practical terms, this testing will focus on two areas of the software:

1. Its high-value components;

2. Its inter-component interfaces, and its extra-component interfaces.

Test techniques that are particularly useful for risk-based testing include

- Code security reviews, using static and dynamic analysis techniques. These reviews should include type checking and static checking to expose consequential and inconsequential security faults;

- White box and black box security fault injection, with fault propagation analysis;

- Fuzz testing;

- Penetration testing;
- Automated vulnerability scanning.

These test techniques are all discussed in Section 8.2.

SUGGESTED GENERAL RESOURCES ON SOFTWARE SECURITY TESTING

- BuildSecurityIn Security Testing resources. Accessed 21 January 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/testing.html
- *Software Security Assurance*, Section 5.5.
- Gallagher, Tom, Lawrence Landauer, and Bryan Jeffries. *Hunting Security Bugs.* Redmond, Washington: Microsoft Press, 2006.
- Wysopal, Chris, Lucas Nelson, Dino Dai Zovi, and Elfriede Dustin. The Art of Software Security Testing: Identifying Software Security Flaws. Cupertino, California: Symantec Press, 2006 —*and*— Wysopal, Chris, *et al.* "Finding software security flaws" (excerpt from *The Art of Software Security Testing). ComputerWorld*, 28 December 2006. Accessed 11 December 2007 at: http://www.computerworld.com/action/article.do?command=viewArticleBasic&articleId=9006870
- Dowd, Mark, John McDonald, and Justin Schuh. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities.* Indianapolis, Indiana: Addison-Wesley Professional, 2006.
- van der Linden, Maura A. *Testing Code Security.* Boca Raton, Florida: Auerbach Publications, 2007.
- Andrews, Mike and James A. Whittaker. *How to Break Web Software: Functional and Security Testing of Web Applications and Web Services.* Boston, Massachusetts: Addison-Wesley Professional, 2006.
- Stuttard, Dafydd and Marcus Pinto. *The Web Application Hacker's Handbook: Discovering and Exploiting Security Flaws.* Indianapolis, Indiana: Wiley Publishing, 2008.
- OWASP Testing Project page (includes OWASP Testing Guide v2). Accessed 11 December 2007 at: http://www.owasp.org/index.php/Category:OWASP_Testing_Project
- TestingSecurity.com—Teaching How to Perform Security Testing Webpage. Accessed 11 December 2007 at: http://www.testingsecurity.com/
- QASec.com—Software Security Testing in Quality Assurance and Development Webpage. Accessed 11 December 2007 at: http://www.qasec.com/

- Basirico, Joe. "Software security testing: Finding your inner evildoer". SearchSoftwareQuality.com, 6 August 2007. Accessed 11 December 2007 at: http://searchsoftwarequality.techtarget.com/originalContent/0,289142,sid92_gci1265911,00.html

- de Vries, Stephen. "Software Testing for Security". *Network Security*, Volume 2007, Issue 3, March 2007, pages 11-15.

- "Learning Guide: Application security testing techniques". SearchSoftwareQuality.com, 14 September 2006. Available at: http://searchsoftwarequality.techtarget.com/loginMembersOnly/1,289498,sid92_gci1215847,00.html

- Buchanan, Sam. "Web Application Security Testing". Based on presentation to Minnesota State Colleges and Universities IT Conference, April 2005. Accessed 11 December 2007 at: http://afongen.com/writing/Webappsec/2005/

- Dickson, John B. "Application Security: What does it take to build and test secure software?". Presented at Information Systems Audit and Control Association North Alabama Chapter meeting, 6 November 2006. Accessed 16 January 2008 at: http://www.bham.net/isaca/downloads/20061106_DenimGroup_Secure_SW_LG_Org.ppt

8.1 TEST PLANNING

The extent to which a security tester will be able to analyze a program's security properties and check for vulnerabilities depends on the kind of development artifacts, security evidence, and other information he/she has about the system being tested. If the tester has full access to the software's source code (ideally well-commented), specifications, design, and other technical documentation, he/she should be able to gain a fairly detailed understanding of its security properties and the assumptions under which it was developed and under which it operates.

If only the compiled executable and vendor-provided documentation are available, the tester will need to infer much of the information he/she needs by extrapolating it from observations of how the executing software behaves under as many conditions and inputs as can be exercised during testing.

A key goal of the test planner, then, will be to define the combination of tests, test scenarios, and test data that will reveal sufficient information about the software to enable the tester to make reliable judgments about how secure it will be once deployed.

The test plan should include:

- Security test cases and scenarios (based on the misuse/abuse cases and using the attack models developed during the requirements specification of the software);

- Test data (both meaningful and fuzzed) and a test oracle (if one is to be used);[46]
- Identification of the test tools and integrated test environment or "ecosystem" (if one is to be used); The Tool Survey on NIST's SAMATE Website[47] provides extensive information on the full range of software security testing tools available to support the various types of tests described in this document.
- Pass/fail criteria for each test;
- Test report template. This should enable capture and analysis of test results and of actions for addressing failed tests.

The misuse and abuse cases and attack/threat models developed early the software life cycle also provides basis for developing appropriately comprehensive and revealing security test scenarios, test cases, and test oracles. As noted in earlier discussions, attack patterns can provide the basis for much of the malicious input incorporated into those cases and models.

The test environment should as closely as possible duplicate the anticipated execution environment in which the software will be deployed. The test environment should be kept entirely separate from the development environment. If network connectivity is needed between the environments to transfer the software from one to the other, that network connection should be severed once the transfer has occurred. All of the preparation measures for moving software into deployment, as described in Section 9.1, and all of the measures taken to secure the operational execution environment, as discussed in Section 9.3, should be duplicated for the software to be tested and the test environment. These measures will ensure that the observations made of the software under test are as accurately indicative as possible of how the software will behave in "real world" operation.

All test data, test oracle, testing tools, integrated test environment, *etc.*, as well as the test plan itself and all test results (both raw results and test reports) should be maintained under strict configuration management control to prevent tampering or corruption.

8.1.1 Test timing

It has been demonstrated repeatedly that problems found early in the software life cycle are significantly easier and less costly to correct than problems discovered post-implementation or, worse, post-deployment. A thorough regiment of security reviews and tests should begin as early in the software's life cycle as is practicable, and should continue iteratively until the operational software is "retired".

---

46 An open source test tool of interest for developing attack patterns for test cases is the Metasploit attack exploit generation tool and library, which can be downloaded from: http://www.metasploit.com/

47 NIST SAMATE Tool Survey. Accessed 3 September 2008 at: https://samate.nist.gov/index.php/Tools

Figure 8-1 shows a suggested distribution of different security test techniques throughout various life cycle phases.

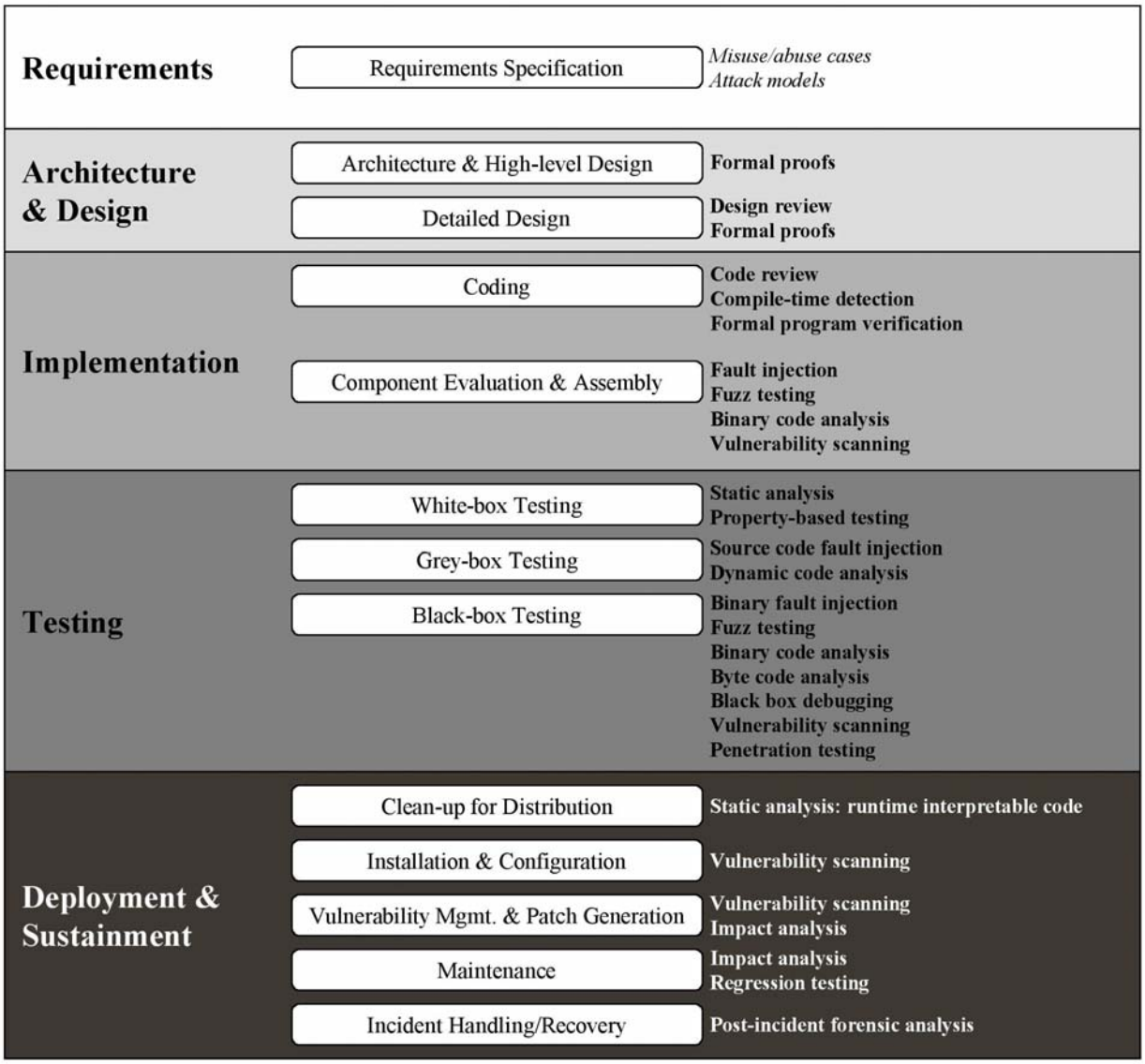


Figure 8-1. Suggested distribution of security test techniques throughout the SDLC

The distribution of security tests throughout the life cycle includes:

- Security reviews of requirements specification, architecture (including assembly option evaluation as discussed in Section 7), design, and development process/controls. This includes verifying that the design conforms with secure design principles;
- Security review of source code, including custom-developed code, open source code, reused legacy code, and, whenever available, source code of COTS, GOTS, and other reused binary components. This includes verifying the code's conformance with secure coding principles and standards;

- Black box security analysis and testing of binary executables;
- Post integration system security testing.

In addition to the specification, architecture, and design reviews discussed in Sections 4 and 5, Code reviews during the software implementation phase, and code reviews and security tests during the testing phase, will ensure that the following artifacts are thoroughly vetted:

1. **Source code modules:** Code security reviews should be performed (1) before accepting any OSS or reused source code components; (2) before compiling modules into object code units;
2. **Compiled object code units:** These should be subjected to white box fault injection tests with dynamic analyses that trace between source the code modules and the object code units compiled from them;
3. **Linked and functional subsystems and components:** These should be subjected to black box fault injection tests with fault propagation analyses;
4. **Integrated software system, pre-deployment:** This should be subjected to automated vulnerability scanning;
5. **Integrated software system, post-deployment:** This should be subjected to penetration testing, as well as iterative vulnerability scans and post-incident forensic analyses to identify new attack patterns, and emergent vulnerabilities, and residual vulnerabilities that are only observable "in the wild".

For all non-custom components, the security reviews and tests should be performed as part of their pre-acquisition/pre-reuse evaluation, *i.e., before* a commitment is made to acquire and reuse them.

Note that outsourcing to expert code security reviewers and software/application security testers may be necessary when in-house expertise is inadequate. Even when in-house expertise is sufficient, including additional security tests as part of Independent Verification and Validation (IV&V) is an extremely good idea, as independent testers are more likely to pinpoint problems overlooked by internal testers due to their familiarity with the software and its documentation.

8.1.2 System and Application Security Checklists

Software security testing needs to go far beyond verifying conformance to system or application security checklists. Indeed, such checklists are of very limited utility to the software security tester, because the usually incorporate system-level verifications of elements that are outside the realm of what is controlled or determined by the software. Moreover, few such checklists have been developed to reflect a software assurance practitioner's point of view.

This said, checklists can be good "thought provokers" during test scenario development, as they may draw attention to scenarios that address for conditions or vulnerabilities not thought of when misuse/abuse cases and attack models were being defined. To this end, the checklists listed below may prove helpful as a resource for test planners. Also, see Appendix D for software security-relevant extracts from two cyber-security checklists.

SOFTWARE AND APPLICATION SECURITY CHECKLISTS

- Internet Security Alliance and U.S. Cyber Consequences Unit. "Cyber Security Check List". Look specifically at Areas 2 and 6. Accessed 25 January 2008 at: http://www.isalliance.org/content/view/144/292
- DISA. *Application Security and Development STIG.* Draft Version 2, Release 0.1, 19 October 2007. Accessed 21 January 2007 at: http://iase.disa.mil/stigs/draft-stigs/application-security-dev-stigv2r0-1-102307.doc *—and— Application Security and Development Checklist*, Draft Version 1 Release 1, 20 April 2007. Accessed 21 January 2008 at: http://iase.disa.mil/stigs/draft-stigs/asd-checklist.doc
- DISA. *Application Security Checklist*, Version 2, Release 1.10, 16 November 2007. Accessed 17 December 2007 at: http://iase.disa.mil/stigs/checklist/application-security-checklist-v2r1-10.doc
- NASA Reducing Software Security Risk Through an Integrated Approach project. Security Checklist for the External Release of Software page. Accessed 17 December 2007 at: http://rssr.jpl.nasa.gov/ssc2/index.html
- Lopienski, Sebastian. "Conseil Européen pour la Recherche Nucléaire[48] Security checklist for software developers". Accessed 17 December 2007 at: http://info-secure-software.Web.cern.ch/info-secure-software/SecurityChecklistForSoftwareDevelopers.pdf
- Ollmann, Gunter. "Application Assessment Questioning". Accessed 17 December 2007 at: http://www.technicalinfo.net/papers/AssessmentQuestions.html
- Mehta, Dharmesh M. "Application Security Testing Cheat Sheet". SmartSecurity.blogspot.com, 2007. Accessed 11 December 2007 at: http://www.edocr.com/doc/264/application-security-testing-cheat-sheet
- Kobus, Walter S. "Total Enterprise Security Solutions Applications Security Checklist". Accessed 17 December 2007 at: http://www.tess-llc.com/Application%20Security%20ChecklistV4.pdf
- SANS (SysAdmin, Audit, Networking and Security) Institute Security Consensus Operational Readiness Evaluation Web Application Security Checklist Webpage. Accessed 21 January 2008 at: http://www.sans.org/score/Webappschecklist.php
- Beaver, Kevin. "Web Application Security Testing Checklist". SearchSoftwareQuality.com, 19 March 2007. Available at: http://searchsoftwarequality.techtarget.com/tip/0,289483,sid92_gci1247920,00.html

48 Now the European Organization for Nuclear Research.

- OWASP. *Testing Guide*, Version 2.0. Accessed 8 September 2008 at: http://www.lulu.com/content/1375886

8.2 SOFTWARE SECURITY TEST TECHNIQUES

TESTING TERMINOLOGY

White box: The source code is available for analysis.

Grey box: Both source code and executable binary are available for analysis.

Black box: Only the binary executable or intermediate bytecode is available for analysis.

White box analyses, and especially static code analyses (discussed in Section 7.1.15.1), are most useful when performed iteratively as the software is being written. This said, independent (non-developer) analyses of completed source code of high-consequence modules should also be included in the software's security test plan, as source code analysis is the only way to trace issues directly to the implemented source code at fault, and thereby identify exactly what needs to be fixed.

Black box tests are performed on the binary executable alone. The tests are limited to those that can "poke at" software from the outside to observe its state changes, behaviors, and outputs in response to changes in its environment and inputs from external entities (including changes associated with attacks). For COTS and GOTS binary components (and many reused legacy components), black box tests may be the only tests that are feasible (binary analysis being a technique that is too resource- and labor-intensive to be practical for anything except binaries that implement high-consequence functions). Black box tests, while extremely important, do not on their own provide a basis for identifying what part of a program can be rewritten to eliminate an identified vulnerability or insecure behavior. The actions taken to address failed box tests must necessarily focus on working around the problem as identified through testing, for example by filtering problematic inputs.

Grey box testing should be performed on all custom-developed software as well as software compiled from open source code. Grey box testing can confirm through actual observation whether assumptions made about the software's execution and the software's interactions with external entities during source code analysis and other white box tests are accurate. Runtime behaviors and interactions can never be adequately simulated or extrapolated through white box analysis alone.

SUGGESTED RESOURCES

- NIST. Software Assurance Metrics and Tool Evaluation Website. Accessed 19 December 2007 at: http://samate.nist.gov/index.php/Main_Page

8.2.1 White and grey box testing techniques

These tests are in addition to post-implementation static code analysis, which was discussed in Section 7.1.15.1.

8.2.1.1 Source code fault injection

Fault injection is a testing technique originated by the software safety community to used to induce stress in the software, create interoperability problems among components, and simulate faults in the execution environment, thereby revealing safety-threatening faults that are not made apparent by traditional testing techniques. Security fault injection extends standard fault injection by adding error injection, thereby enabling tester to analyze the security of the behaviors and state changes that result in the software when it is exposed to various perturbations of its environment data. These data perturbations are intended to simulate the types of faults that would result during unintentional user errors as well as intentional attacks on the software *via* its environment, as well as attacks on the environment itself.

A data perturbation is simply the alteration of the data the execution environment passes to the software, or that one software component passes to another. Fault injection can reveal both the effects of security faults on individual component behaviors, and the behavior of the system as a whole.

The tester uses a fault injection tool to maintain a list of candidate faults: these should be developed by a security expert so that they reflect likely "real world" data perturbations. The tool then "injects" the fault at any point in the executing code at which it encounters a particular call, with the specific fault to be injected selected based on the parameters to that call. Each fault is designed to modify the data that the environment returns to the executing software. When exposed to these faults, the software may behave differently than it would when receiving normal data; this deviant behavior represents a vulnerability that could be exploited.

There are two varieties of fault injection: source code fault injection and binary fault injection. In source code fault injection, the tester deterministically decides (based on information in the software's source code and environment) when various environment faults should be triggered. The tester then "instruments" the source code by non-intrusively inserting changes into the program that reflect the changed environment data that would result from those faults. The instrumented source code is then compiled and executed, and the tester observes the ways in which the executing software's state changes when the instrumented portions of code are executed. In this way, the tester can observe and even quantify the software's secure *vs.* non-secure state changes resulting from changes in its environment (including changes of the type associated with intentional errors and failures).

Also analyzed during source code fault injection are the ways in which faults propagate through the source code. Fault propagation analysis involves two source code fault injection techniques: extended propagation analysis and interface propagation analysis. The objective of both techniques is to trace how state changes resulting from a given fault propagate through the source code tree.

To prepare for fault propagation analysis, the tester must generate a fault tree from the program's source code. To perform an extended propagation analysis, the tester injects faults into the fault tree, then traces how each injected fault propagates through the tree. From this, he/she can extrapolate outward to anticipate the overall impact a particular fault may have on the behavior of the software as a whole.

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.

Source code fault injection is particularly useful in detecting incorrect use of pointers and arrays, use of dangerous calls, and race conditions. Like all source code analyses, is most effective when used iteratively throughout the code implementation process. When new threats (attack types and intrusion techniques) are discovered, the source code can be re-instrumented with faults representative of those new threat types.

Binary fault injection is most useful when performed as an adjunct to security penetration testing to enable the tester to obtain a more complete picture of how the software responds to attack.

Recognizing that a software program interacts with its execution environment though operating system calls, remote procedure calls, application programmatic interfaces, man-machine interfaces, *etc.*, binary fault injection involves monitoring the fault-injected software's execution at runtime. For example, by monitoring system call traces, the tester can decipher the names of system calls (which reveal the types of resources being accessed by the calling software); the parameters to each call (which reveal the names of the resources, usually in the first parameter, and how the resources are being used); and the call's return code/value (which reveals success or failure of the access attempt).

In binary fault injection, faults are injected into the environment resources that surround the program. 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.

Environment perturbation *via* fault injection provides the tester with several benefits:

1. The ability to simulate environment anomalies without understanding how those anomalies occur in the real world. This enables fault injection by testers who do not have a deep knowledge of the environment whose faults are being simulated. Fault injection can emulate environment anomalies without needing reference to how those anomalies occur in the "real world".

2. The tester can decide which environment faults to emulate at which times, thus avoiding the problem that arises when doing a full environment emulation in which the environment state when the software interacts with it may not be what is expected, or may not have the expected effect on the software's behavior.

3. Unlike penetration tests, fault injection tests can be automated with relative ease.

Binary fault injection tools include source and binary fault injectors and brute force testers. The main challenges in fault injection testing are determining the most meaningful number and combination of faults to be injected and, as with all tests, interpreting the results. Also, just because a fault injection does not cause the software to behave in a non-secure manner, or fail into a non-secure state, the tester cannot interpret this to mean that the software will be similarly "well-behaved" when exposed to the more complex inputs it typically receives during operational execution. For this reason, security tests that are performed with the software deployed in its actual target environment (*e.g.,* penetration tests and vulnerability scans) are critical for providing a complete picture of how the software will behave under such conditions.

SUGGESTED RESOURCES

- Ollmann, Gunter. "Second-order Code Injection: Advanced Code Injection Techniques and Testing Procedures". Undated whitepaper posted on his Weblog. Accessed 16 January 2008 at: http://www.technicalinfo.net/papers/SecondOrderCodeInjection.html

8.2.1.2 Dynamic code analysis

Dynamic code analysis entails the execution of the software, with the tester tracing the external interfaces in the source code to the corresponding interactions in the executing code, so that any vulnerabilities or anomalies that arise in the executing interfaces are simultaneously located in the source code, where they can then be fixed.

Unlike static analysis, dynamic analysis enables the tester to "exercise" the software in ways that expose vulnerabilities introduced by interactions with users and changes in the configuration or behavior of environment components. Because the software isn't fully linked and deployed in its actual target environment, these interactions and their associated inputs and environment conditions are essentially simulated by the testing tool.

Current dynamic analysis tools test only conditions that are likely to occur during the software's execution. This means that dynamic analysis enables the tester to validate assumptions about how the software is likely behave only under *anticipated* operating conditions. [49]

49 An example of a dynamic code analysis toolset is the open source Valgrind, which can be downloaded from: http://valgrind.org/

SUGGESTED RESOURCES

- Stytz, Martin R. and Sheila B. Banks. "Dynamic Software Security Testing". *IEEE Security and Privacy*, Volume 4 Issue 3, May 2006, pages 77-79.
- Secure Software, Inc. "Risk in the Balance: How the Right Mix of Static Analysis and Dynamic Analysis Technologies Can Strengthen Application Security". 2004. Accessed 3 January 2008 at: http://secureitalliance.org/blogs/files/164/1137/Risk%20in%20the%20Bal_wp.pdf

8.2.1.3 Property-based testing

Property-based testing is a formal analysis technique developed by University of California at Davis. It is intended for use after the software's functionality has been implemented, to narrowly examine desirable security-relevant properties revealed by the source code, such as the absence of insecure state changes. The test then compares these desirable properties in the code against the corresponding portions of the software's requirements specification and design, to aid the tester in determining whether the security assumptions reflected in the specification and design have, in fact, held true the implemented code.

Property-based testing is detailed and time-consuming, so as with direct code analysis, it will probably be limited in usefulness to the small subset of the overall software code base that implements high-consequence functions. To be effective, the property-based tests themselves must be formally verified to be complete.

8.2.2 Black box testing techniques

The following tests can be performed when only binary executables are available for analysis, making them the only tests possible for the vast majority of COTS and GOTS software, as well as for legacy components for which source code is no longer accessible. Black box tests should also be performed on the executables compiled from custom-developed source or open source code.

8.2.2.1 Binary fault injection

Binary fault injection is most useful when performed as an adjunct to security penetration testing to enable the tester to obtain a more complete picture of how the software responds to attack.

Recognizing that a software program interacts with its execution environment though operating system calls, remote procedure calls, application programmatic interfaces, man-machine interfaces, *etc.,* binary fault injection involves monitoring the fault-injected software's execution at runtime. For example, by monitoring system call traces, the tester can decipher the names of system calls (which reveal the types of resources being accessed by the calling software); the parameters to each call (which reveal the names of the resources, usually in the first parameter, and how the resources are being used); and the call's return code/value (which reveals success or failure of the access attempt).

In binary fault injection, faults are injected into the environment resources that surround the program. 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.

Environment perturbation *via* fault injection provides the tester with several benefits:

1. The ability to simulate environment anomalies without understanding how those anomalies occur in the real world. This enables fault injection by testers who do not have a deep knowledge of the environment whose faults are being simulated. Fault injection can emulate environment anomalies without needing reference to how those anomalies occur in the "real world".

2. The tester can decide which environment faults to emulate at which times, thus avoiding the problem that arises when doing a full environment emulation in which the environment state when the software interacts with it may not be what is expected, or may not have the expected effect on the software's behavior.

3. Unlike penetration tests, fault injection tests can be automated with relative ease.

Binary fault injection tools include source and binary fault injectors and brute force testers. The main challenges in fault injection testing are determining the most meaningful number and combination of faults to be injected and, as with all tests, interpreting the results. Also, just because a fault injection does not cause the software to behave in a non-secure manner, or fail into a non-secure state, the tester cannot interpret this to mean that the software will be similarly "well-behaved" when exposed to the more complex inputs it typically receives during operational execution. For this reason, security tests that are performed with the software deployed in its actual target environment (*e.g.,* penetration tests and vulnerability scans) are critical for providing a complete picture of how the software will behave under such conditions.

SUGGESTED RESOURCES

- Wysopal, Chris, *et al.* "Testing Fault Injection in Local Applications". SecurityFocus, 23 January 2007 (excerpt from *The Art of Software Security Testing).* Accessed 14 December 2007 at: http://www.securityfocus.com/infocus/1886

- Du, Wenliang and Aditya P. Mathur. "Vulnerability Testing of Software Systems Using Fault Injection". Technical Report COAST TR98-02, 6 April 1998. Accessed 2 January 2008 at: https://www.cerias.purdue.edu/tools_and_resources/bibtex_archive/archive/98-02.pdf

- Sivaramakrishnan, Hariharan. *On the Use of Fault Injection to Discover Security Vulnerabilities in Applications.* University of Maryland master of science thesis, May

2006. Accessed 31 December 2007 at: https://drum.umd.edu/dspace/bitstream/1903/3566/1/umi-umd-3404.pdf

8.2.2.2 Fuzz testing

As in binary fault injection, fuzz testing inputs random invalid data (usually produced by modifying valid input) to the software *via* its environment or *via* another software component. Fuzz testing is implemented a "fuzzer"—a program or script that submits a combination of inputs to the software to reveal how that software responds. Fuzzers are generally specific to a particular type of input, such as HTTP input, and are written to be used to test a specific program; they are not easily reusable. But their value is in their specificity, because they can often reveal security vulnerabilities that generic testing tools such as vulnerability scanners and fault injectors cannot.

Effective fuzz testing requires the tester to have a thorough understanding of the software being tested, and how that interfaces with external entities whose data will be simulated by the fuzzer. As with other simulation-based security tests, the software's secure behavior in the face of fuzzed input data should not be interpreted as entirely indicative of how the software will behave when exposed to more complex real-world inputs.

SUGGESTED RESOURCES

- Sutton, Michael, Adam Greene, and Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery.* Indianapolis, Indiana: Addison-Wesley Professional, 2007.

8.2.2.3 Binary code analysis

Tools that support reverse engineering and analysis of binary executables include decompilers, disassemblers, and binary code scanners, reflecting the varying degrees of reverse engineering that can be performed on binaries:

The least intrusive technique is binary scanning. Binary scanners, such as those used by Veracode, analyze machine code to model a language-neutral representation of the program's behaviors, control and data flows, call trees, and external function calls. Such a model may then be traversed by an automated vulnerability scanner in order to locate vulnerabilities caused by common coding errors and simple back doors. A source code emitter can use the model to generate a human-readable source code representation of the program's behavior, enabling manual code review for design-level security weaknesses and subtle back doors that cannot be found by automated scanners.

The next least intrusive technique is assembly, in which binary code is reverse engineered to intermediate assembler. The disadvantage of disassembly is that the resulting assembler code can only be meaningfully analyzed by an expert who both thoroughly understands that particular assembler language and who is skilled in detecting security-relevant constructs within assembler code;

The most intrusive reverse engineering technique is decompilation, in which the binary code is reverse engineered all the way back to source code, which can then be subjected to the same security code review techniques and other white box tests as original source code. Note, however, that decompilation is technically problematical: the quality of the source code generated through decompilation is often very poor. Such code is rarely as navigable or comprehensible as the original source code, and may not accurately reflect the original source code. This is particularly true when the binary has been obfuscated or an optimizing compiler has been used to produce the binary. Such measures, in fact, may make it impossible to generate meaningful source code. In any case, the analysis of decompiled source code will always be significantly more difficult and time consuming than review of original source code. For this reason, decompilation for security analysis only makes sense for the most critical of high-consequence components.

Reverse engineering may also be legally prohibited. Not only do the vast majority of software vendors' license agreements prohibit reverse engineering to source code and assembler) form, software vendors have repeatedly cited the Digital Millennium Copyright Act (DMCA) of 1999 to reinforce such prohibitions, even though the DMCA explicitly exempts reverse engineering as well as "encryption research" (which involves intentional breaking of encryption applied to the software being reverse-engineered) from its prohibitions against copy-protection circumvention.[50]

SUGGESTED RESOURCES

- Eliam, Eldad. *Reversing: Secrets of Reverse Engineering.* Indianapolis, Indiana: Wiley Publishing, 2005.
- McGraw, Gary and Greg Hoglund. "Chapter 3: Reverse Engineering and Program Understanding" (excerpt from *Exploiting Software: How to Break Code).* Accessed 19 January 2008 at: http://www.amazon.com/Exploiting-Software-Break-Addison-Wesley-Security/dp/0201786958/ref=pd_sim_b_title_3
- Wysopal, Chris. "Putting trust in software code". *USENIX ;login:*, Volume 29, Number 6, December 2004. Accessed 26 December 2007 at: http://www.usenix.org/publications/login/2004-12/pdfs/code.pdf
- Carnegie Mellon University Software Engineering Institute. Function Extraction Webpage. Accessed 21 January 2008 at: http://www.cert.org/sse/fxmc.html

8.2.2.4 Byte code analysis

The Java language is compiled into a platform-independent byte code format. Much of the information contained in the original Java source code is preserved in the compiled byte code, thus making decompilation by attackers easy. Byte code scanners enable the tester to examine the byte code for presence of information that would be useful in a reconnaissance attack.

50 See 17 U.S.C. (United States Code) § 1201(f) and (g) for the language of these exemptions.

8.2.2.5 Black box debugging

When only binary is available, and particularly when that binary was compiled from code with no compiler symbols or debug flags set (which would ease reverse engineering), traditional debugging is not possible. Black box debugging, however, provides a technique whereby the analyst can monitor behaviors external to the binary component or system while it is executing, and thereby observe the data that passes between that component/system and external entities.

By observing how data passes across the software's boundary, the analyst can also determine ways in which externally-sourced data might be manipulated to force the software down certain execution paths, or to cause the software to fail, in this way revealing errors and failures that originate not in the software itself, but which are forced by the external entities with which it interacts, or by an incorrectly implemented programmatic interface (errors in interface implementations are often the result of the developer relying on inaccurate documentation).

SUGGESTED RESOURCES

- Whittaker, James A. and Herbert H. Thompson. "Black Box Debugging". *Queue*, Volume 1 Number 9, December/January 2003-2004.

8.2.2.6 Vulnerability scanning

Automated vulnerability scanning is supported for application-level software, as well as for Web servers, database management systems, and some operating systems. Application vulnerability scanners[51] are the most useful for software security testing. These tools scan the executing application software for input and output of known patterns that are associated with known vulnerabilities. These vulnerability patterns, or "signatures", are comparable to the signatures searched for by virus scanners, or the "dangerous coding constructs" searched for by automated source code scanner, making the vulnerability scanner, in essence, an automated pattern-matching tool.

While they can find simple patterns associated with vulnerabilities, automated vulnerability scanners are unable to pinpoint risks associated with aggregations of vulnerabilities, or to identify vulnerabilities that result from unpredictable combinations of input and output patterns.

---

51 An open source Web vulnerability scanner of interest is the OWASP's WebScarab which can be downloaded from: http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project. A number of commercial scanners are also available, almost exclusively targeting Web applications and Web services *vs.* other types of software. Examples include Watchfire's AppScan, SPI Dynamics' WebInspect, N-Stalker's Web Application Security Scanner, Acunetix's Web Vulnerability Scanner, Cenzic's Hailstorm, and NT Objectives' NTOSpider.

In addition to signature-based scanning, some Web application vulnerability scanners attempt to perform "automated stateful application assessment" using a combination of simulated reconnaissance attack patterns and fuzz testing techniques to "probe" the application for known and common vulnerabilities. Like signature-based scans, stateful assessment scans can detect only known classes of attacks and vulnerabilities.

This said, most vulnerability scanners do attempt to provide a mechanism for aggregating vulnerability patterns. The current generation of scanners is able to perform fairly unsophisticated analyses of risks associated with aggregations of vulnerabilities. In many cases, especially with COTS vulnerability scanners, the tools also provide information and guidance on how to mitigate the vulnerabilities they do detect.

Typical application vulnerability scanners are able to identify only 30% of the types of vulnerabilities that exist in large applications: they focus on vulnerabilities that need to be truly remediated *vs.* those that can be mitigated through patching. As with other signature-based scanning tools, application vulnerability scanners usually have a high false positive rate, unless recalibrated by the tester, in which case they may swing too far in the other direction, and manifest an excessive false negative rate. In both cases, the tester must have enough software and security expertise to meaningfully interpret the scanner's results to weed out the false positives and negatives, so as not to identify as a vulnerability what is actually a benign issue, and not to ignore a true vulnerability that has been overlooked by the tool. This is why, as stressed before, it is important to combine tests to examine the software for vulnerabilities in a variety of ways, none of which is adequate on its own, but which in combination can greatly increase the likelihood of vulnerabilities being found.

Because automated vulnerability scanners are signature-based, as with virus scanners, they need to be frequently updated with new signatures from their vendor. Two important evaluation criteria for selecting a vulnerability scanner are (1) how extensive the tool's signature database is, and (2) how often the supplier issues new signatures.

Vulnerability scanners are most effective when used:

1. During the security assessment of binary components prior to acquisition/reuse;

2. Before penetration testing, in order to locate straightforward common vulnerabilities, and thereby eliminate the need to run penetration test scenarios that check for such vulnerabilities.

In the software's target environment, vulnerabilities in software are often masked by environmental protections such as network- and application-level firewalls. Moreover, environment conditions may create novel vulnerabilities that cannot be found by a signature-based tool, but must be sought using a combination of other black box tests, most especially penetration testing of the deployed software in its actual target environment.

As noted earlier, in addition to application vulnerability scanners, there are scanners for operating systems, databases, Web servers, and networks. Most of the vulnerabilities checked

by these other scanners focus on configuration deficiencies and information security vulnerabilities (*e.g.,* is data disclosed that should not be). However, these scanners do often look for conditions such as buffer overflows, race conditions, privilege escalations, *etc.,* at the environment level that can have an impact on the security of the software executing in that environment; therefore, it is a good idea to run execution environment scanners with an eye towards observing problems at the environment's interfaces with the hosted application, in order to get an "inside out" view of how vulnerabilities in the environment around the application may be exploited to "get at" the application itself.

SUGGESTED RESOURCES

- BuildSecurityIn Black Box Testing tools resources. Accessed 21 January 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/tools/black-box.html
- Anantharaju, Srinath. "Automating Web application security testing". Google Online Security Blog, 16 July 2007. Accessed 11 December 2007 at: http://googleonlinesecurity.blogspot.com/2007/07/automating-Web-application-security.html
- Suto, Larry. "Analyzing the Effectiveness and Coverage of Web Application Security Scanners". October 2007. Accessed 14 July 2008 at: http://www.stratdat.com/Webscan.pdf
- Grossman, Jeremiah. "The Best Web Application Vulnerability Scanner in the World". On his Weblog, 23 October 2007. Accessed 14 July 2008 at: http://jeremiahgrossman.blogspot.com/2007/10/best-Web-application-vulnerability.html

8.2.2.7 Penetration testing

In penetration testing, the whole software system in its "live" execution environment is the target of the tests. Penetration testing observes whether the system resists attacks successfully, and how it behaves when it cannot resist an attack.

Penetration testing should focus on those aspects of system behavior, interaction, and vulnerability that cannot be observed through other tests performed outside of the live operational environment. Penetration testers should subject the system to sophisticated multi-pattern attacks designed to trigger complex series of behaviors across system components, including non-contiguous components, as these are the types of behaviors that cannot be forced and observed by any other testing technique. Penetration testing should also attempt to find security problems that are likely to originate in the software's architecture and design (*vs.* coding flaws that manifest as vulnerabilities), as it is this type of problem that tends to be overlooked by other testing techniques.

The penetration test plan should include "worst case" scenarios that reproduce threat vectors (attacks, intrusions) that are considered highly damaging, such as insider threat scenarios. The test plan should capture:

- The security policy the system is supposed to respect or enforce;

- Anticipated threats to the system;
- The sequences of likely attacks that are expected to target the system.

Actual test techniques employed by penetration testers include: spidering, querying for known vulnerable scripts or components, testing for conditions like forceful browsing, directory traversal, running input validation checks, and using the results of spidering to identify all points of user input to test for flaws like SQL injection, cross-site scripting, CSRF, command execution, *etc.* A combination of fuzzing and injection of strings known to cause error conditions may be used.

SUGGESTED RESOURCES

- BuildSecurityIn Penetration Testing resources. Accessed 21 January 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/penetration.html
- Andreu, Andres. *Professional Pen Testing for Web Applications.* Indianapolis, Indiana: Wiley Publishing, 2006.
- Thompson, Herbert H. "Application Penetration Testing". *IEEE Security and Privacy*, Volume 3 Number 1, January/February 2005, pages 66-69.

8.3 INTERPRETING AND USING TEST RESULTS

As soon as each test is completed, test results and the exact version of the software artifact tested should be checked in to the configuration management.

The results of the requirements- and risk-based software security testing regimen should provide adequate information for the software's developers to:

- identify missing security requirements that need to be evaluated to determine the risk posed to the software if those requirements are not added to the specification and the software is not modified to implement them. These could include requirements for constraints, countermeasures, or protections to eliminate or mitigate weaknesses, vulnerabilities, and unsafe behaviors/state changes;
- determine the amount and nature of reengineering (or refactoring) needed to satisfy the new requirements.

Note that just because software successfully passes all of its security tests, this does not mean that novel attack patterns and anomalies will never arise in deployment to compromise the software. For this reason, iterative testing throughout the software's life time is imperative to ensure that its security posture does not degrade over time.

In terms of providing meaningful security metrics for risk management, the aggregation of test results should provide an indication of whether the software is able to resist, withstand, and/or recover from attacks to the extent that risk can be said to be mitigated to an acceptable

level. This is the best that can be hoped for given the current state of the art of software security test technique and tools.

SUGGESTED RESOURCES

- Alhazmi, O.H., Y.K. Malaiya, and I. Ray. "Measuring, analyzing and predicting security vulnerabilities in software systems". *Computers and Security*, Volume 26 Issue 3, May 2007, pages 219-228. Preprint version accessed 26 December 2007 at: http://www.cs.colostate.edu/~malaiya/pub/com&security_darticle.pdf

- Jones, Jeffrey R. "Estimating Software Vulnerabilities". *IEEE Security and Privacy*, July/August 2007, pages 28-32. Accessed 11 September 2008 at: http://www.computer.org/portal/cms_docs_security/security/2007/n4/j4jones.pdf

9 SECURE DISTRIBUTION, DEPLOYMENT, AND SUSTAINMENT

This section discusses what can be done to prepare for distribution, deployment, and sustainment to ensure that software remains secure throughout these phases, until it is retired.

9.1 PREPARATIONS FOR SECURE DISTRIBUTION

The following are sound practices for preparing the software to be distributed/deployed.

- Before deployment, change any default configuration settings, *etc.,* not already addressed prior to whole-system testing. Distribute all software in a default configuration that is as secure and restrictive as possible. Document a set of secure configuration instructions, to be delivered with the software, that explain the risk associated with each possible change to a secure default. This includes:
    - Setting the default parameters for the software's environment to different values than provided to those parameters in the development environment; this will prevent developers from being able to see any details of the deployed software once it has been installed its operational environment.
    - Setting all default account passwords to unpredictable values. Moreover, the software should be written to force the administrator or user to reset all default passwords upon first software invocation. There should be no way for the default passwords to be retained after that first invocation.
    - Setting the default privileges assigned to the software's executable files to be execute-only for any role other than "administrator".
- Deliver all default passwords in encrypted form "out of band", *i.e.,* separate from software itself.
- Provide an automated installation tool with a dialogue that prompts the installer to set OS directory privileges as restrictively as possible.
    - Establish a means for strong authentication of the individual who will run the 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.

- The configuration interfaces provided by the tool or installation script should be clear and secure. 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".

- Review and "sanitize" all user-viewable source code (*e.g.*, client-side Web application code), and apply countermeasures to source code copying, if desired. See Section 9.1.1 for specific practices to this end.

SUGGESTED RESOURCES

- *Software Security Assurance*, Section 5.6.

9.1.1 Review and sanitization of user-viewable source code

If a client includes a function such as a browser's "view source" function, plaintext HTML code and embedded scripting code are made visible to the user by that function should be examined to ensure that it cannot be assist the reconnaissance attacker by increasing his/her knowledge of the software component specifics that would help him/her determine whether the component contained known vulnerabilities, or of the platform's directory structures that could be targeted, or any information that could be leveraged in a social engineering attack.

Content that should not be included in viewable HTML and scripting code includes:

1. **Sensitive comments:** Comments that include sensitive or potentially exploitable information, such as information about the file system directory structure, problems associated with the software's development, the software's configuration, release, and version details (COTS and OSS components), the location of root, cookie structures, or personal information about the code's developers (names, email addresses, phone numbers, or any other information the disclosure of which would compromise the developer's privacy). Of particular concern are comments within source code that can be viewed by attackers using a Web browser.

   The following types of comments may be included in user-viewable source files:

   - **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:** Comments automatically added to viewable source pages by many commercial Web application 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 the HTTP daemon *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 less complicated filter can be written to locate and strip out *all* comments from user-viewable source code. In the case of automatically-generated comments, an active filter may be required to locate and remove comments on an ongoing basis after the application has been deployed. This is particularly true of code that will be maintained using the same package that originally generated it because the package is likely to add undesirable comments when it is used to generate new versions of the code.

2. **Pathname references:** Pathname references can 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 COTS or OSS component 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 relational database management system 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.

   Relative pathnames are a particular issue. If the software calls a library or another component, that call should be explicit and specific. The software should not contain calls to relative pathnames (pointing to the current directory) nor rely or search paths (such constructs render network-based software vulnerable to cross-site scripting, directory traversal, and similar attacks). Full pathnames should be used for URL/URIs and other file paths that users will reference. As search rules for dynamic link libraries and other library routines become more complex, vulnerabilities will be more easily introduced into system routines that can parse filenames that contain embedded spaces.

   Other pathnames of concern are those indicating paths directories that are not explicitly intended to be accessed by users and those indicating paths to unreferenced, hidden, and unused files. If the software includes user-viewable source code, all

pathname/Uniform Resource Identifier (URI) references that point to unused and hidden files that could be exploited in enumeration attacks in which the attacker searches for files or programs that may be exploitable or otherwise useful in constructing an attack.

3. **Explicit and implicit debug information or elements:** Web applications should have been implemented to validate the content of name-value pairs within the URL or URI submitted by the user. Because of this validation, the 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

   This change results in the inserted "debug=on" command forcing the application into debug mode, which then enables the attacker to observe its behavior more closely, *i.e.,* to discover exploitable faults or other weaknesses.

   Debug constructs may also be planted within the HTML, eXtensible 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.

   If the source code containing the implicit debugger commands is user-viewable, such as that of Web pages in HTML/XHTML, Java Server Pages (JSP), or ASP (Active Server Pages), as well as 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 well aware to such obfuscation attempts (which constitute "security through obscurity"; please note that security through obscurity is inadequate to hinder any but the most casual and novice attackers).

4. **Hard-coded credentials:** Basic authentication should not have been used in a Web application, even over SSL/TLS-encrypted connections. However, if basic authentication has been used, hard-coded credentials may appear in the application's HTML or XHTML pages or other user-viewable source code. Such credentials must be flagged by testers so they can be removed before deployment.
5. **Data-collecting trapdoors:** Non-policy-compliant[52] cookies must be located, along with "Web bugs",[53] spyware, and trapdoors (particularly 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. All <IMG> tags in HTML/XHTML code should be checked to ensure that they do not implement Web bugs.

The source code reviewer should also keep an eye out for any other information that could be exploited by an attacker to target the software, its data, or its environment.

9.1.1.1 Preventing disclosure of user-viewable source code and copying of all browser-displayed content

---

52 Federal government policy (Office of Management and Budget 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 Websites operated by the federal government or by contractors on behalf of federal agencies, unless certain conditions are met:

- There is a compelling need to gather the data on the site;
- Appropriate, publicly disclosed privacy safeguards have been implemented to handle information extracted from cookies;
- The head of the agency owning the Website has personally approved use of data collecting cookies.

This policy was particularized for U.S. DoD in the Office of the Secretary of Defense memorandum, dated 13 July 2000, "Privacy Polices and Data Collection on DoD Public Websites" (13 July 2000).

53 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">
```

In addition to removing sensitive content from user-viewable source code, some access control measures may be implemented to protect such code, as well as other Web content, from being copied and misused (*e.g.,* Web page defacement; plagiarism of Web content). These techniques cannot prevent determined users from manipulating viewable source code, even if it means printing the content of a Web page and scanning it with an optical character reader or retyping it. However, these techniques should inhibit less determined, less resourceful attackers and casual plagiarists from:

- Copying HTML source code;
- Cutting and pasting text content;
- Screen capture.

To inhibit copying of user-viewable HTML content, including viewable source code, consider use of an HTML authoring tool or add-on that enables source code encryption (*e.g.,* Authentica's NetRecall, Andreas Wulf Software's HTML Guard).

Unfortunately, there is no way the Web server can prevent the browser from being able to display HTML source code. The user can choose to turn off source viewing in the browser, but this cannot be controlled by the server. If HTML source viewing is seen as a major problem, you may want to code the main elements of the Website (navigation, header, *etc.*) in a Java applet, if possible, instead of using HTML. Unlike HTML source, Java applet source code, as well as CGI source code, cannot be displayed using a browser's *VIEW SOURCE* function (though use of a Turing technology like Java *vs.* a non-Turing technology like HTML presents its own security issues).

Another countermeasure for inhibiting the cutting and pasting of Web content is to serve content as Portable Document Format (PDF) files instead of HTML. Although it is possible to cut and paste PDF files downloaded and displayed in an Acrobat Reader (*versus via* the Acrobat browser plug-in, which does prevent PDF cut and paste), the process is somewhat awkward and may inhibit casual plagiarism.

An even more effective approach is to scan any text documents that you do not want to be cut and paste into image files such as Graphics Interchange Format (GIF) or Joint Photographic Experts Group (JPEG, or JPG) files, which would enable the text to be served as images and thereby bypass any text-based filtering software. While these image files in their entirety can be copied and pasted, the text within them cannot be extracted electronically.

None of these technical countermeasures can prevent a user from printing a hard copy of a Web page (even one posted as an image file) and scanning it with an optical character reader. However, some COTS products designed to control copyright-protected material do advertise the ability to prevent browsers from sending protected Web content to a printer. However, even such products cannot prevent a determined user from simply re-keying the text contained in a Web page.

To prevent screen capture by a browser, write a plug-in that wraps the system-level commands that implement the browser's screen capture function. This wrapper plug-in, when installed in the browser, will effectively disable those commands and thus, prevent the screen capture process from occurring.

9.1.2 Bytecode obfuscation to deter reverse engineering

Bytecode obfuscation is a technique designed to help protect Java bytecode from decompilation. Preventing bytecode decompilation is a countermeasure both against disclosure and tampering (*i.e.,* confidentiality and integrity issues).

9.2 SECURE DISTRIBUTION

Several of the principles and practices of trusted distribution as described in the National Computer Security Center's *Guide to Understanding Trusted Distribution in Trusted Systems* (see reference at end of this section) 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 COTS 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.,* HTTP-Secure 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.

9.2.1 Protection for online distributions

If the software will be distributed *via* a network download, establish a protected channel for distribution. This protection should include at a minimum encryption, digital hash and/or digital watermarking of the package to be downloaded (to enable tamper-detection), and digital code signature to provide.

Authentication of the individual downloading the software may also be desirable, in which case a mechanism needs to be provided that uses a realistic basis for identification and authentication, such as the name and purchase/order number associated with the individual. Digital rights management may be used to deter tampering (*vs.* providing tamper-detection).

While code signatures cannot guarantee that the code was not malicious to begin with, or is otherwise error-free, they do provide validated evidence to the entity (user or process) that

will execute the software that the code came from a trusted source. Code signatures also act as tamper-detection mechanisms. Code signatures cannot, however, attest to the quality or trustworthiness of the entity (person or signature application) that affixed the signature to the code.

9.2.2 Protection for offline distributions

If the software will be distributed on physical storage media, the easiest way to tamperproof the content is to use a storage medium that is non-rewritable (*e.g.,* non-rewritable compact disc-read-only memory [CD-ROM]). Use of CD-ROM has the additional advantage of being the least expensive physical storage medium. It may also be desirable to apply a digital code signature so that the installer can verify that the code on the CD-ROM came from the expected, trusted source, and was not corrupted or tampered with while it was being written to the disc.

SUGGESTED RESOURCES

- Menendez, James N. and Scott Wright. *A Guide to Understanding Trusted Distribution in Trusted Systems* (the "Dark Lavender Book"). NCSC-TG-008, 15 December 1988. Accessed 19 December 2007 at: http://handle.dtic.mil/100.2/ADA392816 —*and*— http://www.fas.org/irp/nsa/rainbow/tg008.htm

9.3 SECURE INSTALLATION AND CONFIGURATION

The software may have been designed and developed to be extremely secure, but it will not remain secure if its configuration parameters are not set as the designer intended. Similarly, the configuration parameters of its execution environment must be set so the software is not unnecessarily exposed to potential threats.

The main purpose of reconfiguration of the software upon installation is to reset the default parameters set in the delivered software; this includes establishing and setting access control parameter values for the accounts, roles, groups, file system directories, *etc.* associated with the software in its installed environment.

The installation documentation for the software should specify configuration parameters that are as restrictive as possible, to make sure the software is as resistant as possible to anticipated attacks and exploits. The installation documentation should also strongly encourage the use of secure installation procedures, not just for the initial installation, but for every additional component, update, patch, *etc.* installed after initial deployment.

Ideally, the configuration procedures will require the administrator or user to explicitly approve the software installation before that installation can occur. This is of particular importance for software that is retrieved *via* download.

The installation and configuration procedures for the software system should include the specific instructions described in the sections below.

If there is an operational security team responsible for "locking down" the execution environment, the developer should provide that team with any system-specific configuration settings of COTS, GOTS, OSS, or legacy execution environment components. Such settings include all additional environment constraints required above and beyond those in the operational security team's standard "locked down" configuration.

Moreover, the developer's configuration information for the security team should document any circumstances in which the violation of one of the "lock down" configuration parameters was truly unavoidable, *i.e.*, because the software could not be implemented in a way that would enable it to run correctly with that particular environment configuration setting (*e.g.,* a particular service or interface disabled).

9.3.1 Instructions for configuring restrictive file system access controls for initialization files and 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.

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).

9.3.2 Instructions for validating install-time 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 (OS) 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.

9.3.3 Instructions for removing all unused and unreferenced files

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;
- Dynamic link libraries, extensions, and any other type of executable that is not explicitly allowed.

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.

9.3.4 Instructions for changing of passwords and account names on default accounts

Note that a lot of COTS 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.

Some software vendors have begun delivering their products with default accounts locked and expired, to force the installer to change them. Others provide tools to scan for the default passwords. For example, the Oracle Installer automatically locks and expires most accounts and/or forces password resets on newer Oracle products, while the Oracle Default Password Scanner that can be run against legacy Oracle installations to locate unchanged default passwords.

9.3.5 Instructions for deleting unused default accounts

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" the Web server's correct operation, its "nobody" account should be renamed to something less obvious.

Instructions for ensuring the confidentiality of configuration data and include files: Place *include* files and configuration files outside of the Web documentation root in the Web server directory tree. This will prevent the Web server from serving these files as Web pages. For example, on the Apache Web server, add a *handler* or an *action* for *.inc* (*include*) files:

```
<Files *.inc> Order allow,deny Deny from all</Files>
```

This places the include files in a protected directory (*e.g., .htaccess*), and designates them as files that will not be served.

Use a filter to deny access to the files. For example, on the Apache Web server use:

```
<Files ~ "\.phpincludes"> Order allow,deny Deny from all</Files>
```

If full regular expressions must match filenames, also use the Apache *FilesMatch* directive.

If the include file is a valid script file to be parsed by the server, make sure it is designed securely, and does not act on user-supplied parameters. In addition, change all file permissions to eliminate world-readable permissions. Ideally, the permissions will be set so that only the *uid/gid* of the Web server can read the files.

Regardless of these configuration parameters, an attacker who is able to get the Web server to run his/her own scripts to access the files will be able to circumvent such permission limitations. One countermeasure to this problem is to run different copies of the Web server program: one for trusted users, and a second for untrusted users, each with appropriate permissions. This approach is difficult to administer, however. Yet, if the perceived threat is great, the additional administrative overhead may be worth it. Knowing that two (or more) versions of the program may be run at different sensitivity levels, the developer may want to tailor each version to its specific intended environment, *e.g.,* to provide greater integrity assurance for the less sensitive version that will be subjected to greater risk from its untrusted users than the more sensitive version will from its trusted users.

9.3.6 Other considerations for locking down the execution environment

The software's installation instructions may need to include environment lockdown procedures, if no such guidelines/procedures already exist elsewhere. Specific environment lock-down considerations include:

- Configuring the required security protections and services, and their interfaces, for protecting or interacting with the software;
- Disabling all non-essential services;
- Configuring the available file system directory access controls, virtual machine monitor, TPM, or other environment compartmentalization mechanisms to isolate the trusted components of the software from its untrusted components and from other higher-risk entities on the same host, and to constrain the execution of untrusted components of the software;
- Setting access privileges on the directory in which the software executables will be stored to ensure that the software is execute-only accessible to all individuals except the administrator;
- Disabling any non-secure protocols;
- Moving production data if necessary to separate it from the program's control/management data;
- Installation of all current security patches for environment components;

- Establishment and testing of all environment recovery procedures;
- Configuring of all intrusion detection, anomaly detection, firewall, honeypot/honeynet/honeytoken, insider threat/security monitoring, and event/incident reporting capabilities

The developer responsible for documenting environment lockdown procedures may wish to consult one or more of the existing secure configuration guides and/or checklists for the environment in question; some are listed in the Suggested Resources below.

SUGGESTED RESOURCES

- BuildSecurityIn Deployment and Operation resources. Accessed 21 January 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/deployment.html
- NIST. National Vulnerability Database National Checklist Program Repository Webpage. Accessed 26 January 2008 at: http://checklists.nist.gov/ncp.cfm?repository
- DISA. *Application Services STIG*, Version 1, Release 1.1, 17 January 2006. Accessed 21 January 2008 at: http://iase.disa.mil/stigs/stig/application-services-stig-v1r1.pdf
- Microsoft Corporation. Baseline Security Analyzer 2.0 Webpage. Accessed 12 December 2007 at: http://www.microsoft.com/technet/security/tools/mbsa2/default.mspx
- Oracle Software Configuration Manager. Accessed 26 August 2008 at: http://www.oracle.com/support/premier/software-configuration-manager.html
- Oracle Enterprise Configuration Management Pack. Accessed 26 August 2008 at: http://www.oracle.com/technology/products/oem/pdf/ds_config_pack.pdf

9.4 SECURE SUSTAINMENT CONSIDERATIONS

Although the developer's main concern will be with producing software that starts out secure, he/she also has a role to play in ensuring that the software's security is preserved during its sustainment.

SUGGESTED RESOURCES

- Black, Paul. "Software Assurance During Maintenance". *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, Philadelphia, PA: September 2006, pages 70-72. Accessed 9 September 2008 at: http://hissa.nist.gov/~black/Papers/softAssurDuringMaintICSM06.html

9.4.1 Vulnerability management

Object-oriented programming models that enforce principles such as inversion of control can complicate the problems of vulnerabilities emerging over time. This is because such models as the Spring framework for Java allows components to be "wired" together declaratively, so that individual components can be replaced without reengineering the system. When components are replaced in this way, the original security assumptions made by the developer regarding those components may be invalidated. For this reason, a security impact analysis needs to be performed whenever a component is replaced, no matter how little impact that replacement may have on other properties of the system.

Similarly, security impact analyses should be performed before deploying any patches, updates, or maintenance changes that affect the software's design, and any negative impact on the system's security posture (*i.e.,* by introducing vulnerabilities, non-secure behaviors/state changes) should be mitigated with the mitigations delivered as part of the patch or update. As each patch, update, and maintenance change is produced, the patch's impact on the software

architecture and design should be evaluated, and the appropriate system documentation should be updated to include a description and rationale for the patch or change. Note also that risk exposure increases whenever a software product or software-intensive system has had multiple patches installed without first evaluating the impact of the patch on the system architecture and design.

Review of the findings of regularly-scheduled periodic security audits of operational software (which should be performed using automated vulnerability scanners, penetration tests, and other post-deployment appropriate tools and techniques) will enable the developer responsible for software maintenance to observe and analyze how the software has behaved in real world usage and environment conditions over time, and compare that against the expectations engendered by the software's pre-deployment and immediate post-deployment security test results. Analysis of audit results will also enable the developer to verify that the original secure deployment configuration of the software has not been changed in ways that introduce previously unobserved vulnerabilities. It is also the responsibility of the software's vulnerability manager, configuration manager, or maintainer to issue reports to customers about vulnerabilities discovered; these reports should include information about plans for patch issuance to address the reported vulnerabilities.

In addition to performing ongoing post-deployment vulnerability assessments and security audits, the developer responsible for the software's sustainment should track and reacting to all COTS and OSS component suppliers' vulnerability and patch notifications, as well as those of trusted third parties. Similarly, customer and CERT and CSIRT incident reports should be reviewed.

The developer should also study the results of forensic analyses of security incidents involving the software. In all cases, the findings of the developer's analysis should form the basis for identifying new security requirements for future software releases, as well as for strategizing security-focused refactoring or reengineering to satisfy those new requirements. Forensic security analyses assist the analyst in determining which vulnerabilities in the software's functionality or interfaces were exploited in the attack; the focus is to analyze proven vulnerabilities, rather than to locate vulnerabilities that may or may not exist. The analyst should consider three areas for analysis:

1. **Intra-component:** If the exploited vulnerability is suspected to lie within the component itself, the focus of the analysis should be on attaining static and dynamic visibility into behavior and state changes within the component to pinpoint where the vulnerability may lie (the vulnerability's location in source code should be traced if the code is available);

2. **Inter-component:** If the location of the vulnerability is suspected to lie in the interface between two components, the analysis should focus on the communication or programmatic interface mechanisms and protocols used between the components, and should attempt to reveal any incompatibilities between the implementation of those mechanisms/protocols from one component to another;

3. **Extra-component:** If the vulnerability is suspected to lie in an environment component or in the combined interactions of a series of components, the analyst should review audit and event logs for the software and its environment's components, focusing on system-level security-relevant behaviors to reveal points of vulnerability in the configuration of the software and it environment components, and in the interaction of the software with its environment.

The same secure software principles and disciplined secure development practices should be adhered to in the production of patches as were followed in the original development of the software.

During regression testing, the developer should re-run a subset of the original software security tests to ensure that the software modifications cause only intended changes in the system's behavior and do not inadvertently cause any unintended changes. In developing the regression security test plan, the misuse/abuse cases and attack scenarios (based in part on relevant attack patterns) used in earlier test cases should be augmented by any new abuse/misuse cases and attack scenarios suggested by real world attacks that have emerged since the software was last tested.

SUGGESTED RESOURCES

- BuildSecurityIn Deployment and Operation resources. Accessed 21 January 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/deployment.html
- Hanebutte, Nadine 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.

9.4.2 Software aging and security

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 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 *rejuvenation* and *reconfiguration.*

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.

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:

- 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);
- Repetitive execution of the same program while checking for consistent outputs and behaviors;
- 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.

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.

SUGGESTED RESOURCES

- Castelli, V., 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. Accessed 19 December 2007 at: http://www.research.ibm.com/journal/rd/452/castelli.pdf

APPENDIX A: ABBREVIATIONS, ACRONYMS, AND DEFINITIONS

A.1 ABBREVIATIONS AND ACRONYMS

Acronym / Abbreviation	Amplification
ACM	Association for Computing Machinery
AEGIS	Appropriate and Effective Guidance for Information Security
AJAX	Asynchronous JavaScript And XML
AOM	Aspect Oriented Modeling
AOSD	Aspect Oriented Software Development
API	Application Programmatic Interface
ARINC	Aeronautical Radio Incorporated
ASP	Active Server Pages
ASP .NET	Active Server Pages for .NET
BIOS	Basic Input Output System
CAPEC	Common Attack Pattern Enumeration and Classification
CBK	Common Body of Knowledge
CD-ROM	Compact Disc-Read-Only Memory
CERT	Computer Emergency Response Team
CGI	Common Gateway Interface
CLASP	Comprehensive, Lightweight Application Security Process
CLR	Common Language Runtime
CM	Configuration Management
CMU	Carnegie Mellon University
CORAS	Consultative Object Risk Analysis System
COTS	Commercial-Off-The-Shelf
CPU	Central Processing Unit
CSIRT	Computer Security Incident Response Team
CSRF	Cross Site Request Forgery
CWE	Common Weakness Enumeration
DACS	Data and Analysis Center for Software
DbC	Design by Contract
D.C.	District of Columbia

Acronym / Abbreviation	Amplification
DCE	Distributed Computing Environment
DCID	Director of Central Intelligence Directive
DHS	Department of Homeland Security
DISA	Defense Information Systems Agency
DoD	Department of Defense
DoS	Denial of Service
DTE	Domain and Type Enforcement
DTIC	Defense Technical Information Center
e.g.	*exempla grata* (Latin term meaning "provided as an example")
FDD	Feature-Driven Development
FIPS	Federal Information Processing Standard
FMEA	Failure Modes and Effects Analysis
GIF	Graphics Interchange Format
GOTS	Government Off The Shelf
GSSP	Global Secure Software Programmer
GUI	Graphical User Interface
HTML	Hyper Text Markup Language
HTTP	Hyper Text Transfer Protocol
i.e.	*id est* (Latin term meaning "that is")
IA	Information Assurance
IATAC	Information Assurance Technology Analysis Center
IAVA	Information Assurance Vulnerability Alert
IBM	International Business Machines
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
Inc.	Incorporated
ISO	International Organization for Standardization
IT	Information Technology
IV&V	Independent Verification and Validation
JAR	Java ARchive
Java EE	Java Platform, Enterprise Edition

Acronym / Abbreviation	Amplification
JPEG	Joint Photographic Experts Group
JPG	Joint Photographic Experts Group
JSP	Java Server Pages
JVM	Java Virtual Machine
KAOS	Knowledge Acquisition in autOmated Specification
Mac OS X	Macintosh Operating System version 10
MDA	Model Driven Architecture
MDD	Model Driven Development
MILS	Multiple Independent Layers of Security
MISRA	Motor Industry Software Reliability Association
ML	MetaLanguage
MSSDM	Motorola Secure Software Development Model
NASA	National Aeronautics and Space Administration
NIST	National Institute of Standards and Technology
NSA	National Security Agency
OS	Operating System
OSS	Open Source Software
OWASP	Open Web Application Vulnerability Project
PDF	Portable Document Format
Perl	Practical extraction and report language
PHP	a recursive acronym that amplifies to PHP Hypertext Processor; PHP originally stood for Personal Home Page
PITAC	President's Information Technology Advisory Committee
RAII	Resource Acquisition Is Initialization
REVEAL	Requirements Engineering VErification and vALidation
RPC	Remote Procedure Call
RSA	Rivest-Shamir-Adleman
RUP	Rational Unified Process
S2e	Secure Software Engineering
SAFECode	Static Analysis For safe Execution of Code
SANS	SysAdmin, Audit, Networking and Security
SAX	Simple API for XML

Acronym / Abbreviation	Amplification
SCADA	Supervisory Control And Data Acquisition
SCM	Software Configuration Management
SDL	Security Development Lifecycle
SDLC	Software Development Life Cycle
SOA	Service Oriented Architecture
SQL	Structured Query Language
SQUARE	System QUAlity Requirements Engineering
SSE-CMM	System Security Engineering Capability Maturity Model
SSL/TLS	Secure Socket Layer/Transport Security Layer
STIG	Security Technical Implementation Guide
TCL	Tool Command Language
TDD	Test Driven Development
T-MAP	Threat Modeling based on Attacking Path Analysis
TPM	Trusted Processor Module
TRIAD	Trustworthy Refinement through Intrusion-Aware Design
TSP-Secure	Team Software Process for Secure Software Development
U.S.	United States
UDDI	Universal Description, Discovery, and Integration
UML	Unified Modeling Language
URI	Universal Resource Identifier
URL	Universal Resource Locator
VB.NET	Visual Basic for .NET
VBScript	Visual Basic Script
VM	Virtual Machine
VMM	Virtual Machine Monitor
vs.	Abbreviation of *versus*, the Latin word meaning "against" (indicating a contrast)
WSDL	Web Service Definition Language
XHTML	eXtensible HyperText Markup Language
XML	eXtensible Markup Language
XP	eXtreme Programming
XSD	XML Schema Design

A.2 DEFINITIONS

The following are definitions of terms used in this document, as they are intended to be understood.

- **ABUSE:** Malicious misuse, usually with the objective of alteration, disruption, or destruction.
- **ACQUISITION:** The procurement or purchase of a product or service under contract or licensing agreement.
- **ASSET:** Anything that has value (e.g. data, executing process) to a stakeholder (*e.g.,* organization who owns it).
- **ASSURANCE ARGUMENT:** A justification that a given assurance claim (or sub-claim) is true or false.
- **ASSURANCE CASE:** The set of assurance claims of critical system/software assurance properties (requirements of the system), assurance arguments that justify the claims (including assumptions and context), and assurance evidence supporting the arguments.
- **ASSURANCE CLAIM:** The critical system/software requirements for assurance, including the maximum level of uncertainty permitted. . [NDIA]
- **ASSURANCE EVIDENCE:** Information that demonstrably substantiate the arguments in an assurance case. [adapted from NDIA]
- **ASSURANCE:** (1) Grounds for confidence that an entity meets its security objectives. (2) Justifiable grounds for confidence that the required properties of the software have been adequately exhibited. In some definitions, assurance also incorporates the activities that enable the software to achieve an assurable state and/or result in the verification of the software's required properties.
- **ATTACK:** An attempt to gain unauthorized access to a system's services or to compromise one of the system's required properties (integrity, availability, correctness, predictability, reliability, *etc.*). When a software-intensive system or component is the target, the attack will most likely manifest as an intentional error or fault that exploits a vulnerability or weakness in the targeted software.
- **AVAILABILITY:** (1) Consistently timely and reliable access to and use of a system, its data, and its resources. (2) The degree to which the services of a system or component are operational and accessible when needed by their intended users. When availability is considered as a security property, the intended users must be authorized to access the specific services they attempt to access, and to perform the specific actions they attempt

to perform. The need for availability generates the requirements that the system or component be able to resist or withstand attempts to delete, disconnect, or otherwise render the system/component inoperable or inaccessible, regardless of whether those attempts are intentional or accidental. The violation of availability is referred to as *Denial of Service* or *sabotage*.

- **BUFFER OVERFLOW:** A condition at an interface under which more input can be placed into a buffer or data holding area than the capacity allocated, overwriting other information. Attackers exploit such a condition to crash a system or to insert specially crafted code that allows them to gain control of the system. A buffer overflow condition exists when a program attempts to put more data in a buffer than it can hold or when a program attempts to put data in a memory area past a buffer. In this case, a buffer is a sequential section of memory allocated to contain anything from a character string to an array of integers.

- **BUG:** A problem, often the result of an error, that exists in the software's code that may or may not represent a vulnerability.

- **COMMERCIAL-OFF-THE-SHELF (COTS):** Commercial products that are ready-made and available for sale to the general public.

- **COMPONENT ASSEMBLY:** Process of organizing and configuring components (by the strict definition of that term) to use their built-in interfaces to communicate/interact with each other. Also referred to as "composition" and "integration".

- **COMPONENT:** A part or element within a larger system. A component may be constructed of hardware or software and may be divisible into smaller components. In the strictest definition, a component must have a contractually-specified interface (or interfaces), explicit context dependencies, the ability to be deployed independently, the ability to be assembled/composed with other components by someone other than its developer. In the less restrictive definition used in this document, a component may also be a code module or code unit. A code unit is either: a separately testable element of a software component, *or* a software component that cannot be further decomposed into constituent components, *or* a logically-separable part of a computer program. A code module is either: a program unit that is discrete and identifiable with respect to compilation, combination with other units, and loading, *i.e.,* a code unit, *or* a logically separable part of a computer program, *i.e.,* a code unit.

- **COMPROMISE:** A violation of the security policy of a system, or an incident in which any of the security properties of the system are violated.

- **CONFIGURATION MANAGEMENT:** Management of security features and assurances through control of changes made to hardware, software, firmware, and documentation, test, test fixtures, and test documentation throughout the life cycle of an information system.

- **CORRECTNESS:** (1) The degree to which software is free from errors or inadequacies in its specification, design, and implementation. (2) The degree to which software, documentation, or other items satisfy their specified requirements. (3) The degree to which software, documentation, or other items meet user needs and expectations, whether those needs and expectations are specified or not. (4) **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* software, documentation and other development artifacts satisfy their specified requirements; *or* 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 will be free of faults, and will operate consistently with its specification.

- **COUNTERMEASURE:** An action, device, procedure, technique, or other measure that reduces the vulnerability/weakness of a component or system.

- **CRITICAL SOFTWARE:** Software the failure of which could have a negative impact on national security or human safety, or could result in a large financial or social loss. Critical software is also referred to as *high-consequence* software.

- **CUSTOM SOFTWARE:** Software developed either for a specific organization or function. It is generally not targeted to the mass market, but usually created for a specific customer to satisfy that customer's unique needs.

- **DENIAL OF SERVICE (DOS):** (1) Prevention of authorized access to a system or resource by making that system/resource unavailable or inaccessible at its expected level of operation capacity and performance, *e.g.,* by delaying system operations and functions, terminating system operations, or interfering with connectivity to/from the system. Any action or series of actions that prevents any part of a system from functioning constitutes a DoS. (2) The intentional violation of the software's availability resulting from an action or series of actions that has one of the following outcomes: the system's intended users cannot gain access to the system; *or* one or more of the system's time-critical operations is delayed; *or* a critical function in the system fails. Also referred to as *sabotage*.

- **DEPENDABILITY:** The ability of a system to perform its intended functionality or deliver its intended service correctly and predictably whenever it is called upon to do so, including under hostile conditions such as when the software comes under attack or runs on a malicious host. The following properties of software directly contribute to its dependability: availability, integrity, reliability, survivability, trustworthiness, safety, fault-tolerance.

- **EMBEDDED SOFTWARE:** Software that is part of a larger physical system and performs some of the requirements of that system, such as monitoring, measuring, or controlling the actions of the system's physical components, *e.g.,* software used in an aircraft or

rapid transit system. Typically, such software does not provide an interface with the user; however, this limitation is changing with some modern embedded software.

- **ERROR:** (1) Deviation of one of the software's states from correct to incorrect. (2) Discrepancy between the condition or value actually computed, observed, or measured by the software, and the true, specified, or theoretically correct value or condition. (3) A human action that leads to the presence of an error (*e.g.,* a coding error) in software, or a failure of operational software.

- **EVENT:** An occurrence of some specific situation, activity, or data handling.

- **EXECUTION ENVIRONMENT:** The aggregation of hardware, software, and networking entities surrounding the software which directly affect or influence its execution.

- **EXPLOIT:** A technique, which may be implemented by software code (often in the form of a script), that takes advantage of a vulnerability or security weakness in a piece of target software. If implemented by software code, the code itself (rather than the activity it performs) is sometimes referred to as the exploit.

- **FAILURE:** (1) Non-performance by a system or component of an intended/required function or service within the operational parameters specified for that function or service. (2) Deviation of the system's performance from its specified, expected parameters (such as its timing constraints).

- **FAULT:** The adjudged or hypothesized cause of an error.

- **FLAW:** An error of commission, omission, or oversight in the creation of the software's requirements, architecture, or design specification that results in an inadequate, and often weak, software design, or in one or more errors in its implementation. Some software assurance practitioners object to the word "flaw" because it is often confused with "error", "fault", and "defect". (Just as "defect" is sometimes similarly confused with "flaw".) A flaw may or may not represent a vulnerability.

- **FORMAL METHOD:** A process by which the system architecture or design is mathematically modeled and specified, and/or the high-level implementation of the system is verified, through use of mathematical proofs, to be consistent with its specified requirements, architecture, design, or security policy.

- **FORMAL:** Based on mathematics. (This narrow definition is used in this SOAR to avoid the confusion that arises when "formal" is used both to mean "mathematically-based" and as a synonym for "structured" or "disciplined".)

- **IMPLEMENTATION:** The SDLC phase at the end of which the software is able to operate. In this document, "implementation" is used to designate the activities that follow the design phase and precede the testing phase, *e.g.,* coding and software integration.

- **INDEPENDENT TESTING:** A common practice of software testing is that it is performed by an independent group of testers after the functionality is developed but before it is shipped to the customer. This practice often results in the testing phase being used as project buffer to compensate for project delays, thereby compromising the time devoted to testing.

- **INDEPENDENT VERIFICATION AND VALIDATION (IV&V):** Verification and Validation performed by a third party, *i.e.,* an entity that is neither the developer of the system being verified and validated, nor the acquirer/user of that system.

- **INFORMATION ASSURANCE:** Measures that protect and defend information and information systems by ensuring their availability, integrity, authentication, confidentiality, and non-repudiation. These measures include providing for restoration of information systems by incorporating protection, detection, and reaction capabilities. Often used interchangeably with "information security", which is the protection of information and information systems from unauthorized access, use, disclosure, disruption, modification, or destruction in order to provide confidentiality, integrity, and availability.

- **INFORMATION SYSTEM:** A discrete set of information resources organized for the collection, processing, maintenance, use, sharing, dissemination, or disposal of information.

- **INFORMATION TECHNOLOGY:** Any equipment or interconnected system or subsystem of equipment that is used in the automatic acquisition, storage, manipulation, management, movement, control, display, switching, interchange, transmission, or reception of data or information by the executive agency. Information technology includes computers, peripherals, software, firmware, services (including support services), and related resources.

- **INPUT VALIDATION:** The act of determining that data input to a program is sound (*e.g.,* for example, might include: the length, format, physical content of the data do not vary from the acceptable parameters defined for length, format, and physical content).

- **INTEGRITY:** (1) Guarding against improper modification or destruction. (2) The property of a system or component that reflects its logical correctness and reliability, completeness, and consistency. Integrity as a security property generates the requirement for the system or component to be protected against intentional attempts to either alter or modify the software in an improper or unauthorized manner (note that attempts to destroy the software in an improper or unauthorized manner are considered attacks on the system's availability, *i.e.,* Denial of Service attacks); *or* through improper or unauthorized manipulation to cause the software to either perform its intended function(s) in a manner inconsistent with the system's specifications and the intended users' expectations, or to perform undocumented or unexpected functions.

- **JUSTIFIABLE CONFIDENCE:** The actions, arguments and evidence that provides a basis for a defensible reduction in uncertainty.

- **LEAST PRIVILEGE:** The principle whereby each subject (*i.e.*, actor) in the system is granted only the most restrictive set of privileges needed by the subject to perform its authorized tasks, and whereby the subject is allowed to retain those privileges for no longer than it needs them.

- **MALICIOUS CODE:** (1) Software or firmware intended to perform an unauthorized process that will have adverse impact on the confidentiality, integrity, or availability of an information system. (2) A virus, worm, Trojan horse, or other code-based malicious entity that successfully infects the host. (3) Undocumented software or firmware intended to perform an unauthorized or unanticipated process that will have adverse impact on the dependability of a component or system. Malicious code may be self – contained (as with viruses, worms, malicious bots, and Trojan horses), or may be embedded in another software component (as with logic bombs, time bombs, and some Trojan horses).

- **MALICIOUS CODE:** Undocumented software or firmware intended to perform an unauthorized or unanticipated process that will have adverse impact on the dependability of a component or system. Malicious code may be self-contained (as with viruses, worms, malicious bots, and Trojan horses), or it may be embedded in another software component (as with logic bombs, time bombs, and some Trojan horses). Also referred to as *malware*.

- **MALWARE:** A malicious program that is inserted into a system, usually covertly, with the intention of compromising the specified operation of that system, including its ability to protect the confidentiality, integrity, and availability of the system's data, applications, or operating system or of otherwise annoying or inhibiting the operational abilities of the system's users. Often used interchangeably with "malicious code".

- **MISTAKE:** An error committed by a person as the result of a bad or incorrect decision or judgment by that person. Contrast with "error", which is used in this document to indicate the result of a "mistake" committed by software (*i.e.,* as the result of an incorrect calculation or manipulation).

- **MISUSE:** Usage that deviates from what is expected (with that expectation usually based on the software's specification). If the misuse is maliciously motivated, it is referred to as *abuse.*

- **MOBILE CODE:** Software modules obtained from remote systems, transferred across a network, and then downloaded and executed on local systems without explicit installation or execution by the recipient. In particular, "mobile code" is used to describe applets within web browsers based upon Microsoft's ActiveX, Sun's Java, or Netscape's JavaScript technologies.

- **NON-REPUDIATION:** For purposes of information security, assurance the sender of data is provided with proof of delivery and the recipient is provided with proof of the sender's identity, so neither can later deny having processed the data. In terms of software's activities, non-repudiation extends to the inability of software to deny having performed a specific action.

- **OPEN SOURCE SOFTWARE:** Publicly-available software the source code of which can be obtained by license permitting users to study and change (improve) the software, as well as redistribute it in modified or unmodified form.

- **OUTSOURCING:** The delegation of operations or jobs from internal production within a business to an external entity usually by contract.

- **PENETRATION TESTING:** Security testing in which evaluators mimic real-world attacks to attempt to identify methods for circumventing the security features of an application, system, or network. Penetration testing often involves issuing real attacks on real systems and data, using common tools and techniques used by attackers. Most penetration tests involve looking for combinations of vulnerabilities on a single system or multiple systems that can be used to gain more access than could be achieved through any single vulnerability.

- **PREDICTABILITY:** The properties, states, and behaviors of the system or component never deviate from what is expected.

- **PROBLEM:** Used interchangeably with anomaly, though "problem" has a more negative connotation, and implies that the anomaly is, or results from, a flaw, defect, fault, error, or failure.

- **PROGRAM:** The umbrella structure established to manage a series of related projects. The program does not produce any project deliverables. The project teams produce them all. The purpose of the program is to provide overall direction and guidance, to make sure the related projects are communicating effectively, to provide a central point of contact and focus for the client and the project teams, and to determine how individual projects should be defined to ensure all the work gets completed successfully. A program may also be an executable software entity.

- **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.

- **RELIABILITY:** (1) The ability of a software system, including all of its individual components, to accomplish their objectives without failure, degradation, or unspecified behavior. Software that possesses the characteristic of reliability to the extent that it can be expected to consistently perform its intended functions satisfactorily. This implies a time factor in that reliable software is expected to perform correctly over a period of time. It also encompasses environmental considerations in that the software is required

to perform correctly in whichever conditions it finds itself - this is sometimes termed robustness. (2) 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.

- **REQUIREMENT:** A statement that identifies an operational, functional, or design characteristic or constraint of a product or process. Ideally, a requirement should be unambiguous, testable or measurable, and necessary to the acceptability of the process or product (by consumers or those responsible for verifying the product's/process' conformance to internal quality assurance guidelines.

- **RISK:** (1) The possibility that a particular threat will adversely impact an information resource (including information systems, information, and software, whether embedded or part of an information systems) by exploiting a particular vulnerability. (2) The level of impact on an organization's business operations (including mission, functions, image, or reputation), assets, or individuals resulting from the operation of a system, given the potential impact of a threat and the likelihood of that threat occurring. The potential that a given threat will exploit vulnerabilities of an asset or group of assets and thereby cause harm to the organization. It is measured in terms of a combination of the probability of an event and its consequence.

- **RISK:** The likelihood that a particular threat will adversely impact a system by exploiting a particular vulnerability.

- **ROBUSTNESS:** The degree to which a component or system can function correctly in the presence of invalid inputs or stressful environmental conditions, including inputs or conditions that are intentionally and maliciously created.

- **ROLE:** An abstract definition of a set of functions performed and work products or deliverables owned. Roles are typically realized by an individual, or a set of individuals, working together as a team. Roles are not individuals; instead, they describe how individuals behave in the business and what responsibilities these individuals have.

- **SABOTAGE:** *See* Denial of Service.

- **SAFETY:** Persistence of dependability in the face of realized hazards (unsponsored, unplanned events, accidents, mishaps) that result in death, injury, illness, damage to the environment, or significant loss or destruction of property.

- **SANDBOXING:** A method of isolating application-level components into distinct execution domains, the separation of which is enforced by software. When run in a sandbox, all of the component's code and data accesses are confined to memory segments within that sandbox. In this way, sandboxes provide a greater level of isolation 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 trusted programs. Virtual machines (VMs) are sometimes used to implement sandboxing, with each VM providing an isolated execution domain.

- **SECURE CODING PRINCIPLES:** A set of philosophical imperatives that collectively govern how coding is done by the programmer so that the resulting software will behave and function as securely as possible.

- **SECURE CODING:** Software programming practices that reduce or eliminate software defects/programming errors as well as other programming practices that lead to software vulnerabilities.

- **SECURE DESIGN PRINCIPLES:** A set of philosophical imperatives that collective govern how the design is conceived by the developer so that the resulting software will behave and function as securely as possible.

- **SECURE SOFTWARE PROJECT MANAGEMENT:** Systematic, disciplined, and quantified" application of management activity that ensures the software being developed conforms to security policies and meets security requirements.

- **SECURE SOFTWARE:** Software that consistently exhibits the properties of dependability, trustworthiness, and reliability. For purposes of software assurance, secure software needs only realize these properties with a justifiably high confidence without having to guarantee absolutely the substantial set of explicit security properties and functionality, including all those required for its intended usage.

- **SECURE STATE:** The condition in which no subject can access another entity in an unauthorized manner for any purpose.

- **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.

- **SECURITY:** Protection against subversion or sabotage (which includes denial of service). Security is a composite of four attributes – confidentiality, integrity, availability, and accountability plus aspects of a fifth, usability, all of which have the related issue of their assurance. To be considered secure, software must exhibit three properties: dependability, trustworthiness, and survivability (also referred to as resilience).

- **SERVICE:** A set of one or more functions, tasks, or activities performed to achieve one or more objectives that benefit a user (human or process).

- **SOFTWARE ASSURANCE:** 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 the software functions in the intended manner.

- **SOFTWARE DEVELOPMENT LIFE CYCLE PROCESS:** The process by which user needs are translated into a software product. the process involves translating user needs into software requirements, transforming the software requirements into architecture and design, implementing the design in code or obtaining existing code that is able to perform the actions specified in the design, integrating the code components, testing the code, and sometimes installing and checking out the software for operational activities. NOTE: Depending on the SDLC process in use, these activities may overlap or may be performed iteratively.

- **SOFTWARE PEDIGREE:** Background/lineage of the software being acquired. This includes such considerations as how the version of the software under consideration at a given point in time was originally conceived and implemented, and by whom. While the software's pedigree is extended, and thus changed, each time the software is modified in some way by its developer, at any given point in time, the software as it exists in that point in time, can be said to have a fixed pedigree.

- **SOFTWARE PROVENANCE:** Experience of the software being acquired after it leaves the control of its developer(s) and enters the supply chain. This includes such considerations as how the software is licensed, how it is installed and configured in its execution environment, and how it is modified through patching and updating, and by whom. Provenance also reflects changes in responsibility for the ongoing development of the software (new versions, patches, *etc.*)----for example, if this responsibility shifts from the software's original developer to an integrator or a new development organization (as when one software firm buys another).

- **SOFTWARE SECURITY ASSURANCE:** Justifiable grounds for confidence that software's security property, including all of security's constituent properties (*e.g.*, attack-resistance, attack-tolerance, attack-resilience, lack of vulnerabilities, lack of malicious logic, dependability despite the presence of sponsored faults, *etc.*) has been adequately exhibited. Often abbreviated to *software assurance.*

- **SOFTWARE:** A set of instructions, written in some form of symbolic language (*i.e.*, a "programming language" or "scripting language"), which are ultimately interpreted or compiled into the low-level binary language directly understood by the hardware of the processor on which the software executes, in order for that processor to accomplish the functional tasks specified by the software.

- **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.

- **STANDARD:** An agreement among any number of organizations that defines certain characteristics, specification, or parameters related to a particular aspect of computer technology.

- **STATE:** (1) A condition or mode of existence that a system or component may be in, for example the input state of a given channel. (2) The values assumed at a given instant by the variables that define the characteristics of a component or system.

- **SUBVERSION:** (1) The intentional violation of the software's integrity. (2) Changing a process or product so as to provide a means to compromise a required property, such as security.

- **SURVIVABILITY:** (1) Resilience sufficient for the software to resist or tolerate most known attacks and as many novel attacks as possible, or if unable to resist or tolerate them, to recovery as quickly as possible with as little damage as possible. In most cases, this will require the software to be able to isolate the source of the attack. (2) The ability to continue correct, predictable operation despite the presence of realized hazards and threats.

- **SYSTEM:** (1) A collection of components organized to accomplish a specific function or set of functions. (2) **System:** A combination of interacting elements organized to achieve one or more stated purposes. These elements include hardware, software, data, computing resources, human users, *etc.*

- **TESTING:** An activity performed for assessing the conformance of software with any or all of its required properties and/or behaviors, and for improving it, by identifying defects and problems.

- **THREAT MODELING:** The analysis, assessment and review of audit trails and other information collected for the purpose of searching out system events that may constitute violations of system security. The artifact of threat modeling is the threat model.

- **THREAT:** (1) Any entity, circumstance or event with the potential to adversely impact harm the software system or component through its unauthorized access, destruction, modification, and/or denial of service. (2) An actor, agent, circumstance, or event with the potential to cause harm to a software-intensive system or to the data or resources to which it has or enables access. If intentional and malicious, the threat is likely to be realized by an attack that exploits a vulnerability in software.

- **TRUST:** The confidence one element has in another that the second element will behave as expected.

- **TRUSTWORTHINESS**: (1) Containing few if any vulnerabilities or weaknesses that can be intentionally exploited to subvert or sabotage the software's dependability, and containing no malicious logic that would cause the software to behave in a malicious

manner. (2) 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 ability to be compromised and the execution of unexpected and unspecified malicious logic renders prediction of its correct behavior impossible.

- **USER:** Any person or process authorized to access an operational system.

- **VERIFICATION AND VALIDATION (V&V):** The process of confirming, by examination and provision of objective evidence, that each step in the process of building or modifying the software yields the right products (verification). Verification asks and answers the question "Was the software built right?" (*i.e.,* correctness); *and* 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. 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. See also "independent testing".

- **VULNERABILITY:** (1) Weakness in a software system that could be exploited by an attacker. Bugs and flaws collectively form the basis of most software vulnerabilities. (2) A development error, bug, flaw, or weakness in deployed software that can be exploited with malicious intent by a threat with the objective of subverting (violation of integrity) or sabotaging (violation of availability) the software, often as a step towards gaining unauthorized access to the information handled by that software. Vulnerabilities can originate from weaknesses in the software's design, faults in its implementation, or problems in its operation.

- **WEAKNESS:** (1) 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). (2) An underlying condition or construct in software that has the potential for degrading the security of the software.

- **WEB SERVICE:** A software component or system designed to support interoperable machine- or application-oriented interaction over a network. A Web service has in interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its descriptions using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

APPENDIX B: RESOURCES AND BIBLIOGRAPHY

This appendix contains a complete listing of all online and print resources referenced in this document, both in the "Suggested Resources" boxes and in the footnotes.

B.1 FREELY-ACCESSIBLE ONLINE RESOURCES

The following resources can be freely accessed on the Web. In the very few cases (noted here) in which online registration is required, this registration is free of charge.

- Aaby, Anthony A. "Security and the Design of Secure Software". In *Software: A Fine Art,* Draft Version 1.0, 9 February 2007. Accessed 19 January 2008 at: http://cs.wwc.edu/~aabyan/FAS/book/node5.html#SECTION05500000000000000000 – *and* – http://moonbase.wwc.edu/~aabyan/FAS/book.pdf – *and* – http://cs.wwc.edu/~aabyan/FAS/book.pdf

- Acegi project Website. Accessed 26 Mar 2008 at: http://www.acegisecurity.org

- Ada Standards. Accessed 8 September 2008 at: http://www.adaic.org/standards/

- AdaCore GNAT Pro High-Integrity Edition. Accessed 5 September 2008 at: http://www.adacore.com/home/gnatpro/development_solutions/safety-critical/

- Aggarwal, Nakul and Ritesh Arora. *Ruby on Rails Security Guide.* Published on QuarkRuby Weblog, 20 September 2007. Accessed 26 March 2008 at: http://www.quarkruby.com/2007/9/20/ruby-on-rails-security-guide

- Aime, Marco. "Modelling services for trust and security assurance". Presented at Workshop on Software and Service Development, Security and Dependability, Maribor, Slovenia, 10-11 July 2007. Accessed 19 January 2008 at: http://www.esfors.org/esfors_ws2/D1_S1_Aime-ModellingServicesTrustAndSecurityAssurance.pdf

- Alexander, Ian. "Modelling the interplay of conflicting goals with use and misuse cases". *Proceedings of 8th International Workshop on Requirements Engineering Foundation for Software Quality,* Essen, Germany, 9-10 September 2002, pages 145-15. Accessed 11 September 2008 at: http://ftp.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-109/paper1.pdf

- Alhazmi, O.H., Y.K. Malaiya, and I. Ray. "Measuring, analyzing and predicting security vulnerabilities in software systems". *Computers and Security,* Volume 26 Issue 3, May 2007, pages 219-228. Preprint version accessed 26 December 2007 at: http://www.cs.colostate.edu/~malaiya/pub/com&security_darticle.pdf

- Anantharaju, Srinath. "Automating Web application security testing". Google Online Security Blog, 16 July 2007. Accessed 11 December 2007 at:

http://googleonlinesecurity.blogspot.com/2007/07/automating-Web-application-security.html

- Ankrum, T. Scott, and Alfred H. Kromholz. "Structured assurance cases: three common standards". Slides presented at the Association for Software Quality (ASQ) Section 509 Software Special Interest Group meeting. 2006 January 23; McLean, VA. Accessed 31 July 2007 at: http://www.asq509.org/ht/action/GetDocumentAction/id/2132

- Apache Axis Web Service Security page. Accessed 11 December 2007 at: http://ws.apache.org/axis/java/security.html

- Apple Computer. *Secure Coding Guide.* 23 May 2003. Accessed 12 December 2007 at: http://developer.apple.com/documentation/Security/Conceptual/SecureCodingGuide/SecureCodingGuide.pdf

- Araujo, Rudolph. "Security Requirements Engineering: A Road Map". *Software Mag.com*, July 2007. Accessed 13 December 2007 at: http://www.softwaremag.com/L.cfm?Doc=1067-7/2007

- Association for Computing Machinery (ACM) Workshops on Formal Methods in Security Engineering. Accessed 26 February 2008 at: http://www.cs.utexas.edu/~shmat/FMSE08/

- Avaya Labs' Libsafe research Webpage. Accessed 21 January 2008 at: http://www.research.avayalabs.com/gcm/usa/en-us/initiatives/all/nsr.htm&Filter=ProjectTitle:Libsafe&Wrapper=LabsProjectDetails&View=LabsProjectDetails

- Balzarotti, Davide, Mattia Monga, and Sabrina Sicari, Università di Catania. "Assessing the Risk of Using Vulnerable Components". *Proceedings of the First Workshop on Quality of Protection,* Milan, Italy, 15 September 2005. Accessed 19 December 2007 at: http://dabalza.net/publications/download/risk-qop05.pdf - *and* - http://homes.dico.unimi.it/~monga/lib/qop.pdf

- Barnes, John. *Safe and Secure Software: An Introduction to Ada 2005.* Accessed 8 September 2008 at: http://www2.adacore.com/home/ada_answers/ada_2005/safe_secure/

- Barnum, Sean and Amit Sethi. "Attack Pattern Glossary", 7 November 2006. Accessed 25 January 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/attack/590.html

- Barnum, Sean and Amit Sethi. "Attack Patterns as a Knowledge Resource for Building Secure Software" (whitepaper). Ashburn, Virginia: Cigital, Inc., 2007. Accessed 26 December 2007 at: http://capec.mitre.org/documents/Attack_Patterns-Knowing_Your_Enemies_in_Order_to_Defeat_Them-Paper.pdf

- Basirico, Joe. "Software security testing: Finding your inner evildoer". SearchSoftwareQuality.com, 6 August 2007. Accessed 11 December 2007 at: http://searchsoftwarequality.techtarget.com/originalContent/0,289142,sid92_gci1265911,00.html

- Beaver, Kevin. "Web Application Security Testing Checklist". SearchSoftwareQuality.com, 19 March 2007. Available at: http://searchsoftwarequality.techtarget.com/tip/0,289483,sid92_gci1247920,00.html

- Beznosov, Konstantin and Philippe Kruchten. "Towards agile security assurance". *Proceedings of the 11th Workshop on New Security Paradigms,* Nova Scotia, Canada, September 2004. Accessed 21 January 2008 at: http://konstantin.beznosov.net/professional/works/shared/biblio_view.php?bibid=10&tab=home - *and* - http://konstantin.beznosov.net/old-professional/papers/Towards_Agile_Security_Assurance.html

- Beznosov, Konstantin. "Extreme security engineering: on employing XP practices to achieve "good enough security" without defining it". Presented at the First ACM Workshop on Business Driven Security Engineering, Washington, D.C. (District of Columbia), 31 October 2003. Accessed 4 April 2007 at: http://konstantin.beznosov.net/professional/doc/papers/eXtreme_Security_Engineering-BizSec-paper.pdf

- Bistarelli, Stefano. Classroom Materials from "Secure Programming" course taught at Università degli Studi "G. d'Annunzio" Italy [in English and Italian]. Accessed 14 December 2007 at: http://www.sci.unich.it/~bista/didattica/secure-programming/materiale/

- Black, Paul. "Software Assurance During Maintenance". *Proceedings of the 22nd IEEE International Conference on Software Maintenance,* Philadelphia, PA: September 2006, pages 70-72. Accessed 9 September 2008 at: http://hissa.nist.gov/~black/Papers/softAssurDuringMaintICSM06.html

- Boström, Gustav. *Simplifying development of secure software – Aspects and Agile methods.* Licentiate Thesis for the Stockholm (Sweden) University and Royal Institute of Technology. undated. Accessed 9 April 2008 at: http://www.diva-portal.org/diva/getDocument?urn_nbn_se_kth_diva-3913-3__fulltext.pdf

- Breu, Ruth, Klaus Burger, Michael Hafner, Jan Jürjens, Gerhard Popp, Guido Wimmel, and Volkmar Lotz. "Key Issues of a Formally Based Process Model for Security Engineering". *Proceedings of 16th International Conference on Software and Systems Engineering and their Applications,* Paris, France, 2-4 December 2003. Accessed 7 July 2008 at: http://www4.informatik.tu-muenchen.de/~popp/publications/workshops/icssea03.pdf

- Brosgol, Benjamin M. and Robert B. K. Dewar. "Need Secure Software?". *Application Software Developer,* June 2008. Accessed 8 September at: http://www.applicationsoftwaredeveloper.com/features/june07/article2.html

- Buchanan, Sam. "Web Application Security Testing". Based on presentation to Minnesota State Colleges and Universities IT Conference, April 2005. Accessed 11 December 2007 at: http://afongen.com/writing/Webappsec/2005/

- Building bug-free O-O software: An introduction to Design by Contract. Accessed 7 July 2008 at: http://archive.eiffel.com/doc/manuals/technology/contract/

- BuildSecurityIn Architectural Risk Analysis resources. Accessed 21 January 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/architecture.html

- BuildSecurityIn Assembly, Integration, and Evolution resources. Accessed 21 January 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/assembly.html

- BuildSecurityIn Assurance Case resources. Accessed 21 January 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/assurance.html

- BuildSecurityIn Attack Patterns resources. Accessed 21 January 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/attack.html

- BuildSecurityIn Black Box Testing tools resources. Accessed 21 January 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/tools/black-box.html

- BuildSecurityIn Code Analysis resources. Accessed 21 January 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/code.html

- BuildSecurityIn Coding Practices resources. Accessed 21 January 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/coding.html

- BuildSecurityIn Coding Rules resources. Accessed 21 January 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi-rules/home.html

- BuildSecurityIn Deployment and Operation resources. Accessed 21 January 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/deployment.html

- BuildSecurityIn Design Guidelines resources. Accessed 21 January 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi-rules/home.html

- BuildSecurityIn Design Principles resources. Accessed 21 January 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/principles.html

- BuildSecurityIn Legacy Systems resources. Accessed 21 January 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/legacy.html

- BuildSecurityIn Modeling tools resources. Accessed 21 January 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/tools/modeling.html

- BuildSecurityIn Penetration Testing resources. Accessed 21 January 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/penetration.html

- BuildSecurityIn Requirements Engineering resources. Accessed 21 January 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/requirements.html

- BuildSecurityIn SDLC Process resources. Accessed 21 January 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/sdlc.html

- BuildSecurityIn Security Testing resources. Accessed 21 January 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/testing.html

- BuildSecurityIn Training and Awareness resources. Accessed 21 January 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/training.html

- BuildSecurityIn White Box Testing resources. Accessed 21 January 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/white box.html

- Bumgarner, John and Scott Borg. *The U.S. Cyber Consequences Unit Cyber-Security Check List.* Final Version, 2007. Accessed 28 January 2008 at: http://www.isalliance.org/content/view/144/292

- Bundesamt für Sicherheit in der Informationstechnik. *Sicherheit von Webanwendungen: Maßnahmenkatalog und Best Practices,* Version 1, August 2006 [in German]. Accessed 26 December 2007 at: http://www.bsi.de/literat/studien/Websec/WebSec.pdf

- C4J: Design by Contract for Java. Accessed 7 July 2008 at: http://c4j.sourceforge.net/

- Carnegie Mellon University Software Engineering Institute. Function Extraction Webpage. Accessed 21 January 2008 at: http://www.cert.org/sse/fxmc.html

- Carnegie Mellon University Software Engineering Institute. Predictable Assembly from Certifiable Components Webpage. Accessed 21 January 2008 at: http://www.sei.cmu.edu/pacc/

- Carnegie Mellon University Software Engineering Institute/Computer Emergency Response Team Secure Coding Standards Webpage. Accessed 21 January 2008 at: https://www.securecoding.cert.org/

- Carnegie Mellon University. Scenario and Attack Graphs project Webpage. Accessed 26 January 2008 at: http://www.cs.cmu.edu/~scenariograph/

- Castelli, V., 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. Accessed 19 December 2007 at: http://www.research.ibm.com/journal/rd/452/castelli.pdf

- CCured Documentation Web page. Accessed 21 January 2008 at: http://manju.cs.berkeley.edu/ccured

- Chen, Yue. *Software Security Economics and Threat Modeling Based on Attack Path Analysis: A Stakeholder Value Driven Approach.* University of Southern California Doctoral Dissertation, December 2007. Accessed 25 January 2008 at: http://sunset.usc.edu/csse/TECHRPTS/PhD_Dissertations/files/ChenY_Dissertation.pdf

- Chess, Brian. "Twelve Java Technology Security Traps and How to Avoid Them". Presented at JavaOne 2006. Accessed 11 December 2007 at: http://developers.sun.com/learning/javaoneonline/j1sessn.jsp?sessn=TS-1660&yr=2006&track=coreenterprise

- Ciphent Certified Secure Software Developer certification. Accessed 21 January 2008 at: http://www.ciphent.com/training/certification

- Codase BETA Java Source Code Search Engine. Entry for javax.xml Class XMLConstants#FEATURE_SECURE_PROCESSING. Accessed 8 April 2008 at: http://www.codase.com/java/javax/xml/XMLConstants.html#FEATURE_SECURE_PROCESSING

- ComputerWeekly.com. Learning Guide: Developing secure enterprise Java applications, 19 April 2006. Accessed 11 December 2007 at: http://www.computerweekly.com/Articles/2006/04/19/218968/developing-secure-enterprise-java-applications.htm

- Congressional Research Service. *Creating a National Framework for Cybersecurity: An Analysis of Issues and Options.* Report for Congress, Order Code RL32777, 22 February 2005. Accessed 28 January 2008 at: http://fpc.state.gov/documents/organization/43393.pdf

- Croxford, Martin and Roderick Chapman. "Correctness by Construction: a manifesto for high-integrity software". *CrossTalk: The Journal of Defense Software Engineering,* Volume 18 Number 12, December 2005. Accessed 21 January 2008 at: http://www.stsc.hill.af.mil/CrossTalk/2005/12/0512CroxfordChapman.html

- CSRF-killer plug-in. Accessed 1 July 2008 at: http://activereload.net/2007/3/6/your-requests-are-safe-with-us

- CYCLONE Website. Accessed 21 January 2008 at: http://cyclone.thelanguage.org/

- Damodaran, Meledath. "Secure Software Development Using use cases and Misuse Cases". *Issues in Information Systems,* Volume VII, Number 1, 2006, pages 150-154. Accessed 13 December 2007 at: http://www.iacis.org/iis/2006_iis/PDFs/Damodaran.pdf

- Daniel P.F. «Análisis y Modelado de Amenazas» [in Spanish], Version 1.0, 18 December 2006. Accessed 25 January 2007 at: http://metal.hacktimes.com/files/Analisis-y-Modelado-de-Amenazas.pdf

- Davidson, Jack W. and Jason D. Hiser. "Securing Embedded Software Using Software Dynamic Translation". Position Paper for the Army Research Organization Planning Workshop on Embedded Systems and Network Security, 22-23 February 2007. Accessed 31 December 2007 at: http://moss.csc.ncsu.edu/~mueller/esns07/davidson.pdf

- Davidson, Michelle. "Secure agile software development an oxymoron?" Application Security Tech Target, 25 October 2006. Accessed 4 April 2007 at: http://searchappsecurity.techtarget.com/originalContent/0,289142,sid92_gci1226109,00.html

- Dean, J. and L. Li. "Issues in Developing Security Wrapper Technology for COTS Software Products". *Proceedings of the First International Conference on COTS-Based Software Systems,* Orlando, Florida, 4-6 February 2002, pages 76-85. Accessed 19 December 2007 at: http://iit-iti.nrc-cnrc.gc.ca/iit-publications-iti/docs/NRC-44924.pdf

- Dewar, Robert B.K. and Roderick Chapman. "Building secure software: Your language matters!" *Military Embedded Systems,* Winter 2006. Accessed 8 September 2008 at: http://www.mil-embedded.com/articles/id/?2012

- DHS National Cyber Security Division. *IT Security Essential Body of Knowledge.* Final Draft Version 1.1, October 2007. Accessed 28 January 2008 at: http://www.us-cert.gov/ITSecurityEBK/EBK2007.pdf

- Di Paola, Stefano and Giorgio Fedon. "Subverting Ajax". *Proceedings of the 23rd Chaos Communication Conference,* Berlin, Germany, 27-30 December 2006. Accessed 26 January 2008 at: http://events.ccc.de/congress/2006/Fahrplan/attachments/1158-Subverting_Ajax.pdf

- Dickson, John B. "Application Security: What does it take to build and test secure software?". Presented at Information Systems Audit and Control Association North Alabama Chapter meeting, 6 November 2006. Accessed 16 January 2008 at:

http://www.bham.net/isaca/downloads/20061106_DenimGroup_Secure_SW_LG_Org.ppt

- Director of Central Intelligence Directive 6/3 (DCID 6/3), *Protecting Sensitive Compartmented Information within Information Systems – Manual.* Accessed 28 January 2008 at: http://www.fas.org/irp/offdocs/dcid-6-3-manual.pdf

- DISA. *Application Security and Development STIG.* Draft Version 2, Release 0.1, 19 October 2007. Accessed 21 January 2007 at: http://iase.disa.mil/stigs/draft-stigs/application-security-dev-stigv2r0-1-102307.doc *– and –* *Application Security and Development Checklist,* Draft Version 1 Release 1, 20 April 2007. Accessed 21 January 2008 at: http://iase.disa.mil/stigs/draft-stigs/asd-checklist.doc

- DISA. *Application Security Checklist,* Version 2, Release 1.10, 16 November 2007. Accessed 17 December 2007 at: http://iase.disa.mil/stigs/checklist/application-security-checklist-v2r1-10.doc

- DISA. *Application Services STIG,* Version 1, Release 1.1, 17 January 2006. Accessed 21 January 2008 at: http://iase.disa.mil/stigs/stig/application-services-stig-v1r1.pdf

- Dobson, John E. and Brian Randell. "Building Reliable Secure Computing Systems out of Unreliable Insecure Components". *Proceedings of the 17th Annual Computer Security Applications Conference (ACSAC),* New Orleans, Louisiana, 10-14 December 2001. Accessed 19 December 2007 at: http://www.cs.ncl.ac.uk/research/pubs/inproceedings/papers/355.pdf

- DoD Instruction 8500.2, *Information Assurance (IA) Implementation.* 6 February 2003. Accessed 25 January 2008 at: http://www.dtic.mil/whs/directives/corres/html/850002.htm.

- DoD Instruction 8552.01, "Use of Mobile Code Technologies in DoD Information Systems". 23 October 2006. Accessed 19 January 2008 at: http://www.dtic.mil/whs/directives/corres/html/855201.htm

- Du, Wenliang and Aditya P. Mathur. "Vulnerability Testing of Software Systems Using Fault Injection". Technical Report COAST TR98-02, 6 April 1998. Accessed 2 January 2008 at: https://www.cerias.purdue.edu/tools_and_resources/bibtex_archive/archive/98-02.pdf

- Dubin, Joel. "Java security: Is it getting worse?". SearchSecurity.com, 12 July 2007. Accessed 11 December 2007 at: http://searchsecurity.techtarget.com/tip/0,289483,sid14_gci1263607,00.html

- Dustin, Elfriede. "The Software Trustworthiness Framework". 30 January 2007. Accessed 11 December 2007 at: http://www.veracode.com/Weblog/?p=22

- Dwaikat, Zaid. "Attacks and Countermeasures". *CrossTalk: The Journal of Defense Software Engineering,* October 2005. Accessed 21 December 2007 at: http://www.stsc.hill.af.mil/crosstalk/2005/10/0510Dwaikat.html

- Edge, Kenneth S. *A Framework for Analyzing and Mitigating the Vulnerabilities of Complex Systems via Attack and Protection Trees.* Air Force Institute of Technology doctoral thesis, July 2007. Accessed 21 January 2008 at: http://handle.dtic.mil/100.2/ADA472310

- Elgazzar, Mohamed. "Security in Software Localization". Microsoft Global Development and Computing portal, no date. Accessed 19 December 2007 at: http://www.microsoft.com/globaldev/handson/dev/secSwLoc.mspx

- Ellison, Robert J. "Trustworthy Composition: The System is Not Always the Sum of Its Parts". September 2005. Accessed 4 February 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/50.html?branch=1&language=1

- Ellison, Robert J. and Andrew P. Moore. "Trustworthy Refinement through Intrusion-Aware Design (TRIAD)". Technical Report CMU/SEI-2003-TR-002, March 2003. Accessed 22 March 2008 at: http://www.sei.cmu.edu/publications/documents/03.reports/03tr002.html

- Epstein, Jeremy. "SOA security: the only thing we have to fear is fear itself". *SOA Web Services Journal,* 3 December 2005. Accessed 21 January 2008 at: http://Webservices.sys-con.com/read/155630.htm

- Esterle, Alain, Hanno Ranck, and Burkard Schmitt. "Information Security: A New Challenge for the European Union". European Union Institute for Security Studies Chaillot Paper Number 76, March 2005. Accessed 28 January 2008 at: http://www.iss-eu.org/chaillot/chai76.pdf

- Fail-Safe C Version 1.0 Webpage. Accessed 21 January 2008 at: http://homepage.mac.com/t.sekiguchi/fsc/index.html

- Fedchak, Elaine, Thomas McGibbon, and Robert Vienneau. *Software Project Management for Software Assurance: A DACS State-of-the-Art Report,* Data & Analysis Center for Software Report Number 347617, 30 September 2007. Accessed 7 July 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/dhs/906-BSI.html

- Fernandez, Eduoardo B., J.C. Pelaez, and M.M. Larrondo-Petrie. "Attack patterns: A new forensic and design tool". *Proceedings of the Third Annual IFIP WG 11.9 International Conference on Digital Forensics,* Orlando, Florida, 29-31 January 2007. Accessed 11 September 2008 at: http://www.springerlink.com/content/mv0541345hx15345/

- Fiskiran, A. Murat and Ruby B. Lee. "Runtime execution monitoring (REM) to detect and prevent malicious code execution". *Proceedings of the 22nd IEEE International Conference on Computer Design,* San Jose, California, 11-13 October 2004, pages 452-457.

Accessed 21 January 2008 at:
http://palms.ee.princeton.edu/PALMSopen/fiskiran04runtime.pdf

- Flayer Web page. Accessed 21 January 2008 at: http://code.google.com/p/flayer/

- Fléchais, Ivan, Cecilia Mascolo, and M. Angela Sasse. "Integrating security and usability into the requirements and design process". *Proceedings of the Second International Conference on Global E-Security, London, United Kingdom, April 2006.* Accessed 21 January 2008 at: http://www.softeng.ox.ac.uk/personal/Ivan.Flechais/downloads/icges.pdf

- Fléchais, Ivan. *Designing Secure and Usable Systems*. University of London doctoral thesis, February 2005. Accessed 19 December 2007 at:
http://www.softeng.ox.ac.uk/personal/Ivan.Flechais/downloads/thesis.pdf

- Fléchais, Ivan. *Designing Secure and Usable Systems.* University of London doctoral thesis, February 2005. Accessed 19 December 2007 at:
http://www.softeng.ox.ac.uk/personal/Ivan.Flechais/downloads/thesis.pdf

- Fortify Software. Fortify Taxonomy: Software Security Errors Webpage. Accessed 21 January 2008 at: http://www.fortifysoftware.com/vulncat/

- Gansler, Jacques S. and Hans Binnendijk, editors.: "Information Assurance: Trends in Vulnerabilities, Threats and Technologies". National Defense University Working Paper, May 2003. Accessed 28 January 2008 at:
http://www.ndu.edu/ctnsp/IAverMay03.pdf

- Goertzel, Karen Mercedes, Theodore Winograd, *et al. Software Security Assurance: A State-of-the-Art Report.* Herndon, Virginia: Information Assurance Technology Analysis Center (IATAC) of the DTIC, 31 July 2007. Accessed 28 January 2008 at:
http://iac.dtic.mil/iatac/download/security.pdf

- Grossman, Jeremiah. "Input validation or output filtering, which is better?" On his Weblog, 30 January 2007. Accessed 14 December 2007 at:
http://jeremiahgrossman.blogspot.com/2007/01/input-validation-or-output-filtering.html

- Grossman, Jeremiah. "The Best Web Application Vulnerability Scanner in the World". On his Weblog, 23 October 2007. Accessed 14 July 2008 at:
http://jeremiahgrossman.blogspot.com/2007/10/best-Web-application-vulnerability.html

- Hafiz, Munawar. "Security Patterns and Secure Software Architecture." Tutorial presented at ACM Special Interest Group on Programming Languages International Conference on Object-Oriented Programming, Systems, Languages, and Applications, Portland, Oregon, 22-26 October 2006. Accessed 23 January 2008 at:

https://netfiles.uiuc.edu/mhafiz/www/ResearchandPublications/Security%20Patterns%20Talk.ppt

- Haley, Charles B. *Arguing Security: A Framework for Analyzing Security Requirements.* Open University doctoral thesis, March 2007. Accessed 23 January 2008 at: http://www.the-haleys.com/chaley/papers/Thesis-Final-DS.pdf

- Haley, Charles, Jonathan D. Moffett, Robin C. Laney, and Bashar Nuseibeh. "A Framework for Security Requirements Engineering". *Proceedings of the 2006 Software Engineering for Secure Systems Workshop,* Shanghai China, 20-21 May 2006, pages 35-42. Accessed 17 January 2008 at: http://www.the-haleys.com/chaley/papers/Haley-SESS06-p35.pdf

- Haley, Charles B., Robin C. Laney, Jonathan D. Moffett, and Bashar Nuseibeh. "Using Trust Assumptions with Security Requirements". *Requirements Engineering Journal,* Volume 11 Number 2, April 2006, pages 138-151. Accessed 11 September 2008 at: http://www.the-haleys.com/chaley/papers/REJ06.pdf

- Halkidis, Spyros T., Alexander Chatzigeorgiou, and George Stephanides. "A practical evaluation of security patterns". *Proceedings of the Sixth International Conference on Artificial Intelligence and Digital Communications,* Thessaloniki, Greece, 18-20 August 2006. Accessed 21 January 2008: http://www.inf.ucv.ro/~aidc/proceedings/2006/5%20shalkidis.pdf

- Hall, Anthony and Roderick Chapman. "Correctness by Construction: developing a commercial secure system". *IEEE Software,* January-February 2002, pages 18-25. Accessed 26 February 2008 at: http://www.anthonyhall.org/c_by_c_secure_system.pdf

- Hardened-PHP Project Month of PHP Bugs Website. Accessed 26 March 2008 at: http://www.php-security.org/

- Harold, Elliotte. "Configure SAX parsers for secure processing (Prevent entity resolution vulnerabilities and overflow attacks)". IBM DeveloperWorks, 27 May 2005. Accessed 9 April 2008 at: http://www-128.ibm.com/developerworks/xml/library/x-tipcfsx.html

- Hayre, Jaswinder S. and Jayasankar Kelath. "Ajax Security Basics". *SecurityFocus,* 19 June 2006. Accessed 13 December 2007 at: http://www.securityfocus.com/infocus/1868

- Heckman, Rocky. "Is Agile Development Secure?". CNET Builder.au. 8 August 2005. Accessed 4 April 2007 at: http://www.builderau.com.au/manage/project/soa/Is_Agile_development_secure_/0,39024668,39202460,00.htm *– and –* http://www.builderau.com.au/architect/sdi/soa/Is_Agile_development_secure_/0,39024602,39202460,00.htm

- Heitmeyer, Constance Heitmeyer. Automatic Construction of High Assurance Systems from Requirements Specifications Web page. Accessed 26 February 2008 at: http://chacs.nrl.navy.mil/personnel/heitmeyer.html

- Heitmeyer, Constance, Myla Archer, Elizabeth Leonard, and John McLean. "Applying Formal Methods to a Certifiably Secure Software System". *IEEE Transactions on Software Engineering,* Volume 34 Number 1, January 2008, pages 82-98. Accessed 26 February 2007 at: http://chacs.nrl.navy.mil/publications/CHACS/2008/2008heitmeyer-TSE.pdf

- Hennebrueder, Sebastian. "Java security in Web application, typical attacks, Tomcat security". Tutorial, 3 March 2007. Accessed 11 December 2007 at: http://www.laliluna.de/java-security-Web-application.html

- Hermes publications Webpage. Accessed 21 January 2008 at: http://www.research.ibm.com/people/d/dfb/hermes-publications.html

- Hope, Paco, Gary McGraw, and Annie I. Antón. Misuse and abuse cases: getting past the positive. *IEEE Security and Privacy,* May-June 2004, pages 32-34. Accessed 21 January 2008 at: http://www.cigital.com/papers/download/bsi2-misuse.pdf

- Hsu, Francis. "Input validation of client-server Web applications through static analysis". Presented at Web 2.0 Security and Privacy 2007, Oakland, California, 24 May 2007. Accessed 14 December 2007 at: http://seclab.cs.rice.edu/w2sp/2007/papers/paper-210-z_9464.pdfSecure

- Hu, Wei, Jason Hiser, Dan Williams, *et al.* "Secure and Practical Defense Against Code-Injection Attacks Using Software Dynamic Translation". *Proceedings of the 2006 ACM Virtual Execution Environments Conference,* Ottawa, Ontario, Canada, 10-13 June 2006. Accessed 31 December 2007 at: http://dependability.cs.virginia.edu/publications/2006/strata-isr.vee2006.pdf

- Hudson, Paul. "PHP—Secure Coding". *Linux Format,* Issue 56, August 2004. Accessed 22 March 2008 at: http://www.linuxformat.co.uk/wiki/index.php/PHP_-_Secure_coding

- IBM alphaWorks Security Workbench Development Environment for Java (SWORD4J). Accessed 11 December 2007 at: http://alphaworks.ibm.com/tech/sword4j/

- IBM Java Security Research Webpage. Accessed 11 December 2007 at: http://domino.research.ibm.com/comm/research_projects.nsf/pages/javasec.index.html

- IBM. Non-Functional Requirements High Availability Runtime patterns. Accessed 28 June 2008 at: http://www.ibm.com/developerworks/patterns/edge/at1-runtime.html

- International Council of Electronic Commerce Consultants (EC-Council) Certified Secure Programmer (CSP) and Certified Secure Application Developer (CSAD) certifications. Accessed 21 January 2008 at: http://www.eccouncil.org/ecsp/index.htm

- International Institute of Training, Assessment, and Certification Certified Secure Software Engineering Professional certification. Accessed 17 December 2007 at: http://www.iitac.org/content/view/146/lang,en/

- International Organization for Standards/International Electrotechnical Commission (ISO/IEC). "ISO/IEC TR 15942:2000: Information technology—Programming languages—Guide for the use of the Ada programming language in high integrity systems". Accessed 9 September 2008 at: http://standards.iso.org/ittf/PubliclyAvailableStandards/c029575_ISO_IEC_TR_15942_2000(E).zip

- Internet Security Alliance and U.S. Cyber Consequences Unit. "Cyber Security Check List". Look specifically at Areas 2 and 6. Accessed 25 January 2008 at: http://www.isalliance.org/content/view/144/292

- ISO/IEC. "ISO/IEC TR 24718:2005: Information technology—Programming languages—Guide for the use of the Ada Ravenscar Profile in high integrity systems". Accessed 9 September 2008 at: http://standards.iso.org/ittf/PubliclyAvailableStandards/c038828_ISO_IEC_TR_24718_2005(E).zip

- Jansen, Wayne A., Theodore Winograd, Karen Scarfone. *Guidelines on Active Content and Mobile Code.* NIST Special Publication 800-28, Version 2, March 2008. Accessed 26 March 2008 at: http://csrc.nist.gov/publications/nistpubs/800-28-ver2/SP800-28v2.pdf

- Jezequel, Jean-Marc and Bertrand Meyer. "Design by Contract: the lessons of Ariane". *IEEE Computer,* January 1997, pages 129-130. Accessed 11 September 2008 at: http://archive.eiffel.com/doc/manuals/technology/contract/ariane/page.html

- JSecurity Java security application development framework project Website. Accessed 11 December 2007 at: http://www.jsecurity.org/

- Jürjens, Jan, Joerg Schreck, and Peter Bartmann. "Model-based Security Analysis for Mobile Communications". *Proceedings of the 30th IEEE International Conference on Software Engineering,* Leipzig, Germany, 10-18 May 2008. Accessed 25 August 2008 at: http://mcs.open.ac.uk/jj2924/publications/papers/icsetele08-jurjens.pdf

- Kevin McFarlane's Design by Contract Framework. Accessed 7 July 2008 at: http://www.codeproject.com/KB/cs/designbycontract.aspx

- Khan, R.A. and K. Mustafa. "Secured Requirement Specification Framework (SRSF)". *American Journal of Applied Sciences,* Volume 5 Number 12, 2008, pages 1622-1629. Accessed 7 July 2008 at: http://www.scipub.org/fulltext/ajas/ajas5121622-1629.pdf

- Kienzle, Darrell M., Matthew C. Elder, David Tyree, and James Edwards-Hewitt. *Security Patterns Repository,* Version 1.0 *– and –* "Security Patterns for Web Application Development: Final Technical Report", 4 November 2003. Accessed 21 January 2008 at: http://www.modsecurity.org/archive/securitypatterns/ *- and -* http://www.scrypt.net/~celer/securitypatterns/

- Kis, Miroslav. Information Security Antipatterns in Software Requirements Engineering. *Proceedings of the Ninth Conference on Pattern Language of Programs,* Monticello, Illinois, 8-12 September 2002. Accessed 19 December 2007 at: http://jerry.cs.uiuc.edu/~plop/plop2002/final/mkis_plop_2002.pdf

- Kobus, Walter S. "Total Enterprise Security Solutions Applications Security Checklist". Accessed 17 December 2007 at: http://www.tess-llc.com/Application%20Security%20ChecklistV4.pdf

- Kocher, Paul, Ruby B. Lee, Gary McGraw, Anand Raghunathan, and Srivaths Ravi. "Security as a New Dimension in Embedded System Design". *Proceedings of the 41st Design Automation Conference,* San Diego, California, 7-11 June 2004. Accessed 17 January 2008 at: http://palms.ee.princeton.edu/PALMSopen/Lee-41stDAC_46_1.pdf

- Kortti, Heikki. "Input Attack Trees". Presented at Black Hat Japan, 5-6 October 2006. Accessed 21 January 2008 at: http://www.blackhat.com/presentations/bh-jp-06/BH-JP-06-Kortti-up.pdf

- Kumar, Naveen and Bruce Childers. Flexible Instrumentation for Software Dynamic Translation. *Proceedings of the Workshop on Exploring the Trace Space for Dynamic Optimization Techniques,* at the ACM International Conference on Supercomputing, New York, New York, June 2003. Accessed 31 December 2007 at: http://www.cs.pitt.edu/coco/papers/traces-kumar.pdf

- Kurz, John. "Dynamic Client-Side Input Validation". *ColdFusion Developer's Journal,* 1 May 2003. Accessed 14 December 2007 at: http://coldfusion.sys-con.com/read/41599.htm

- Laney, Robin C., Janet van der Linden, and Pete Thomas. "Evolving Legacy System Security Concerns Using Aspects". Open University Technical Report Number 2003/13, 11 November 2003. Accessed 9 April 2008 at: http://computing-reports.open.ac.uk/index.php/content/download/82/322/file/2003_13.pdf

- Lewis, Richard. "Temporary files security in-depth". Posted on his Application Security blog, 12 October 2006. Accessed 31 December 2007 at: http://secureapps.blogspot.com/2006/10/temporary-files-security-in-depth.html

- Li, Ninghui, Ziqing Mao, and Hong Chen. "Usable Mandatory Integrity Protection for Operating Systems". *Proceedings of the 2007 IEEE Symposium on Security and Privacy,* Oakland, California, 20-23 May 2007. Accessed 20 March 2008 at: http://www.cs.purdue.edu/homes/ninghui/papers/umip_oakland07.pdf

- Li, Zhenmin, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. "Have things changed now? An empirical study of bug characteristics in modern open source software". *Proceedings of Workshop on Architectural and System Support for Improving Software Dependability,* New York, New York, October 2006, pages 25-33. Accessed 11 September 2008 at: http://opera.cs.uiuc.edu/~lintan2/publications/bugchar_asid06.pdf —*and*— *Presentation slides:* http://opera.cs.uiuc.edu/~lintan2/publications/bugchar_asid06_slides.pdf

- Lieberman, Danny. "Software security assessment of production systems". 2006. Accessed 9 April 2008 at: http://www.software.co.il/content/view/195/41/. Also published by the Control Policy Group as "Practical Software Security Assessment". 2007. Accessed 9 April 2008 at: http://www.controlpolicy.com/practicalsoftwaresecurityassessment

- Lipner, Steve and Michael Howard. "The Trustworthy Computing Security Development Lifecycle". Microsoft Developer Network, March 2005. Accessed 17 December 2007 at: http://msdn2.microsoft.com/en-us/library/ms995349.aspx

- Lopienski, Sebastian. "Conseil Européen pour la Recherche Nucléaire[54] Security checklist for software developers". Accessed 17 December 2007 at: http://info-secure-software.Web.cern.ch/info-secure-software/SecurityChecklistForSoftwareDevelopers.pdf

- Luu, Tieu. "Separation of Concerns in Web Service Implementations". *OnJava,* 6 September 2006. Accessed 11 December 2007 at: http://www.onjava.com/pub/a/onjava/2006/09/06/separation-of-concerns-in-Web-services.html

- Mathers, Greg, Joseph K. Simpson, *et al.* "Framework for the Application of Systems Engineering in the Commercial Aircraft Domain", DRAFT Version 1.2a, 28 July 2000 - Section 4.0, "Commercial Aircraft Process Relationships". Accessed 25 August 2008 at: http://www.incose.org/ProductsPubs/pdf/techdata/SEApps-TC/FrameworkForApplicOfSEToCommercialAircraftDomain.pdf

- McGraw, Gary and Greg Hoglund. "Chapter 3: Reverse Engineering and Program Understanding" (excerpt from *Exploiting Software: How to Break Code).* Accessed 19

54 Now the European Organization for Nuclear Research.

January 2008 at: http://www.amazon.com/Exploiting-Software-Break-Addison-Wesley-Security/dp/0201786958/ref=pd_sim_b_title_3

- Mead, Nancy R., Eric D. Hough, Theodore R. Stehney, II. "Security Quality Requirements Engineering (SQUARE) Methodology". Technical Report CMU/SEI-2005-TR-009|ESC-TR-2005-009, November 2005. Accessed 19 December 2007 at: http://www.cert.org/archive/pdf/05tr009.pdf

- Mehta, Dharmesh M. "Application Security Testing Cheat Sheet". SmartSecurity.blogspot.com, 2007. Accessed 11 December 2007 at: http://www.edocr.com/doc/264/application-security-testing-cheat-sheet

- Meier, J.D., Alex Mackman, Michael Dunner, Srinath Vasireddy, Ray Escamilla, and Anandha Murukan. "Code Access Security in Practice". Chapter 8 of *Improving Web Application Security: Threats and Countermeasures,* Microsoft Developer Network, January 2006. Accessed 8 April 2008 at: http://msdn2.microsoft.com/en-us/library/aa302424.aspx

- Memory Safe C Compiler Webpage. Accessed 21 January 2008 at: http://www.seclab.cs.sunysb.edu/mscc/

- Menendez, James N. and Scott Wright. *A Guide to Understanding Trusted Distribution in Trusted Systems* (the "Dark Lavender Book"). NCSC-TG-008, 15 December 1988. Accessed 19 December 2007 at: http://handle.dtic.mil/100.2/ADA392816 *– and –* http://www.fas.org/irp/nsa/rainbow/tg008.htm

- Meyer, Bertrand. "Applying 'Design by Contract'". *IEEE Computer,* Volume 25 Issue 10, October 1992, pages 40-51. Accessed 26 August 2008 at: http://se.ethz.ch/~meyer/publications/computer/contract.pdf

- Microsoft Corporation. "Secure Coding Guidelines". Chapter in *Visual Studio 2008 .NET Framework Developer's Guide.* Accessed 22 March 2008 at: http://msdn2.microsoft.com/en-us/library/d55zzx87.aspx

- Microsoft Corporation. Baseline Security Analyzer 2.0 Webpage. Accessed 12 December 2007 at: http://www.microsoft.com/technet/security/tools/mbsa2/default.mspx

- Microsoft SDL Weblog. Accessed 17 December 2007 at: http://blogs.msdn.com/sdl/

- Microsoft Security Developer Center Application Threat Modeling Webpage. Accessed 25 January 2007 at: http://msdn2.microsoft.com/en-us/security/aa570413.aspx

- Microsoft Security Developer Center. Writing Secure Code Webpage. Accessed 12 December 2007 at: http://msdn2.microsoft.com/en-us/security/aa570401.aspx

- Mikhalenko, Peter V. "Understanding the Java security model". TechRepublic, 18 April 2007. Accessed 11 December 2007 at: http://articles.techrepublic.com.com/5100-3513-6177275.html

- Mile2: Secure Coding Training. Accessed 12 December 2007 at: http://www.mile2.com/IT_Security_Training.html

- Minkiewicz, Arlene F. "Security in a COTS-Based Software System". *CrossTalk: The Journal of Defense Software Engineering,* November 2005. Accessed 17 December 2007 at: http://www.stsc.hill.af.mil/crosstalk/2005/11/0511Minkiewicz.html

- Minkiewicz, Arlene F. "Security in a COTS-Based Software System". *CrossTalk: The Journal of Defense Software Engineering,* November 2005. Accessed 4 February 2008 at: http://www.stsc.hill.af.mil/crosstalk/2005/11/0511Minkiewicz.html

- MISRA C Website. Accessed 21 January 2008 at: http://www.misra-c2.com/

- Moffett, Jonathan D., Charles B. Haley, and Bashar Nuseibeh. "Core Security Requirements Artefacts". Open University Technical Report Number 2004/23, 21 June 2004. Accessed 26 January 2008 at: http://computing-reports.open.ac.uk/index.php/content/download/166/999/file/2004_23.pdf

- Mouratidis, Haralambos and Paolo Giorgini (2007) "Secure Tropos: A Security-Oriented Extension of the Tropos Methodology". *International Journal of Software Engineering and Knowledge Engineering,* Volume 17 No. 2, April 2007, pages 285-309. Accessed 25 August 2008 at: http://www.dit.unitn.it/~pgiorgio/papers/IJSEKE06-1.pdf

- Mouratidis, Haralambos, Michael Weiss, and Paolo Giorgini. "Modelling Secure Systems Using an Agent-Oriented Approach and Security Patterns". *International Journal of Software Engineering and Knowledge Engineering,* Volume 16 Number 3, 2006, pages 471-498. Accessed 21 January 2008 at: http://www.scs.carleton.ca/~weiss/papers/ijseke06.pdf

- Mouratidis, Haralambos, Paolo Giorgini, and Gordon Manson. "Using Security Attack Scenarios to Analyse Security During Information Systems Design". *Proceedings of the International Conference on Enterprise Information Systems,* Porto, Portugal, April 2004. Accessed 17 December 2007 at: http://homepages.uel.ac.uk/H.Mouratidis/Paper91_CR.pdf

- Nagappan, Ramesh. "Demystifying Java security—Part 1". SearchSoftwareQuality.com, 21 June 2006. Accessed 11 December 2007 at: http://searchsoftwarequality.techtarget.com/originalContent/0,289142,sid92_gci1195320,00.html *—and—* "Demystifying Java security—Part 2". SearchSoftwareQuality.com, 23 June 2006. Accessed 11 December 2007 at: http://searchsoftwarequality.techtarget.com/originalContent/0,289142,sid92_gci1195332,00.html

- NASA Reducing Software Security Risk Through an Integrated Approach project. Security Checklist for the External Release of Software page. Accessed 17 December 2007 at: http://rssr.jpl.nasa.gov/ssc2/index.html

- National Academy of Sciences Computer Science and Telecommunications Board. *Cybersecurity Today and Tomorrow: Pay Now or Pay Later* (excerpts). Washington, D.C.: National Academies Press, 2002. Accessed 28 January 2008 at: http://books.nap.edu/openbook.php?isbn=0309083125 —*and*– http://books.nap.edu/html/cybersecurity/

- National Computer Security Center. *A Guide to Understanding Configuration Management in Trusted Systems* (the "Amber Book"). NCSC-TG-006, 28 March 1988. Accessed 19 December 2007 at: http://handle.dtic.mil/100.2/ADA392775 —*and*– http://www.fas.org/irp/nsa/rainbow/tg006.htm

- National Defense Industrial Association System Assurance Committee. *Engineering for System Assurance,* Version 0.90, 22 April 2008. Accessed 30 May 2008 at: http://www.acq.osd.mil/sse/ssa/docs/SA+guidebook+v905-22Apr08.pdf

- National Institute of Advanced Industrial Science (Tokyo, Japan) Technology Research Center for Information Security. "Fail-Safe C: a memory-safe compile for the C language". Accessed 21 January 2008 at: http://www.rcis.aist.go.jp/project/FailSafeC-en.html

- Neuhaus, Stephan, Thomas Zimmermann, Christian Holler, and Andreas Zeller. "Predicting vulnerable software components". *Proceedings of the 14th ACM Conference on Computer and Communications Security,* Alexandria, Virginia, 29 October-2 November 2007, pages 529-540. Accessed 11 September 2008 at: http://www.st.cs.uni-sb.de/publications/files/neuhaus-ccs-2007.pdf

- Neumann, Peter G. "Principled assuredly trustworthy composable architectures". CDRL A001 Final Report, 28 December 2004. Accessed 21 January 2008 at: http://www.csl.sri.com/users/neumann/chats4.html

- Neumann, Peter G. and Richard J. Feiertag. "PSOS (Provably Secure Operating System) Revisited". *Proceedings of the 19th Annual Computer Security Applications Conference,* Las Vegas, Nevada, 8-12 December 2003. Accessed 4 February 2008 at: http://www.csl.sri.com/users/neumann/psos03.pdf

- Nice Language. Accessed 7 July 2008 at: http://c2.com/cgi/wiki?NiceLanguage

- Niski, Joe. "Application Security Frameworks". Burton Group Application Platform Strategies Reference Architecture Technical Position Paper, January 2008. Accessed 2 February 2008 at: http://www.burtongroup.com/Research/PublicDocument.aspx?cid=21

- NIST Computer Security Division. *Minimum Security Requirements for Federal Information Systems.* FIPS Publication 200, March 2006. Accessed 28 January 2008 at: http://csrc.nist.gov/publications/fips/fips200/FIPS-200-final-march.pdf

- NIST Computer Security Division. *Standards for Security Categorization of Federal Information and Information Systems.* FIPS 199, February 2004. Accessed 26 January 2008 at: http://csrc.nist.gov/publications/fips/fips199/FIPS-PUB-199-final.pdf

- NIST SAMATE Tool Survey. Accessed 3 September 2008 at: https://samate.nist.gov/index.php/Tools

- NIST, National Vulnerability Database (NVD). Accessed 8 September 2008 at: http://nist.nvd.gov

- NIST. *Generally Accepted Principles and Practices for Securing Information Technology Systems.* Special Publication 800-14, September 1996. Accessed 29 August 2008 at: http://csrc.nist.gov/publications/nistpubs/800-14/800-14.pdf

- NIST. *An Introduction to Computer Security: The NIST Handbook.* Special Publication 800-12, October 1995. Accessed 28 January 2008 at: http://csrc.nist.gov/publications/nistpubs/800-12/handbook.pdf

- NIST. *Information Security Handbook: A Guide for Managers.* Special Publication 800-100, October 2006. Accessed 29 August 2008 at: http://csrc.nist.gov/publications/nistpubs/800-100/SP800-100-Mar07-2007.pdf

- NIST. National Vulnerability Database National Checklist Program Repository Webpage. Accessed 26 January 2008 at: http://checklists.nist.gov/ncp.cfm?repository

- NIST. *Recommended Security Controls for Federal Information Systems.* Special Publication 800-53 Rev. 2, December 2007. Accessed 29 August 2008 at: http://csrc.nist.gov/publications/nistpubs/800-53-Rev2/sp800-53-rev2-final.pdf

- NIST. *Security Considerations in the Information System Development Life Cycle.* Special Publication 800-64 DRAFT Rev 2, June 2004. Accessed 29 August 2008 at: http://csrc.nist.gov/publications/drafts/800-64-rev2/draft-SP800-64-Revision2.pdf

- NIST. Software Assurance Metrics and Tool Evaluation Website. Accessed 19 December 2007 at: http://samate.nist.gov/index.php/Main_Page

- Norton, Francis. "Implementing Real world Data Input Validation Using Regular Expressions" (for .NET). *Simple-Talk,* 14 May 2007. Accessed 14 December 2007 at: http://www.simple-talk.com/dotnet/.net-framework/implementing-real world-data-input-validation-using-regular-expressions/

- NSA. "National Information Security Education and Training Program: Introduction to Information Assurance". Online presentation, 1998. Accessed 28 January 2008 at: http://security.isu.edu/ppt/shows/information_assurance_files/frame.htm *– or –* http://security.isu.edu/ppt/pdfppt/information_assurance.pdf

- octotrike.org Tools Webpage. Accessed 25 January 2008 at: http://www.octotrike.org/

- Oertli, Thomas. "Secure Programming in PHP". 30 January 2002. Accessed 11 April 2008 at: http://www.cgisecurity.com/lib/php-secure-coding.html

- Ollmann, Gunter. "Application Assessment Questioning". Accessed 17 December 2007 at: http://www.technicalinfo.net/papers/AssessmentQuestions.html

- Ollmann, Gunter. "Second-order Code Injection: Advanced Code Injection Techniques and Testing Procedures". Undated whitepaper posted on his Weblog. Accessed 16 January 2008 at: http://www.technicalinfo.net/papers/SecondOrderCodeInjection.html

- Open Web Application Security Project (OWASP) CLASP Project Webpage. Accessed 17 December 2007 at: http://www.owasp.org/index.php/Category:OWASP_CLASP_Project

- Oracle Enterprise Configuration Management Pack. Accessed 26 August 2008 at: http://www.oracle.com/technology/products/oem/pdf/ds_config_pack.pdf

- Oracle Software Configuration Manager. Accessed 26 August 2008 at: http://www.oracle.com/support/premier/software-configuration-manager.html

- Over, James W. "TSP for Secure Systems Development" (presentation). Accessed 17 December 2007 at: http://www.sei.cmu.edu/tsp/tsp-secure-presentation/tsp-secure.pdf

- OWASP AJAX Security Project Webpage. Accessed 13 December 2007 at: http://www.owasp.org/index.php/Category:OWASP_AJAX_Security_Project

- OWASP Code Review Project page. Accessed 14 December 2007 at: http://www.owasp.org/index.php/Category:OWASP_Code_Review_Project

- OWASP Java Project page. Accessed 11 December 2007 at: http://www.owasp.org/index.php/Category:OWASP_Java_Project

- OWASP Testing Project page (includes OWASP Testing Guide v2). Accessed 11 December 2007 at: http://www.owasp.org/index.php/Category:OWASP_Testing_Project

- OWASP Threat Risk Modeling Webpage. Accessed 25 January 2008 at: http://www.owasp.org/index.php/Threat_Risk_Modeling

- OWASP WebScarab. Accessed 23 July 2008 at: http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project

- OWASP. *Guide to Building Secure Web Applications,* Version 2.0. Accessed 8 September 2008 at: http://www.owasp.org/index.php/Category:OWASP_Guide_Project#OWASP_Development_Guide_2.0_Downloads

- OWASP. *Testing Guide,* Version 2.0. Accessed 8 September 2008 at: http://www.lulu.com/content/1375886

- Patterns and Practices Guidance Library Wiki. "Reg Ex Input Val Code—Validate User Input with Regular Expressions" [C#]. Accessed 14 December 2007 at: http://www.guidancelibrary.com/default.aspx/Home.RegExInputValCode

- Patternshare Repository of Security Patterns. Accessed 17 December 2007 at: https://netfiles.uiuc.edu/mhafiz/www/ResearchandPublications/Patternshare_Security_Patterns.htm

- Paul, Nathanael and David Evans. "Comparing Java and .NET Security: Lessons Learned and Missed". *Computers and Security,* Volume 25 Issue 5, July 2006. Accessed 11 December 2007 at: http://www.cs.virginia.edu/~nrp3d/papers/computers_and_security-net-java.pdf

- Pauli Joshua J. and Dianxiang Xu. "Misuse Case-Based Design and Analysis of Secure Software Architecture". *Proceedings of the International Conference on Information Technology Coding and Computing,* Las Vegas, Nevada, April 2005. Accessed 17 December 2007 at: http://www.homepages.dsu.edu/paulij/pubs/pauli-xu-ITCC05.pdf

- Pauli, Joshua and Dianxiang Xu. "Threat-Driven Architectural Design of Secure Information Systems". *Proceedings of the Seventh International Conference on Enterprise Information Systems,* Miami, Florida, 24-28 May 2005. Accessed 19 December 2007 at: http://cs.ndsu.edu/%7Edxu/publications/pauli-xu-ICEIS05.pdf

- Pauli, Joshua J. and Dianxiang Xu. "Trade-off Analysis of Misuse Case-based Secure Software Architectures: A Case Study". *Proceedings of the 3rd International Workshop on Modeling, Simulation, Verification and Validation of Enterprise Information Systems,* Miami, Florida, May 2005, pages 89-95. Accessed 13 December 2007 at: http://cs.ndsu.edu/~dxu/publications/pauli-xu-MSVVEIS05.pdf

- Peeters, Johann. "Agile Security Requirements Engineering". Presented at the Symposium on Requirements Engineering for Information Security, Paris, France, 29 August 2005. Accessed 21 January 2008 at: http://johanpeeters.com/papers/abuser%20stories.pdf

- Petrovic, Mark. "Discovering a Java Application's Security Requirements". *OnJava,* 3 January 2007. Accessed 11 December 2007 at: http://www.onjava.com/pub/a/onjava/2007/01/03/discovering-java-security-requirements.html

- PHP Security Consortium Website. Accessed 26 March 2008 at: http://phpsec.org/

- Polydys, Mary Linda and Stan Wisseman. *Software Assurance in Acquisition: Mitigating Risks to the Enterprise,* Draft Version 1.0, 10 September 2007. Accessed 30 May 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/908.html?branch=1&language=1

- Practical Threat Analysis Website. Accessed 25 January 2008 at: http://www.ptatechnologies.com/

- Praxis High Integrity Systems. Accessed 27 January 2008 at: http://www.praxis-his.com/

- Praxis High Integrity Systems. Requirements Engineering VErification and VAlidation (REVEAL) Webpage. Accessed 19 December 2007 at: http://www.praxis-his.com/reveal/

- Praxis High Integrity Systems. SafSec: Integration of Safety and Security Certification webpage. Accessed 21 January 2008 at: http://www.praxis-his.com/safsec/index.asp

- Praxis High Integrity Systems. SPARKAda. Accessed 8 September 2008 at: http://www.praxis-his.com/sparkada/index.asp

- QASec.com—Software Security Testing in Quality Assurance and Development Webpage. Accessed 11 December 2007 at: http://www.qasec.com/

- Reza Ayatollahzadeh Shirazi, Mohammad, Pooya Jaferian, Golnaz Elahi, Hamid Baghi, and Babak Sadeghian. "RUPSec: An Extension on RUP for Developing Secure Systems—Requirements Discipline". *Proceedings of the World Academy of Science, Engineering, and Technology,* Volume 4, February 2005. Accessed 7 July 2008 at: http://www.waset.org/pwaset/v4/v4-51.pdf. *Farsi version accessed 7 July 2008 at: http://hamkelasy.com/files/pdfarticles/fani_moh/61013860203_(www.hamkelasy.com).pdf*

- Ricci, Lawrence and Larry McGinnes. "Embedded System Security: Designing Secure Systems with Windows CE". AppliedData.net white paper, not dated. Accessed 9 April 2008 at: http://www.applieddata.net/WP/Whitepaper_Secure_Windows_CE.pdf

- Ruby on Rails Security Project Website. Accessed 28 January 2008 at: http://www.rorsecurity.info

- Ruderman, Jesse. "JavaScriptSecurity: Same Origin". Mozilla.org Website, undated. Accessed 26 March 2008 at: http://www.mozilla.org/projects/security/components/same-origin.html

- Safe C String Library v1.0.3 Web page. Accessed 21 January 2008 at: http://www.zork.org/safestr/

- SAFECode (Static Analysis For safe execution of Code). "Software Assurance: An Overview of Current Industry Best Practices". February 2008. Accessed 14 February 2008 at: http://www.safecode.org/publications/SAFECode_BestPractices0208.pdf

- Safonov, Vladimir O. Classroom Materials from "Secure Software Engineering" course taught at St. Petersburg University, Russia [in English]. Accessed 17 January 2008 at: http://www.academicresourcecenter.net/curriculum/pfv.aspx?ID=6753

- Saltzer, Jerome H. and Michael D. Schroeder. "The protection of information in computer systems." *Proceedings of the IEEE,* 1975, Volume 63 Issue 9, pages 1278-1308. Accessed 11 September 2008 at: http://www.acsac.org/secshelf/papers/protection_information.pdf –*and*– http://www.ece.rutgers.edu/~parashar/Classes/03-04/ece572/papers/protection.pdf

- SANS (SysAdmin, Audit, Networking and Security) Institute Security Consensus Operational Readiness Evaluation Web Application Security Checklist Webpage. Accessed 21 January 2008 at: http://www.sans.org/score/Webappschecklist.php

- SANS Institute. "C Secure Coding Tasks, Skills and Knowledge." April 2007. Accessed 4 April 2008 at: http://www.sans.org/gssp/SANS-SSI%20C%20Blueprint%20(9-07).pdf

- SANS Software Security Institute GIAC Secure Programmer certifications. Accessed 21 January 2008 at: http://www.sans-ssi.org/

- Schneider, Thorsten. *(S2e) Integrated: Process Oriented Secure Software Development Model.* Version 1.0, 2007 [in German]. Accessed 17 December 2007 at: http://model.secure-software-engineering.com/

- Schneider, Thorsten. "Secure Software Engineering Processes: Improving the Software Development Life Cycle to Combat Vulnerability". *Software Quality Professional.* Volume 8 Issue 1, December 2006. Available (with free registration) from: http://www.asq.org/pub/sqp/past/vol9_issue1/sqpv9i1schneider.pdf

- Schneier, Bruce. "Attack trees: Modeling security threats". *Dr. Dobbs Journal,* December 1999. Accessed 21 January 2008: http://www.schneier.com/paper-attacktrees-ddj-ft.html –*and*– http://www.ddj.com/184411129

- Seacord, Robert, Noopur Davis, Chad Dougherty, Nancy Mead, and Robert Mead. "How to write secure C/C++ application code for your embedded design: Part 1—

Some secure software design principles". *Embedded.com,* 10 September 2007. Accessed 13 December 2007 at: http://www.embedded.com/design/202300629; "Part 2—Systems Quality Requirements Engineering". *Programmable Logic Design Line*, 15 October 2007. Accessed 13 December 2007 at: http://www.pldesignline.com/202402853

- SearchSoftwareQuality.com. Learning Guide: Application security testing techniques, 14 September 2006. Available at: http://searchsoftwarequality.techtarget.com/loginMembersOnly/1,289498,sid92_gci1215847,00.html [Requires registration before download.]

- secologic. "A Short Guide to Input Validation". Version 1.0, 25 April 2007. Accessed 14 December 2007 at: http://www.secologic.org/downloads/Web/070509_secologic-short-guide-to-input-validation.pdf

- Secure PHP Wiki. Accessed 14 December 2007 at: http://www.securephpwiki.com/index.php/Main_Page

- Secure Software Inc. *CLASP: Comprehensive Lightweight Application Security Process.* Version 2.0, 2006. Accessed 17 December 2007 at: http://searchappsecurity.techtarget.com/searchAppSecurity/downloads/clasp_v20.pdf

- Secure Software, Inc. "Risk in the Balance: How the Right Mix of Static Analysis and Dynamic Analysis Technologies Can Strengthen Application Security". 2004. Accessed 3 January 2008 at: http://secureitalliance.org/blogs/files/164/1137/Risk%20in%20the%20Bal_wp.pdf

- Secure University Software Security Expert Bootcamp certification. Accessed 21 January 2008 at: http://www.securityuniversity.net/classes_SI_SoftwareSecurity_Bootcamp.php

- Security (C# Programming Guide). Accessed 25 March 2008 at: http://msdn2.microsoft.com/en-us/library/ms173195.aspx

- Security Tutorial. Accessed 25 March 2008 at: http://msdn2.microsoft.com/en-us/library/aa288469(VS.71).aspx

- SecurityConcerns in Ruby on Rails Wiki. 10 January 2008. Accessed 26 March 2008 at: http://wiki.rubyonrails.org/rails/pages/SecurityConcerns

- SecurityPatterns.org Website. Accessed 19 December 2007 at: http://www.securitypatterns.org/index.html

- Shah, Shreeraj. "Top 10 Ajax Security Holes and Driving Factors". *Help Net Security,* 10 November 2006. Accessed 13 December 2007 at: http://www.net-security.org/article.php?id=956

- Shakil, Kamran. "Security Features in C#." Csharphelp.com. Accessed 25 March 2008 at: http://www.csharphelp.com/archives/archive189.html

- Shi, Weidong, Hsien-Hsin S. Lee, Chenghuai Lu, and Mrinmoy Ghosh. "Towards the Issues in Architectural Support for Protection of Software Execution". Georgia Institute of Technology Report Number CERCS;GIT-CERCS-04-29, 2004. Accessed 31 December 2007 at: http://smartech.gatech.edu/bitstream/1853/4949/1/git-cercs-04-29.pdf

- Shiflett, Chris. "Cross-Domain Ajax Insecurity". On his PHP and Web Application Security Weblog, 9 August 2006. Accessed 26 January 2008 at: http://shiflett.org/Weblog/2006/aug/cross-domain-ajax-insecurity

- Shiva, Sajjan G. and Lubna Abou Shala. "Adding Security to the Rational Unified Process". *Proceedings of the First Annual Computer Security Conference,* Myrtle Beach, South Carolina, 11-13 April 2007. Accessed 7 July 2008 at: http://computersecurityconference.com/Papers2007/CSC2007Shala.doc

- Sindre, Guttorm and Andreas L. Opdahl. "Capturing Security Requirements through Misuse Cases". *Proceedings of the Norsk Informatikkonferanse,* Tromsø, Norway, 26-28 November 2001. Accessed 21 January 2008 at: http://folk.uio.no/nik/2001/21-sindre.pdf

- Sindre, Guttorm and Andreas L. Opdahl. "Templates for misuse case description". *Proceedings of the Seventh International Workshop on Requirements Engineering Foundation for Software Quality,* Interlaken, Switzerland, 4-5 June 2001. Accessed 11 September 2008 at: http://swt.cs.tu-berlin.de/lehre/saswt/ws0506/unterlagen/TemplatesforMisuseCaseDescription.pdf

- Singhal, Anoop, Theodore Winograd, and Karen Scarfone. *Guide to Secure Web Services.* NIST Special Publication 800-95, August 2007. Accessed 26 January 2008 at: http://www.csrc.nist.gov/publications/nistpubs/800-95/SP800-95.pdf

- Sivaramakrishnan, Hariharan. *On the Use of Fault Injection to Discover Security Vulnerabilities in Applications.* University of Maryland master of science thesis, May 2006. Accessed 31 December 2007 at: https://drum.umd.edu/dspace/bitstream/1903/3566/1/umi-umd-3404.pdf

- Software Assurance Workforce Education and Training Working Group. *SoftwareAssurance: A Curriculum Guide to the Common Body of Knowledge to Produce, Acquire and Sustain Secure Software.* Draft Version 1.2, 29 October 2007. Accessed 10 March 2008 at: https://buildsecurityin.us-cert.gov/daisy/bsi/940/version/1/part/4/data/CurriculumGuideToTheCBK.pdf?branch=main&language=default

- *Software Security: Building Security In* Webpage. Accessed 17 December 2007 at: http://www.swsec.com/

- SPARKAda Webpage. Accessed 21 January 2008 at: http://www.praxis-his.com/sparkada/

- Srivastava, Amitabh. "Engineering Quality Software". Presented at the 6th International Conference on Formal Engineering Methods, Seattle, Washington, November 2004. Accessed 26 February 2008 at: http://research.microsoft.com/conferences/icfem2004/Presentations/AmitabhSrivastava.ppt

- Stamos, Alex and Zane Lackey. "Attacking AJAX Web Applications: Vulns 2.0 or Web 2.0". Presented at Black Hat USA 2006, 3 August 2006. Accessed 26 January 2008 at: http://www.isecpartners.com/files/iSEC-Attacking_AJAX_Applications.BH2006.pdf

- Stanford University Center for Professional Development Software Security Foundations Certificate. Accessed 21 January 2008 at: http://scpd.stanford.edu/scpd/courses/ProEd/compSec/

- Sun Developer Network Java SE Security page. Accessed 11 December 2007 at: http://java.sun.com/javase/technologies/security/

- Sun Java 6 Security Documentation page. Accessed 11 December 2007 at: http://java.sun.com/javase/6/docs/technotes/guides/security/index.html

- Sun Microsystems. "JavaScript Security". Chapter 14 of *Client-Side JavaScript Guide,* 27 May 1999. Accessed 26 March 2008 at: http://docs.sun.com/source/816-6409-10/sec.htm

- Sun Microsystems. Secure Coding Guidelines for the Java Programming Language, Version 2.0. Accessed 22 March 2008 at: http://java.sun.com/security/seccodeguide.html

- Suto, Larry. "Analyzing the Effectiveness and Coverage of Web Application Security Scanners". October 2007. Accessed 14 July 2008 at: http://www.stratdat.com/Webscan.pdf

- Taylor, John. "JavaScript Security in Mozilla". Mozilla.org Website, undated. Accessed 26 March 2008 at: http://www.mozilla.org/projects/security/components/jssec.html

- TenorLogic.com Java Security Resources page. Accessed 11 December 2007 at: http://www.tenorlogic.com/

- TestingSecurity.com—Teaching How to Perform Security Testing Webpage. Accessed 11 December 2007 at: http://www.testingsecurity.com/

- The Code Project. "Understanding .NET Code Access Security". 14 January 2004. Accessed 8 April 2008 at: http://www.codeproject.com/KB/security/UB_CAS_NET.aspx

- The CORAS Project Webpage. Accessed 25 January 2008 at: http://coras.sourceforge.net/

- The Jass Page. Accessed 7 July 2008: http://csd.informatik.uni-oldenburg.de/~jass/

- The Java Modeling Language (JML). Accessed 7 July 2008 at: http://www.eecs.ucf.edu/~leavens/JML/index.shtml

- The MITRE Corporation. CAPEC-Common Attack Pattern Enumeration and Classification Website. Accessed 14 December 2007 at: http://capec.mitre.org/index.html

- The MITRE Corporation. Common Weakness Enumeration Website. Accessed 21 January 2008 at: http://cwe.mitre.org/

- The PHP Group. "A Note on Security in PHP". Not dated. Accessed 26 March 2008 at: http://www.php.net/security-note.php

- The Secure Programming Council. "Essential Skills for Secure Programmers Using Java/JavaEE." November 2007. Accessed 4 April 2008 at: http://www.sans.org/gssp/essential_skills_java.pdf

- University of Southern California Center for Systems and Software Engineering. Security Economics and Threat Modeling for Information Technology (IT) Systems—A Stakeholder Value Driven Approach project Webpage. Accessed 25 January 2008 at: http://sunset.usc.edu/csse/research/COTS_Security/index.html

- Vaas, Lisa. "Java Security Traps Getting Worse". *eWeek,* 9 May 2007. Accessed 11 December 2007 at: http://www.eweek.com/article2/0,1759,2128071,00.asp

- van Lamsweerde "A. Elaborating Security Requirements by Construction of Intentional Anti-Models". *Proceedings of the 26th International Conference on Software Engineering,* Edinburgh, Scotland, May 2004, pages 148-157. Accessed 19 January 2008 at: http://www.info.ucl.ac.be/Research/Publication/2004/avl-Icse04-AntiGoals.pdf

- van Lamsweerde, Axel, Simon Brohez, Renaud De Landtsheer, David Janssens. "From System Goals to Intruder Anti-Goals: Attack Generation and Resolution for Security Requirements Engineering". *Proceedings of the 2003 Workshop on Requirements for High Assurance Systems,* Monterey, California, September 2003, pages 49-56. Accessed 17 January 2008 at: http://www.cs.toronto.edu/~jm/2507S/Readings/avl-RHAS03.pdf – *and* – http://www.cs.ndsu.nodak.edu/~vgoel/Security-Engineering/avl-RHAS03.pdf – *and* – http://www.info.ucl.ac.be/Research/Publication/2003/avl-RHAS03.pdf

- Vault: a programming language for reliable systems. Accessed 21 January 2008 at: http://research.microsoft.com/vault/

- Verbauwhede, I. and Patrick Schaumont. "Design Methods for Security and Trust". *Proceedings of the Design Automation and Test Conference in Europe,* Nice, France, April 2007. Accessed 11 September 2008 at: http://www.cosic.esat.kuleuven.be/publications/article-875.pdf

- Viega, John and Matt Messier. "Input Validation in C and C++". Chapter excerpt from *Secure Programming Cookbook for C and C++,* posted on O'Reilly Network Website, 20 May 2003. Accessed 19 December 2007 at: http://www.oreillynet.com/pub/a/network/2003/05/20/secureprogckbk.html

- Walker, Kenneth M., Daniel F. Sterne, M. Lee Badger, Michael J. Petkac, David L. Shermann, and Karen A. Oostendorp. "Confining Root Programs with Domain and Type Enforcement (DTE)". *Proceedings of the 6th USENIX Unix Security Symposium,* San Jose, California, July 1996. Accessed 20 March 2008 at: http://www.usenix.org/publications/library/proceedings/sec96/full_papers/walker/walker.ps

- Wallace, Dolores R. and Laura M. Ippolito. *A Framework for the Development and Assurance of High Integrity Software.* NIST Special Publication 500-223, December 1994, Section 3.4 "Software Configuration Management Process". Accessed 26 January 2008 at: http://hissa.nist.gov/publications/sp223/

- Wäyrynen, J., Bodén, M., and G. Boström. "Security engineering and eXtreme Programming: an impossible marriage?". In Zannier, C., Erdogmus, H., and L. Lindstrom, editors. *Extreme Programming and Agile Methods – XP/Agile Universe 2004.* Berlin, Germany: Springer-Verlag, 2004. pages 117-128. Accessed 11 September 2008 at: http://is.dsv.su.se/PubsFilesFolder/612.pdf

- Webb, Warren. "Hack this: secure embedded systems". *EDN,* 22 July 2004. Accessed 9 April 2008 at: http://www.edn.com/index.asp?layout=article&articleid=CA434871

- Wilander, John and Pia Fåk. "Pattern Matching Security Properties of Code using Dependence Graphs". *Proceedings of the First International Workshop on Code Based Software Security Assessments,* Pittsburgh, Pennsylvania, 7 November 2005, pages 5-8. Accessed 3 January 2008 at: http://www.ida.liu.se/~johwi/research_publications/paper_cobassa2005_wilander_fak.pdf

- Williams, L., R.R. Kessler, W. Cunningham, and E. Jeffries. "Strengthening the Case for Pair-Programming". *IEEE Software.* Volume 17 Issue 4, July/August 2000, pages 19-25. Accessed 11 September 2008 at: http://www.cs.utah.edu/~lwilliam/Papers/ieeeSoftware.PDF *– and –* http://rockfish-cs.cs.unc.edu/COMP290-agile/ieeeSoftware.pdf

- Wilson, David L. *Risk Perception and Trusted Computer Systems: Is Open Source Software Really More Secure than Proprietary Software?* Purdue University master's thesis, CERIAS TR 2004-07, 2004. Accessed 19 December 2007 at: https://www.cerias.purdue.edu/tools_and_resources/bibtex_archive/view_entry.php?bibtex_id=2686

- Wing, Jeannette M. *A Symbiotic Relationship between Formal Methods and Security.* Technical Report CMU-CS-98-188, December 1998. Accessed 4 April 2007 at: http://reports-archive.adm.cs.cmu.edu/anon/1998/abstracts/98-188.html

- Wu, Dan. "Security Functional Requirements Analysis for Developing Secure Software". Qualification Exam Report for University of Southern California, December 2006. Accessed 17 December 2007 at: http://sunset.usc.edu/csse/TECHRPTS/2006/usccse2006-622/usccse2006-622.pdf

- Wysopal, Chris, *et al.* "Testing Fault Injection in Local Applications". SecurityFocus, 23 January 2007 (excerpt from *The Art of Software Security Testing).* Accessed 14 December 2007 at: http://www.securityfocus.com/infocus/1886

- Wysopal, Chris, Lucas Nelson, Dino Dai Zovi, and Elfriede Dustin. The Art of Software Security Testing: Identifying Software Security Flaws. Cupertino, California: Symantec Press, 2006 *—and—* Wysopal, Chris, *et al.* "Finding software security flaws" (excerpt from *The Art of Software Security Testing). ComputerWorld,* 28 December 2006. Accessed 11 December 2007 at: http://www.computerworld.com/action/article.do?command=viewArticleBasic&articleId=9006870

- Wysopal, Chris. "Putting trust in software code". *USENIX ;login:,* Volume 29, Number 6, December 2004. Accessed 26 December 2007 at: http://www.usenix.org/publications/login/2004-12/pdfs/code.pdf

- Xu, Dianxiang and Joshua J. Pauli. "Threat-Driven Design and Analysis of Secure Software Architectures". *Journal of Information Assurance,* Volume 1 Issue 3, 2006. Accessed 17 December 2007 at: http://www.homepages.dsu.edu/paulij/pubs/xu-pauli-JIAS.pdf

- Yin, Jian, Chunqiang Tang, Xiaolan Zhang, and Michael McIntosh. "On Estimating the Security Risks of Composite Software Services". *Proceedings of the Program Analysis for Security and Safety Workshop Discussion,* Nantes, France, 4 July 2006. Accessed 23 January 2008 at: http://research.ihost.com/password/papers/Yin.pdf

- Yumerefendi, Aydan R. and Jeffrey S. Chase. "Trust but Verify: Accountability for Network Services". *Proceedings of the Eleventh ACM SIGOPS European Workshop,* Leuven, Belgium, 19-22 September 2004. Accessed 7 July 2008 at: http://issg.cs.duke.edu/publications/trust-ew04.pdf

B.2 RESTRICTED-ACCESS ONLINE RESOURCES

The following resources are available online only to paid subscribers, conference attendees, *etc.*, or to those who pay a fee-for-use. In a few cases, which are noted, the documents must be requested from their custodial organizations or authors.

- de Vries, Stephen. "Software Testing for Security". *Network Security,* Volume 2007 Issue 3, March 2007, pages 11-15. Purchase online at: http://dx.doi.org/10.1016/S1353-4858(07)70027-2

- DISA Application Security Project. *Application Security Developer's Guides,* Version 3.1 DRAFT, October 2004. For information, contact: goertzel_karen@bah.com

- Fernandez, Eduoardo B., Preethi Cholmondeley, and Olaf Zimmermann. "Extending a secure system development methodology to SOA". *Proceedings of the First International Workshop on Secure Systems Methodologies Using Patterns,* Regensburg, Germany, 6 September 2007, pages 749-754. Purchase online at: http://ieeexplore.ieee.org/xpls/abs_all.jsp?tp=&arnumber=4312994&isnumber=4312839

- Ge, X., R.F. Paige, F.A.C. Polack, H. Chivers, and P.J. Brooke. "Agile Development of Secure Web Applications". *Proceedings of the ACM International Conference on Web Engineering,* Palo Alto, California, 11-14 July 2006. Purchase online at: http://doi.acm.org/10.1145/1145581.1145641

- Gegick, Michael and Laurie Williams. "Matching attack patterns to security vulnerabilities in software-intensive system designs". *Proceedings of the Workshop on Software Engineering for Secure Systems,* St. Louis, Missouri, 15-16 May 2005. Purchase online at: http://doi.acm.org/10.1145/1083200.1083211

- Gegick, Michael and Laurie Williams. "On the design of more secure software-intensive systems by use of attack patterns". *Information and Software Technology,* Volume 49 Issue 4, April 2007, pages 381-397. Purchase online at: http://dx.doi.org/10.1016/j.infsof.2006.06.002

- Gilliam, David P. "Security risks: management and mitigation in the software life cycle". *Proceedings of the 13th International Workshop on Enabling Technologies,* Modena, Italy, 14-16 June 2004, pages 211-216. Purchase online at: http://doi.ieeecomputersociety.org/10.1109/ENABL.2004.55

- Gupta, Suvajit and Joel Winstead. "Using Attack Graphs to Design Systems". *IEEE Security and Privacy,* July/August 2007, pages 80-83. Purchase online at: http://ieeexplore.ieee.org/xpls/abs_all.jsp?tp=&arnumber=4288052&isnumber=4288029

- Hanebutte, Nadine 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. Purchase online at: http://www3.interscience.wiley.com/journal/112136396/abstract

- Jaferian, Pooya, Golnaz Elahi, Mohammad Reza Ayatollahzadeh Shirazi, and Babak Sadeghian, "RUPSec: Extending Business Modeling and Requirements Disciplines of RUP for Developing Secure Systems". *Proceeding of the 31th IEEE Conference of EuroMicro,* Porto, Portugal, 31 August-2 September 2005. Purchase online at: http://doi.ieeecomputersociety.org/10.1109/EURMIC.2005.52

- Jarzombek, Joe and Karen Mercedes Goertzel. "Security Considerations in the Use of Open Source Software". Presented at the Technology Training Corporation Military Open Source Software Conference, Washington, D.C., 21 April 2008. For information, contact: goertzel_karen@bah.com

- Jeong, Gu-Beom and Guk-Boh Kim. "A Framework for Security Assurance in Component Based Development". *Proceedings of Workshop on Approaches or Methods of Security Engineering,* Singapore, China, 9-12 May 2005, pages 42-51. Purchase online at: http://www.springerlink.com/content/wtktbh69y37xayy4/

- Johansson, Jesper M. and E. Eugene Schultz. "Dealing with contextual vulnerabilities in code: distinguishing between solutions and pseudosolutions". *Computers and Security,* Volume 22 Number 2, 2003, pages 152-159. Purchase online at: http://dx.doi.org/10.1016/S0167-4048(03)00213-X

- Kellerman, Thomas. "Technology Risk Checklist, Version 13.0". Washington, DC: The World Bank, 2008. For information, contact: goertzel_karen@bah.com

- Kolb, Ronny, Dirk Muthig, Thomas Patzke, and Kazuyuki Yamauchi. "Refactoring a legacy component for reuse in a software product line: a case study: Practice Articles". Presented at the 2005 IEEE International Conference on Software Maintenance. Published in *Journal of Software Maintenance and Evolution: Research and Practice,* Volume 18 Issue 2, March 2006, pages 109-132. Purchase online at: http://www3.interscience.wiley.com/journal/112561657/abstract

- Kongsli, Vidar. "Towards agile security in Web applications". *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications,* Portland, Oregon, 22-26 October 2006. Purchase online at: http://doi.acm.org/10.1145/1176617.1176727

- Le Traon, Yves, Benoit Baudry, and Jean-Marc Jezequel. "Design by Contract to improve software vigilance". *IEEE Transactions on Software Engineering,* Volume 32 Issue 8 August 2006, pages 571-586. Purchase online at: http://doi.ieeecomputersociety.org/10.1109/TSE.2006.79

- Paes, Carlos Eduardo de Barros and Celso Massaki Hirata. "RUP Extension for the Development of Secure Systems". *International Journal of Web Information Systems.* Volume 3 Issue 4, 2007, pages 293-314. Purchase online at: http://www.emeraldinsight.com/10.1108/17440080710848099

- Rothbart, K., U. Neffe, C. Steger, R. Weiss, E. Rieger, and A. Muehlberger. "High level fault injection for attack simulation in smart cards". In *Proceedings of the 13th Asian Test Symposium*, Kenting, Taiwan, 15-17 November 2004, pages 118-121. Purchase online at: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?tp=&arnumber=1376546&isnumber=30045

- Shina, Michael E. and Hassan Gomaab. "Software requirements and architecture modeling for evolving non-secure applications into secure applications". *Science of Computer Programming,* Volume 66 Issue 1, 15 April 2007, pages 60-70. Purchase online at: http://dx.doi.org/10.1016/j.scico.2006.10.009

- Sindre, Guttorm and Opdahl, Andreas L. "Eliciting security requirements by misuse cases". *Proceedings of the 37th International Conference on Technology of Object-Oriented Languages and Systems,* Sydney, Australia, 20-23 November 2000, pages 120-131. Purchase online at: http://doi.ieeecomputersociety.org/10.1109/TOOLS.2000.891363

- Stytz, Martin R. and Sheila B. Banks. "Dynamic Software Security Testing". *IEEE Security and Privacy,* Volume 4 Issue 3, May 2006, pages 77-79. Purchase online at: http://doi.ieeecomputersociety.org/10.1109/MSP.2006.64

- Tappenden, A., P. Beatty, J. Miller, A. Geras, and M. Smith. "Agile security testing of Web-based systems *via* HTTPUnit". *Proceedings of the AGILE 2005 Conference,* Denver, Colorado, 24-29 July 2005. Purchase online at: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?tp=&arnumber=1609802&isnumber=33795

- Thompson, Herbert H. "Application Penetration Testing". *IEEE Security and Privacy,* Volume 3 Number 1, January/February 2005, pages 66-69. Purchase online at: http://doi.ieeecomputersociety.org/10.1109/MSP.2005.3

- Whittaker, James A. and Herbert H. Thompson. "Black Box Debugging". *Queue,* Volume 1 Number 9, December/January 2003-2004. Purchase online at: http://doi.acm.org/10.1145/966789.966807

- Xu, Dianxiang, Vivek Goel and K. Nygard. "An Aspect-Oriented Approach to Security Requirements Analysis". In *Proceedings of the 30th Annual International Computer Software and Applications Conference,* Chicago, Illinois, 17-21 September 2006, Volume 2, pages 79-82. Purchase online at: http://doi.ieeecomputersociety.org/10.1109/COMPSAC.2006.109

B.3 BIBLIOGRAPHY

The following is a list of books that may be of interest to readers of this document.

- Allen, Julia H., Sean Barnum, Robert J. Ellison, Gary McGraw, and Nancy R. Mead. *Software Security Engineering: A Guide for Project Managers.* Indianapolis, Indiana: Addison-Wesley Professional, 2008. Information available at: http://www.sei.cmu.edu/publications/books/cert/software-security-engineering.html

- Anderson, Ross. J. *Security Engineering: A Guide to Building Dependable Distributed Systems*, Second Edition. New York, New York: John Wiley & Sons Inc., 2008. Information available at: http://www.cl.cam.ac.uk/~rja14/book.html

- Andreu, Andres. *Professional Pen Testing for Web Applications.* Indianapolis, Indiana: Wiley Publishing, 2006.

- Andrews, Mike and James A. Whittaker. *How to Break Web Software: Functional and Security Testing of Web Applications and Web Services.* Boston, Massachusetts: Addison-Wesley Professional, 2006.

- Anley, Chris, John Heasman, Felix Linder, and Gerardo Richarte. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes.* Indianapolis, Indiana: Wiley Publishing, 2007.

- Ashbaugh, Douglas A. *Security Software Development: Assessing and Managing Security Risks.* Boca Raton, Florida: Auerbach Publications, 2008.

- Bidgoli, Hossein. *Handbook of Information Security.* New York, NY: Wiley, 2005.

- Blakley, Bob and Craig Heath, Craig. *Technical Guide to Security Design Patterns.* San Francisco, California: The Open Group, 2004.

- Chess, Brian and Jacob West. *Secure Programming with Static Analysis.* Indianapolis, Indiana: Addison-Wesley, 2007.

- Daswani, Neil, Christoph Kern, and Anita Kesavan. *Foundations of Security: What Every Programmer Needs to Know.* Berkeley, California: Apress, 2007. Slides to accompany the book available at: http://code.google.com/edu/submissions/daswani/index.html

- Debbabi, M., M. Saleh, C. Talhi, and S. Zhioua. *Embedded Java Security: Security for Mobile Devices.* Berlin, Germany: Springer-Verlag, 2007.

- Dowd, Mark, John McDonald, and Justin Schuh. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities.* Indianapolis, Indiana: Addison-Wesley Professional, 2006.

- Eliam, Eldad. *Reversing: Secrets of Reverse Engineering.* Indianapolis, Indiana: Wiley Publishing, 2005.

- Firesmith, Donald G. Security and Safety *Requirements for Software-Intensive Systems.* Boca Raton, Florida: Auerbach Publications, 2008.

- Fisher, Matt. *Developer's Guide to Web Application Security.* Rockland, Massachusetts: Syngress Publishing, 2007.

- Gallagher, Tom, Lawrence Landauer, and Bryan Jeffries. *Hunting Security Bugs.* Redmond, Washington: Microsoft Press, 2006.

- Gasser, Morrie. *Building a Secure Computer System.* New York, New York: Van Nostrand Reinhold; 1988. A digital copy of this book can be downloaded from: http://cs.unomaha.edu/~stanw/gasserbook.pdf

- Graff, Mark G. and Kenneth R. Van Wyk. *Secure Coding, Principles and Practices.* Sebastopol, California: O'Reilly and Associates, 2003. Information available at: http://www.securecoding.org/

- Haralambos, Mouratidis and Paolo Giorgini, editors. *Integrating Security and Software Engineering.* Hershey, Pennsylvania: Idea Group Publishing; 2007.

- Hoffmann, Billy and Bryan Sullivan. *Ajax Security.* Boston, Massachusetts: Pearson Education, 2008.

- Howard, Michael and David LeBlanc. *Designing Secure Software.* San Francisco, California: McGraw-Hill Osborne Media, 2007.

- Howard, Michael and David LeBlanc. *Writing Secure Code, Second Edition*. Redmond, Washington: Microsoft Press, 2003. Information available at: http://www.microsoft.com/mspress/books/5957.aspx

- Howard, Michael and Steve Lipner. *The Security Development Lifecycle.* Redmond, Washington: Microsoft Press, 2006.

- Howard, Michael, John Viega, and David LeBlanc. *19 Deadly Sins of Software Security.* San Francisco, California: McGraw-Hill Osborne Media, 2005.

- Kanneganti, Ramarao and Prasad Chodavarapu. *SOA Security.* Greenwich, Connecticut: Manning Publications, 2007.

- Kriha, Walter and Roland Schmitz. *Internet-Security aus Software-Sicht: Grundlagen der Software-Erstellung für sicherheitskritische Bereiche* [in German]. Heidelberg/Berlin, Germany: Springer, 2008.

- Lee, Edward A. "Embedded Software" in M. Zelkowitz, editor. *Advances in Computers*. Volume 56. London, United Kingdom: Academic Press, 2002.

- McGraw, Gary and Greg Hoglund. *Exploiting Software: How to Break Code.* Indianapolis, Indiana: Addison-Wesley Professional, 2004.

- McGraw, Gary. *Software Security: Building Security In.* Boston, Massachusetts: Addison-Wesley Professional, 2006.

- McGraw, Gary. *Software Security: Building Security In*. Indianapolis, Indiana: Addison-Wesley Professional, 2006.

- Mouratidis, Haralambos and Paolo Giorgini, editors. *Integrating Security and Software Engineering: Advances and Future Vision.* Hershey, Pennsylvania: Idea Group Publishing, 2007.

- Schumacher, Markus, Eduardo Fernandez-Buglioni, Duane Hybertson, Frank Buschmann, and Peter Sommerlad. *Security Patterns: Integrating Security and Systems Engineering.* New York, New York: John Wiley & Sons; 2005.

- Seacord, Robert C. *Secure Coding in C and C++.* Indianapolis, Indiana: Addison-Wesley Professional, 2005.

- Sekar, R. and Ravi Sandhu. *Program Transformation Techniques for Enhancing Software Security.* San Rafael, California: Morgan & Claypool Publishers, 2008.

- Seo, Seong Chae, Jin Ho You, Young Dae Kim, Jun Yong Choi, Sang Jun Lee, and Byung Ki Kim. "Building Security Requirements Using State Transition Diagram at Security Threat Location". In *Lecture Notes in Computer Science: Computational Intelligence and Security*, Volume 3802/2005, pages 451-456. Heidelberg, Germany: Springer-Verlag, 2005.

- Sinn, Richard. *Software Security Technologies.* Boston, Massachusetts: Course Technology, 2007.

- Siponen, M., R. Baskerville, and T. Kuivalainen. "Extending Security in Agile Software Development Methods". In Mouratidis, Haralambos and Paolo Giorgini, editors. *Integrating Security and Software Engineering: Advances and Future Visions.* Hershey, Pennsylvania: IGI Global Publishing, 2007.

- Steel, Chris, Ramesh Nagappan, and Ray Lai. *Core Security Patterns.* Indianapolis, Indiana: Prentice-Hall Professional, 2005.

- Stuttard, Dafydd and Marcus Pinto. *The Web Application Hacker's Handbook: Discovering and Exploiting Security Flaws.* Indianapolis, Indiana: Wiley Publishing, 2008.

- Sutton, Michael, Adam Greene, and Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery.* Indianapolis, Indiana: Addison-Wesley Professional, 2007.

- Thompson, Herbert H. and Scott G. Chase. *The Software Vulnerability Guide.* Boston, Massachusetts: Charles River Media, 2005.

- van der Linden, Maura A. *Testing Code Security.* Boca Raton, Florida: Auerbach Publications, 2007.

- Viega, John and Matt Messier. *Secure Programming Cookbook for C and C++.* Sebastopol, California: O'Reilly, 2003.

- Zulkernine, Mohammad and Sheikh Iqbal Ahamed. "Software security engineering: toward unifying software engineering and security engineering". Chapter 14 of *Enterprise Information Systems Assurance and System Security: Managerial and Technical Issues* (Warkentin, Merrill and Rayford B. Vaughn, editors). Hershey, Pennsylvania: Idea Group Publishing, 2006.

APPENDIX C: SECURITY CONCERNS ASSOCIATED WITH SPECIFIC SOFTWARE TECHNOLOGIES, METHODOLOGIES, AND PROGRAMMING LANGUAGES

C.1 SECURITY CONCERNS ASSOCIATED WITH WEB SERVICE SOFTWARE

The security concerns associated with Web services and service oriented architectures (SOA), as well as the standard and vendor-specific mechanisms being defined and implemented to address those concerns, are described in the Suggested Resources at the end of this section. The security issues of Web services as software are discussed in: Goertzel, Karen. "The Security of Web Services as Software". *CrossTalk: The Journal of Defense Software Engineering,* September 2007. Accessed 9 April 2008 at: http://www.stsc.hill.af.mil/CrossTalk/2007/09/0709Goertzel.html

SUGGESTED RESOURCES

- Singhal, Anoop, Theodore Winograd, and Karen Scarfone. *Guide to Secure Web Services.* NIST Special Publication 800-95, August 2007. Accessed 26 January 2008 at: http://www.csrc.nist.gov/publications/nistpubs/800-95/SP800-95.pdf
- Kanneganti, Ramarao and Prasad Chodavarapu. *SOA Security.* Greenwich, Connecticut: Manning Publications, 2007.
- Epstein, Jeremy. "SOA security: the only thing we have to fear is fear itself". *SOA Web Services Journal,* 3 December 2005. Accessed 21 January 2008 at: http://Webservices.sys-con.com/read/155630.htm
- Luu, Tieu. "Separation of Concerns in Web Service Implementations". *OnJava*, 6 September 2006. Accessed 11 December 2007 at: http://www.onjava.com/pub/a/onjava/2006/09/06/separation-of-concerns-in-Web-services.html
- Fernandez, Eduoardo B., Preethi Cholmondeley, and Olaf Zimmermann. "Extending a secure system development methodology to SOA". *Proceedings of the First International Workshop on Secure Systems Methodologies Using Patterns,* Regensburg, Germany, 6 September 2007, pages 749-754.
- Aime, Marco. "Modelling services for trust and security assurance". Presented at Workshop on Software and Service Development, Security and Dependability, Maribor, Slovenia, 10-11 July 2007. Accessed 19 January 2008 at: http://www.esfors.org/esfors_ws2/D1_S1_Aime-ModellingServicesTrustAndSecurityAssurance.pdf

C.2 SECURITY CONCERNS ASSOCIATED WITH EMBEDDED SYSTEM SOFTWARE[55]

By virtue of their usage profile, embedded devices have a much higher reliability expectation than most of the other software systems. This means that embedded software must continue to operate in spite of security threats. It also renders the common software security policy of patching after a failure irrelevant.

Embedded devices, especially portable devices, face many more security threats than a non-embedded system, *e.g.,* an information system. Hackers may use sensitive test equipment to steal, disassemble, and probe small devices to remove memory elements from the system to extract their contents, use debugging ports and software to read sensitive data or force unintended operation, measure electromagnetic radiation or power consumption to gain clues about hidden functions and concealed information. By forcing the system to operate outside its design parameters through introduction of extreme temperatures, voltage excursions, and clock variations the attacker can expose anomalous and vulnerable behaviors.

Memory devices are a favorite target of attack because they store both the system's firmware and software and its sensitive data. Many devices can be read while in circuit and may provide temporary plain text data during operation. If the device includes a tamper sensor, the designer can incorporate hardware or software resources that will rapidly erase sensitive data before it can be extracted.

A threat unique to software within embedded systems is the fact that the physical system may fall into an adversary's hands, allowing the scrutiny and reverse engineering of the system, including its software component. Once the system is well-understood, the adversary will be able to devise countermeasures to the system's security protections or to the system itself (*e.g.*, in the case of a SCADA or weapons system). There is also a possibility that the adversary will use information gained through the reverse engineering to design a new version of the system which can be employed against the copied system or its originator.

Embedded software may be designed according to any of several dozen software architectures and hosted on any of numerous operating systems. These architectures and operating systems may provide good security protection or none at all. This abundance of existing and emerging embedded-system architectures is increasing the number of available attack paths, and inhibiting the development of an industry-wide security protection scheme for embedded software. These attack paths are targeted by an increasing number and variety of security threats intended to cause embedded system disruptions, force unplanned operations, or extract sensitive data.

---

55 The source for most of this discussion is Webb, Warren. "Hack this: secure embedded systems". *EDN,* 22 July 2004. Accessed 9 April 2008 at: http://www.edn.com/index.asp?layout=article&articleid=CA434871

Some key secure software design principles are particularly relevant to embedded software design:

- Differentiation between low- and high-consequence functions and public and private data. By denying user access to high-consequence functions and private data, the designer can reduce the risks to these elements of the embedded system;

- The design must provide protection during the embedded system's normal operation, during attack through a network connection, and during electronic probing (*e.g.,* in an attacker's laboratory).

Most embedded operating system security specifications focus on implementing security protections defined in the Common Criteria; one of these, from LynuxWorks, is intended to achieve certified assurance at Evaluation Assurance Level 7. The MILS system standard requires a partitioned real-time operating system that can be certified as secure through a set of rigorous tests. MILS systems provide memory protection and guaranteed resource availability, enabling the user to securely host both trusted and untrusted data on the same processor. Developers can create formally-verified, always-invoked, tamperproof application code with non-bypassable security features for MILS platforms. Green Hills Software, LynuxWorks, and Wind River Software are among vendors working on MILS-compliant RTOSs.

At a loss for how to improve the security of their software, embedded systems designers are relying increasingly to strengthened physical packaging to protect their embedded software.

TPMs, such as those conforming to the open industry standards defined by the Trusted Computing Platform Alliance can be used as secure platforms for embedded programs. An embedded TPM monitors the boot process to create hash values or checksums for important elements, such as BIOS, device drivers, and operating-system loaders. The TPM stores these values and compares them with the reference values that define the trustworthy status of the platform. The TPM also provides public/private-key Rivest-Shamir-Adleman (RSA)-based encryption and decryption along with a tamperproof on-chip memory for storing keys and passwords.

Security concerns and precautions have changed the basic design guidelines for embedded products. Traditional criteria for evaluating embedded systems designs—smallness of circuitry, efficiency of code, and long mean times between failures—are no longer adequate; these criteria must be augmented by the requirement for dependability, trustworthiness, and resilience during not only normal operation, but also under attack.

In the UK, Praxis High Integrity Systems[56] provides development methodologies, tools, services, and resources expressly intended to support the development of secure embedded software.

Researchers at Virginia Tech are investigating the use of hardware-software co-design techniques using a design environment with specific support for secure embedded system design. According to the researchers, the objective of secure embedded system design is to protect the root-of-trust from being compromised by obtaining a systematic deployment of countermeasures that will protect the root-of-trust at different levels of abstraction. Specific techniques the researchers are applying include:

1. Partitioning of systems into secure and non-secure parts;

2. Development of secure interfaces to integrate those partitions back into a single system.

Another major challenge in secure embedded system design being addressed in the Virginia Tech research is the implementation of end-point-security, i.e. the creation of trusted channels that reach out into the peripherals and off-chip, off-board interface. Hardware-software co-design techniques are being used to build trusted channels to securely cross the hardware-software interface.

SUGGESTED RESOURCES

- Lee, Edward A. "Embedded Software" in M. Zelkowitz, editor. *Advances in Computers*. Volume 56. London, United Kingdom: Academic Press, 2002.

- Kocher, Paul, Ruby B. Lee, Gary McGraw, Anand Raghunathan, and Srivaths Ravi. "Security as a New Dimension in Embedded System Design". *Proceedings of the 41st Design Automation Conference*, San Diego, California, 7-11 June 2004. Accessed 17 January 2008 at: http://palms.ee.princeton.edu/PALMSopen/Lee-41stDAC_46_1.pdf

- Seacord, Robert, Noopur Davis, Chad Dougherty, Nancy Mead, and Robert Mead. "How to write secure C/C++ application code for your embedded design: Part 1—Some secure software design principles". *Embedded.com*, 10 September 2007. Accessed 13 December 2007 at: http://www.embedded.com/design/202300629 —*and*— "Part 2—Systems Quality Requirements Engineering". *Programmable Logic Design Line*, 15 October 2007. Accessed 13 December 2007 at: http://www.pldesignline.com/202402853

- Debbabi, M., M. Saleh, C. Talhi, and S. Zhioua. *Embedded Java Security: Security for Mobile Devices.* Berlin, Germany: Springer-Verlag, 2007.

- Ricci, Lawrence and Larry McGinnes. "Embedded System Security: Designing Secure Systems with Windows CE". AppliedData.net white paper, not dated. Accessed 9 April 2008 at: http://www.applieddata.net/WP/Whitepaper_Secure_Windows_CE.pdf

56 Website accessed 26 August 2008 at: http://www.praxis-his.com/

- Rothbart, K., U. Neffe, C. Steger, R. Weiss, E. Rieger, and A. Muehlberger. "High level fault injection for attack simulation in smart cards". In *Proceedings of the 13th Asian Test Symposium*, Kenting, Taiwan, 15-17 November 2004, pages 118-121.
- Davidson, Jack W. and Jason D. Hiser. "Securing Embedded Software Using Software Dynamic Translation". Position Paper for the Army Research Organization Planning Workshop on Embedded Systems and Network Security, 22-23 February 2007. Accessed 31 December 2007 at: http://moss.csc.ncsu.edu/~mueller/esns07/davidson.pdf
- Verbauwhede, I. and Patrick Schaumont. "Design Methods for Security and Trust". *Proceedings of the Design Automation and Test Conference in Europe*, Nice, France, April 2007. Accessed 11 September 2008 at: http://www.cosic.esat.kuleuven.be/publications/article-875.pdf

C.3 FORMAL METHODS AND SECURE SOFTWARE

The main purposes to which formal methods are put in the SDLC include:

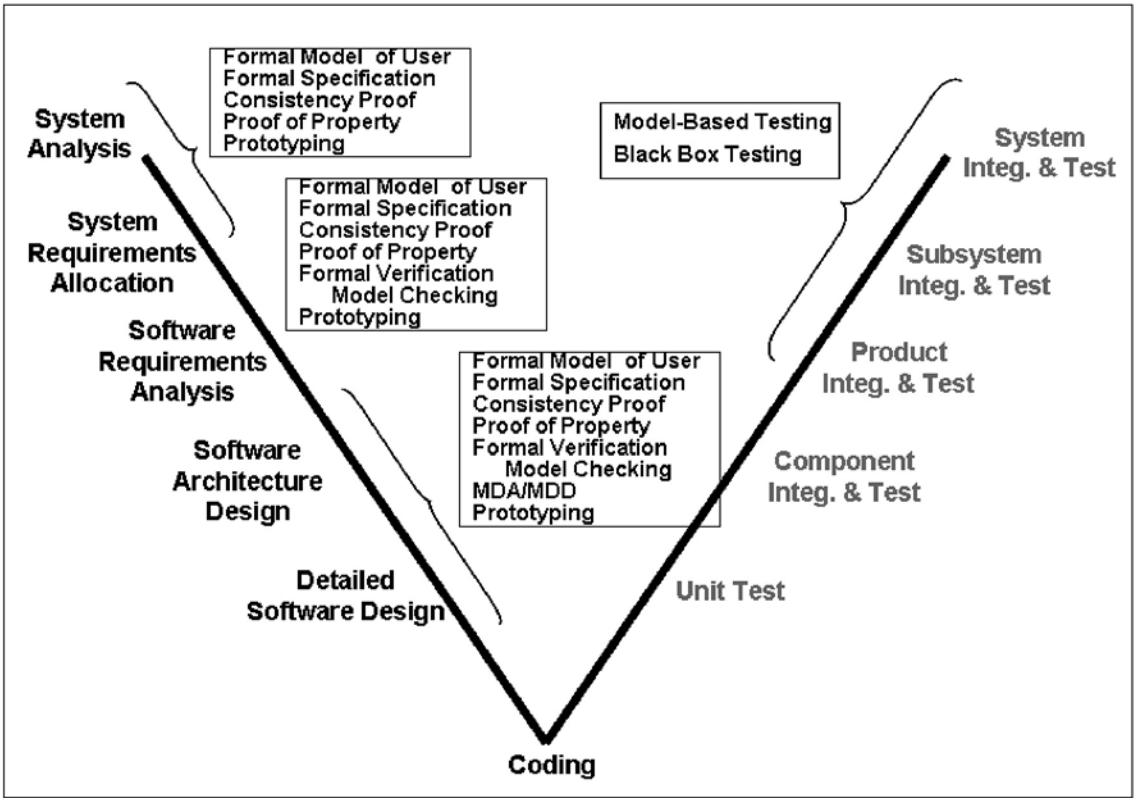
1. Writing the software's formal specification;
2. Proving properties about the software's formal specification;
3. Constructing the software program through mathematical manipulation of its formal specification;
4. Using mathematical arguments and proofs to verify the properties of a program.

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

The essence of a formal method is its ability to provide one or more of the following tools:

- A mathematically-based specification language in which the developer can specify, with mathematical precision, the required properties (*e.g.,* correctness, safety, security) of the software he/she will build and the mathematical proofs whereby the software's exhibition of those properties can be verified;
- The reasoning logic needed to model the software that needs to have its properties formally verified.
- The tools to perform the proof-based formal verification of the formally-specified, modeled software artifacts.

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



SOURCE: Software Security Assurance: A State-of-the-Art Report

Figure C-1. Formal methods in the SDLC

Table C-1 describes each formal method artifact referenced in Figure C-1. Table C-2 maps formal method artifacts to SDLC activities.

Table C-1. Formal methods artifacts defined

Formal Method Artifact	Description
Formal models of user behavior	Often describe sequences in which users invoke the functionality of a system. For example, decision tables, finite state machines, Markov models, or Petri nets can characterize user actions.
Formal specifications	Rigorously describe the functionality of a system or system component. Languages used, such as in the Vienna Development Method and Z, often involve extensions of a mixture of predicate logic and set theory.
Consistency proofs	Examine the components of a system in a formal specification developed at a single level of abstraction. They are useful at every phase in which a formal model is developed.

Formal Method Artifact	Description
Proofs of properties	Prove that some proposition over states or combination of states is always maintained true in a system. For example, a formal method for safety might include the proof that some state never arises in a system.
Formal verification: The development of proofs that a formal specification at a lower level of abstraction fulfills the requirements of a formal specification of a system at a higher level of abstraction. Formal verification can be used to check software requirements against the system requirements allocated to software, architecture design against software requirements, the detailed design against the architecture design, and code against the detailed design.	• Software requirements • Software detailed design • Coding
Model checking results	A practical technique for automated formal verification. Model checking tools use symbolic expressions in propositional logic to explore a large state space. Model checking can be performed as part of formal verification is used, with the result of the model checking analyzed to determine whether the model is accurate and trustworthy enough to form the basis for positive assurance (note that the determination of assurance is not intrinsic to model checking).
Formally-generated prototypes	While prototyping is not a formal method *per se*, some formal tools can generate prototypes, especially when formal operational semantics are used.
Model driven architecture (MDA)-generated architectural design	The automatic generation of an architecture from a Unified Modeling Language (UML) specification of a system.
Model driven development (MDD)-generated implementation	Supports the construction of a system or system component by transforming a formal or semi-formal model into an implementation.

Formal Method Artifact	Description
Formally-generated black box test cases	Test cases based on specifications of the system or component being tested, rather than on pre-existing knowledge of the internal implementation of the system or component. In so far as the original specifications are formal, formal techniques can be used to generate the associated black box test cases.
Model-based test cases	Efficient test cases automatically generated from models of requirements and functionality, based on the formal model of user behavior developed during the requirements phase.

Table C-2. Formal methods artifacts mapped to SDLC activities

SDLC Activity	Formal Method Artifacts
System analysis	• Formal models of user behavior • Formal specifications • Consistency proofs • Proofs of properties • Formal tool-generated prototypes
System requirements allocation	• Formal models of user behavior • Formal specifications • Consistency proofs • Proofs of properties • Formal tool-generated prototypes
Software requirements specification	• Formal models of user behavior • Formal specifications • Consistency proofs • Proofs of properties • Formal verifications • Model checking results • Formally-generated prototypes
Test case generation (especially black box tests, integration tests)	• Formal (checked) models of user behavior • Formally-generated black box test cases • Model-based test cases
Software architectural design	• Formal specifications • Consistency proofs • Proofs of properties • Model checking results • Formal tool-generated prototypes • MDA-generated architectural design

SDLC Activity	Formal Method Artifacts
Software detailed design	• Consistency proofs • Proofs of properties • Formal verifications • Model checking results • Formal tool-generated prototypes • Model-driven development (MDD)-generated implementation
Coding	• Consistency proofs • Proofs of properties • Formal verifications • Model checks • MDD-generated implementation

The effectiveness of formal methods is determined in large part by the expertise and skill of the developer who uses them. It is quite possible for the reasoning behind formal proofs to be erroneous, and thus for those proofs to produce spurious results. Moreover, current formal methods verify properties in *models* of software, not implemented software itself. For this reason even software specified, modeled, and verified by formal methods will require adequate security testing, with some test cases based on formal models of user behavior and generated by formal tools.

Barry W. Boehm of University of Southern California has also observed that formal methods can be useful for *verification* (demonstrating that each step in the development process has satisfied the requirements imposed by previous steps) of a system as it is developed, but they are not useful for *validation* of an implemented system (demonstrating that the system actually satisfies its objectives, including its security objectives).

Moreover, while a formal verification can prove that an abstract description (*i.e.,* model) of a software implementation satisfies its formal specification, or that some formal property is satisfied in the implementation, formal verification cannot prove that the formal specification itself has captured the users' intuitive understanding of their requirements for the specified system. Nor can formal methods prove that the implementation of the formally-specified system will run correctly under all conditions including the unexpected conditions associated with novel attacks.

Because software security requirements are often expressed in negative terms, as constraints defining what the software must not do, such requirements can be particularly difficult to specify formally, and it may not be possible to mathematically prove that such requirements have been correctly satisfied in implemented software. The same is true of safety properties, which tend to be stated in terms of "Nothing bad must happen as a result of *x*."

As with other methods, such as aspect-oriented software development (AOSD) and secure UML variants (*e.g.,* UMLsec), that are promoted for engineering "secure software", when case studies and papers describe real world usage of those methods, it becomes clear that what some of these methods were actually used for was to improve the correctness of IT security functions implemented by software, *e.g.,* role-based access control, encryption. In some of the documented projects, little if any consideration was given to whether the methods also improved the trustworthiness and survivability of the software or the system as a whole.

While the reader of such studies and papers should not be deceived into believing that the experiences of their authors have much to teach about the effectiveness of formal methods in the production of secure software *vs.* rather functionally-correct security software, the same reader should also be aware that in other projects,[57] more emphasis was placed on the security of the software architecture on the whole, and not just the correctness of its security mechanisms.

C.4 SECURITY CONCERNS AND BENEFITS ASSOCIATED WITH SPECIFIC PROGRAMMING LANGUAGES

This section focuses on features of individual programming languages that affect the likelihood that the programmer will be able to write secure code in that language. The section also discusses the security benefits and concerns associated with popular programming languages, as well as secure language variants, and tools that can be used to enable the secure use of otherwise non-secure languages.

C.4.1 C and C++

C is one of the most widely used programming languages in the World. C is one of the primary languages used in the Windows, Macintosh Operating System Version 10 (Mac OS X), Berkeley Software Distribution, Linux, and Unix kernels. Introduced in the early 1970s, C was developed to require a straightforward compiler, low-level memory access, and language constructs that easily map to assembly-code constructs. Through these design goals, developers could create efficient and portable applications within the restrictive processing environments at the time. As such, C and C++ (the latter was originally an enhancement to C but is now considered a separate programming language) have a large installed-base of libraries and developers. Nevertheless, many of the design choices made in the 1970s to meet its original design goals (*e.g.,* failing to perform run-time checks on arrays) have led many to claim that C and C++ are insecure programming languages. Due to its status as one of the

57 An example of a project in which the end-goal was defining a software architecture that would result in a more trustworthy, resilient software system is described in Jürjens, Jan, Joerg Schreck, and Peter Bartmann. "Model-based Security Analysis for Mobile Communications". *Proceedings of the 30th IEEE International Conference on Software Engineering,* Leipzig, Germany, 10-18 May 2008. Accessed 25 August 2008 at: http://mcs.open.ac.uk/jj2924/publications/papers/icsetele08-jurjens.pdf

most popular programming languages in use, there are a wealth of resources on secure programming in C and C++.

In 2007, the SysAdmin, Audit, Networking and Security (SANS) Institute introduced the Global Information Assurance Certification Global Secure Software Programmer (GSSP) C certification. This certification aims to recognize C developers who can mitigate C-specific security concerns. The GSSP C certification's essential skills include eight *tasks*, which address specific aspects of C development:

- **Environment and input:** Covers the skills and mechanisms necessary to securely interface with the environment and external input to the application;
- **Dynamic allocated resources:** Covers the skills and mechanisms necessary to securely use dynamic resources in the C stack and heap memory spaces;
- **Input, output, and files:** Covers the skills and mechanisms necessary to securely interact with output functions such as *printf* and reading and writing to the file system;
- **Security mechanisms:** Covers identification, authentication, authorization, privacy, encryption and secure designs;
- **Concurrency:** Covers the skills and mechanisms necessary to securely implement multi-threaded applications;
- **C types:** Covers the skills and mechanisms necessary to securely interact with built-in C types (*e.g.,* null-terminated strings, pointers and arrays);
- **Error conditions:** Covers the skills and mechanisms necessary to securely identify and handle error conditions in applications;
- **Coding correctness and style:** Covers the skills and mechanisms necessary to write simple, readable and correct code to minimize the introduction of vulnerabilities (*e.g.,* avoid dangerous functions and do not mix assignment and comparison operators).

SUGGESTED RESOURCES

- Carnegie Mellon University Software Engineering Institute/Computer Emergency Response Team Secure Coding Standards Webpage. Accessed 21 January 2008 at: https://www.securecoding.cert.org/
- SANS Institute. "C Secure Coding Tasks, Skills and Knowledge." April 2007. Accessed 4 April 2008 at: http://www.sans.org/gssp/SANS-SSI%20C%20Blueprint%20(9-07).pdf
- Seacord, Robert C. *Secure Coding in C and C++.* Indianapolis, Indiana: Addison-Wesley Professional, 2005.

C.4.2 Java

Java allows the user to have almost full control of the virtual environment in which the Java bytecode is run. Many of the security flaws discovered in Java have been implementation problems in the Java Virtual Machine (JVM). Because portions of the JVM are written in C, it is subject to the same coding errors and security flaws as any other C program. Much of Java's security mechanisms can be traced back to one of its original uses: running untrusted Java applets on client machines. Java has since become a popular platform for enterprise applications, where language portability and security are important concerns.

While the Java language itself provides a number of security features, applications can only be secured when these features are used appropriately. Java applets run with almost all of Java's security features enabled while server-side code is often run in less restrictive configurations. There are common requirements when the Java code runs on client or on the server. Input from untrusted sources should always be validated.

Increasingly, Java is being used to implement Web services, enabling server-to-server communication *via* language-independent mechanisms. Web services are a conglomeration of software systems designed to support interoperable machine-to-machine interaction over a network. This encompasses services that use SOAP-formatted XML envelopes and have their interfaces described by WSDL. Inherently, Web services suffers from many of the same vulnerabilities as Web applications, with the methods for exploitation differing. Therefore, it is critical to ensure Web services are securely implemented.

C.4.2.1 Secure Java development

As one of the most popular development languages, a number of organizations have put together secure coding guidelines and considerations for Java developers. In 2007, the SANS Institute introduced the GIAC GSSP Java certification. This certification aims to recognize Java developers who understand Java security mechanisms and can mitigate Java-specific security concerns. The GSSP Java certification's essential skills include nine *tasks*, which address specific aspects of Java development:

- Input handling, which covers the skills and mechanisms necessary to deal with potentially malicious input;
- Authentication and session management, which covers the skills and mechanisms necessary to securely provide authentication and session support in Java;
- Access control, which covers the skills and mechanisms available for access control in Java, specifically the Java EE controls and Java Authentication and Authorization Service API;
- Java Types and JVM Management, which covers the security implications of Java types and garbage collection;
- Application Faults and Logging, which covers the skills and mechanisms available for securely handling application faults and logging output;

- Encryption Services, which covers the skills and mechanisms necessary to take full advantage of the Java Cryptographic Extensions API;
- Concurrency and Threading, which covers the skills and mechanisms necessary to properly structure multi-threaded Java programs;
- Connection Patterns, which covers the skills and mechanisms necessary to securely interface with other applications;
- Miscellaneous, which covers language-specific security mechanisms, such as access modifiers, Java Archive (JAR) protections and Java EE filters.

SUGGESTED RESOURCES

- Sun Microsystems. Secure Coding Guidelines for the Java Programming Language, Version 2.0. Accessed 22 March 2008 at: http://java.sun.com/security/seccodeguide.html
- Sun Developer Network Java SE Security page. Accessed 11 December 2007 at: http://java.sun.com/javase/technologies/security/
- Sun Java 6 Security Documentation page. Accessed 11 December 2007 at: http://java.sun.com/javase/6/docs/technotes/guides/security/index.html
- OWASP Java Project page. Accessed 11 December 2007 at: http://www.owasp.org/index.php/Category:OWASP_Java_Project
- The Secure Programming Council. "Essential Skills for Secure Programmers Using Java/JavaEE." November 2007. Accessed 4 April 2008 at: http://www.sans.org/gssp/essential_skills_java.pdf
- TenorLogic.com Java Security Resources page. Accessed 11 December 2007 at: http://www.tenorlogic.com/
- Mikhalenko, Peter V. "Understanding the Java security model". TechRepublic, 18 April 2007. Accessed 11 December 2007 at: http://articles.techrepublic.com.com/5100-3513-6177275.html
- Nagappan, Ramesh. "Demystifying Java security—Part 1". SearchSoftwareQuality.com, 21 June 2006. Accessed 11 December 2007 at: http://searchsoftwarequality.techtarget.com/originalContent/0,289142,sid92_gci1195320,00.html —*and*— "Demystifying Java security—Part 2". SearchSoftwareQuality.com, 23 June 2006. Accessed 11 December 2007 at: http://searchsoftwarequality.techtarget.com/originalContent/0,289142,sid92_gci1195332,00.html
- Chess, Brian. "Twelve Java Technology Security Traps and How to Avoid Them". Presented at JavaOne 2006. Accessed 11 December 2007 at: http://developers.sun.com/learning/javaoneonline/j1sessn.jsp?sessn=TS-1660&yr=2006&track=coreenterprise

- Vaas, Lisa. "Java Security Traps Getting Worse". *eWeek*, 9 May 2007. Accessed 11 December 2007 at: http://www.eweek.com/article2/0,1759,2128071,00.asp

- Dubin, Joel. "Java security: Is it getting worse?". SearchSecurity.com, 12 July 2007. Accessed 11 December 2007 at: http://searchsecurity.techtarget.com/tip/0,289483,sid14_gci1263607,00.html

- IBM Java Security Research Webpage. Accessed 11 December 2007 at: http://domino.research.ibm.com/comm/research_projects.nsf/pages/javasec.index.html

- Paul, Nathanael and David Evans. "Comparing Java and .NET Security: Lessons Learned and Missed". *Computers and Security*, Volume 25 Issue 5, July 2006. Accessed 11 December 2007 at: http://www.cs.virginia.edu/~nrp3d/papers/computers_and_security-net-java.pdf

- Hennebrueder, Sebastian. "Java security in Web application, typical attacks, Tomcat security". Tutorial, 3 March 2007. Accessed 11 December 2007 at: http://www.laliluna.de/java-security-Web-application.html

- "Learning Guide: Developing secure enterprise Java applications". ComputerWeekly.com, 19 April 2006. Accessed 11 December 2007 at: http://www.computerweekly.com/Articles/2006/04/19/218968/developing-secure-enterprise-java-applications.htm

- IBM alphaWorks Security Workbench Development Environment for Java (SWORD4J). Accessed 11 December 2007 at: http://alphaworks.ibm.com/tech/sword4j/

- JSecurity Java security application development framework project Website. Accessed 11 December 2007 at: http://www.jsecurity.org/

- Acegi project Website. Accessed 26 Mar 2008 at: http://www.acegisecurity.org

C.4.3 C# and VB.NET

C# was developed by Microsoft as part of its .NET initiative to provide developers with the power of languages like Java and C++ for rapid application development. Like Java, C# runs in a secure environment through code access security, which provides a sandbox in which to run Common Language Runtime (CLR) applications, such as C# and VB.NET (Visual Basic for .NET). Like Java, C# provides improvements over C and C++ through strong type checking, array bounds checking, detection of attempts to use uninitialized variables, source code portability (through the .NET CLR) and garbage collection. Unlike Java, C# applications also support accessing raw memory—although these portions of the code must be marked as unsafe, allowing reviewers to easily locate such code. Through code access security, C# offers security similar to Java's while providing native access to all functionality that the .NET framework provides.

Visual Basic .NET was released along side of C# as a successor to the legacy Visual Basic language. While the VB.NET syntax is very similar to the original Visual Basic language, VB.NET provides a fully object-oriented language in place of the Component Object Model-

based language of Visual Basic. While Visual Basic has long been notorious for its security concerns, VB.NET's support of garbage collection, object-oriented design, code access security, and the .NET framework place it on equal footing with C#. Through the .NET framework and the CLR, C# and VB.NET have access to the same security libraries and both C# and VB.NET applications are subject to code access security.

Like in Java, code written in C# or VB.NET can be downloaded and run from untrusted sources. To alleviate this problem, Microsoft has integrated code access security into the .NET Framework. Code access security provides varying levels of trust for code based on where the code originates and allows individual users to specify what permissions will be given to an application. Because code access security is part of the .NET Framework, all applications that access the .NET Framework can be subject to code access security. Because policies are defined on a per-machine basis, libraries are provided that allow applications to determine whether the application has a particular permission prior to performing a potentially restricted act—allowing .NET applications to alter their behavior rather than simply throw a security exception.

Windows security is principal-oriented with authorization decisions being based on the identity of an authenticated principal. Code Access Security adds another dimension of security by allowing authorization decisions to be based on the identity of the code rather than the user who runs the code.

Code is authenticated and its identity is determined using attributes of the code called *evidence*. Evidence can include an assembly's public key, which is part of its strong name, its download URL, or its installed application directory, among other attributes. Once all the evidence is gathered and passed through security policy, the policy is then used to determine the capabilities of the code and the permissions it has to access secure resources.

Default policy ensures that all code installed on a local machine is fully trusted and granted an unrestricted set of permissions to access secure resources. As a result, any resource accessed is only subject to operating system security. Code installed on a local machine is fully trusted because a conscious decision is required by an administrator to install the software in the first place.

The .NET Framework uses principal objects, which contain identity objects, to represent users when .NET code is running. Together they provide the backbone of .NET role-based authorization. For ASP.NET Web applications, a principal and identity object attached to the current thread and Web request represents the authenticated user.

Identity and principal objects must implement the *IIdentity* and *IPrincipal* interfaces respectively. These interfaces are defined within the *System.Security.Principal* namespace.

SUGGESTED RESOURCES

- Microsoft Corporation. "Secure Coding Guidelines". Chapter in *Visual Studio 2008 .NET Framework Developer's Guide.* Accessed 22 March 2008 at: http://msdn2.microsoft.com/en-us/library/d55zzx87.aspx

- Security (C# Programming Guide). Accessed 25 March 2008 at: http://msdn2.microsoft.com/en-us/library/ms173195.aspx

- Security Tutorial. Accessed 25 March 2008 at: http://msdn2.microsoft.com/en-us/library/aa288469(VS.71).aspx

- Shakil, Kamran. "Security Features in C#." Csharphelp.com. Accessed 25 March 2008 at: http://www.csharphelp.com/archives/archive189.html

- Meier, J.D., Alex Mackman, Michael Dunner, Srinath Vasireddy, Ray Escamilla, and Anandha Murukan. "Code Access Security in Practice". Chapter 8 of *Improving Web Application Security: Threats and Countermeasures*, Microsoft Developer Network, January 2006. Accessed 8 April 2008 at: http://msdn2.microsoft.com/en-us/library/aa302424.aspx

- The Code Project. "Understanding .NET Code Access Security". 14 January 2004. Accessed 8 April 2008 at: http://www.codeproject.com/KB/security/UB_CAS_NET.aspx

C.4.4 Ada

Ada was designed from its introduction in 1983, with the first Ada standard, Ada-83, to support sound software engineering techniques and promote program reliability, predictability, and analyzability. Since its introduction in 1983, Ada has been used in the development of safety-critical systems. Many of the language issues that Ada addresses in safety-critical development are also relevant for secure software. For example, Ada includes a number of features that help address common software security vulnerabilities, including:

- Strong typing that prevents many range errors that lead to buffer overflows in weakly typed or un-typed languages, such as C and C++;

- Clear syntax for numeric literals;

- Compile-time detection of typing errors;

- Runtime checking that raises exceptions, thereby allowing controlled recovery;

- Language semantics that are precisely-defined (by an international standard), and thus clearly understandable;

- Encouragement for programming of well-defined, predictable program behaviors;

- Ability to declare scalar data using explicit ranges, assisting in program readability and range analysis - a key differentiator between Ada and C, C++, and Java;

- Parameter passing whereby parameter modes are specified based on the direction of data flow ("in", "out", or "in out"), which facilitates data flow analysis;

- Inclusion in Ada of a high-level model ("tasking") that supports structured concurrent programming, enabling security functions to easily be mapped to individual Ada tasks, elegantly supporting one-security-function-per-thread-of-control. By contrast, C and C++ lack this facility entirely, and Java's thread model is low-level and error prone;

- Direct support for fixed point binary and decimal data;

- A compiler directive, the pragma Normalise_Scalars, that directs the compiler to use an invalid value as the default initialization for scalar objects whenever possible.

- Ability to easily specify use of only a complete, usable subset of the full language from which individual features have been removed, *e.g.,* to impose restraints needed to achieve security objectives. This "subsetting" is accomplished through a compiler directive, pragma Restrictions set with arguments identifying the specific features to be excluded; this pragma can be used in concert with a higher-level directive, pragma Profile, which defines a collection of Restrictions pragmas.

While there is no predefined secure subset of Ada, Ada's subsetting feature enables developers to define their own subsets based on the expressability desired and the analysis techniques they plan to use. This said, there is a high-integrity Ada subset, SPARKAda (from Praxis High Integrity Systems). SPARK Ada is an Ada subset augmented with special annotations that identify various program properties, such as pre-conditions, post-conditions, and data dependencies. Through these annotations SPARKAda eliminates ambiguities that arise from language constructs that are not fully specified by the Ada standard.

The Ada-95 standard Language Reference Manual includes Annex H, "Safety and Security", which "addresses requirements for systems that are safety critical or have security constraints." The specific information provided in the Annex includes information on:

- How to leverage the pragma *Normalize_Scalars* to better understand an Ada program's execution;

- How to document the program's effect when a bounded error is present, or the standard Ada language rules otherwise leave the effect undocumented. Among the situations that the Annex recommends be documented are parameter-passing conventions, runtime storage management methods, and the method used to evaluate numeric expressions when that evaluation requires extended range or extra precision.

- How to leverage the pragmas *Reviewable* and *Inspection_Point* to produce object code that is more easily reviewed for security and safety concerns;

- Which pragmas and other Ada language constructs should be restricted to ease the program analysis and verification of correctness and predictable execution;

The Annex also notes that the standard Ada *Valid* attribute is useful in avoiding safety and security problems that could arise from use of scalars that have values that fall outside their declared range constraints.

The Language Reference Manual for newest version of the Ada standard, Ada-2005, has re-titled Annex H "High Integrity Systems", which the Manual states include "safety-critical systems and security-critical systems". Otherwise the Annex is unchanged in its essentials.

SUGGESTED RESOURCES

- Ada Standards. Accessed 8 September 2008 at: http://www.adaic.org/standards/
- Praxis High Integrity Systems. SPARKAda. Accessed 8 September 2008 at: http://www.praxis-his.com/sparkada/index.asp
- Barnes, John. *Safe and Secure Software: An Introduction to Ada 2005.* Accessed 8 September 2008 at: http://www2.adacore.com/home/ada_answers/ada_2005/safe_secure/
- Dewar, Robert B.K. and Roderick Chapman. "Building secure software: Your language matters!" *Military Embedded Systems*, Winter 2006. Accessed 8 September 2008 at: http://www.mil-embedded.com/articles/id/?2012
- Brosgol, Benjamin M. and Robert B. K. Dewar. "Need Secure Software?". *Application Software Developer*, June 2008. Accessed 8 September at: http://www.applicationsoftwaredeveloper.com/features/june07/article2.html
- International Organization for Standards/International Electrotechnical Commission (ISO/IEC). "ISO/IEC TR 15942:2000: Information technology—Programming languages—Guide for the use of the Ada programming language in high integrity systems". Accessed 9 September 2008 at: http://standards.iso.org/ittf/PubliclyAvailableStandards/c029575_ISO_IEC_TR_15942_2000(E).zip
- ISO/IEC. "ISO/IEC TR 24718:2005: Information technology—Programming languages—Guide for the use of the Ada Ravenscar Profile in high integrity systems". Accessed 9 September 2008 at: http://standards.iso.org/ittf/PubliclyAvailableStandards/c038828_ISO_IEC_TR_24718_2005(E).zip

C.4.5 Ruby

Ruby is an object oriented scripting language developed in 1995 by Japanese developer Yukihiro "Matz" Matsumoto. It incorporates features provided by Perl, SmallTalk, Eiffel, Ada, and Lisp. Within the past few years, Ruby has become an increasingly popular for Web development using the Ruby on Rails framework, developed by David Heinemeier Hansson.

Ruby on Rails provides a Web development stack similar to that provided by the Java EE or .NET Web development frameworks. It provides built-in support for integrating with relational databases, Web services and AJAX.

Ruby on Rails aims to simplify Web development by relying on "convention over configuration." In Java EE and .NET applications, developers are required to use a number of configuration mechanisms to describe how the application interacts with a database and how the Model, View and Controller components are related to one another. In Ruby on Rails, users are expected to follow a standard pattern in naming objects, variables and SQL tables. This provides allows for the rapid development of Web applications using little code. A

corollary to this is that the Web application has the potential to be more secure due to the reduced complexity of developed code.

One important item to note regarding Ruby on Rails is that it is one of the first Web application frameworks to support a mechanism for protecting against Cross Site Request Forgery (CSRF) attacks. The CSRF-killer plug-in[58] provides a transparent mechanism for supplying and verifying tokens to protect against CSRF attacks.

While there are many benefits to Ruby on Rails, it faces a number of criticisms. One of the primary criticisms is that it is not as robust as its older counterparts (Ruby on Rails 1.0.0 was released December 2005). Another criticism is that Ruby on Rails relies on the RubyGems framework for distributing libraries for the Ruby language; while RubyGems provides support for digital signatures, it is not yet widely used. Additionally, Ruby is considered slower than other scripting languages (*e.g.,* Perl and Python) and slower than other Web application frameworks such as Java EE and .NET.

Some of the security concerns are alleviated by running Ruby on Rails on different Ruby interpreters. Two common interpreters are IronRuby, which will run Ruby applications on .NET environments; and JRuby, which runs supports Ruby on Rails on top of the Java Virtual Machine, including Java EE applications.

SUGGESTED RESOURCES

- Ruby on Rails Security Project Website. Accessed 28 January 2008 at: http://www.rorsecurity.info
- Aggarwal, Nakul and Ritesh Arora. *Ruby on Rails Security Guide.* Published on QuarkRuby Weblog, 20 September 2007. Accessed 26 March 2008 at: http://www.quarkruby.com/2007/9/20/ruby-on-rails-security-guide
- Anonymous author. *SecurityConcerns in Ruby on Rails Wiki.* 10 January 2008. Accessed 26 March 2008 at: http://wiki.rubyonrails.org/rails/pages/SecurityConcerns

C.4.6 Server-side scripting languages

A script that runs on or interacts with a networked client is always a vulnerable target of attack, and should be written extremely carefully to minimize exploitable vulnerabilities that threaten the integrity of the system as a whole, or enable the attacker to gain unauthorized access to the underlying server.

The following recommendations should increase the security of scripts:

- Do not include or call out to shell scripts;

58 Downloadable from: http://activereload.net/2007/3/6/your-requests-are-safe-with-us

- Server-side include statements that display environmental variables and file statistics present a low security risk; these functions are issued with directives such as *include* and *echo var*. A significant security risk arises, however, when using server-side includes with the *virtual* and *exec* directives to execute programs on the server. Allowing the server to execute files in this way also gives attackers the means direct the server to disclose private information or issue unauthorized commands. The article "Security Hints for Server-Side Includes" provides useful guidance on mitigating the risk posed by using server-side includes with the *exec* directive.[59]

- Do not allow "<" and ">" in user input; either reject the input, or escape the characters by prepending them with a back slash ("\");

- Only allow expected characters (A-Z, 0-9 and punctuation). In cases where minimal HTML is allowed, all characters should be properly encoded (*e.g.*, "'", "<", and ">");

- Remove all comments from client-side script code (also remove comments from HTML, XHTML, and any other user-viewable source code);

  Only trust a client/browser for which there is strong evidence of trustworthiness. Browsers are particularly vulnerable to compromise by malicious code and attackers due to the frequent combination of lack of robustness of the client platform protections and the browser's inherent configurability by their users, who are not always security-savvy. A well-behaved browser or other client should escape all characters in query strings that have special meaning to the underlying operating system shell. This will avoid the possibility of the script misinterpreting those characters, and will thwart any malicious user's attempt to include special characters in query strings in order to confuse the script and gain unauthorized access to the system shell;

- Never trust data supplied to a script by a user, even if the script does not invoke the system shell;

- Do not use the *HTTP_REFERER* header for authentication in a CGI program. This header originates in the browser, not the server, and thus can be easily tampered with;

- Validate all data. Do not accept non-validated data or meta characters, or pass them to the system shell. Instead of simply detecting the meta characters, escape them;

- The "\" (back slash), which acts as the shell escape character, may itself be present in input data and cause problems. If it is found in input (during input validation) it should be stripped out and the data following it should be validated;

- Encode dynamic output to prevent the passing of malicious scripts to the user.

---

59 Accessed 12 September 2008 at:
http://www.workz.com/content/view_content.html?section_id=482&content_id=5814

In 2006, the DHS Science and Technology Directorate established its Vulnerability Discovery and Remediation: Open Source Hardening Project, contracting Coverity to evaluate a number of popular open source applications, programs, and tools, to locate their software security vulnerabilities, and Symantec and Stanford University to remediate those vulnerabilities. As of March 2008, Coverity had evaluated over 40 open-source software packages, including the "LAMP" stack (Linux, Apache, MySQL, and Perl/Python/PHP). As of March 2008, the Project had reviewed, remediated, and reported on eleven open source software packages, including the Perl, PHP, Python, and Tool Command Language (TCL) scripting languages.

C.4.6.1 Perl

In addition to the secure scripting guidance above, there are language-specific practices that will help secure applications written in Perl. Chief among these is the use of Perl taint mode and escaping of HTML data input to Perl scripts.

Taint mode causes Perl to perform extra security validations when accessing variables and making function calls. Taint mode ensures that any tainted data received from outside the program are not used, directly or indirectly, to modify files, processes, or directories.

Specifically, taint mode prevents data derived outside the program from accidentally affecting anything else inside the program. It marks such data as tainted. All externally obtained input is marked tainted, including:

- Command-line arguments,
- Environment variables,
- Locale information (see *perllocale()*),
- Results of the *readdir* and *readlink* system calls,
- Results of the *gecos* field of *getpw** calls,
- All file input.

Tainted data may not be used directly or indirectly in any command that invokes a sub-shell, nor in any command that modifies files, directories, or processes. There is one important exception: if the script passes a list of arguments to either *system()* or *exec(),* the elements of that list will not be checked for tainting. Be especially careful when using *system()* or *exec()* while in taint mode. Any data value derived from tainted data becomes tainted also.

To untaint data, perform a regular expression against a substring of the tainted data. Do not simply use "*" as the substring will defeat the taint mode mechanism altogether. Instead, identify safe patterns allowed by the script, and use them to extract good values. Check all extracted values to ensure that they do not contain unsafe characters or exceed safe lengths.

When invoking Perl (particularly when running the script as a Unix/Linux *setuid* program), place the *-T* flag at the end of the command line to put Perl into taint mode. For example:

```
#!/usr/bin/perl –T
```

See the *perlsec()* manual page for details. It is important to note that Perl does not consider "print" function to be an unsafe function.

Perl and mod_perl systems enable the developer to add input validation of HTML-escaped input data. For example, insert the following line of code before any output to eliminate any input that is not an alphanumeric character or a space:

```
$text =~ s/[^A-Za-z0-9 ]*/ /g;
```

Use the *HTML::Entities::encode()* function in the *HTML::Entities* module of the *libwww-perl CPAN* distribution to escape HTML characters in input data. This encodes data into HTML entity references.

When using the *Apache::Registry* script or mod_perl handler, use *Apache::Util::escape_html()* in the *Apache::Registry* to encode all HTML input.

Use the *Apache::TaintRequest* module to automate escaping of HTML data. This will HTML-escape any tainted data item found and pass all untainted data to the browser without altering it.

Also void the functions *open(), glob,* and backtick (`) that call the system shell to expand filename wildcard characters. Instead of *open()* use *sysopen(),* or in Perl v5.6 or later, use *open()* with three parameters (consult the *perlopentut()* manual page). Instead of backticks, use *system()* or a safer call.

C.4.6.2 PHP

PHP v4.1.0 and earlier versions are less secure than most scripting languages due to the way earlier versions of PHP load data into the PHP namespace. All environment variables and values sent to PHP over the Web as global variables are automatically loaded into the same namespace as normal variables, enabling attackers to set those variables to arbitrary values that persist until they are explicitly reset by a PHP program.

When a variable is first requested, PHP automatically creates that variable with a default value. Therefore, PHP programs often do not initialize variables. If the programmer forgets to set a variable, PHP must be explicitly configured to report the problem. Otherwise, by default, it will simply ignore it. These behaviors in PHP allow attackers to completely control the values of all variables in the program, unless the program was written to explicitly reset all PHP default variables upon execution. Failing to reset a single variable may create a vulnerability in the PHP program.

The following PHP program illustrates the problem. The program is intended to implement an authorization check to ensure that only users who submit the correct password are allowed access to sensitive information. However, by simply setting *auth* in his/her Web browser, the attacker can subvert this authorization check:

```
<?php
if ($pass == "hello"
$auth = 1;
...
if ($auth == 1)
echo "sensitive information";
?>
```

C.4.6.2.1 Setting register_globals to "Off"

It is possible to eliminate the most common PHP attacks by setting *register_globals* to "off"; however, older versions of PHP are hard to use with *register_globals* off. In PHP 6.0, *register_globals* will be removed from the language.

In PHP v4.7.0 and later versions, external variables received from the environment, or from an HTTP request, a cookie, or the Web server are no longer registered in the global scope by default. Instead, they are accessed using the language's new *Superglobal* arrays. Several other special arrays—most notably *$_REQUEST*—make it easier to develop PHP programs with *register_globals* set to "off".

Note that many third-party PHP applications will not operate correctly, and in some cases not at all, with *register_globals* set to "off"; it may be necessary to use this setting selectively for only those programs that can operate with *register_globals* off. For example, on an Apache Web server, insert the following lines into the *.htaccess* file in the PHP directory:

```
php_flag register_globals Off
php_flag track_vars On
```

Also consider using directory directives for further control. Also, note that the *.htaccess* file itself will be ignored unless the Apache Web server has been configured to permit overrides. Check to ensure that the Apache global configuration does not set *AllowOverride* to "none". Instead, configure the *AllowOverride* options in the Apache configuration file so as to be able to write helper functions that load only the data needed by the PHP programs.

Later versions of PHP also provide functionality that makes it easier to specify and limit the input the program should accept from external sources. Routines can be placed in the PHP library to enable users to list the input variables they want to accept. Then, functions can be written to check the validity of the patterns and types of variables before coercing the program to use them.

If *register_globals* cannot be set to "off", write the program to accept only values not provided by the user, and do not trust PHP default values. Do not trust any variable that has not been explicitly set. Remember that these variables must be set for every entry point into the program itself and in every HTML file that uses the program.

Following the guidelines below should provide additional safety and security to PHP scripts.

- Begin each PHP program by resetting all variables, including global variables referenced transitively in included files and libraries, even if this means simply resetting those variables to their default values. Before doing this, the developer needs to learn and understand all the global variables that might be used by the functions called in the program. Search through the *HTTP_GET_VARS, HTTP_POST_VARS, HTTP_COOKIE_VARS,* and *HTTP_POST_FILES* to determine the originator of the data—it should not be a user. This search should be repeated whenever a new version of PHP is released, because it may add a new data source;

- Write the program to record all errors by piping them to error reports in a log file;

- Prevent remote file access by filtering all information that is used to create filenames. PHP defaults to functionality that allows it to use commands like *fopen()* to invoke Web or FTP requests from other sites;

- Use only the *HTTP_POST_FILES* array and related functions to upload files and access them. PHP allows attackers to temporarily upload files with arbitrary content. This cannot be prevented in the language itself;

- Place only protected entry points in the document tree, and place all other code—that is the majority of code—outside the document tree. Do not place *include* (*.inc*) files inside the document tree;

- Do not use PHP's session mechanism, which contains security vulnerabilities;

- Because PHP is loosely typed, after validating all inputs use typecasting to coerce non-string data into the type that data should have. Develop helper functions to check and import a selected list of expected inputs;

- Use *magic_quotes_gpc()* where appropriate to help protect the script against several types of attacks;

- Modify the *php.ini* file to disable file uploads, *i.e., file_uploads = Off;*

- Be very careful when using the following functions (avoid using them if possible):

  Code Execution:
  eval()
  include()
  preg_replace() [with or without */e* flag]
  require()

Command Execution:
` [backtick]
exec()
passthru()
popen()
system()

File Opening:
file()
fopen()
readfile().

This list is not exhaustive; refer to the PHP specification for a complete list of all PHP code and command execution functions and file opening commands.

C.4.6.2.2 PHP implementation of input validation

To ensure that the design conforms with the following principles:

- Non-bypassability of input validation;
- Inability to mistake tainted (bad/unfiltered) data for validated data;
- All data origins can be determined.

There are two main input validation methods possible in PHP: *Dispatch* and *Include*.

In PHP *Dispatch*, a single PHP script is available to the public (*via* URL). Everything else is a module included as needed with include or require. A *GET* variable is passed with every URL to identify the task (in modern frameworks and techniques, this requirement may be eliminated). This *GET* variable can be considered as a replacement for the script name. The following is a code example:

```
http://example.org/dispatch.php?task=print_form
```

Because *dispatch.php* is the only resource available to the public, the design of this application ensures that any global security measures taken at the top cannot be bypassed. It also lets a developer easily see the control flow for a specific task, so he/she can ensure that input filtering takes place without having to review a lot of code.

The dispatch method leverages existing mechanisms that have been proven reliable, such as the fact that only files within document root can be accessed *via* URL. It also relies less on the developer remembering to do something about security.

In the PHP *Include* method, a single module is included at the top of every public script within document root. This module is responsible for all global security measures, such as ensuring that input filtering cannot be bypassed. If input filtering is placed in a separate module, the developer should make sure that no path through the application's logic bypasses the initialization of that module.

C.4.6.2.3 Naming conventions for variables

Only choose methods that will not make it easy to mistakenly use tainted (unfiltered) data. One approach is to rename any variable that is filtered to something that distinguishes it as being clean.

SUGGESTED RESOURCES

- Hudson, Paul. "PHP—Secure Coding". *Linux Format,* Issue 56, August 2004. Accessed 22 March 2008 at: http://www.linuxformat.co.uk/wiki/index.php/PHP_-_Secure_coding
- Oertli, Thomas. "Secure Programming in PHP". 30 January 2002. Accessed 11 April 2008 at: http://www.cgisecurity.com/lib/php-secure-coding.html
- PHP Security Consortium Website. Accessed 26 March 2008 at: http://phpsec.org/
- Hardened-PHP Project Month of PHP Bugs Website. Accessed 26 March 2008 at: http://www.php-security.org/
- The PHP Group. "A Note on Security in PHP". Not dated. Accessed 26 March 2008 at: http://www.php.net/security-note.php
- Secure PHP Wiki. Accessed 14 December 2007 at: http://www.securephpwiki.com/index.php/Main_Page

C.4.6.3 Python

As with other languages, be very careful when using functions and calls that allow data to be executed as parts of a program, including the following:

Functions:	**Calls:**
exec()	*compile()*
eval()	*input().*
execfile()	

Privileged Python programs that can be invoked by unprivileged users must not import the user module, which causes the *pythonrc.py* file to be read and executed. Doing so could allow an unprivileged attacker to force the trusted program to run arbitrary code.

Python does very little compile-time checking. It implements no static typing or compile-time type checking. Nor does it check that the number of parameters passed is legal for a given

function or method. There is an open source program, *PyChecker*, which can be used to check for common bugs in Python source code (see *Bibliography and References*).

Python does support restricted execution through its *RExec* class. *RExec* is primarily intended for executing applets and mobile code, but can also be used to limit privileges in a program when the code originated external to the program. By default, a restricted execution environment permits reading (but not writing) of files, and does not allow operations for network access or graphical user interface (GUI) interaction. If these defaults are changed, beware of creating security vulnerabilities in the restricted environment.

Python's implementation calls many hidden methods and passes most objects by reference. When inserting a reference to a mutable value into a restricted program's environment, first copy that mutable value or use the *Bastion* module. Otherwise, the program may change the object in a way that is visible outside the restricted environment.

C.4.6.4 ASP and ASP.NET

ASP is a fairly simple and secure language and is not prone to many of the vulnerabilities that plague other languages such as C/C++, Visual Basic, or Perl. The greatest extent of security violations for ASP occurs in the non-secure configuration of the Web server and in the failure to apply necessary patches.

ASP.NET is very similar to the syntax and theory of ASP, with the addition of numerous security features that allow quick and easy deployment. Almost every Web application that has sensitive data is going to have some concept of access privileges and restricted user input. The ASP.NET model allows for scalable and efficient deployment regarding both of these topics. The security features of ASP.NET include:

- Input validators;
- Authentication modes and authorization options: ASP.NET works in conjunction with Microsoft's Internet Information Services, the .NET Framework, and the underlying security services provided by the operating system, to provide a range of authentication and authorization mechanisms;
- Gates and gatekeepers.

When a client makes a request, the Web server is responsible for handling the file type requested. When creating a Website that uses ASP pages, the Web server must process all documents that end in the .asp extension before they are handed over to the client. Requesting pages or files that do not end in the .asp extension can circumvent the processing that is performed by the ASP engine. In such an instance, the unprocessed page will be sent to the client in text format. The two most common mistakes that cause such problems are as follows:

- Most ASP files that are *included* in other ASP files have the *.inc* extension; this allows a client to make a direct request to the file and return the source code in text format;

- When files are modified directly on the server backup (*.bak*) files may be created by the development environment. These files may be requested by the client and returned in source form.

Both of these vulnerabilities deal with files that are simply not processed by the ASP engine because they do not have the correct extension. This can be avoided by either appending .asp to the filename, deleting the file from the Web root, or placing the extension in the Web server configuration so that it must be processed before being sent to the client.

If the ASP application accesses a Microsoft Access Database, it is critical that the folders on the Web server in which the *MDB* files are stored are not publicly accessible. Otherwise anyone who can guess the name and location of the database will be able to download it *via* a browser. Set up a "safe" data directory outside the Web root with permissions set that allows access by the Web server but denies access by any client, then store the *MDB* files in that "safe" directory.

Despite all safeguards, given the history of IAVAs and CERT and CSIRT reports associated with Microsoft products, one must expect there to be exploitable vulnerabilities in the Windows operating system, Internet Information Server Web server, ASP components, and/or Access Database used in the ASP or ASP.NET application. Moreover, one should expect an attacker to successfully exploit one of these vulnerabilities at some point in its lifetime in order to compromise the application or its data. There are new patches and fixes coming out daily from Microsoft, and it is critical to apply each one in a timely fashion in order to ensure as secure a Web server, database, and Web application as possible.

Even with all patches are kept up to date, the most restrictive user account should be used when connecting to the database and processing page requests. Unauthenticated "anonymous" access to Web server resources is usually controlled *via* the *IUSR_MachineName* or *IWAM_MachineName* Windows user account (where *MachineName* is the name of the Windows server). *IUSR_MachineName* is used by default, but if the Web server is configured to run out-of-process (*i.e.*, with Application Protection set to "High Isolate" on the Home Directory tab of the Web server) then *IWAM_MachineName* will be used instead.

For the highest level of security, disable all permissions for these accounts and other anonymous accounts, and strictly control their access through Windows user and group access control lists. Disable anonymous access *via* the Internet Services Control Panel. Remember when testing and troubleshooting that permissions can be inherited from multiple groups and/or from the parent folders.

C.4.7 Client-side scripting languages

With the advent of AJAX and other Web 2.0 technologies, client side scripting languages are becoming increasingly important. In the past, it has been sufficient to simply disable client-side scripting to improve security, but many enterprise-level Web applications have begun introducing Web 2.0 concepts, requiring JavaScript or VBScript to be enabled. As such, it is

important for Web developers to understand the implications of relying on client-side processing in Web applications.

Other scripting languages, such as shell and TCL scripts, are equally necessary for non-Web applications in Unix environments. Shell scripts perform vital functions in most—if not all—Unix-like environments. As such, these scripting languages are used by many developers due to the implicit guarantee they will be part of the deployment environment (*e.g.,* languages such as Perl, Python and Java, while very widespread, may not be as readily available). As such, this guide provides some insight into the security considerations that must be taken into account when using these languages.

C.4.7.1 JavaScript

JavaScript is a general purpose, cross-platform scripting language available in most Web browsers. It can be embedded within standard Web pages to create interactive documents. Each JavaScript interpreter supplies the needed objects to control the execution environment, and because of differences the functionality can vary considerably. Within the context of a Web browser, the language is powerful, allowing prepared scripts to modify the Web page and perform calculations on the client side, improving perceived performance and potentially reducing the load on the server. Within a browser context, JavaScript does not have methods for directly accessing a client file system or for directly opening connections to other computers besides the host that provided the content source. Moreover, the browser normally confines a script's execution to the page in which it was downloaded.

The name JavaScript is a misnomer since the language has little relationship to Java technology and arose independently from it. Netscape developed JavaScript for its Navigator browser, and eventually JScript, a variation of JavaScript, appeared in Microsoft's Internet Explorer. Standardizing the core language and facilities of JavaScript and JScript resulted in the ECMAScript Language Specification,[60] which most modern Web browsers support. Design and implementation bugs have been discovered in the commercial scripting products provided with many browsers, including those from Microsoft, Apple, Mozilla, and Opera.

Visual Basic Script (VBScript) is a programming language developed by Microsoft for creating scripts that can be embedded in Web pages for viewing with the Internet Explorer browser. Alternative browsers do not support VBScript. Like JavaScript, VBScript is an interpreted language able to process client-side scripts. VBScript is a subset of the widely used Microsoft Visual Basic programming language and works with Microsoft ActiveX controls. The language is similar to JavaScript and poses similar risks.

In theory, confining a scripting language to boundaries of a Web browser should provide a relatively secure environment. In practice, this has not been the case. Many browser-based attacks stem from the use of a scripting language in combination with a security vulnerability.

---

60 Available at: http://www.ecma-international.org/publications/standards/Ecma-262.htm

The main sources of problems have been twofold: the prevalence of implementation flaws in the execution environment and the close binding of the browser to related functionality such as an email client. Past exploits include sending a user's URL history list to a remote site, and using the mail address of the user to forge email. The increasing use of scripting technologies for various Websites has opened new avenues for exploits through embedded scripts.

SUGGESTED RESOURCES

- Taylor, John. "JavaScript Security in Mozilla". Mozilla.org Website, undated. Accessed 26 March 2008 at: http://www.mozilla.org/projects/security/components/jssec.html
- Ruderman, Jesse. "JavaScriptSecurity: Same Origin". Mozilla.org Website, undated. Accessed 26 March 2008 at: http://www.mozilla.org/projects/security/components/same-origin.html
- Sun Microsystems. "JavaScript Security". Chapter 14 of *Client-Side JavaScript Guide*, 27 May 1999. Accessed 26 March 2008 at: http://docs.sun.com/source/816-6409-10/sec.htm

C.4.7.2. AJAX (Asynchronous JavaScript And XML)

JavaScript is one of the main components of Asynchronous JavaScript And XML (AJAX), a collection of technologies that allows Web developers to improve the response times of Web pages. Through AJAX, developers rely on asynchronous JavaScript calls to fetch XML documents from the Web server in the background, without reloading the existing browser page.

AJAX relies on the *XMLHttpRequest* object, which was originally developed by Microsoft for Outlook Web Access 2000. Since its introduction, other Web browsers have provided compatible implementations. In 2006, the World Wide Web Consortium published a working draft specification. Because there is no final specification, each browser implements the *XMLHttpRequest* object in a slightly different manner (*e.g.,* IE uses an ActiveX object while Mozilla Firefox offers a built-in implementation). One of primary techniques for mitigating the effects of these differences is to rely on an AJAX framework, including Google's Web Toolkit, ASP.NET AJAX, and the Yahoo! User Interface Library.

In most cases, the Web browser will load an initial Web page containing JavaScript. This JavaScript code will instantiate the *XMLHttpRequest* object and send an HTTP request back to the Web server for additional data. Once the request is sent, a callback function is specified that will handle the data once it has been returned (allowing AJAX applications to handle multiple *XMLHttpRequest* calls at the same time). The content of these messages is arbitrary and can be defined by the developer. Traditionally, developers have relied on XML data but plaintext, HTML, and JavaScript Object Notation are becoming increasingly popular. Once the response has been received, the JavaScript code modifies the content within the browser, allowing the Web page to dynamically change without appearing to reload.

Due to its support for XML, AJAX applications can interact with Web services (both REST and SOAP services) for data interaction, allowing the Web browser to act as a GUI to an organization's SOA. In fact, many organizations have begun exposing Web service endpoints for these purposes, allowing Web developers to create "mashups," or applications that rely on Web services from multiple organizations. Popular mashups use the GoogleMaps API to provide a visual representation of data from a second Web service.

AJAX allows Web content to behave more like traditional applications, while potentially reducing the load on the Web server by offloading the majority HTML generation to the browser. However, AJAX also typically increases the attack surface of the Web server. In particular:

- AJAX increases the number of points where a client interacts with the application;
- AJAX may reveal details of internal functions within the Web application;
- AJAX inherits JavaScript security problems;
- By some estimates, approximately 70% of today's malicious code is downloaded *via* AJAX;
- Inexperienced developers may not design AJAX applications securely;
- In many cases, cross-site scripting attacks against Web applications in AJAX may be easier to accomplish, particularly when *XMLHttpRequest* calls retrieve JavaScript code;
- Most processing occurs in the background, preventing the user from detecting the download of malicious software or the execution of harmful operations (*e.g.,* CSRF).

It is important to note that while AJAX does have a built-in security model, it is increasingly being subverted by developers. AJAX relies on the same-origin restriction applied in most Web browsers. According to the Mozilla.org Website, the same origin policy prevents documents or scripts loaded from one origin from getting or setting the properties of a document from a different origin. This restricts AJAX HTTP requests to a single Web server. Nevertheless, many developers feel this is a "defective" security mechanism and have instituted a number of workarounds, allowing for cross-domain AJAX. In fact, Safari allows cross-domain AJAX without any workarounds. Should an attacker be able to succeed in a cross-site scripting attack that inserts malicious AJAX code, the attacker can perform any Web operation from the browser—potentially resulting in increasingly dangerous CSRF attacks. Without stringent protections against cross-site scripting, AJAX CSRF attacks can access any Website from the browser, including those corporate intranets, which may rely too much on the assumption that attackers will be stopped by the firewall or virtual private network.

While AJAX has introduced some potential vulnerabilities, many of the trends in AJAX development may eventually lead to more secure Web applications. For example, relying primarily on *XMLHttpRequest* endpoints that perform simple distinct functions, developers have begun implementing more rigid interfaces—potentially allowing for the development of improved testing mechanisms. Similarly, developers have begun to rely on AJAX frameworks for generating much of the AJAX JavaScript code. This allows testers to focus primarily on these frameworks, increasing their robustness and security along with that of the Web applications that rely on them. In addition, AJAX applications can be platform-independent—relying primarily on XML interfaces at the Web server. As such, whether a Web application is

written in PHP, Python, Java, or .NET is becoming increasingly irrelevant, as functionality can be re-implemented without requiring developers to alter any of the AJAX code.

SUGGESTED RESOURCES

- Hoffmann, Billy and Bryan Sullivan. *Ajax Security.* Boston, Massachusetts: Pearson Education, 2008.
- OWASP AJAX Security Project Webpage. Accessed 13 December 2007 at: http://www.owasp.org/index.php/Category:OWASP_AJAX_Security_Project
- Shah, Shreeraj. "Top 10 Ajax Security Holes and Driving Factors". *Help Net Security*, 10 November 2006. Accessed 13 December 2007 at: http://www.net-security.org/article.php?id=956
- Hayre, Jaswinder S. and Jayasankar Kelath. "Ajax Security Basics". *SecurityFocus*, 19 June 2006. Accessed 13 December 2007 at: http://www.securityfocus.com/infocus/1868
- Stamos, Alex and Zane Lackey. "Attacking AJAX Web Applications: Vulns 2.0 or Web 2.0". Presented at Black Hat USA 2006, 3 August 2006. Accessed 26 January 2008 at: http://www.isecpartners.com/files/iSEC-Attacking_AJAX_Applications.BH2006.pdf
- Di Paola, Stefano and Giorgio Fedon. "Subverting Ajax". *Proceedings of the 23rd Chaos Communication Conference*, Berlin, Germany, 27-30 December 2006. Accessed 26 January 2008 at: http://events.ccc.de/congress/2006/Fahrplan/attachments/1158-Subverting_Ajax.pdf
- Apache Axis Web Service Security page. Accessed 11 December 2007 at: http://ws.apache.org/axis/java/security.html
- Shiflett, Chris. "Cross-Domain Ajax Insecurity". On his PHP and Web Application Security Weblog, 9 August 2006. Accessed 26 January 2008 at: http://shiflett.org/Weblog/2006/aug/cross-domain-ajax-insecurity

C.4.7.3 VBScript

Like JavaScript, VBScript contains no functions that enable disk or network access. However, unlike JavaScript, which "inherits" the benefits of the Java security model, VBScript benefits from no formal security model, because Visual Basic does not provide one. Therefore, as with Visual Basic, security is entirely dependent on the programmer's adherence to secure coding guidelines, and on secure implementation by the browser. Rather than using VBScript, use a client-side scripting language that has more extensive security features built into the language itself.

C.4.7.4 TCL

TCL comprises two parts: the language and the library. The language is a simple text language used to issue commands to interactive programs and includes basic programming capabilities. The TCL library can be embedded in application programs. In an effort to make TCL as small

and simple as possible, however, its designers have created a language that is, from a security standpoint, somewhat limited.

Because TCL was designed to be a scripting language, it has few of the capabilities of a full-blown programming language. It has no arrays or other structures from which to create linked lists. It simulates numbers, which slows TCL programs. As a result, TCL is only suitable for small, simple programs.

In TCL, there is only one data type, *string*. This and its other limitations make it difficult to program anything other than very simple scripts in TCL. With TCL, developers can accidentally create programs that are susceptible to malicious input strings. For example, an attacker can send characters with special meanings to TCL such as:

- embedded spaces,
- double quotes (“.”),
- curly braces ({.}),
- dollar signs ($),
- brackets ([.]).

An attacker might even send input that causes these characters to be created during processing, triggering unexpected and even dangerous behavior in the TCL program.

For all these reasons, TCL should not be used to write programs that perform security functions, such as mediating a security boundary.

There is a promising alternative to generic TCL, *Safe-TCL*, which creates a sandbox in which the TCL program operates. Safe-TCL should be implemented in conjunction with *Safe-TK*, which implements a sandboxed portable GUI for Safe-TCL. Because it contains its own sandbox feature, Safe-TCL may be a good language in which to implement simple mobile code constructs.

C.4.8 Mobile code security

The main security considerations in development of client applications that include mobile code and/or active content are:

- How to establish the trustworthiness of mobile code and active content;
- How to protect integrity of mobile code and active content during distribution and in execution;
- How to protect the browser and its host from untrustworthy mobile code and active content. This includes how to constrain mobile code/active content execution, so that the software internals and data of the browser and the application software are not exposed or disclosed.

The primary countermeasures for addressing these considerations are digital signing of mobile code and signature validation prior to execution, and sandboxing to isolate executing mobile code from the rest of the software application. Sandboxing has been discussed previously. The following discussion focuses on signing and signature validation for mobile code.

C.4.8.1 Digital signing of mobile code

Beyond code signing and validation of code signatures to determine, after the fact, whether openly transported mobile code has been tampered with in transit, secure distribution techniques can help prevent (rather than detect) code tampering, misrouting of mobile code, illicit copying, *etc.*

The predominant commercial model for mobile code distribution identifies dynamically linkable parts of mobile programs by a URI. This model is based on the assumption that all constituent parts of the mobile program will be downloaded to a single location, where they will be verified, linked, possibly dynamically compiled, and ultimately executed at that location.

Aside from the obvious defects of basing a distribution management and versioning scheme on untrustworthy URIs, this approach is neither flexible nor scalable enough to support other modes of mobile-code dissemination and deployment.

If the Web application incorporates the use of mobile code, you will need to address the security of the distribution of that code. The examples below are provided to help stimulate the imagination while undertaking the definition an approach to solving the secure distribution problem for mobile code in Web applications.

C.4.8.2 Mobile code signature validation

Ideally, browsers used to interact with the Web server application will be able to verify that a mobile code executable has been signed by a certificate from a trusted certificate authority *before* allowing that executable to run. Optimally, these browsers will be able to distinguish between certificates to select only those that they will accept as code signers, while rejecting code signed by any other certificate.

Unfortunately, most COTS browsers do not perform mobile code signature processing very well. It is pointless to have the server sign mobile code if the browsers to which that code will be served cannot process the code signature. For this reason, and remembering to "never trust the browser", the safest approach to using mobile code that executes in the browser context is to avoid using any Category 1 and Category 2 mobile code. Instead, limit any client-side mobile code to Category 3, such as JavaScript, VBScript, and PDF (the latter for serving documents).

SUGGESTED RESOURCES

- Jansen, Wayne A., Theodore Winograd, Karen Scarfone. *Guidelines on Active Content and Mobile Code.* NIST Special Publication 800-28, Version 2, March 2008. Accessed 26 March 2008 at: http://csrc.nist.gov/publications/nistpubs/800-28-ver2/SP800-28v2.pdf
- DoD Instruction 8552.01, "Use of Mobile Code Technologies in DoD Information Systems". 23 October 2006. Accessed 19 January 2008 at: http://www.dtic.mil/whs/directives/corres/html/855201.htm

C.4.9 Shell scripting languages

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.

SUGGESTED RESOURCES

- Anley, Chris, John Heasman, Felix Linder, and Gerardo Richarte. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes.* Indianapolis, Indiana: Wiley Publishing, 2007.

C.4.10 Secure language variants and derivatives

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 the 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 widely adopted, and until recently their use was limited almost exclusively to academia and the science and technology community. With the emergence of languages such as MISRA (Motor Industry Software Reliability Association) C and Praxis High Integrity Systems' SPARKAda, however, secure programming language use has expanded into industries in which the imperative for

software safety has expanded to encompass software security concerns. Table C-2 lists the most noteworthy secure programming languages. With the exception of SPARKAda, these are all secure variants on C or C++. Also refer back to the discussion of Ada in C.4.4 for a description of how Ada in particular supports programmer definition of fully-featured secure Ada subsets.

Table C-3. Noteworthy secure language variants and derivatives

Language	Resource
CCured	CCured Documentation Web page. Accessed 21 January 2008 at: http://manju.cs.berkeley.edu/ccured
CYCLONE	CYCLONE Website. Accessed 21 January 2008 at: http://cyclone.thelanguage.org/
Hermes	Hermes publications Webpage. Accessed 21 January 2008 at: http://www.research.ibm.com/people/d/dfb/hermes-publications.html
MISRA C	MISRA C Website. Accessed 21 January 2008 at: http://www.misra-c2.com/
SPARKAda	SPARKAda Webpage. Accessed 21 January 2008 at: http://www.praxis-his.com/sparkada/
Vault	Vault: a programming language for reliable systems. Accessed 21 January 2008 at: http://research.microsoft.com/vault/

C.5. LEVERAGING DESIGN BY CONTRACT™ FOR SOFTWARE SECURITY

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.

An example of a design contract is:

- Each component has preconditions which must be satisfied by the caller.
- One such precondition is: "All data delivered to the called function must be successfully validated before it is used by that called function."
- The intent of this precondition is to allow the called function to forego input validation because it trusts that the input it receives has already been validated.

For a design contract to be enforceable, the assumption on which it is based must hold throughout the component's lifecycle.

Replacement of components (with substitutes or newer versions) is a point at which such a contract is likely to be violated. For this reason:

- Change management must guarantee that all changes of the contract are inherited to all contracts with dependent modules.
- The validating function must "know" what is considered unacceptable input is for all subsequent callable functions.
- New calling functions may need to be changed to incorporate input validation for downstream callable functions.

There are potential pitfalls with this approach, the most obvious being the difficulty in assuring the enforcement of a design contract over time, particularly in a component-based system. Furthermore such contract relationships between functionally unrelated modules are undesirable because they make it difficult (if not impossible) to decouple the components.

Tools supporting use of DbC are now available. These are included among the Suggested Resources below.

SUGGESTED RESOURCES

- Kevin McFarlane's Design by Contract Framework. Accessed 7 July 2008 at: http://www.codeproject.com/KB/cs/designbycontract.aspx
- The Jass Page. Accessed 7 July 2008: http://csd.informatik.uni-oldenburg.de/~jass/
- C4J: Design by Contract for Java. Accessed 7 July 2008 at: http://c4j.sourceforge.net/
- The Java Modeling Language (JML). Accessed 7 July 2008 at: http://www.eecs.ucf.edu/~leavens/JML/index.shtml
- Nice Language. Accessed 7 July 2008 at: http://c2.com/cgi/wiki?NiceLanguage
- Le Traon, Yves, Benoit Baudry, and Jean-Marc Jezequel. "Design by Contract to improve software vigilance". *IEEE Transactions on Software Engineering,* Volume 32 Issue 8 August 2006, pages 571-586.
- Building bug-free O-O software: An introduction to Design by Contract. Accessed 7 July 2008 at: http://archive.eiffel.com/doc/manuals/technology/contract/
- Meyer, Bertrand. "Applying 'Design by Contract'". *IEEE Computer,* Volume 25 Issue 10, October 1992, pages 40-51. Accessed 26 August 2008 at: http://se.ethz.ch/~meyer/publications/computer/contract.pdf
- Jezequel, Jean-Marc and Bertrand Meyer. "Design by Contract: the lessons of Ariane". *IEEE Computer,* January 1997, pages 129-130. Accessed 11 September 2008 at: http://archive.eiffel.com/doc/manuals/technology/contract/ariane/page.html

APPENDIX D: SECURITY CHECKLIST EXCERPTS

The Internet Security Alliance and U.S. Cyber Consequences Unit Cyber-Security Checklist[61] categorizes its entries by what it terms "areas" and "avenues". There are a number of avenues within three areas that provide checklist entries that are relevant for software (as contrasted with system) security testing, and secure SDLC process verification. These checklist entries have been captured in Table D-1 below.

The software assurance-relevant excerpts from the recent revision of the World Bank's Technology Risk Checklist[62] may provide useful in the SDLC phase and activity indicated. These appear in Table D-2 below.

Table D-1. Secure SDLC-relevant sections of the U.S. Cyber Consequences Unit Cyber-Security Checklist

Area	Avenue	Subheading(s)	Applicability to secure software development
Two: Software Access Vulnerabilities	5: Application Privileges	Customizing of Privileges; General Control of Privileges	Useful in defining secure configuration parameters and in performing deployment-time verification of secure environment configuration
	6: Input Validation		Useful in input validation design and implementation, and unit- and system-level test scenarios to verify correctness and effectiveness of input validation
	7: Appropriate Behavior Patterns		Useful in design and implementation of error and exception handling, and in development of penetration test scenarios
Six: Software Supply Vulnerabilities	15: Internal Policies for Software Development	Secure Procedures for Developing New Software	Useful in defining secure development practices for security-enhancing the SDLC process
		Security Features to Build into New Software	Useful in specifying functional security requirements and designing software's security functionality
		Security Testing of New Software	Useful in developing security test plans for software

61 Bumgarner, John and Scott Borg. *The U.S. Cyber Consequences Unit Cyber-Security Check List.* Final Version, 2007. Accessed 28 January 2008 at: http://www.isalliance.org/content/view/144/292

62 Kellerman, Thomas. "Technology Risk Checklist, Version 13.0". Washington, DC: The World Bank, 2008.

Table D-2. Secure SDLC-relevant sections of World Bank Technology Risk Checklist

Checklist Section	Useful in	Specific checks
Patch Management	*Post-Deployment Phase, Patch Generation and Issuance*	When applying a patch to any system vulnerability, do you have a process for verifying the integrity, and testing the proper functioning of the patch?
		Have you verified that the patch will not negatively affect or alter other system configurations?
		Are patches tested on test beds before being released into the network?
		Do you make a backup of your system before applying patches?
		Do you conduct another vulnerability test after you apply a patch?
		Do you keep a log file of any system changes and updates?
		Are patches prioritized?
		Do you disseminate patch update information throughout organization's local systems administrators?
		Do you add timetables to patch potential vulnerabilities?
		Are external partners required to patch all non-critical patches within 30 days?
Patch Management (continued)	*Post-Deployment Phase, Patch Generation and Issuance* (continued)	Are external partners required to patch critical patches to servers and clients within 48 hours?

Checklist Section	Useful in	Specific checks
VI. Active Content Filtering	*Implementation Phase, Development of Input Validation Logic*	Is your system configured to filter (examine content) hostile ActiveX?
		Is your system configured to filter JavaScript?
		Is your system configured to filter Remote Procedure Calls (RPCs)?
		Is your system configured to filter Perimeter-Based Security?
		Is your system configured to filter Berkeley Internet Name Domain?
		Is your system configured to filter Simple Network Management Protocol?
		Is your system configured to filter the Java Virtual Machine vulnerability?
		Does your organization have a standard desktop configuration and software standards?
		Do you employ enterprise level desktop configuration management?
		Do you filter all .exe, .zip, and .doc attachments?
		Do you implement XML filtering and layered security?
B. Active Content Filtering	*Implementation Phase, Development of Input Validation Logic*	Does your system filter the following Simple Network Management Protocol? If so, have you turned off RPC?
Web Application Security	*Distribution/Deployment Phase, Defining Secure Configuration; Testing Phase, Post-Deployment Security Test Planning and Testing*	Do you check the lengths of all input? If greater than the maximum length, do you stop processing and return as failure?
		Do you allow source packets coming from outside to have internal IP addresses. Conversely, do not allow inside packets to go out that do not have valid internal IP source addresses.
		Are user names and passwords sent in plaintext over an insecure channel?
		Do you restrict user access to system-level resources?
		Do you limit session lifetimes?
		Do you encrypt sensitive cookie states?

Checklist Section	Useful in	Specific checks
X. Vulnerability and Penetration Testing	*Testing Phase, System and Post-Deployment Security Penetration Test and Vulnerability Assessment Planning and Testing; Sustainment Phase: Ongoing Vulnerability Assessments*	Are vulnerability tests conducted on a quarterly basis?
		Are penetration tests conducted on a bi-annual basis?
		If they are conducted do they address the following: (a) Describing threats in terms of who, how and when; (b) Establishing into which threat class a threat falls; (c) Determining the consequences on the business operations should a threat be successful; (d) Assessing the impact of the consequences as less serious, serious or exceptionally grave injury; (e) Assigning an exposure rating to each threat, in terms of the relative severity to the business prioritization of the impacts according to the exposure rating.
		Do penetration tests assess both the external and insider threat?
		Do your tests include performing a network survey, port scan, application and code review, router, firewall, intrusion detection system, trusted system, and password cracking?
		Do you employ network sniffers to evaluate network protocols along with the source and destination of various protocols for stealth port scanning and hacking activity?
		Are penetration tests conducted upon hosting provider systems and existing partner systems before connecting them to the organization's network?
		Are vulnerability/penetration testing results shared with all appropriate security and network administrators?
		Do your penetration tests encompass social engineering?
		Are the results acted upon?
		Is there a timetable for acting upon the above results?