SEI SERIES • A CERT® BOOK

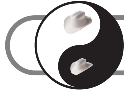
SOFTWARE SECURITY SERIES

Software Security Engineering

A Guide for Project Managers

Julia H. Allen • Sean Barnum
Robert J. Ellison • Gary McGraw
Nancy R. Mead

The SEI Series in Software Engineering



The Addison-Wesley Software Security Series

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

CMM, CMMI, Capability Maturity Model, Capability Maturity Modeling, Carnegie Mellon, CERT, and CERT Coordination Center are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

ATAM; Architecture Tradeoff Analysis Method; CMM Integration; COTS Usage-Risk Evaluation; CURE; EPIC; Evolutionary Process for Integrating COTS Based Systems; Framework for Software Product Line Practice; IDEAL; Interim Profile; OAR; OCTAVE; Operationally Critical Threat, Asset, and Vulnerability Evaluation; Options Analysis for Reengineering; Personal Software Process; PLTP; Product Line Technical Probe; PSP; SCAMPI; SCAMPI Lead Appraiser; SCAMPI Lead Assessor; SCE; SEI; SEPG; Team Software Process; and TSP are service marks of Carnegie Mellon University.

Special permission to reproduce portions of Build Security In, © 2005–2007 by Carnegie Mellon University, in this book is granted by the Software Engineering Institute.

Special permission to reproduce portions of Build Security In, © 2005–2007 by Cigital, Inc., in this book is granted by Cigital, Inc.

Special permission to reprint excerpts from the article "Software Quality at Top Speed," © 1996 Steve McConnell, in this book is granted by Steve McConnell.

The authors and publisher have taken care in the preparation of this book, but make no express or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact: U.S. Corporate and Government Sales, (800) 382-3419, corpsales@pearsontechgroup.com.

For sales outside the United States, please contact: International Sales, international@pearsoned.com.

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data
Software security engineering : a guide for project managers / Julia H. Allen ... [et al.].
p. cm.
Includes bibliographical references and index.
ISBN 978-0-321-50917-8 (pbk. : alk. paper) 1. Computer security. 2. Software engineering. 3. Computer networks—Security measures. I. Allen, Julia H.

QA76.9.A25S654 2008
005.8—dc22

2008007000

Copyright © 2008 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to: Pearson Education, Inc., Rights and Contracts Department, 501 Boylston Street, Suite 900, Boston, MA 02116, Fax: (617) 671-3447.

ISBN-13: 978-0-321-50917-8
ISBN-10: 0-321-50917-X
Text printed in the United States on recycled paper at Courier in Stoughton, Massachusetts.
First printing, April 2008

Foreword

Everybody knows that software is riddled with security flaws. At first blush, this is surprising. We know how to write software in a way that provides a moderately high level of security and robustness. So why don't software developers practice these techniques?

This book deals with two of the myriad answers to this question. The first is the meaning of secure software. In fact, the term "secure software" is a misnomer. Security is a product of software *plus environment*. How a program is used, under what conditions it is used, and what security requirements it must meet determine whether the software is secure. A term like "security-enabled software" captures the idea that the software was designed and written to meet specific security requirements, but in other environments where the assumptions underlying the software—and any implied requirements—do not hold, the software may not be secure. In a way that is easy to understand, this book presents the need for accurate and meaningful security requirements, as well as approaches for developing them. Unlike many books on the subject of secure software, this book does not assume the requirements are given *a priori*, but instead discusses requirements derivation and analysis. Equally important, it describes their validation.

The second answer lies in the roles of the executives, managers, and technical leaders of projects. They must support the introduction of security enhancements in software, as well as robust coding practices (which is really a type of security enhancement). Moreover, they must understand the processes and make allowances for it in their scheduling, budgeting, and staffing plans. This book does an excellent job of laying out the process for the people in these roles, so they can realistically assess its impact. Additionally, the book points out where the state of the art is too new or lacks enough experience to have approaches that are proven to work, or are not generally accepted to work. In those cases, the authors suggest ways to think about the issues in order to develop effective approaches. Thus, executives, managers, and technical leaders can figure out what should work best in their environment.

An additional, and in fact crucial, benefit of designing and implementing security in software from the very beginning of the project is the increase in assurance that the software will meet its requirements. This will greatly reduce the need to patch the software to fix security holes—a process that is itself fraught with security problems, undercuts the reputation of the vendor, and adversely impacts the vendor financially. Loss of credibility, while intangible, has tangible repercussions. Paying the extra cost of developing software correctly from the start reduces the cost of fixing it after it is deployed—and produces a better, more robust, and more secure product.

This book discusses several ways to develop software in such a way that security considerations play a key role in its development. It speaks to executives, to managers at all levels, and to technical leaders, and in that way, it is unique. It also speaks to students and developers, so they can understand the process of developing software with security in mind and find resources to help them do so.

The underlying theme of this book is that the software we all use could be made much better. The information in this book provides a foundation for executives, project managers, and technical leaders to improve the software they create and to improve the quality and security of the software we all use.

Matt Bishop
Davis, California
March 2008

Preface

The Problem Addressed by This Book

Software is ubiquitous. Many of the products, services, and processes that organizations use and offer are highly dependent on software to handle the sensitive and high-value data on which people's privacy, livelihoods, and very lives depend. For instance, national security—and by extension citizens' personal safety—relies on increasingly complex, interconnected, software-intensive information systems that, in many cases, use the Internet or Internet-exposed private networks as their means for communication and transporting data.

This ubiquitous dependence on information technology makes software security a key element of business continuity, disaster recovery, incident response, and national security. Software vulnerabilities can jeopardize intellectual property, consumer trust, business operations and services, and a broad spectrum of critical applications and infrastructures, including everything from process control systems to commercial application products.

The integrity of critical digital assets (systems, networks, applications, and information) depends on the reliability and security of the software that enables and controls those assets. However, business leaders and informed consumers have growing concerns about the scarcity of practitioners with requisite competencies to address software security [Carey 2006]. Specifically, they have doubts about suppliers' capabilities to build and deliver secure software that they can use with confidence and without fear of compromise. Application software is the primary gateway to sensitive information. According to a Deloitte survey of 169 major global financial institutions, titled *2007 Global Security Survey: The Shifting Security Paradigm* [Deloitte 2007], current application software countermeasures are no longer adequate. In the survey, Gartner identifies application security as the number one issue for chief information officers (CIOs).

Selected content in this preface is summarized and excerpted from *Security in the Software Lifecycle: Making Software Development Processes—and Software Produced by Them—More Secure* [Goertzel 2006].

The absence of security discipline in today's software development practices often produces software with exploitable weaknesses. Security-enhanced processes and practices—and the skilled people to manage them and perform them—are required to build software that can be trusted to operate more securely than software being used today.

That said, there is an economic counter-argument, or at least the perception of one: Some business leaders and project managers believe that developing secure software slows the software development process and adds to the cost while not offering any apparent advantage. In many cases, when the decision reduces to "ship now" or "be secure and ship later," "ship now" is almost always the choice made by those who control the money but have no idea of the risks. The opposite side of this argument, including how software security can potentially reduce cost and schedule, is discussed in Chapter 1 (Section 1.6, "The Benefits of Detecting Software Security Defects Early") and Chapter 7 (Section 7.5.3, in the "Knowledge and Expertise" subsection discussing Microsoft's experience with its Security Development Lifecycle) in this book.

Software's Vulnerability to Attack

The number of threats specifically targeting software is increasing, and the majority of network- and system-level attacks now exploit vulnerabilities in application-level software. According to CERT analysts at Carnegie Mellon University,[1] most successful attacks result from targeting and exploiting known, unpatched software vulnerabilities and insecure software configurations, a significant number of which are introduced during software design and development.

These conditions contribute to the increased risks associated with software-enabled capabilities and exacerbate the threat of attack. Given this atmosphere of uncertainty, a broad range of stakeholders need justifiable confidence that the software that enables their core business operations can be trusted to perform as intended.

Why We Wrote This Book: Its Purpose, Goals, and Scope

The Challenge of Software Security Engineering

Software security engineering entails using practices, processes, tools, and techniques to address security issues in every phase of the software

1. CERT (www.cert.org) is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

development life cycle (SDLC). Software that is developed with security in mind is typically more resistant to both intentional attack and unintentional failures. One view of secure software is software that is engineered "so that it continues to function correctly under malicious attack" [McGraw 2006] and is able to recognize, resist, tolerate, and recover from events that intentionally threaten its dependability. Broader views that can overlap with software security (for example, software safety, reliability, and fault tolerance) include the notion of proper functioning in the face of unintentional failures or accidents and inadvertent misuse and abuse, as well as reducing software defects and weaknesses to the greatest extent possible regardless of their cause. This book addresses the narrower view.

The goal of software security engineering is to build better, defect-free software. Software-intensive systems that are constructed using more securely developed software are better able to do the following:

- Continue operating correctly in the presence of most attacks by either *resisting* the exploitation of weaknesses in the software by attackers or *tolerating* the failures that result from such exploits
- Limit the damage resulting from any failures caused by attack-triggered faults that the software was unable to resist or tolerate and recover as quickly as possible from those failures

No single practice offers a universal "silver bullet" for software security. With this caveat in mind, *Software Security Engineering: A Guide for Project Managers* provides software project managers with sound practices that they can evaluate and selectively adopt to help reshape their own development practices. The objective is to increase the security and dependability of the software produced by these practices, both during its development and during its operation.

What Readers Can Expect

Readers will increase their awareness and understanding of the security issues in the design and development of software. The book's content will help readers recognize how software development practices can either contribute to or detract from the security of software.

The book (and material referenced on the Build Security In Web site described later in this preface) will enable readers to identify and compare potential new practices that can be adapted to augment a

project's current software development practices, thereby greatly increasing the likelihood of producing more secure software and meeting specified security requirements. As one example, assurance cases can be used to assert and specify desired security properties, including the extent to which security practices have been successful in satisfying security requirements. Assurance cases are discussed in Chapter 2 (Section 2.4, "How to Assert and Specify Desired Security Properties").

Software developed and assembled using the practices described in this book should contain significantly fewer exploitable weaknesses. Such software can then be relied on to more capably resist or tolerate and recover from attacks and, therefore, to function more securely in an operational environment. Project managers responsible for ensuring that software and systems adequately address their security requirements throughout the SDLC should review, select, and tailor guidance from this book, the Build Security In Web site, and the sources cited throughout this book as part of their normal project management activities.

The five key take-away messages for readers of this book are as follows:

1. Software security is about more than eliminating vulnerabilities and conducting penetration tests. Project managers need to take a systematic approach to incorporate the sound practices discussed in this book into their development processes (all chapters).
2. Network security mechanisms and IT infrastructure security services do not sufficiently protect application software from security risks (Chapters 1 and 2).
3. Software security initiatives should follow a risk management approach to identify priorities and determine what is "good enough," while understanding that software security risks will inevitably change throughout the SDLC (Chapters 1, 4, and 7).
4. Developing secure software depends on understanding the operational context in which it will be used (Chapter 6).
5. Project managers and software engineers need to learn to think like an attacker to address the range of things that software should *not* do and identify how software can better resist, tolerate, and recover when under attack (Chapters 2, 3, 4, and 5).

Who Should Read This Book

Software Security Engineering: A Guide for Project Managers is primarily intended for project managers who are responsible for software development and the development of software-intensive systems. Lead requirements analysts, experienced software and security architects and designers, system integrators, and their managers should also find this book useful. It provides guidance for those involved in the management of secure, software-intensive systems, either developed from scratch or through the assembly, integration, and evolution of acquired or reused software.

This book will help readers understand the security issues associated with the engineering of software and should help them identify practices that can be used to manage and develop software that is better able to withstand the threats to which it is increasingly subjected. It presumes that readers are familiar with good general systems and software engineering management methods, practices, and technologies.

How This Book Is Organized

This book is organized into two introductory chapters, four technical chapters, a chapter that describes governance and management considerations, and a concluding chapter on how to get started.

Chapter 1, Why Is Security a Software Issue?, identifies threats that target most software and the shortcomings of the software development process that can render software vulnerable to those threats. It describes the benefits of detecting software security defects early in the SDLC, including the current state of the practice for making the business case for software security. It closes by introducing some pragmatic solutions that are further elaborated in the chapters that follow.

Chapter 2, What Makes Software Secure?, examines the core and influential properties of software that make it secure and the defensive and attacker perspectives in addressing those properties, and discusses how desirable traits of software can contribute to its security. The chapter introduces and defines the key resources of attack patterns and assurance cases and explains how to use them throughout the SDLC.

Chapter 3, Requirements Engineering for Secure Software, describes practices for security requirements engineering, including processes

that are specific to eliciting, specifying, analyzing, and validating security requirements. This chapter also explores the key practice of misuse/abuse cases.

Chapter 4, Secure Software Architecture and Design, presents the practice of architectural and risk analysis for reviewing, assessing, and validating the specification, architecture, and design of a software system with respect to software security, and reliability.

Chapter 5, Considerations for Secure Coding and Testing, summarizes key practices for performing code analysis to uncover errors in and improve the quality of source code, as well as practices for security testing, white-box testing, black-box testing, and penetration testing. Along the way, this chapter references recently published works on secure coding and testing for further details.

Chapter 6, Security and Complexity: System Assembly Challenges, describes the challenges and practices inherent in the design, assembly, integration, and evolution of trustworthy systems and systems of systems. It provides guidelines for project managers to consider, recognizing that most new or updated software components are typically integrated into an existing operational environment.

Chapter 7, Governance, and Managing for More Secure Software, describes how to motivate business leaders to treat software security as a governance and management concern. It includes actionable practices for risk management and project management and for establishing an enterprise security framework.

Chapter 8, Getting Started, summarizes all of the recommended practices discussed in the book and provides several aids for determining which practices are most relevant and for whom, and where to start.

The book closes with a comprehensive bibliography and glossary.

Notes to the Reader

Navigating the Book's Content

As an aid to the reader, we have added descriptive icons that mark the book's sections and key practices in two practical ways:

- Identifying the content's relative "maturity of practice":

  L1 The content provides guidance for how to think about a topic for which there is no proven or widely accepted approach. The

intent of the description is to raise awareness and aid the reader in thinking about the problem and candidate solutions. The content may also describe promising research results that may have been demonstrated in a constrained setting.

L2 The content describes practices that are in early (pilot) use and are demonstrating some successful results.

L3 The content describes practices that are in limited use in industry or government organizations, perhaps for a particular market sector.

L4 The content describes practices that have been successfully deployed and are in widespread use. Readers can start using these practices today with confidence. Experience reports and case studies are typically available.

- Identifying the designated audiences for which each chapter section or practice is most relevant:
  - E Executive and senior managers
  - M Project and mid-level managers
  - L Technical leaders, engineering managers, first-line managers, and supervisors

As the audience icons in the chapters show, we urge executive and senior managers to read all of Chapters 1 and 8, plus the following sections in other chapters: 2.1, 2.2, 2.5, 3.1, 3.7, 4.1, 5.1, 5.6, 6.1, 6.6, 7.1, 7.3, 7.4, 7.6, and 7.7.

Project and mid-level managers should be sure to read all of Chapters 1, 2, 4, 5, 6, 7, and 8, plus these sections in Chapter 3: 3.1, 3.3, and 3.7.

Technical leaders, engineering managers, first-line managers, and supervisors will find useful information and guidance throughout the entire book.

Build Security In: A Key Resource

Since 2004, the U.S. Department of Homeland Security Software Assurance Program has sponsored development of the Build Security In (BSI) Web site (https://buildsecurityin.us-cert.gov/), which was one of the significant resources used in writing this book. BSI content is based on the principle that software security is fundamentally a software engineering problem and must be managed in a systematic way throughout the SDLC.

BSI contains and links to a broad range of information about sound practices, tools, guidelines, rules, principles, and other knowledge to help project managers deploy software security practices and build secure and reliable software. Contributing authors to this book and the articles appearing on the BSI Web site include senior staff from the Carnegie Mellon Software Engineering Institute (SEI) and Cigital, Inc., as well as other experienced software and security professionals.

Several sections in the book were originally published as articles in *IEEE Security & Privacy* magazine and are reprinted here with the permission of IEEE Computer Society Press. Where an article occurs in the book, a statement such as the following appears in a footnote:

> This section was originally published as an article in *IEEE Security & Privacy* [citation]. It is reprinted here with permission from the publisher.

These articles are also available on the BSI Web site.

Articles on BSI are referenced throughout this book. Readers can consult BSI for additional details, book errata, and ongoing research results.

Start the Journey

A number of excellent books address secure systems and software engineering. *Software Security Engineering: A Guide for Project Managers* offers an engineering perspective that has been sorely needed in the software security community. It puts the entire SDLC in the context of an integrated set of sound software security engineering practices.

As part of its comprehensive coverage, this book captures both standard and emerging software security practices and explains why they are needed to develop more security-responsive and robust systems. The book is packed with reasons for taking action early and revisiting these actions frequently throughout the SDLC.

This is not a book for the faint of heart or the neophyte software project manager who is confronting software security for the first time. Readers need to understand the SDLC and the processes in use within their organizations to comprehend the implications of the various techniques presented and to choose among the recommended practices to determine the best fit for any given project.

Other books are available that discuss each phase of secure software engineering. Few, however, cover all of the SDLC phases in as concise and usable a format as we have attempted to do here. Enjoy the journey!

Acknowledgments

We are pleased to acknowledge the support of many people who helped us through the book development process. Our organizations, the CERT Program at the Software Engineering Institute (SEI) and Cigital, Inc., encouraged our authorship of the book and provided release time as well as other support to make it possible. Pamela Curtis, our SEI technical editor, diligently read and reread each word of the entire manuscript and provided many valuable suggestions for improvement, as well as helping with packaging questions and supervising development of figures for the book. Jan Vargas provided SEI management support, tracked schedules and action items, and helped with meeting agendas and management. In the early stages of the process, Petra Dilone provided SEI administrative support as well as configuration management for the various chapters and iterations of the manuscript.

We also appreciate the encouragement of Joe Jarzombek, the sponsor of the Department of Homeland Security Build Security In (BSI) Web site. The Build Security In Web site content is a key resource for this book.

Much of the material in this book is based on articles published with other authors on the BSI Web site and elsewhere. We greatly appreciated the opportunity to collaborate with these authors, and their names are listed in the individual sections that they contributed to, directly or indirectly.

We had many reviewers, whose input was extremely valuable and led to many improvements in the book. Internal reviewers included Carol Woody and Robert Ferguson of the SEI. We also appreciate the inputs and thoughtful comments of the Addison-Wesley reviewers: Chris Cleeland, Jeremy Epstein, Ronda R. Henning, Jeffrey A. Ingalsbe, Ron Lichty, Gabor Liptak, Donald Reifer, and David Strom. We would like to give special recognition to Steve Riley, one of the Addison-Wesley reviewers who reviewed our initial proposal and all iterations of the manuscript.

We would like to recognize the encouragement and support of our contacts at Addison-Wesley. These include Peter Gordon, publishing

partner; Kim Boedigheimer, editorial assistant; Julie Nahil, full-service production manager; and Jill Hobbs, freelance copyeditor. We also appreciate the efforts of the Addison-Wesley and SEI artists and designers who assisted with the cover design, layout, and figures.

Chapter 1

Why Is Security a Software Issue?

1.1 Introduction

Software is everywhere. It runs your car. It controls your cell phone. It's how you access your bank's financial services; how you receive electricity, water, and natural gas; and how you fly from coast to coast [McGraw 2006]. Whether we recognize it or not, we all rely on complex, interconnected, software-intensive information systems that use the Internet as their means for communicating and transporting information.

Building, deploying, operating, and using software that has not been developed with security in mind can be high risk—like walking a high wire without a net (Figure 1–1). The degree of risk can be compared to the distance you can fall and the potential impact (no pun intended).

This chapter discusses why security is increasingly a software problem. It defines the dimensions of software assurance and software security. It identifies threats that target most software and the shortcomings of the

Selected content in this chapter is summarized and excerpted from *Security in the Software Lifecycle: Making Software Development Processes—and Software Produced by Them—More Secure* [Goertzel 2006]. An earlier version of this material appeared in [Allen 2007].

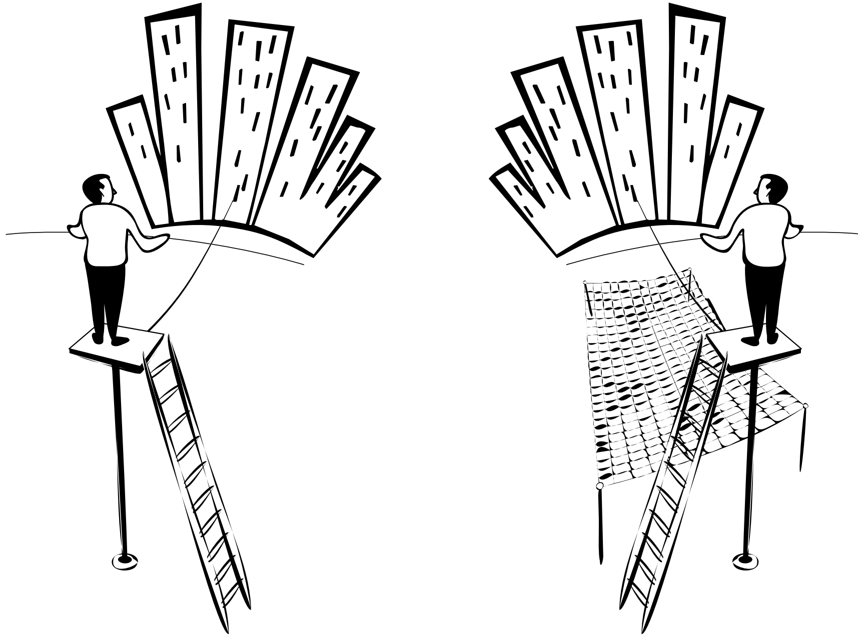


Figure 1–1: *Developing software without security in mind is like walking a high wire without a net*

software development process that can render software vulnerable to those threats. It closes by introducing some pragmatic solutions that are expanded in the chapters to follow. This entire chapter is relevant for executives (E), project managers (M), and technical leaders (L).

1.2 The Problem

Organizations increasingly store, process, and transmit their most sensitive information using software-intensive systems that are directly connected to the Internet. Private citizens' financial transactions are exposed via the Internet by software used to shop, bank, pay taxes, buy insurance, invest, register children for school, and join various organizations and social networks. The increased exposure that comes with global connectivity has made sensitive information and the software systems that handle it more vulnerable to unintentional and unauthorized use. In short, software-intensive systems

and other software-enabled capabilities have provided more open, widespread access to sensitive information—including personal identities—than ever before.

Concurrently, the era of information warfare [Denning 1998], cyberterrorism, and computer crime is well under way. Terrorists, organized crime, and other criminals are targeting the entire gamut of software-intensive systems and, through human ingenuity gone awry, are being successful at gaining entry to these systems. Most such systems are not attack resistant or attack resilient enough to withstand them.

In a report to the U.S. president titled *Cyber Security: A Crisis of Prioritization* [PITAC 2005], the President's Information Technology Advisory Committee summed up the problem of nonsecure software as follows:

> 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.

Software defects with security ramifications—including coding bugs such as buffer overflows and design flaws such as inconsistent error handling—are ubiquitous. Malicious intruders, and the malicious code and botnets[1] they use to obtain unauthorized access and launch attacks, can compromise systems by taking advantage of software defects. Internet-enabled software applications are a commonly exploited target, with software's increasing complexity and extensibility making software security even more challenging [Hoglund 2004].

The security of computer systems and networks has become increasingly limited by the quality and security of their software. Security defects and vulnerabilities in software are commonplace and can pose serious risks when exploited by malicious attacks. Over the past six years, this problem has grown significantly. Figure 1–2 shows the number of vulnerabilities reported to CERT from 1997 through 2006. Given this trend, "[T]here is a clear and pressing need to change the

1. http://en.wikipedia.org/wiki/Botnet

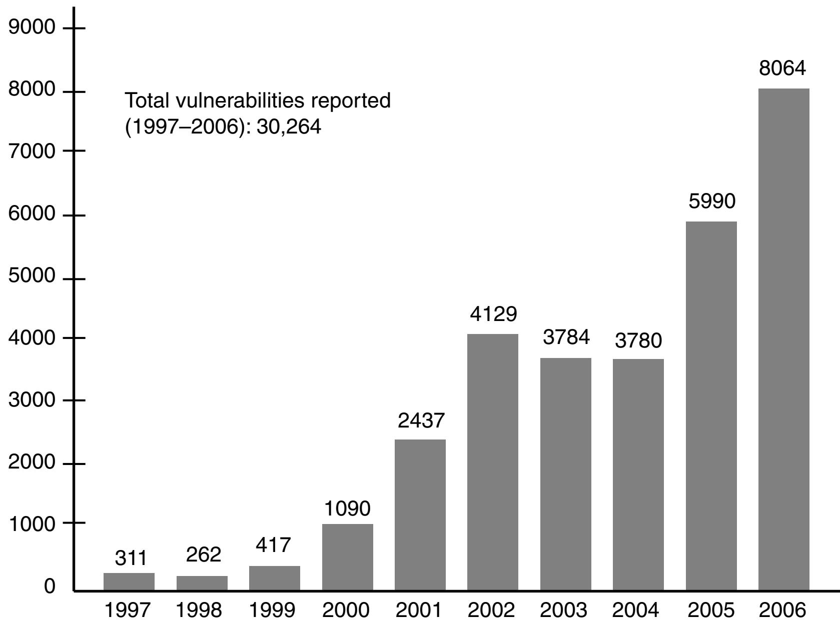


Figure 1–2: *Vulnerabilities reported to CERT*

way we (project managers and software engineers) approach computer security and to develop a disciplined approach to software security" [McGraw 2006].

In Deloitte's *2007 Global Security Survey*, 87 percent of survey respondents cited poor software development quality as a top threat in the next 12 months. "Application security means ensuring that there is secure code, integrated at the development stage, to prevent potential vulnerabilities and that steps such as vulnerability testing, application scanning, and penetration testing are part of an organization's software development life cycle [SDLC]" [Deloitte 2007].

The growing Internet connectivity of computers and networks and the corresponding user dependence on network-enabled services (such as email and Web-based transactions) have increased the number and sophistication of attack methods, as well as the ease with which an attack can be launched. This trend puts software at greater risk. Another risk area affecting software security is the degree to which systems accept updates and extensions for evolving capabilities. Extensible systems are attractive because they provide for the addition of new features and services, but each new extension adds new

capabilities, new interfaces, and thus new risks. A final software security risk area is the unbridled growth in the size and complexity of software systems (such as the Microsoft Windows operating system). The unfortunate reality is that in general more lines of code produce more bugs and vulnerabilities [McGraw 2006].

1.2.1 System Complexity: The Context within Which Software Lives

Building a trustworthy software system can no longer be predicated on constructing and assembling discrete, isolated pieces that address static requirements within planned cost and schedule. Each new or updated software component joins an existing operational environment and must merge with that legacy to form an operational whole. Bolting new systems onto old systems and Web-enabling old systems creates systems of systems that are fraught with vulnerabilities. With the expanding scope and scale of systems, project managers need to reconsider a number of development assumptions that are generally applied to software security:

- Instead of centralized control, which was the norm for large stand-alone systems, project managers have to consider multiple and often independent control points for systems and systems of systems.
- Increased integration among systems has reduced the capability to make wide-scale changes quickly. In addition, for independently managed systems, upgrades are not necessarily synchronized. Project managers need to maintain operational capabilities with appropriate security as services are upgraded and new services are added.
- With the integration among independently developed and operated systems, project managers have to contend with a heterogeneous collection of components, multiple implementations of common interfaces, and inconsistencies among security policies.
- With the mismatches and errors introduced by independently developed and managed systems, failure in some form is more likely to be the norm than the exception and so further complicates meeting security requirements.

There are no known solutions for ensuring a specified level or degree of software security for complex systems and systems of systems,

assuming these could even be defined. This said, Chapter 6, Security and Complexity: System Assembly Challenges, elaborates on these points and provides useful guidelines for project managers to consider in addressing the implications.

1.3 Software Assurance and Software Security

The increasing dependence on software to get critical jobs done means that software's value no longer lies solely in its ability to enhance or sustain productivity and efficiency. Instead, its value also derives from its ability to continue operating dependably even in the face of events that threaten it. The ability to trust that software will remain dependable under all circumstances, with a justified level of confidence, is the objective of software assurance.

Software assurance has become critical because dramatic increases in business and mission risks are now known to be attributable to exploitable software [DHS 2003]. The growing extent of the resulting risk exposure is rarely understood, as evidenced by these facts:

- Software is the weakest link in the successful execution of interdependent systems and software applications.
- Software size and complexity obscure intent and preclude exhaustive testing.
- Outsourcing and the use of unvetted software supply-chain components increase risk exposure.
- The sophistication and increasingly more stealthy nature of attacks facilitates exploitation.
- Reuse of legacy software with other applications introduces unintended consequences, increasing the number of vulnerable targets.
- Business leaders are unwilling to make risk-appropriate investments in software security.

According to the U.S. Committee on National Security Systems' "National Information Assurance (IA) Glossary" [CNSS 2006], software assurance is

> the level of confidence that software is free from vulnerabilities, either intentionally designed into the software or accidentally

inserted at any time during its life cycle, and that the software functions in the intended manner.

Software assurance includes the disciplines of software reliability[2] (also known as software fault tolerance), software safety,[3] and software security. The focus of *Software Security Engineering: A Guide for Project Managers* is on the third of these, software security, which is the ability of software to resist, tolerate, and recover from events that intentionally threaten its dependability. The main objective of software security is to build more-robust, higher-quality, defect-free software that continues to function correctly under malicious attack [McGraw 2006].

Software security matters because so many critical functions are completely dependent on software. This makes software a very high-value target for attackers, whose motives may be malicious, criminal, adversarial, competitive, or terrorist in nature. What makes it so easy for attackers to target software is the virtually guaranteed presence of known vulnerabilities with known attack methods, which can be exploited to violate one or more of the software's security properties or to force the software into an insecure state. Secure software remains dependable (i.e., correct and predictable) despite intentional efforts to compromise that dependability.

The objective of software security is to field software-based systems that satisfy the following criteria:

- The system is as vulnerability and defect free as possible.
- The system limits the damage resulting from any failures caused by attack-triggered faults, ensuring that the effects of any attack are not propagated, and it recovers as quickly as possible from those failures.
- The system continues operating correctly in the presence of most attacks by either *resisting* the exploitation of weaknesses in the software by the attacker or *tolerating* the failures that result from such exploits.

---

2. Software reliability means 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. Sources for this definition can be found in [Goertzel 2006], appendix A.1.

3. Software safety means the persistence of dependability in the face of accidents or mishaps—that is, unplanned events that result in death, injury, illness, damage to or loss of property, or environmental harm. Sources for this definition can be found in [Goertzel 2006], appendix A.1.

Software that has been developed with security in mind generally reflects the following properties throughout its development life cycle:

- *Predictable execution.* There is justifiable confidence that the software, when executed, functions as intended. The ability of malicious input to alter the execution or outcome in a way favorable to the attacker is significantly reduced or eliminated.
- *Trustworthiness.* The number of exploitable vulnerabilities is intentionally minimized to the greatest extent possible. The goal is no exploitable vulnerabilities.
- *Conformance.* Planned, systematic, and multidisciplinary activities ensure that software components, products, and systems conform to requirements and applicable standards and procedures for specified uses.

These objectives and properties must be interpreted and constrained based on the practical realities that you face, such as what constitutes an adequate level of security, what is most critical to address, and which actions fit within the project's cost and schedule. These are risk management decisions.

In addition to predictable execution, trustworthiness, and conformance, secure software and systems should be as attack resistant, attack tolerant, and attack resilient as possible. To ensure that these criteria are satisfied, software engineers should design software components and systems to recognize both legitimate inputs and known attack patterns in the data or signals they receive from external entities (humans or processes) and reflect this recognition in the developed software to the extent possible and practical.

To achieve attack resilience, a software system should be able to recover from failures that result from successful attacks by resuming operation at or above some predefined minimum acceptable level of service in a timely manner. The system must eventually recover full service at the specified level of performance. These qualities and properties, as well as attack patterns, are described in more detail in Chapter 2, What Makes Software Secure?

1.3.1 The Role of Processes and Practices in Software Security

A number of factors influence how likely software is to be secure. For instance, software vulnerabilities can originate in the processes

and practices used in its creation. These sources include the decisions made by software engineers, the flaws they introduce in specification and design, and the faults and other defects they include in developed code, inadvertently or intentionally. Other factors may include the choice of programming languages and development tools used to develop the software, and the configuration and behavior of software components in their development and operational environments. It is increasingly observed, however, that *the most critical difference between secure software and insecure software lies in the nature of the processes and practices used to specify, design, and develop the software* [Goertzel 2006].

The return on investment when security analysis and secure engineering practices are introduced early in the development cycle ranges from 12 percent to 21 percent, with the highest rate of return occurring when the analysis is performed during application design [Berinato 2002; Soo Hoo 2001]. This return on investment occurs because there are fewer security defects in the released product and hence reduced labor costs for fixing defects that are discovered later.

A project that adopts a security-enhanced software development process is adopting a set of practices (such as those described in this book's chapters) that initially should reduce the number of exploitable faults and weaknesses. Over time, as these practices become more codified, they should decrease the likelihood that such vulnerabilities are introduced into the software in the first place. More and more, research results and real-world experiences indicate that *correcting potential vulnerabilities as early as possible in the software development life cycle, mainly through the adoption of security-enhanced processes and practices, is far more cost-effective* than the currently pervasive approach of developing and releasing frequent patches to operational software [Goertzel 2006].

1.4 Threats to Software Security

In information security, the threat—the source of danger—is often a person intending to do harm, using one or more malicious software agents. Software is subject to two general categories of threats:

- *Threats during development* (mainly insider threats). A software engineer can sabotage the software at any point in its development

life cycle through intentional exclusions from, inclusions in, or modifications of the requirements specification, the threat models, the design documents, the source code, the assembly and integration framework, the test cases and test results, or the installation and configuration instructions and tools. The secure development practices described in this book are, in part, designed to help reduce the exposure of software to insider threats during its development process. For more information on this aspect, see "Insider Threats in the SDLC" [Cappelli 2006].

- *Threats during operation* (both insider and external threats). Any software system that runs on a network-connected platform is likely to have its vulnerabilities exposed to attackers during its operation. Attacks may take advantage of publicly known but unpatched vulnerabilities, leading to memory corruption, execution of arbitrary exploit scripts, remote code execution, and buffer overflows. Software flaws can be exploited to install spyware, adware, and other malware on users' systems that can lie dormant until it is triggered to execute.[4]

Weaknesses that are most likely to be targeted are those found in the software components' external interfaces, because those interfaces provide the attacker with a direct communication path to the software's vulnerabilities. A number of well-known attacks target software that incorporates interfaces, protocols, design features, or development faults that are well understood and widely publicized as harboring inherent weaknesses. That software includes Web applications (including browser and server components), Web services, database management systems, and operating systems. Misuse (or abuse) cases can help project managers and software engineers see their software from the perspective of an attacker by anticipating and defining unexpected or abnormal behavior through which a software feature could be unintentionally misused or intentionally abused [Hope 2004]. (See Section 3.2.)

Today, most project and IT managers responsible for system operation respond to the increasing number of Internet-based attacks by relying on operational controls at the operating system, network, and database or Web server levels while failing to directly address the insecurity

4. See the Common Weakness Enumeration [CWE 2007], for additional examples.

of the application-level software that is being compromised. This approach has two critical shortcomings:

1. The security of the application depends completely on the robustness of operational protections that surround it.
2. Many of the software-based protection mechanisms (controls) can easily be misconfigured or misapplied. Also, they are as likely to contain exploitable vulnerabilities as the application software they are (supposedly) protecting.

The wide publicity about the literally thousands of successful attacks on software accessible from the Internet has merely made the attacker's job easier. Attackers can study numerous reports of security vulnerabilities in a wide range of commercial and open-source software programs and access publicly available exploit scripts. More experienced attackers often develop (and share) sophisticated, targeted attacks that exploit specific vulnerabilities. In addition, the nature of the risks is changing more rapidly than the software can be adapted to counteract those risks, regardless of the software development process and practices used. To be 100 percent effective, defenders must anticipate *all* possible vulnerabilities, while attackers need find only *one* to carry out their attack.

1.5 Sources of Software Insecurity

Most commercial and open-source applications, middleware systems, and operating systems are extremely large and complex. In normal execution, these systems can transition through a vast number of different states. These characteristics make it particularly difficult to develop and operate software that is consistently correct, let alone consistently secure. The unavoidable presence of security threats and risks means that project managers and software engineers need to pay attention to software security even if explicit requirements for it have not been captured in the software's specification.

A large percentage of security weaknesses in software could be avoided if project managers and software engineers were routinely trained in how to address those weaknesses systematically and consistently. Unfortunately, these personnel are seldom taught how to design and develop secure applications and conduct quality assurance

to test for insecure coding errors and the use of poor development techniques. They do not generally understand which practices are effective in recognizing and removing faults and defects or in handling vulnerabilities when software is exploited by attackers. They are often unfamiliar with the security implications of certain software requirements (or their absence). Likewise, they rarely learn about the security implications of how software is architected, designed, developed, deployed, and operated. The absence of this knowledge means that security requirements are likely to be inadequate and that the resulting software is likely to deviate from specified (and unspecified) security requirements. In addition, this lack of knowledge prevents the manager and engineer from recognizing and understanding how mistakes can manifest as exploitable weaknesses and vulnerabilities in the software when it becomes operational.

Software—especially networked, application-level software—is most often compromised by exploiting weaknesses that result from the following sources:

- Complexities, inadequacies, and/or changes in the software's processing model (e.g., a Web- or service-oriented architecture model).
- Incorrect assumptions by the engineer, including assumptions about the capabilities, outputs, and behavioral states of the software's execution environment or about expected inputs from external entities (users, software processes).
- Flawed specification or design, or defective implementation of
  - The software's interfaces with external entities. Development mistakes of this type include inadequate (or nonexistent) input validation, error handling, and exception handling.
  - The components of the software's execution environment (from middleware-level and operating-system-level to firmware- and hardware-level components).
- Unintended interactions between software components, including those provided by a third party.

Mistakes are unavoidable. Even if they are avoided during requirements engineering and design (e.g., through the use of formal methods) and development (e.g., through comprehensive code reviews and extensive testing), vulnerabilities may still be introduced into software during its assembly, integration, deployment, and operation. No matter how faithfully a security-enhanced life cycle is followed, as long as

software continues to grow in size and complexity, some number of exploitable faults and other weaknesses are sure to exist.

In addition to the issues identified here, Chapter 2, What Makes Software Secure?, discusses a range of principles and practices, the absence of which contribute to software insecurity.

1.6 The Benefits of Detecting Software Security Defects Early[5]

Limited data is available that discusses the return on investment (ROI) of reducing security flaws in source code (refer to Section 1.6.1 for more on this subject). Nevertheless, a number of studies have shown that significant cost benefits are realized through improvements to reduce software defects (including security flaws) throughout the SDLC [Goldenson 2003]. The general software quality case is made in this section, including reasonable arguments for extending this case to include software security defects.

Proactively tackling software security is often under-budgeted and dismissed as a luxury. In an attempt to shorten development schedules or decrease costs, software project managers often reduce the time spent on secure software practices during requirements analysis and design. In addition, they often try to compress the testing schedule or reduce the level of effort. Skimping on software quality[6] is one of the worst decisions an organization that wants to maximize development speed can make; higher quality (in the form of lower defect rates) and reduced development time go hand in hand. Figure 1–3 illustrates the relationship between defect rate and development time.

Projects that achieve lower defect rates typically have shorter schedules. But many organizations currently develop software with defect levels that result in longer schedules than necessary. In the 1970s,

5. This material is extracted and adapted from a more extensive article by Steven Lavenhar of Cigital, Inc. [BSI 18]. That article should be consulted for more details and examples. In addition, this article has been adapted with permission from "Software Quality at Top Speed" by Steve McConnell. For the original article, see [McConnell 1996]. While some of the sources cited in this section may seem dated, the problems and trends described persist today.

6. A similar argument could be made for skimping on software security if the schedule and resources under consideration include software production and operations, when security patches are typically applied.

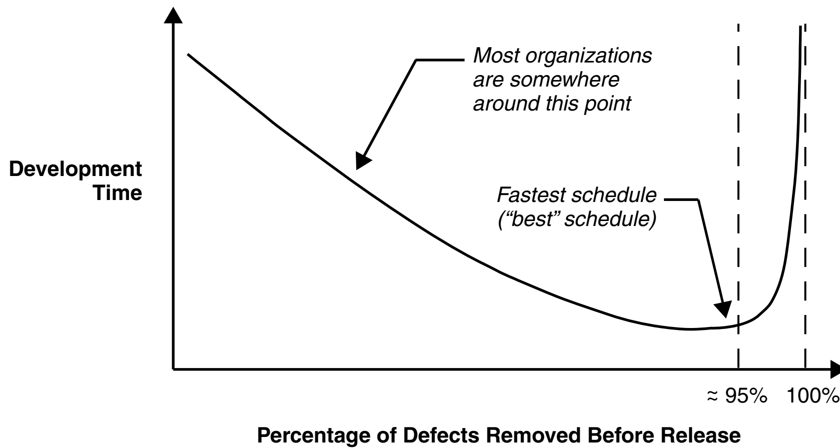


Figure 1–3: *Relationship between defect rate and development time*

studies performed by IBM demonstrated that software products with lower defect counts also had shorter development schedules [Jones 1991]. After surveying more than 4000 software projects, Capers Jones [1994] reported that poor quality was one of the most common reasons for schedule overruns. He also reported that poor quality was a significant factor in approximately 50 percent of all canceled projects. A Software Engineering Institute survey found that more than 60 percent of organizations assessed suffered from inadequate quality assurance [Kitson 1993]. On the curve in Figure 1–3, the organizations that experienced higher numbers of defects are to the left of the "95 percent defect removal" line.

The "95 percent defect removal" line is significant because that level of prerelease defect removal appears to be the point at which projects achieve the shortest schedules for the least effort and with the highest levels of user satisfaction [Jones 1991]. If more than 5 percent of defects are found after a product has been released, then the product is vulnerable to the problems associated with low quality, and the organization takes longer to develop its software than necessary. Projects that are completed with undue haste are particularly vulnerable to shortchanging quality assurance at the individual developer level. Any developer who has been pushed to satisfy a specific deadline or ship a product quickly knows how much pressure there can be to cut corners because "we're only three weeks from the deadline." As many as four times the average number of defects are reported for released software

products that were developed under excessive schedule pressure. Developers participating in projects that are in schedule trouble often become obsessed with working harder rather than working smarter, which gets them into even deeper schedule trouble.

One aspect of quality assurance that is particularly relevant during rapid development is the presence of error-prone modules—that is, modules that are responsible for a disproportionate number of defects. Barry Boehm reported that 20 percent of the modules in a program are typically responsible for 80 percent of the errors [Boehm 1987]. On its IMS project, IBM found that 57 percent of the errors occurred in 7 percent of the modules [Jones 1991]. Modules with such high defect rates are more expensive and time-consuming to deliver than less error-prone modules. Normal modules cost about $500 to $1000 per function point to develop, whereas error-prone modules cost about $2000 to $4000 per function point to develop [Jones 1994]. Error-prone modules tend to be more complex, less structured, and significantly larger than other modules. They often are developed under excessive schedule pressure and are not fully tested. If development speed is important, then identification and redesign of error-prone modules should be a high priority.

If an organization can prevent defects or detect and remove them early, it can realize significant cost and schedule benefits. Studies have found that reworking defective requirements, design, and code typically accounts for 40 to 50 percent of the total cost of software development [Jones 1986b]. As a rule of thumb, every hour an organization spends on defect prevention reduces repair time for a system in production by three to ten hours. In the worst case, reworking a software requirements problem once the software is in operation typically costs 50 to 200 times what it would take to rework the same problem during the requirements phase [Boehm 1988]. It is easy to understand why this phenomenon occurs. For example, a one-sentence requirement could expand into 5 pages of design diagrams, then into 500 lines of code, then into 15 pages of user documentation and a few dozen test cases. It is cheaper to correct an error in that one-sentence requirement at the time requirements are specified (assuming the error can be identified and corrected) than it is after design, code, user documentation, and test cases have been written. Figure 1–4 illustrates that the longer defects persist, the more expensive they are to correct.

The savings potential from early defect detection is significant: Approximately 60 percent of all defects usually exist by design time

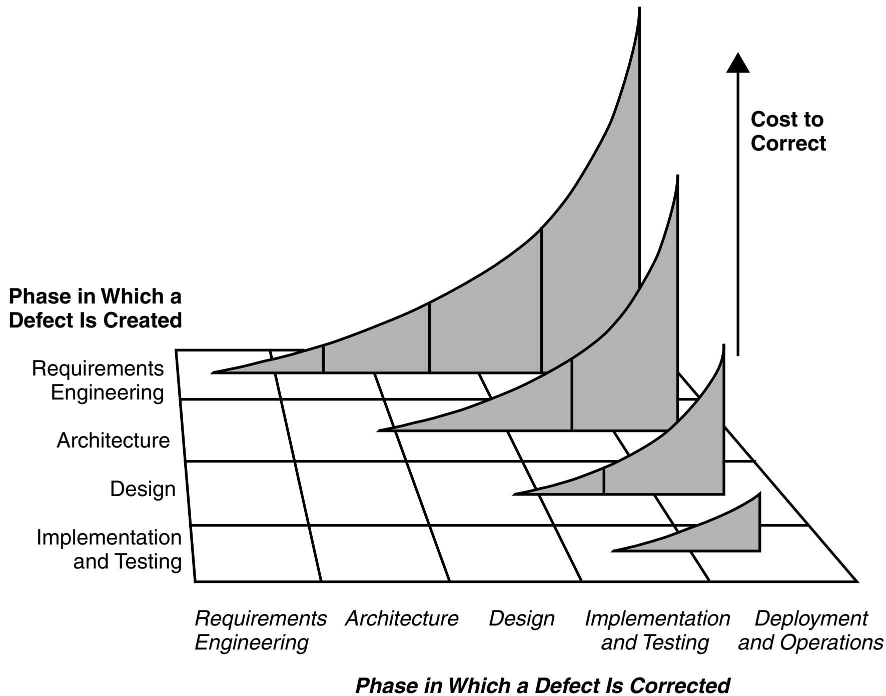


Figure 1–4: *Cost of correcting defects by life-cycle phase*

[Gilb 1988]. A decision early in a project to exclude defect detection amounts to a decision to postpone defect detection and correction until later in the project, when defects become much more expensive and time-consuming to address. That is not a rational decision when time and development dollars are at a premium. According to software quality assurance empirical research, $1 required to resolve an issue during the design phase grows into $60 to $100 required to resolve the same issue after the application has shipped [Soo Hoo 2001].

When a software product has too many defects, including security flaws, vulnerabilities, and bugs, software engineers can end up spending more time correcting these problems than they spent on developing the software in the first place. Project managers can achieve the shortest possible schedules with a higher-quality product by addressing security throughout the SDLC, especially during the early phases, to increase the likelihood that software is more secure the first time.

1.6.1 Making the Business Case for Software Security: Current State[7]

As software project managers and developers, we know that when we want to introduce new approaches in our development processes, we have to make a cost–benefit argument to executive management to convince them that this move offers a business or strategic return on investment. Executives are not interested in investing in new technical approaches simply because they are innovative or exciting. For profit-making organizations, we need to make a case that demonstrates we will improve market share, profit, or other business elements. For other types of organizations, we need to show that we will improve our software in a way that is important—in a way that adds to the organization's prestige, that ensures the safety of troops in the battlefield, and so on.

In the area of software security, we have started to see some evidence of successful ROI or economic arguments for security administrative operations, such as maintaining current levels of patches, establishing organizational entities such as computer security incident response teams (CSIRTs) to support security investment, and so on [Blum 2006, Gordon 2006, Huang 2006, Nagaratnam 2005]. In their article "Tangible ROI through Secure Software Engineering," Kevin Soo Hoo and his colleagues at @stake state the following:

> Findings indicate that significant cost savings and other advantages are achieved when security analysis and secure engineering practices are introduced early in the development cycle. The return on investment ranges from 12 percent to 21 percent, with the highest rate of return occurring when analysis is performed during application design.
>
> Since nearly three-quarters of security-related defects are design issues that could be resolved inexpensively during the early stages, a significant opportunity for cost savings exists when secure software engineering principles are applied during design.

However, except for a few studies [Berinato 2002; Soo Hoo 2001], we have seen little evidence presented to support the idea that investment during software development in software security will result in commensurate benefits across the entire life cycle.

---

7. Updated from [BSI 45].

Results of the Hoover project [Jaquith 2002] provide some case study data that supports the ROI argument for investment in software security early in software development. In his article "The Security of Applications: Not All Are Created Equal," Jaquith says that "the best-designed e-business applications have one-quarter as many security defects as the worst. By making the right investments in application security, companies can out-perform their peers—and reduce risk by 80 percent."

In their article "Impact of Software Vulnerability Announcements on the Market Value of Software Vendors: An Empirical Investigation," the authors state that "On average, a vendor loses around 0.6 percent value in stock price when a vulnerability is reported. This is equivalent to a loss in market capitalization values of $0.86 billion per vulnerability announcement." The purpose of the study described in this article is "to measure vendors' incentive to develop secure software" [Telang 2004].

We believe that in the future Microsoft may well publish data reflecting the results of using its Security Development Lifecycle [Howard 2006, 2007]. We would also refer readers to the business context discussion in chapter 2 and the business climate discussion in chapter 10 of McGraw's recent book [McGraw 2006] for ideas.

1.7 Managing Secure Software Development

The previous section put forth useful arguments and identified emerging evidence for the value of detecting software security defects as early in the SDLC as possible. We now turn our attention to some of the key project management and software engineering practices to aid in accomplishing this goal. These are introduced here and covered in greater detail in subsequent chapters of this book.

1.7.1 Which Security Strategy Questions Should I Ask?

Achieving an adequate level of software security means more than complying with regulations or implementing commonly accepted best practices. You and your organization must determine your own definition of "adequate." The range of actions you must take to reduce software security risk to an acceptable level depends on what the product,

service, or system you are building needs to protect and what it needs to prevent and manage.

Consider the following questions from an enterprise perspective. Answers to these questions aid in understanding security risks to achieving project goals and objectives.

- What is the value we must protect?
- To sustain this value, which assets must be protected? Why must they be protected? What happens if they're not protected?
- What potential adverse conditions and consequences must be prevented and managed? At what cost? How much disruption can we stand before we take action?
- How do we determine and effectively manage residual risk (the risk remaining after mitigation actions are taken)?
- How do we integrate our answers to these questions into an effective, implementable, enforceable security strategy and plan?

Clearly, an organization cannot protect and prevent everything. Interaction with key stakeholders is essential to determine the project's risk tolerance and its resilience if the risk is realized. In effect, security in the context of risk management involves determining what could go wrong, how likely such events are to occur, what impact they will have if they do occur, and which actions might mitigate or minimize both the likelihood and the impact of each event to an acceptable level.

The answers to these questions can help you determine how much to invest, where to invest, and how fast to invest in an effort to mitigate software security risk. In the absence of answers to these questions (and a process for periodically reviewing and updating them), you (and your business leaders) will find it difficult to define and deploy an effective security strategy and, therefore, may be unable to effectively govern and manage enterprise, information, and software security.[8]

The next section presents a practical way to incorporate a reasoned security strategy into your development process. The framework

---

8. Refer to *Managing Information Security Risks: The OCTAVE Approach* [Alberts 2003] for more information on managing information security risk; "An Introduction to Factor Analysis of Information Risk (FAIR)" [Jones 2005] for more information on managing information risk; and "Risk Management Approaches to Protection" [NIAC 2005] for a description of risk management approaches for national critical infrastructures.

described is a condensed version of the Cigital Risk Management Framework, a mature process that has been applied in the field for almost ten years. It is designed to manage software-induced business risks. Through the application of five simple activities (further detailed in Section 7.4.2), analysts can use their own technical expertise, relevant tools, and technologies to carry out a reasonable risk management approach.

1.7.2 A Risk Management Framework for Software Security[9]

A necessary part of any approach to ensuring adequate software security is the definition and use of a continuous risk management process. Software security risk includes risks found in the outputs and results produced by each life-cycle phase during assurance activities, risks introduced by insufficient processes, and personnel-related risks. The risk management framework (RMF) introduced here and expanded in Chapter 7 can be used to implement a high-level, consistent, iterative risk analysis that is deeply integrated throughout the SDLC.

Figure 1–5 shows the RMF as a closed-loop process with five activity stages. Throughout the application of the RMF, measurement and reporting activities occur. These activities focus on tracking, displaying, and understanding progress regarding software risk.

1.7.3 Software Security Practices in the Development Life Cycle

Managers and software engineers should treat all software faults and weaknesses as potentially exploitable. Reducing exploitable weaknesses begins with the specification of software security requirements, along with considering requirements that may have been overlooked (see Chapter 3, Requirements Engineering for Secure Software). Software that includes security requirements (such as security constraints on process behaviors and the handling of inputs, and resistance to and tolerance of intentional failures) is more likely to be engineered to remain dependable and secure in the face of an attack. In addition, exercising misuse/abuse cases that anticipate abnormal and unexpected behavior can aid in gaining a better understanding of how to create secure and reliable software (see Section 3.2).

---

9. This material is extracted and adapted from a more extensive article by Gary McGraw, Cigital, Inc. [BSI 33].

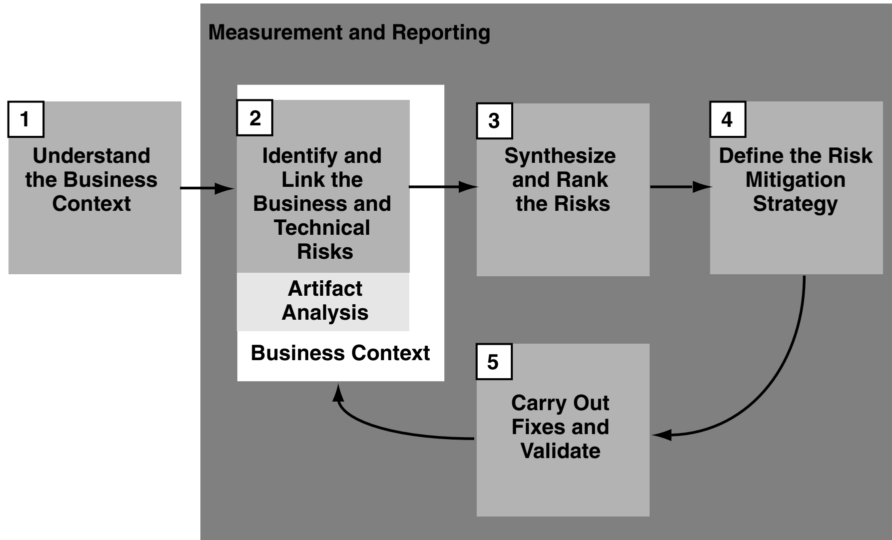


Figure 1–5: *A software security risk management framework*

Developing software from the beginning with security in mind is more effective by orders of magnitude than trying to validate, through testing and verification, that the software is secure. For example, attempting to demonstrate that an implemented system will *never* accept an unsafe input (that is, proving a negative) is impossible. You can prove, however, using approaches such as formal methods and function abstraction, that the software you are designing will never accept an unsafe input. In addition, it is easier to design and implement the system so that input validation routines check *every* input that the software receives against a set of predefined constraints. Testing the input validation function to demonstrate that it is consistently invoked and correctly performed every time input enters the system is then included in the system's functional testing.

Analysis and modeling can serve to better protect your software against the more subtle, complex attack patterns involving externally forced sequences of interactions among components or processes that were never intended to interact during normal software execution. Analysis and modeling can help you determine how to strengthen the security of the software's interfaces with external entities and increase its tolerance of all faults. Methods in support of analysis and modeling

during each life-cycle phase such as attack patterns, misuse and abuse cases, and architectural risk analysis are described in subsequent chapters of this book.

If your development organization's time and resource constraints prevent secure development practices from being applied to the entire software system, you can use the results of a business-driven risk assessment (as introduced earlier in this chapter and further detailed in Section 7.4.2) to determine which software components should be given highest priority.

A security-enhanced life-cycle process should (at least to some extent) compensate for security inadequacies in the software's requirements by adding risk-driven practices and checks for the adequacy of those practices during all software life-cycle phases. Figure 1–6 depicts one example of how to incorporate security into the SDLC using the concept of touchpoints [McGraw 2006; Taylor 2005]. Software security best practices (touchpoints shown as arrows) are applied to a set of software artifacts (the boxes) that are created during the software development process. The intent of this particular approach is that it is

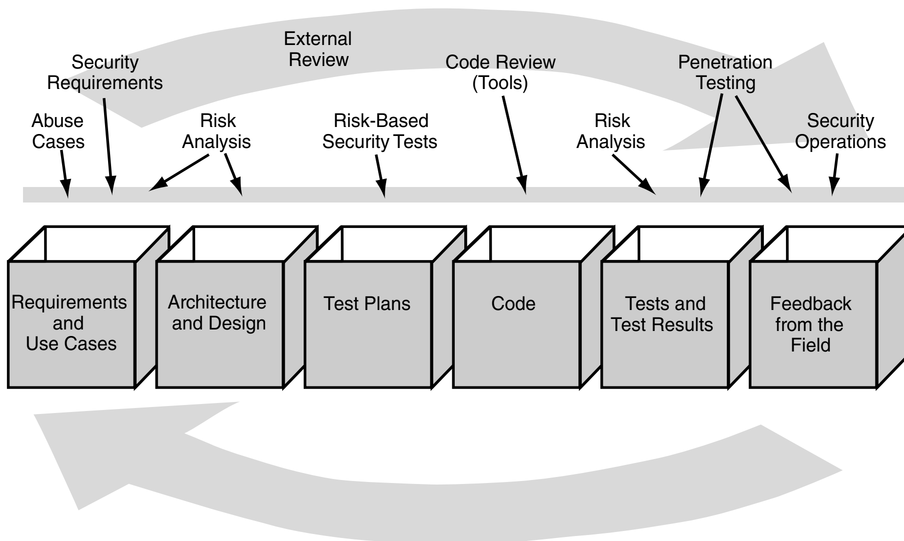


Figure 1–6: *Software development life cycle with defined security touchpoints [McGraw 2006]*

process neutral and, therefore, can be used with a wide range of software development processes (e.g., waterfall, agile, spiral, Capability Maturity Model Integration [CMMI]).

Security controls in the software's life cycle should not be limited to the requirements, design, code, and test phases. It is important to continue performing code reviews, security tests, strict configuration control, and quality assurance during deployment and operations to ensure that updates and patches do not add security weaknesses or malicious logic to production software.[10] Additional considerations for project managers, including the effect of software security requirements on project scope, project plans, estimating resources, and product and process measures, are detailed in Chapter 7.

1.8 Summary

It is a fact of life that software faults, defects, and other weaknesses affect the ability of software to function securely. These vulnerabilities can be exploited to violate software's security properties and force the software into an insecure, exploitable state. Dealing with this possibility is a particularly daunting challenge given the ubiquitous connectivity and explosive growth and complexity of software-based systems.

Adopting a security-enhanced software development process that includes secure development practices will reduce the number of exploitable faults and weaknesses in the deployed software. Correcting potential vulnerabilities as early as possible in the SDLC, mainly through the adoption of security-enhanced processes and practices, is far more cost-effective than attempting to diagnose and correct such problems after the system goes into production. It just makes good sense.

Thus, the goals of using secure software practices are as follows:

- Exploitable faults and other weaknesses are eliminated to the greatest extent possible by well-intentioned engineers.
- The likelihood is greatly reduced or eliminated that malicious engineers can intentionally implant exploitable faults and weaknesses, malicious logic, or backdoors into the software.

10. See the Build Security In Deployment & Operations content area for more information [BSI 01].

- The software is attack resistant, attack tolerant, and attack resilient to the extent possible and practical in support of fulfilling the organization's mission.

To ensure that software and systems meet their security requirements throughout the development life cycle, review, select, and tailor guidance from this book, the BSI Web site, and the sources cited throughout this book as part of normal project management activities.

Index

Index terms should be read within the context of software security engineering. For example, "requirements engineering" refers to *security* requirements engineering.

Numbers

90 percent right, 167
"95 percent defect removal," 14
100-point method, 108

A

Absolute code metrics, 159–160
Abstraction, 102
Abuse cases. *See* Misuse/abuse cases
Accelerated Requirements Method (ARM), 103
Access control, identity management, 197
Accountability, 27, 283
Active responses, 245
Ad hoc testing, 170
Adequate security, defining, 236–238, 278
Age verification case study, 199
AHP, 110
Ambiguity analysis, 128–129, 283
Analyzing
  failures. *See* Failure analysis
  risks. *See* Architectural risk analysis; RMF (risk management framework)
  source code. *See* Source code analysis; Testing security
Application defense techniques, 39–41
Application security, 11, 12, 39
Architectural risk analysis
  architectural vulnerability assessment
    ambiguity, 128–129
    attack resistance, 127–128
    dependency, 129–130
    known bad practices, 127–128
    mapping threats and vulnerabilities, 130
    overview, 126–127
    vulnerability classification, 130
  attack patterns, 147–148
  complete mediation principle, 142
  defense in depth principle, 140
  definition, 283
  economy of mechanism principle, 141
  failing security principle, 139–140
  "forest-level" view, 120–123
  least common mechanism principle, 141
  least privilege principle, 139
  never assuming... [safety] principle, 141–142
  one-page diagram, 122
  overview, 119–120
  project management, 247–249
  promoting privacy principle, 142
  psychological acceptability principle, 142
  reluctance to trust principle, 141
  risk exposure statement, 133–134
  risk impact determination, 132–134
  risk likelihood determination, 130–132
  risk mitigation plans
    contents of, 134–136
    costs, 135
    definition, 123
    detection/correction strategies, 135
    risk impact, reducing, 135
    risk likelihood, reducing, 135
  securing the weakest link principle, 140
  security guidelines, 143–147, 273
  security principles, 137–143, 273
  separation of privilege principle, 140
  software characterization, 120–123
  summary of, 273

Architectural risk analysis *(Continued)*
threats
actors, 126
analyzing, 123–126
business impact, identifying, 132–133
capability for, 131
"hactivists," 126
from individuals, 126
mapping, 130
motivation, 131
organized nonstate entities, 126
script kiddies, 123
state-sponsored, 126
structured external, 126
summary of sources, 123–125
transnational, 126
unstructured external, 126
virtual hacker organizations, 126
vulnerable assets, identifying, 132
useful artifacts, 121
zones of trust, 123
Architectural vulnerability assessment
ambiguity, 128–129
attack resistance, 127–128
dependency, 129–130
known bad practices, 127–128
mapping threats and vulnerabilities, 130
overview, 126–127
vulnerability classification, 130
Architecture
definition, 288
integration mechanism errors, 187
Architecture and design phases
attack patterns, 55–56
issues and challenges, 117–118
management concerns, 118
objectives, 116
risk assessment. *See* Architectural risk analysis
role in developing secure software, 115–117
source of vulnerabilities, 117
summary of, 273
Arguments, assurance cases
developing, 66
vs. other quality arguments, 70
overview, 62
reusing, 69
soundness of, 63
ARM (Accelerated Requirements Method), 103
Artifacts
architectural risk analysis, 121
description, 78
developing, 88–89, 93–94
example, 93–94
for measuring security, 256–257
Assets
critical, 236
definition, 236
threats, identifying, 132
Assurance
quality, 13–15
role in security, 6–7
security. *See* Measuring security; Testing security
software, 7, 289
source code quality. *See* Source code analysis
Assurance cases
arguments
developing, 66
vs. other quality arguments, 70
overview, 62
reusing, 69
soundness of, 63
benefits of, 69–70
building, 62–63
CC (Common Criteria), 69
claims, 62–63
definition, 61, 283
diamond symbol, 66
EAL (Evaluation Assurance Level), 69
evidence, 62–63, 66
example, 63–67
flowchart of, 64–65
GSN (Goal Structuring Notation), 65
maintaining, 69–70
mitigating complexity, 214–215
notation for, 65
parallelogram symbol, 66
protection profiles, 69
reasonable degree of certainty, 69–70

recording defects, 63
responding to detected errors, 63
reusing elements of, 69
rounded box symbol, 65
in the SDLC, 67–68
security evaluation standard, 69
security-privacy, laws and regulations, 68
subclaims, 62
summary of, 270
symbols used in, 65–67

AT&T network outage, 188

Attack patterns
architectural risk analysis, 147–148
benefits of, 46
components, 47–49, 52–53
definition, 45, 283
MITRE security initiatives, 47
recent advances, 46
resistance to using, 46
in the SDLC
architecture phase, 55–56
automated analysis tools, 57–58
black-box testing, 59–60
code reviews, 57–58
coding phase, 57–58
design phase, 55–56
environmental testing, 59–60
implementation phase, 57–58
integration testing, 58
operations phase, 59
requirements phase, 53–54
system testing, 58–59
testing phase, 58–61
unit testing, 58
white-box testing, 59–60
specificity level, 51
summary of, 270, 273
testing security, 173

Attack resilience, 8, 283

Attack resistance
architectural vulnerability assessment, 127–128
defensive perspective, 43
definition, 8, 283

Attack resistance analysis, 283

Attack surface, 35, 284

Attack tolerance, 8, 43, 284

Attack trees, 284

Attacker's perspective. *See also* Defensive perspective; Functional perspective
attacker's advantage, 44–45
definition, 189
identity management, 198–201
misuse/abuse cases, 80–81
representing. *See* Attack patterns
requirements engineering, 76–77
Web services, 192–196

Attackers
behavior, modeling, 188–189. *See also* Attack patterns; Attacker's perspective
motivation, 131
script kiddies, 123
unsophisticated, 123
vs. users, 165
virtual hacker organizations, 126

Attacks. *See also* Threats; *specific attacks*
botnets, 284
cache poisoning, 285
denial-of-service, 285
DNS cache poisoning, 153, 286
elevation (escalation) of privilege, 286
illegal pointer values, 287
integer overflow, 287
known, identifying. *See* Attack patterns
malware, 287
phishing, 287
recovering from. *See* Attack resilience
replay, 288
resistance to. *See* Attack resistance
sources of. *See* Sources of problems
spoofing, 289
SQL injection, 155, 289
stack-smashing, 33, 289
tampering, 289
tolerating. *See* Attack tolerance
usurpation, 196

Attributes. *See* Properties of secure software

Audits, 197, 260–261

Authentication, 284

Availability, 27, 284

Availability threats, 200

B

Bank of America, security management, 226
Benefits of security. *See* Costs/benefits of security; ROI (return on investment)
Best practices, 36–37, 284
Bishop, Matt, 138, 202
Black hats. *See* Attackers
Black-box testing
  definition, 284
  in the SDLC, 177–178
  uses for, 59–60
Blacklist technique, 168–169
Booch, Grady, 215
Botnets, 3, 284
Boundary value analysis, 170
Brainstorming, 81
BSI (Build Security In) Web site, xix–xx, 46
BST (Binary Search Tree), 107
Buffer overflow, 154–155, 284–285
Bugs in code. *See also* Flaws in code
  buffer overflow, 154–155
  deadlocks, 156
  defect rate, 152–153
  definition, 152, 285
  exception handling, 154
  infinite loops, 156
  input validation, 153–154
  introduced by fixes, 169
  known, solutions for, 153–156
  race conditions, 156
  resource collisions, 156
  SQL injection, 155
Building security in
  vs. adding on, 21
  misuse/abuse cases, 80
  processes and practices for, 41–42
  requirements engineering, 79–80
Build Security In (BSI) Web site, xix–xx, 46
Business stresses, 206

C

Cache poisoning, 285
CAPEC (Common Attack Pattern Enumeration and Classification), 46
Case studies
  age verification, 199
  AT&T network outage, 188
  Bank of America, security management, 226
  complexity, 184, 188, 199, 206
  exposure of customer data, 133
  governance and management, 226
  identity management, 199
  managing secure development, 226
  Microsoft, security management, 252
  Network Solutions data loss, 188
  power grid failure, 184, 206
  Skype phone failure, 184
  TJX security breach, 133
  voting machine failure, 184
Categorizing requirements, 272
Causes of problems. *See* Sources of problems
CC (Common Criteria), 69
CDA (critical discourse analysis), 103
CERT
  reported vulnerabilities (1997–2006), 3–4
  Secure Coding Initiative, 161
  sources of vulnerabilities, 161
Characteristics of secure software. *See* Properties of secure software
Claims, assurance cases, 62–63
CLASP (Comprehensive Lightweight Application Security Process), 77, 248
Code. *See* Source code analysis
Code-based testing, 171. *See also* White-box testing
Coding phase, attack patterns, 57–58
Coding practices
  books about, 162–163
  overview, 160–161
  resources about, 161–163
  summary of, 274–275
Common Attack Pattern Enumeration and Classification (CAPEC), 46
Common Weakness Enumeration (CWE), 127–128
Complete mediation principle, 142
Complexity
  business stresses, 206

case studies
  age verification, 199
  AT&T network outage, 188
  identity management, 199
  Network Solutions data loss, 188
  power grid failure, 184, 206–207
  Skype phone failure, 184
  voting machine failure, 184
component interactions, 206
conflicting/changing goals, 205, 213–214
deep technical problems, 215–216
discrepancies, 206–208
evolutionary development, 205, 212–213
expanded scope of security, 204–205
global work process perspective, 208
incremental development, 205, 212–213
integrating multiple systems
  delegating responsibilities, 209–210
  failure analysis, consolidating with mitigation, 211
  generalizing the problem, 211–212
  interoperability, 209
  mitigation, consolidating with failure analysis, 211
  mitigations, 209–212
  "Not my job" attitude, 210
  operational monitoring, 210–211, 215
  partitioning security analysis, 208
  recommended practices, 210–212
interaction stresses, 206
less development freedom, 205
mitigating
  CbyC (Correctness by Construction) method, 216
  change support, 215
  changing/conflicting goals, 214–215
  continuous risk assessment, 215
  coordinating security efforts, 219, 277
  deep technical problems, 216–217
  end-to-end analysis, 216, 218, 276
  failure analysis, 218–219, 276–277
  failure analysis, consolidating with mitigation, 211
  flexibility, 215
  integrating multiple systems, 209–213
  mitigation, consolidating with failure analysis, 211
  operational monitoring, 215
  recommended practices, 218–219
  summary of, 275
  tackling hard problems first, 216, 218, 276
  unavoidable risks, 218–219, 276
overview, 203–205
partitioning security analysis, 208
people stresses, 206–207
recommended practices, 218–219
reduced visibility, 204
service provider perspective, 208
as source of problems, 5–6
stress-related failures, 206
unanticipated risks, 204, 207
user actions, 207
wider spectrum of failures, 204
Component interaction errors, 188
Component interactions, 206
Component-specific integration errors, 187
Comprehensive Lightweight Application Security Process (CLASP), 77, 248
Concurrency management, 144–147
Confidentiality, 26, 285
Conformance, 8, 285
Consolidation, identity management, 190
Control-flow testing, 170
Controlled Requirements Expression (CORE), 101–102
Convergence, 261–262
Coordinating security efforts, 277
CORE (Controlled Requirements Expression), 101–102
Core security properties, 26–28. *See also* Influential security properties; Properties of secure software
Correctness
  definition, 30, 285
  influence on security, 30–34
  specifications, 30–34
  under unanticipated conditions, 30

Costs/benefits of security. *See also* ROI (return on investment)
  budget overruns, 74
  business case, 17–18
  business impacts, 133
  defect rate, effect on development time, 14
  lower stock prices, 18
  preventing defects
    "95 percent defect removal," 14
    early detection, 13–18
    error-prone modules, 15
    inadequate quality assurance, 14
    vs. repairing, 15
  repairing defects
    costs of late detection, 15–16
    by life-cycle phase, 16
    over development time, 14–15
    vs. preventing, 15
  requirements defects, 74
  risk mitigation plans, 135
  schedule problems
    canceled projects, 14
    defect prevention *vs.* repair, 15
    poor quality, effect on, 14–16
    requirements defects, 74
    time overruns, 14
  skimping on quality, 13–14
Coverage analysis, 175–176
Crackers. *See* Attackers
Critical assets, definition, 236
Critical discourse analysis (CDA), 103
Cross-site scripting, 285
Culture of security, 221–225, 227
CWE (Common Weakness Enumeration), 127–128

D

Data-flow testing, 170
Deadlocks, 156
Deception threats, 195
Decision table testing, 170
Defect density, 257
Defect rates
  bugs in code, 152–153
  and development time, 14
  error-prone modules, 15
  flaws in code, 152–153
  and repair time/cost, 15
Defects
  definition, 3, 285
  preventing. *See* Preventing defects
  repairing. *See* Repairing defects
Defense in depth, 140, 285
Defensive perspective. *See also* Attacker's perspective; Functional perspective
  application defense techniques, 39–41
  application security, 39
  architectural implications, 37–43
  attack resistance, 43
  attack tolerance, 43
  expected issues, 37–39
  software development steps, 37
  unexpected issues, 39–43
Delegating responsibilities, 209–210
Denial-of-service attack, 285
Dependability, 29–30, 286
Dependency, 129–130
Dependency analysis, 286
Design phase. *See* Architecture and design phases
Developing software. *See* SDLC (software development life cycle)
Diamond symbol, assurance cases, 66
Discrepancies caused by complexity, 206–208
Disruption threats, 195
DNS cache poisoning, 286
Domain analysis, 102
Domain provisioning, 197–198

E

EAL (Evaluation Assurance Level), 69
Early detection, cost benefits, 13–18
Economy of mechanism principle, 141
Elevation (escalation) of privilege, 286
Elicitation techniques
  comparison chart, 105
  evaluating, 103–104
  selecting, 89, 97
Eliciting requirements
  misuse/abuse cases, 100–101
  overview, 100

SQUARE, 89, 97–98
summary of, 272
tools and techniques, 100–106
Endpoint attacks, 201
End-to-end analysis, 276
Enterprise software security framework (ESSF). *See* ESSF (enterprise software security framework)
Environmental testing, 59–60
Equivalence partitioning, 170
Error-prone modules, 15
Errors
architecture integration mechanisms, 187
categories of, 187–188
component interaction, 188
component-specific integration, 187
definition, 186, 286
operations, 188
specific interfaces, 187
system behavior, 188
usage, 188
ESSF (enterprise software security framework)
assurance competency, 234
best practices, lack of, 228–229
creating a new group, 228
decision making, 229
definition, 230
focus, guidelines, 233
goals, lack of, 227–228
governance competency, 235
knowledge management, 234
pitfalls, 227–229
required competencies, 233–235
roadmaps for, 235
security touchpoints, 234
summary of, 280
tools for, 229
training, 234
"who, what, when" structure, 230–233
Evaluation Assurance Level (EAL), 69
Evidence, assurance cases, 62–63, 66
Evolutionary development, 205, 212–213
Examples of security problems. *See* Case studies
Exception handling, 154
Executables, testing, 175–176
Exploits, 286
Exposure of information, 194

F

Fagan inspection, 90, 98, 286
Failing security principle, 139–140
Failure, definition, 186, 286
Failure analysis
attacker behavior, 188–189
consolidating with mitigation, 211
hardware *vs.* software, 205–208
recommended practices, 218–219
summary of, 276–277
Fault-based testing, 171
Faults, 186, 286
Fault-tolerance testing, 170
Flaws in code, 152–153, 286. *See also* Bugs in code
FODA (feature-oriented domain analysis), 102
"Forest-level" view, 120–123
Formal specifications, 78
Functional perspective. *See also* Attacker's perspective; Defensive perspective
definition, 189
identity management
access control, 197
auditing, 197
challenges, 197–198
consolidation, 190
domain provisioning, 197–198
identity mapping, 197
interoperability, 190
legislation and regulation, 198
objectives, 190
privacy concerns, 198
reporting, 197
technical issues, 197–198
trends, 197–198
Web services, 190–192
Functional security testing
ad hoc testing, 170
boundary value analysis, 170
code-based testing, 171
control-flow testing, 170
data-flow testing, 170

Functional security testing *(Continued)*
decision table testing, 170
equivalence partitioning, 170
fault-based testing, 171
fault-tolerance testing, 170
known code internals. *See* White-box testing
limitations, 168–169
load testing, 171
logic-based testing, 170
model-based testing, 170
overview, 167–168
performance testing, 171
protocol performance testing, 171
robustness testing, 170
specification-based testing, 170
state-based testing, 170
test cases, 170, 173, 274–275
tools and techniques, 170–171
unknown code internals. *See* Black-box testing
usage-based testing, 171
use-case-based testing, 171

G

Generalizing the problem, 211–212
Global work process perspective, 208
Goal Structuring Notation (GSN), 65
Governance and management. *See also* Project management
adequate security, defining, 236–238, 278
adopting a security framework. *See* ESSF (enterprise software security framework)
analysis, 21–22
assets, definition, 236
at Bank of America, 226
building in *vs.* adding on, 21
case study, 226
critical assets, definition, 236
definition, 223–224, 288
effective, characteristics of, 224–225, 279
maturity of practice
audits, 260–261
convergence, 261–262
exemplars, 265
legal perspective, 263
operational resilience, 261–262
overview, 259
protecting information, 259–260
software engineering perspective, 263–264
modeling, 21–22
overview, 221–222
process, definition, 237
protection strategies, 236
risks
impact, 237–238
likelihood, 237–238
measuring, 242–243
reporting, 242–243
tolerance for, 237–238
RMF (risk management framework)
business context, understanding, 239–240
business risks, identifying, 240–241
fixing problems, 242
identifying risks, 240–241
mitigation strategy, defining, 241–242
multi-level loop nature of, 243–244
overview, 238–239
prioritizing risks, 241
synthesizing risks, 241
technical risks, identifying, 240–241
validating fixes, 242
security, as cultural norm, 221–222, 279
security strategy, relevant questions, 18–20
security-enhanced SDLC, 20–23
summary of, 278–280
touchpoints, 22–23
Greenspan, Alan, 134
GSN (Goal Structuring Notation), 65

H

Hackers. *See* Attackers
"Hactivists," 126. *See also* Attackers
Hardened services and servers, 200–201
Hardening, 286
Howard, Michael, 138, 264

I

IBIS (issue-based information systems), 102
Icons used in this book, xix, 268

Identity management
  availability threats, 200
  case study, 199
  endpoint attacks, 201
  functional perspective
    access control, 197
    auditing, 197
    challenges, 197–198
    consolidation, 190
    domain provisioning, 197–198
    identity mapping, 197
    interoperability, 190
    legislation and regulation, 198
    objectives, 190
    privacy concerns, 198
    reporting, 197
    technical issues, 197–198
    trends, 197–198
  hardened services and servers, 200–201
  incident response, 201
  information leakage, 199–200
  risk mitigation, 200–201
  security *vs.* privacy, 199
  software development, 201–203
  usability attacks, 201
Identity mapping, 197
Identity spoofing, 286
Illegal pointer values, 287
Implementation phase, attack patterns, 57–58
Incident response, 201
Incremental development, 205, 212–213
Infinite loops, 156
Information warfare, 3
Influential security properties. *See also* Core security properties; Properties of secure software
  complexity, 35–36
  correctness, 30–34
  definition, 28
  dependability, 29–30
  predictability, 34
  reliability, 34–35
  safety, 34–35
  size, 35–36
  traceability, 35–36
Information leakage, 199–200
Input validation, 153–154
Inspections
  code. *See* Source code analysis
  Fagan, 90, 98, 286
  measuring security, 258
  requirements, 90, 98–99, 272
  SQUARE (Security Quality Requirements Engineering), 90, 98–99, 272
Integer overflow, 287
Integrating multiple systems
  delegating responsibilities, 209–210
  failure analysis, consolidating with mitigation, 211
  generalizing the problem, 211–212
  interoperability, 209
  mitigation, consolidating with failure analysis, 211
  mitigations, 209–212
  "Not my job" attitude, 210
  operational monitoring, 210–211, 215
  partitioning security analysis, 208
  recommended practices, 210–212
Integration testing, 58, 176
Integrity, 27, 287
Interaction stresses, 206
Interoperability
  identity management, 190
  integrating multiple systems, 209
  Web services, 190–191
Issue-based information systems (IBIS), 102

J

JAD (Joint Application Development), 102

K

Known
  attacks, identifying. *See* Attack patterns
  bad practices, 127–128
  bugs, solutions for, 153–156
  code internals, testing. *See* White-box testing
  vulnerabilities, 187

L

Least common mechanism principle, 141
Least privilege principle, 139

LeBlanc, David, 138
Legal perspective, 263
Libraries, testing, 175–176
Lipner, Steve, 264
Load testing, 171
Logic-based testing, 170

M

MacLean, Rhonda, 226
Make the Client Invisible pattern, 49–51, 56
Malware, 287
Managing security. *See* Governance and management; Project management
Manual code inspection, 157
Mapping
- identities, 197
- threats and vulnerabilities, 130

Maturity of practice, governance and management
- audits, 260–261
- convergence, 261–262
- exemplars, 265
- legal perspective, 263
- operational resilience, 261–262
- overview, 259
- protecting information, 259–260
- software engineering perspective, 263–264

McGraw, Gary, 138
Measuring security. *See also* Source code analysis; Testing security
- defect density, 257
- inspections, 258
- Making Security Measurable program, 47
- objectives, 255–256
- overview, 254–255
- phase containment of defects, 257
- process artifacts, 256–257
- process measures, 256–257
- product measures, 257–259
- summary of, 280

Messages, Web services, 191–192, 194
Methods. *See* Tools and techniques
Metric code analysis, 159–160
Microsoft, security management, 252
Minimizing threats. *See* Mitigation
Misuse/abuse cases
- attacker’s perspective, 80–81
- brainstorming, 81
- classification, example, 96–97
- creating, 81–82
- definition, 31, 287
- eliciting requirements, 100–101
- example, 82–84
- security, built in *vs.* added on, 79–80
- system assumptions, 80–81
- templates for, 83

Mitigation. *See also* Risk mitigation plans
- causing new risks, 254
- complexity. *See* Complexity, mitigating
- consolidating with failure analysis, 211
- definition, 287
- identity management risks, 200–201
- integrating multiple systems, 209–212
- threats, 194–196

MITRE security initiatives, 47
Model-based testing, 170
Modeling, 90, 102

N

Negative requirements, 77, 172–173
Network Solutions data loss, 188
Neumann, Peter, 184
Never assuming... [safety] principle, 141–142
Nonfunctional requirements, 75–76
Non-repudiation, 27, 287
“Not my job” attitude, 210
Notation for assurance cases, 65
Numeral assignment, 107

O

Operational monitoring, 210–211, 215
Operational resilience, 261–262
Operations errors, 188
Operations phase, attack patterns, 59
Organized nonstate threats, 126
Origins of problems. *See* Sources of problems

P

Parallelogram symbol, assurance cases, 66
Partitioning security analysis, 208

Passive responses, 245
Penetration testing, 178–179
People stresses, 206–207
Performance testing, 171
Perspectives on security
  attacker's point of view. *See* Attacker's perspective
  complexity, 208
  defender's point of view. *See* Defensive perspective
  functional point of view. *See* Functional perspective
  global work process, 208
  legal, 263
  service provider, 208
  software engineering, 263–264
Phase containment of defects, 257
Phishing, 287
Planning game, 107–108
Positive requirements, 167–168
Power grid failure, 184, 206
Predictability, 34
Predictable execution, 8, 287
Preventing defects. *See also* Mitigation
  "95 percent defect removal," 14
  costs of
    "95 percent defect removal," 14
    early detection, 13–18
    error-prone modules, 15
    inadequate quality assurance, 14
    vs. repair, 15
Prioritizing
  requirements
    overview, 106
    requirements prioritization framework, 109
    SQUARE, 90, 98
    summary of, 272
    tools and techniques, 106–111
  risks, 241
Privacy, 198–199. *See also* Identity management
Problems, causes. *See* Sources of problems
Process, definition, 237
Processes and practices
  role in security, 8–9
  in the SDLC, 20–23
Project management. *See also* Governance and management
  active responses, 245
  architectural risk analysis, 247–249
  continuous risk management, 247–249, 278
  knowledge and expertise, 250–251
  measuring security
    defect density, 257
    inspections, 258
    objectives, 255–256
    overview, 254–255
    phase containment of defects, 257
    process artifacts, 256–257
    process measures, 256–257
    product measures, 257–259
    summary of, 280
  at Microsoft, 252
  miscommunications, 246–247
  passive responses, 245
  planning, 246–250
  recommended practices, 266
  required activities, 249–250
  resource estimation, 251–253
  resources, 250–251
  risks, 253–254
  scope, 245–246
  security practices, 247–249
  tools, 250
Promoting privacy principle, 142
Properties of secure software
  accountability, 27
  asserting. *See* Assurance cases
  attack resilience, 8
  attack resistance, 8
  attack tolerance, 8
  availability, 27
  confidentiality, 26
  conformance, 8
  core properties, 26–28
  influencing, 36–37
  influential properties
    complexity, 35–36
    correctness, 30–34
    definition, 28
    dependability, 29–30
    predictability, 34
    reliability, 34–35
    safety, 34–35

Properties of secure software *(Continued)*
size, 35–36
traceability, 35–36
integrity, 27
non-repudiation, 27
perspectives on. *See* Attacker's perspective; Defensive perspective
predictable execution, 8
specifying. *See* Assurance cases
summary of, 270
trustworthiness, 8
Protection profiles, 69, 287
Protection strategies, 236
Protocol performance testing, 171
Prototyping, 90
Psychological acceptability principle, 142

Q
QFD (Quality Function Deployment), 101
Qualities. *See* Properties of secure software
Quality assurance, 13–15. *See also* Measuring security; Source code analysis; Testing security
Quality requirements. *See* Requirements engineering

R
Race conditions, 156
Reasonable degree of certainty, 69–70, 288
Recognize, 288
Recover, 288
Red teaming, 288
Reduced visibility, 204
Reducing threats. *See* Mitigation
Refinement, 102
Relative code metrics, 159–160
Reliability, 7, 34–35, 289
Reluctance to trust principle, 141
Repairing defects, 14–16
Replay attack, 288
Reporting, identity management, 197
Repudiation, 288
Requirements engineering. *See also* SQUARE (Security Quality Requirements Engineering)
artifacts
description, 78
developing, 88–89, 93–94
example, 93–94
attacker's perspective, 76–77
common problems, 74–75
elicitation techniques
comparison chart, 105
evaluating, 103–104
selecting, 89, 97
eliciting requirements
misuse/abuse cases, 100–101
overview, 100
summary of, 272
tools and techniques, 100–106
formal specifications, 78
importance of, 74–75
negative requirements, 77
nonfunctional requirements, 75–76
prioritizing requirements
overview, 106
recommendations for managers, 111–112
requirements prioritization framework, 109
SQUARE, 90, 98
summary of, 272
tools and techniques, 106–111
quality requirements, 75–76
recommendations for managers, 111–112
security, built in *vs.* added on, 79–80
security requirements, 76–78
summary of, 271–272
tools and techniques, 77–78. *See also* Attack patterns; Misuse/abuse cases
Requirements phase
attack patterns, 53–54
positive *vs.* negative requirements, 53–54
security testing, 173–174
vague specifications, 53–54
Requirements prioritization framework, 109
Requirements triage, 108–109
Resilience
attack, 8, 283
operational, 261–262

Resist, 288
Resource collisions, 156
Resources for project management, 250–253
Return on investment (ROI), 9, 13, 18. *See also* Costs/benefits of security
Reusing assurance cases arguments, 69
Reviewing source code. *See* Source code analysis
Risk assessment
  in architecture and design. *See* Architectural risk analysis
  preliminary, 173
  SQUARE
    description, 89
    sample output, 94–97
    techniques, ranking, 94–96
  summary of, 271
  tools and techniques, 94–97
Risk-based testing, 169–172, 275
Risk exposure statement, 133–134
Risk management framework (RMF). *See* RMF (risk management framework)
Risk mitigation plans. *See also* Mitigation
  contents of, 134–136
  costs, 135
  definition, 123
  detection/correction strategies, 135
  risk impact, reducing, 135
  risk likelihood, reducing, 135
Risks
  exposure, lack of awareness, 6
  impact, 132–134, 237–238
  likelihood, 130–132, 237–238
  managing. *See* RMF (risk management framework)
  measuring, 242–243
  project management, 253–254
  protective measures. *See* Mitigation; Risk mitigation plans
  reporting, 242–243
  tolerance for, 237–238
  unavoidable, 276
  Web services, 193–194
Rittel, Horst, 102
RMF (risk management framework)
  business context, understanding, 239–240
  business risks, identifying, 240–241
  continuous management, 247–249, 278
  fixing problems, 242
  identifying risks, 240–241
  mitigation strategy, defining, 241–242
  multi-level loop nature of, 243–244
  overview, 238–239
  prioritizing risks, 241
  synthesizing risks, 241
  technical risks, identifying, 240–241
  validating fixes, 242
Robustness testing, 170
ROI (return on investment), 13, 17–18. *See also* Costs/benefits of security
Rounded box symbol, assurance cases, 65
Rubin, Avi, 184

S

Safety, 7, 34–35, 289
Saltzer, Jerome, 138
Schedules
  canceled projects, 14
  costs of security problems, 14–16
  poor quality, effect on, 14–16
  prevention *vs.* repair, 15
  time overruns, 14
Schneier, Bruce, 138
Schroeder, Michael, 138
SCR (Software Cost Reduction), 78
Script kiddies, 123. *See also* Attackers
SDLC (software development life cycle). *See also specific activities*
  architecture and design phases
    attack patterns, 55–56
    issues and challenges, 117–118
    management concerns, 118
    objectives, 116
    risk assessment. *See* Architectural risk analysis
    role in developing secure software, 115–117
    source of vulnerabilities, 117
  assurance cases, 67–68
  attack patterns
    architecture phase, 55–56
    automated analysis tools, 57–58
    black-box testing, 59–60
    code reviews, 57–58

SDLC *(Continued)*
coding phase, 57–58
design phase, 55–56
environmental testing, 59–60
implementation phase, 57–58
integration testing, 58
operations phase, 59
requirements phase, 53–54
system testing, 58–59
testing phase, 58–61
unit testing, 58
white-box testing, 59–60
black-box testing, 177–178
coding phase, attack patterns, 57–58
design phase. *See* Architecture and design phases
implementation phase, attack patterns, 57–58
managing secure development, 20–23
operations phase, attack patterns, 59
phase containment of defects, 257
practices by life cycle phase, summary of, 270–280
processes and practices, 20–23
requirements phase, 53–54, 173–174. *See also* Requirements engineering
reviewing source code. *See* Source code analysis
testing for security. *See* Testing security
touchpoints, 22–23
white-box testing, 175
writing code. *See* Coding practices
Secure Coding Initiative, 161
goal, xv
Securing the weakest link principle, 140
Security
application, 11, 12, 39
vs. assurance, 6–7
as cultural norm, 221–222, 279
definition, 289
development assumptions, 5
goals, 7
guidelines, 143–147, 273
principles, 137–143, 273
vs. privacy, 199
nonsecure software, 3
objective, 7
practices, summary of, 270–280
problem overview
complexity of systems, 5–6
report to the U.S. President, 3
survey of financial services industry, 4
vulnerabilities reported to CERT (1997-2006), 3–4
strategy, 18–19
system criteria, 7
Security architecture. *See* Architectural; Architecture and design
Security profiles, 288
Security Quality Requirements Engineering (SQUARE). *See* SQUARE (Security Quality Requirements Engineering)
Security touchpoints
definition, 36–37, 290
ESSF (enterprise software security framework), 234
in the SDLC, 22–23
Separation of privilege principle, 140
Service provider perspective, 208
Shirey's threat categories, 192, 194–196
Simple Script Injection pattern, 58
Size, influence on security, 35–36. *See also* Complexity
Skype phone failure, 184
SOA (service-oriented architecture), 191
SOAP (Simple Object Access Protocol), 192–193
SOAR report, 78
Social engineering, 189, 288
Soft Systems Methodology (SSM), 101
Software
assurance, 6–7, 289
characterization, 120–123
development. *See* SDLC
reliability, 7, 34–35, 289
safety, 7, 34–35, 289
security. *See* Security
trustworthy, 5
Software Cost Reduction (SCR), 78
Software development life cycle (SDLC). *See* SDLC (software development life cycle)
Software engineering perspective, 263–264

"Software Penetration Testing," 179
"Software Security Testing," 179
Sound practice, 289
Source code analysis
  bugs
    buffer overflow, 154–155
    deadlocks, 156
    defect rate, 152–153
    definition, 152
    exception handling, 154
    infinite loops, 156
    input validation, 153–154
    known, solutions for, 153–156
    race conditions, 156
    resource collisions, 156
    SQL injection, 155
  flaws, 152–153
  process diagrams, 160
  source code review
    absolute metrics, 159–160
    automatic analysis, 57–58
    manual inspection, 157
    metric analysis, 159–160
    overview, 156–157
    relative metrics, 159–160
    static code analysis tools, 157–159
    summary of, 274
  vulnerabilities, 152–156. *See also* Bugs in code; Flaws in code
Sources of problems. *See also* Attacks; Complexity; Threats
  architecture and design, 117
  bugs in code, 152–153
  CERT analysis, 161
  complexity of systems, 5–6
  flaws in code, 152–153
  implementation defects, 12
  incorrect assumptions, 12
  mistakes, 12–13
  requirements defects, 74
  specification defects, 12
  summary of, 12
  threats, 123–125
  unintended interactions, 12
Specific interface errors, 187
Specification-based testing, 170
Specifications
  properties of secure software. *See* Assurance cases
  security requirements. *See* Requirements engineering
Spoofing, 289
SQL injection, 155, 289
SQUARE (Security Quality Requirements Engineering). *See also* Requirements engineering
  artifacts, developing, 88–89, 93–94
  categorizing requirements, 89, 97–98, 272
  defining terms, 88, 92
  definition, 84
  elicitation techniques, selecting, 89, 97
  eliciting requirements, 89, 97–98
  goals, identifying, 88, 92
  history of, 84
  inspecting requirements, 90, 98–99, 272
  prioritizing requirements, 90, 98
  process overview, 85–87
  process steps, 88–90
  prototyping, 90
  results, expected, 90–91
  results, final, 99
  risk assessment
    description, 89
    sample output, 94–97
    techniques, ranking, 94–96
  sample outputs, 92–99
  summary of, 271
SSM (Soft Systems Methodology), 101
Stack-smashing attack, 33, 289
Standards
  CC (Common Criteria), 69
  credit card industry, 260
  security evaluation, 69
  software engineering terminology, 34
State-based testing, 170
State-sponsored threats, 126
"Static Analysis for Security," 179
Static code analysis tools, 157–159
Steven, John, 224–225, 281
Stock prices, effects of security problems, 18
Strategies for security, 18–20
Stress testing, 177
Stress-related failures, 206
Structured external threats, 126
Subclaims, 62

System behavior errors, 188
System testing, 58–59, 176–179

T

Tackling hard problems first, 276
Tampering, 289
Techniques. *See* Tools and techniques
Telephone outages, 184, 188
Templates
  misuse/abuse cases, 83
  testing security, 172
Test cases, functional testing, 170, 173, 274–275
Testing security. *See also* Measuring security; Source code analysis
  90 percent right, 167
  attack patterns, 58–61, 173
  black-box testing, 177–178
  blacklist technique, 168–169
  books and publications, 164, 179–180
  bugs, introduced by fixes, 169
  coverage analysis, 175–176
  executables, testing, 175–176
  functional
    ad hoc testing, 170
    boundary value analysis, 170
    code-based testing, 171. *See also* White-box testing
    control-flow testing, 170
    data-flow testing, 170
    decision table testing, 170
    equivalence partitioning, 170
    fault-based testing, 171
    fault-tolerance testing, 170
    limitations, 168–169
    load testing, 171
    logic-based testing, 170
    model-based testing, 170
    overview, 167–168
    performance testing, 171
    protocol performance testing, 171
    robustness testing, 170
    specification-based testing, 170
    state-based testing, 170
    tools and techniques, 170–171
    usage-based testing, 171
    use-case-based testing, 171
    white-box testing, 171
  goals of, 164
  integration testing, 176
  known internals. *See* White-box testing
  libraries, testing, 175–176
  limitations, 168–169
  methods for, 167
  negative requirements, 172–173
  penetration testing, 178–179
  positive requirements, 167–168
  preliminary risk assessment, 173
  required scenarios, 33
  requirements phase, 173–174
  resources about, 179–180
  risk-based, 169–172, 275
  scenarios, from past experiences, 172–173
  vs. software testing, 165–167
  stress testing, 177
  summary of, 274–275
  system testing, 176–179
  templates for, 172
  threat modeling, 173
  throughout the SDLC
    black-box testing, 177–178
    coverage analysis, 175–176
    executables, testing, 175–176
    integration testing, 176
    libraries, testing, 175–176
    penetration testing, 178–179
    preliminary risk assessment, 173
    requirements phase, 173–174
    stress testing, 177
    system testing, 176–179
    unit testing, 174–175
    white-box testing, 175
  unique aspects, summary of, 274
  unit testing, 174–175
  unknown internals. *See* Black-box testing
  users *vs.* attackers, 165
  white-box testing, 175
Theory-W, 108
Threats. *See also* Attacks; Sources of problems
  actors, 126
  analyzing, 123–126, 289
  business impact, identifying, 132–133
  capability for, 131

categories of, 9–10, 194–196
deception, 195
definition, 289
development stage, 9–10
disruption, 195
exposure of information, 194
"hactivists," 126
from individuals, 126
mapping, 130
modeling, 173, 290
motivation, 131
operations stage, 10
organized nonstate entities, 126
protective measures. *See* Mitigation
script kiddies, 123
Shirey's categories, 192, 194–196
state-sponsored, 126
structured external, 126
summary of sources, 123–125
transnational, 126
unstructured external, 126
usurpation, 196
virtual hacker organizations, 126
vulnerable assets, identifying, 132
weaknesses. *See* Vulnerabilities
TJX security breach, 133
Tolerance
attacks, 8, 43, 284
definition, 290
faults, 170
risks, 237–238
Tools and techniques. *See also specific tools and techniques*
100-point method, 108
AHP, 110
ARM (Accelerated Requirements Method), 103
automated analysis, 57–58
bad practices reference, 127–128
BST (binary search tree), 107
CDA (critical discourse analysis), 103
CLASP (Comprehensive Lightweight Application Security Process), 77, 248
CORE (Controlled Requirements Expression), 101–102
CWE (Common Weakness Enumeration), 127–128
domain analysis, 102
eliciting requirements, 100–106
ESSF (enterprise software security framework), 229
FODA (feature-oriented domain analysis), 102
functional security testing, 170–171
IBIS (issue-based information systems), 102
JAD (Joint Application Development), 102
numeral assignment, 107
planning game, 107–108
prioritizing requirements, 106–111
project management, 250
prototyping, 90
QFD (Quality Function Deployment), 101
requirements engineering, 77–78. *See also* SQUARE
requirements prioritization framework, 109
requirements triage, 108–109
risk assessment, 94–97
SSM (Soft Systems Methodology), 101
static code analysis, 157–159
Theory-W, 108
Wiegers' method, 109
win-win, 108
Touchpoints
definition, 36–37, 290
ESSF (enterprise software security framework), 234
in the SDLC, 22–23
Traceability, 35–36
Transnational threats, 126
Trust boundaries, 290
Trustworthiness, 8, 290

U

Unanticipated risks, 204, 207
Unintended interactions, 12
Unit testing, 58, 174–175
Unstructured external threats, 126
Usability attacks, 201
Usage errors, 188
Usage-based testing, 171
Use cases, 290. *See also* Misuse/abuse cases

Use-case diagrams. *See* Misuse/abuse cases
Use-case-based testing, 171
User actions, effects of complexity, 207
Users *vs.* attackers, 165
Usurpation threats, 196

V

Viega, John, 138
Voting machine failure, 184
Vulnerabilities. *See also* Architectural vulnerability assessment
    architectural vulnerability assessment, 130
    classifying, 130
    in code, 152. *See also* Bugs in code; Flaws in code
    definition, 290
    known, list of, 187
    perceived size, 33
    reported to CERT (1997-2006), 3–4
    tolerating, based on size, 32–34
Vulnerable assets, identifying, 132

W

Weaknesses. *See* Vulnerabilities
Web services
    attacker's perspective, 192–196
    distributed systems risks, 193–194
    functional perspective, 190–192
    goal of, 191
    interoperability, 190–191
    message risks, 194
    messages, 191–192
    risk factors, 193–194
    Shirey's threat categories, 192, 194–196
    SOA (service-oriented architecture), 191
    SOAP (Simple Object Access Protocol), 192–193
White-box testing. *See also* Code-based testing
    attack patterns, 59–60
    definition, 290
    functional security testing, 171
    in the SDLC, 175
Whitelist, 290
"Who, what, when" structure, 230–233
Wicked problems, 102
Wiegers' method, 109
Win-win technique, 108

Z

Zones of trust, 123, 290