

Self-Study Course for the Certified Security
Information Professional

ALL-IN-ONE

CSSEP

Certified Security Enterprise Lifecycle Professional

EXAM GUIDE

Third Edition

Complete coverage
of CSSEP
domain knowledge

Includes both an
object-based and an
entity-based representation

Filled with
over 1000 exam questions
and explanations



Self-Study Course
for the Certified Security
Information Professional

CSSEP Exam Guide
Version 3.0

“All-in-One Is All You Need.”

ALL-IN-ONE

CSSLP®

Certified Secure Software Lifecycle Professional

E X A M G U I D E

THIRD EDITION

Online content
includes:

- Test engine that provides full-length practice exams or customized quizzes by chapter or exam domain

*Complete coverage
of all CSSLP
exam domains*

*Ideal as both a
study tool and an
on-the-job reference*

*Filled with
practice exam questions
and explanations*



WM. ARTHUR CONKLIN,
PhD, CompTIA Security+™, CISSP®, CSSLP

DANIEL SHOEMAKER,
PhD

ALL ■ IN ■ ONE

CSSLP®

Certified Secure Software Lifecycle Professional

EXAM GUIDE

Third Edition

Wm. Arthur Conklin
Dan Shoemaker



New York Chicago San Francisco
Athens London Madrid Mexico City
Milan New Delhi Singapore Sydney Toronto

McGraw Hill is an independent entity from the International Information Systems Security Certification Consortium, Inc., (ISC)² and is not affiliated with (ISC)² in any manner. This study/training guide and/or material is not sponsored by, endorsed by, or affiliated with (ISC)² in any manner. This publication and accompanying media may be used in assisting students to prepare for the Certified Secure Software Lifecycle Professional (CSSLP[®]) exam. Neither (ISC)² nor McGraw Hill warrant that use of this publication and accompanying media will ensure passing any exam. (ISC)², CSSLP[®], CISSP[®], CAP[®], ISSAP[®], ISSEP[®], ISSMP[®], SSCP[®] and CBK[®] are trademarks or registered trademarks of (ISC)² in the United States and certain other countries. All other trademarks are trademarks of their respective owners.

Copyright © 2022 by McGraw Hill. All rights reserved. Except as permitted under the United States Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of the publisher, with the exception that the program listings may be entered, stored, and executed in a computer system, but they may not be reproduced for publication.

ISBN: 978-1-26-425821-5
MHID: 1-26-425821-6

The material in this eBook also appears in the print version of this title:
ISBN: 978-1-26-425820-8, MHID: 1-26-425820-8.

eBook conversion by codeMantra
Version 1.0

All trademarks are trademarks of their respective owners. Rather than put a trademark symbol after every occurrence of a trademarked name, we use names in an editorial fashion only, and to the benefit of the trademark owner, with no intention of infringement of the trademark. Where such designations appear in this book, they have been printed with initial caps.

McGraw-Hill Education eBooks are available at special quantity discounts to use as premiums and sales promotions or for use in corporate training programs. To contact a representative, please visit the Contact Us page at www.mhprofessional.com.

Information has been obtained by McGraw Hill from sources believed to be reliable. However, because of the possibility of human or mechanical error by our sources, McGraw Hill, or others, McGraw Hill does not guarantee the accuracy, adequacy, or completeness of any information and is not responsible for any errors or omissions or the results obtained from the use of such information.

TERMS OF USE

This is a copyrighted work and McGraw-Hill Education and its licensors reserve all rights in and to the work. Use of this work is subject to these terms. Except as permitted under the Copyright Act of 1976 and the right to store and retrieve one copy of the work, you may not decompile, disassemble, reverse engineer, reproduce, modify, create derivative works based upon, transmit, distribute, disseminate, sell, publish or sublicense the work or any part of it without McGraw-Hill Education's prior consent. You may use the work for your own noncommercial and personal use; any other use of the work is strictly prohibited. Your right to use the work may be terminated if you fail to comply with these terms.

THE WORK IS PROVIDED "AS IS." McGRAW-HILL EDUCATION AND ITS LICENSORS MAKE NO GUARANTEES OR WARRANTIES AS TO THE ACCURACY, ADEQUACY OR COMPLETENESS OF OR RESULTS TO BE OBTAINED FROM USING THE WORK, INCLUDING ANY INFORMATION THAT CAN BE ACCESSED THROUGH THE WORK VIA HYPERLINK OR OTHERWISE, AND EXPRESSLY DISCLAIM ANY WARRANTY, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. McGraw-Hill Education and its licensors do not warrant or guarantee that the functions contained in the work will meet your requirements or that its operation will be uninterrupted or error free. Neither McGraw-Hill Education nor its licensors shall be liable to you or anyone else for any inaccuracy, error or omission, regardless of cause, in the work or for any damages resulting therefrom. McGraw-Hill Education has no responsibility for the content of any information accessed through the work. Under no circumstances shall McGraw-Hill Education and/or its licensors be liable for any indirect, incidental, special, punitive, consequential or similar damages that result from the use of or inability to use the work, even if any of them has been advised of the possibility of such damages. This limitation of liability shall apply to any claim or cause whatsoever whether such claim or cause arises in contract, tort or otherwise.

*To Susan Conklin, my muse.
You make all journeys worth taking,
and your love carries me forward each day.*
—Art Conklin

*To Tamara,
Your love and support make all I do possible.*
—Dan Shoemaker

ABOUT THE AUTHORS

Wm. Arthur Conklin, PhD, CompTIA Security+, CISSP, CSSLP, GISCP, GCFA, GCIA, GRID, GCIP, CRISC, CASP, is a professor and director of the Center for Information Security Research and Education in the College of Technology at the University of Houston. He holds two terminal degrees, a PhD in business administration (specializing in information security) from The University of Texas at San Antonio (UTSA) and a degree in electrical engineering (specializing in space systems engineering) from the Naval Postgraduate School in Monterey, California. He is a fellow of ISSA and (CS)2AI and a senior member of ASQ, IEEE, and ACM. Dr. Conklin's research interests lie in the areas of critical infrastructure cybersecurity. He has coauthored numerous books on information security and has written and presented numerous conference and academic journal papers. He has more than 20 years of teaching experience at the college level and has assisted in building two information security programs that have been recognized by the National Security Agency (NSA) and Department of Homeland Security (DHS) as Centers of Academic Excellence in Cyberdefense Education.

Dan Shoemaker, PhD (University of Detroit Mercy), is a professor and the director of the Graduate Program in Cybersecurity at UDM's Center for Cyber Security and Intelligence Studies. This center includes the NSA Center of Academic Excellence in Information Assurance Education. Dr. Shoemaker is a full-time professor, with 26 years as department chair. Dr. Shoemaker was one of the earliest academic participants in the development of software engineering as a discipline, starting at the Software Engineering Institute (SEI) in the fall of 1987. As the co-chair for the DHS National Workforce Training and Education Initiative for Software and Supply Chain Assurance, he is one of the three authors of the Software Assurance Common Body of Knowledge (CBK). Dr. Shoemaker is a Michigan man at heart, having received his degrees at the University of Michigan. He was named a

Distinguished Visitor of the IEEE in 2020. He has written 12 books in the field and has more than 200 publications. He speaks extensively on security topics, both in the United States and internationally. He also served as an SME for the Securely Provision Knowledge Area for both versions of the NIST-NICE workforce framework. Finally, he recently served as an SME for the ACM Joint Committee's CSEC2017. He does all of this with the help and support of his wife, Tamara.

About the Technical Editor

Brian Barta, CISM, CSSLP, CISSP, CRISC, CISA, CCSK, CCSP, GROL, CEH, LSSBB, GIAC-GCIH, MCSE, is a proven leader with global security program management experience and a reputation for delivering enhanced results through the strategic implementation of organizational information security technology and process transformation. He has served as the director of Cloud Security Governance, Risk and Compliance for SaaS Cloud Security at Oracle, as well as the manager of Group Health's Governance, Risk and Compliance program where he established a program that aligned to business risks and is understood by business partners. Prior to supporting Group Health's success, he spent three years as the vulnerability management and advisory services security manager for T-Mobile improving the operational effectiveness and efficiencies of the program. Dr. Barta has more than 20 years of hands-on, in-the-weeds information security experience as well as the plain-speak, seasoned management buy-in skills required for security success in federal, state, and private industries. His passion is the measurable improvement of information security and regulatory compliance programs with a focus on security enablement of the business teams. He enjoys finding the problems and working with folks to fix them. He has also served as the chief information security officer at Washington State's Employment Security Department, information technology security architect at Washington State's Department of Corrections, and a variety of information security roles for the Department of Defense. His personal motto is "If you do what you love, you'll never work a day in your life." You can reach out to him via LinkedIn:

<https://www.linkedin.com/in/informationsecurityleadership>.

CONTENTS AT A GLANCE

Part I Secure Software Concepts

Chapter 1 Core Concepts

Chapter 2 Security Design Principles

Part II Secure Software Requirements

Chapter 3 Define Software Security Requirements

Chapter 4 Identify and Analyze Compliance Requirements

Chapter 5 Misuse and Abuse Cases

Part III Secure Software Architecture and Design

Chapter 6 Secure Software Architecture

Chapter 7 Secure Software Design

Part IV Secure Software Implementation

Chapter 8 Secure Coding Practices

Chapter 9 Analyze Code for Security Risks

Chapter 10 Implement Security Controls

Part V Secure Software Testing

Chapter 11 Security Test Cases

Chapter 12 Security Testing Strategy and Plan

Chapter 13 Software Testing and Acceptance

Part VI Secure Software Lifecycle Management

Chapter 14 Secure Configuration and Version Control

Chapter 15 Software Risk Management

Part VII Secure Software Deployment, Operations, Maintenance

Chapter 16 Secure Software Deployment

Chapter 17 Secure Software Operations and Maintenance

Part VIII Secure Software Supply Chain

Chapter 18 Software Supply Chain Risk Management

Chapter 19 Supplier Security Requirements

Part IX Appendix and Glossary

Appendix About the Online Content

Glossary

Index

CONTENTS

Acknowledgments

Introduction

Exam Objective Map

Part I Secure Software Concepts

- Chapter 1** Core Concepts
 - Confidentiality
 - Implementing Confidentiality
 - Integrity
 - Implementing Integrity
 - Availability
 - Authentication
 - Multifactor Authentication
 - Identity Management
 - Identity Provider
 - Identity Attributes
 - Certificates
 - Identity Tokens
 - SSH Keys
 - Smart Cards
 - Implementing Authentication
 - Credential Management
 - Authorization
 - Access Control Mechanisms
 - Accountability (Auditing and Logging)
 - Logging

- Syslog
- Nonrepudiation
- Secure Development Lifecycle
 - Security vs. Quality
 - Security Features != Secure Software
- Secure Development Lifecycle Components
 - Software Team Awareness and Education
 - Gates and Security Requirements
 - Bug Tracking
 - Threat Modeling
 - Fuzzing
 - Security Reviews
 - Mitigations
- Chapter Review
 - Quick Tips
 - Questions
 - Answers

Chapter 2 Security Design Principles

- System Tenets
 - Session Management
 - Exception Management
 - Configuration Management
- Secure Design Tenets
 - Good Enough Security
 - Least Privilege
 - Separation of Duties
 - Defense in Depth
 - Fail-Safe
 - Economy of Mechanism
 - Complete Mediation
 - Open Design
 - Least Common Mechanism
 - Psychological Acceptability
 - Weakest Link

- Leverage Existing Components
- Single Point of Failure
- Security Models
 - Access Control Models
 - Multilevel Security Model
 - Integrity Models
 - Information Flow Models
- Adversaries
 - Adversary Type
 - Adversary Groups
 - Threat Landscape Shift
- Chapter Review
 - Quick Tips
 - Questions
 - Answers

Part II Secure Software Requirements

Chapter 3 Define Software Security Requirements

- Functional Requirements
 - Role and User Definitions
 - Objects
 - Activities/Actions
 - Subject-Object-Activity Matrix
 - Use Cases
 - Sequencing and Timing
 - Secure Coding Standards
- Operational and Deployment Requirements
- Connecting the Dots
- Chapter Review
 - Quick Tips
 - Questions
 - Answers

Chapter 4 Identify and Analyze Compliance Requirements

Regulations and Compliance

Security Standards

ISO

NIST

FISMA

Sarbanes-Oxley

Gramm-Leach-Bliley

HIPAA and HITECH

Payment Card Industry Data Security Standard

Other Regulations

Legal Issues

Intellectual Property

Data Classification

Data States

Data Usage

Data Risk Impact

Data Lifecycle

Generation

Data Ownership

Data Owner

Data Custodian

Labeling

Sensitivity

Impact

Privacy

Privacy Policy

Personally Identifiable Information

Personal Health Information

Breach Notifications

General Data Protection Regulation

California Consumer Privacy Act 2018 (AB 375)

Privacy-Enhancing Technologies

Data Minimization

Data Masking

Tokenization
Anonymization
Pseudo-anonymization

Chapter Review

Quick Tips
Questions
Answers

- Chapter 5** Misuse and Abuse Cases
Misuse/Abuse Cases
Requirements Traceability Matrix
Software Acquisition
Definitions and Terminology
Build vs. Buy Decision
Outsourcing
Contractual Terms and Service Level Agreements
Requirements Flow Down to Suppliers/Providers
- Chapter Review
- Quick Tips
Questions
Answers

Part III Secure Software Architecture and Design

- Chapter 6** Secure Software Architecture
Perform Threat Modeling
Threat Model Development
Attack Surface Evaluation
Attack Surface Measurement
Attack Surface Minimization
Threat Intelligence
Threat Hunting
- Define the Security Architecture
Security Control Identification and Prioritization
Distributed Computing

- Service-Oriented Architecture
- Web Services
- Rich Internet Applications
- Pervasive/Ubiquitous Computing
- Embedded
- Cloud Architectures
- Mobile Applications
- Hardware Platform Concerns
- Cognitive Computing
- Control Systems

Chapter Review

- Quick Tips
- Questions
- Answers

- Chapter 7**
 - Secure Software Design
 - Performing Secure Interface Design
 - Logging
 - Protocol Design Choices
 - Performing Architectural Risk Assessment
 - Model (Nonfunctional) Security Properties and Constraints
 - Model and Classify Data
 - Types of Data
 - Structured
 - Unstructured
 - Evaluate and Select Reusable Secure Design
 - Creating a Practical Reuse Plan
 - Credential Management
 - Flow Control
 - Data Loss Prevention
 - Virtualization
 - Trusted Computing
 - Database Security
 - Programming Language Environment
 - Operating System Controls and Services

Secure Backup and Restoration Planning
Secure Data Retention, Retrieval, and Destruction
Perform Security Architecture and Design Review
Define Secure Operational Architecture
Use Secure Architecture and Design Principles, Patterns, and Tools
Chapter Review
 Quick Tips
 Questions
 Answers

Part IV Secure Software Implementation

Chapter 8 Secure Coding Practices
Declarative vs. Imperative Security
 Bootstrapping
 Cryptographic Agility
 Handling Configuration Parameters
Memory Management
 Type-Safe Practice
 Locality
Error Handling
Interface Coding
Primary Mitigations
Learning from Past Mistakes
Secure Design Principles
 Good Enough Security
 Least Privilege
 Separation of Duties
 Defense in Depth
 Fail Safe
 Economy of Mechanism
 Complete Mediation
 Open Design
 Least Common Mechanism

- Psychological Acceptability
- Weakest Link
- Leverage Existing Components
- Single Point of Failure
- Interconnectivity
 - Session Management
 - Exception Management
 - Configuration Management
- Cryptographic Failures
 - Hard-Coded Credentials
 - Missing Encryption of Sensitive Data
 - Use of a Broken or Risky Cryptographic Algorithm
 - Download of Code Without Integrity Check
 - Use of a One-Way Hash Without a Salt
- Input Validation Failures
 - Buffer Overflow
 - Canonical Form
 - Missing Defense Functions
 - Output Validation Failures
- General Programming Failures
 - Sequencing and Timing
- Technology Solutions
- Chapter Review
 - Quick Tips
 - Questions
 - Answers

Chapter 9 Analyze Code for Security Risks

- Code Analysis (Static and Dynamic)
 - Static Application Security Testing
 - Dynamic Application Security Testing
 - Interactive Application Security Testing
 - Runtime Application Self-Protection
- Code/Peer Review
- Code Review Objectives

Additional Sources of Vulnerability Information

CWE/SANS Top 25 Vulnerability Categories

OWASP Vulnerability Categories

Common Vulnerabilities and Countermeasures

 Injection Attacks

Chapter Review

 Quick Tips

 Questions

 Answers

Chapter 10 Implement Security Controls

 Security Risks

 Implement Security Controls

 Applying Security via the Build Environment

 Integrated Development Environment

 Anti-tampering Techniques

 Code Signing

 Configuration Management: Source Code and
 Versioning

 Code Obfuscation

 Defensive Coding Techniques

 Declarative vs. Programmatic Security

 Bootstrapping

 Cryptographic Agility

 Handling Configuration Parameters

 Interface Coding

 Memory Management

 Primary Mitigations

 Secure Integration of Components

 Secure Reuse of Third-Party Code or Libraries

 System-of-Systems Integration

Chapter Review

 Quick Tips

 Questions

 Answers

Part V Secure Software Testing

Chapter 11 Security Test Cases

Security Test Cases

Attack Surface Evaluation

Penetration Testing

Common Methods

Fuzzing

Scanning

Simulations

Failure Modes

Cryptographic Validation

Regression Testing

Integration Testing

Continuous Testing

Chapter Review

Quick Tips

Questions

Answers

Chapter 12 Security Testing Strategy and Plan

Develop a Security Testing Strategy and a Plan

Functional Security Testing

Unit Testing

Nonfunctional Security Testing

Testing Techniques

White-Box Testing

Black-Box Testing

Gray-Box Testing

Testing Environment

Environment

Standards

ISO/IEC 25010:2011

SSE-CMM

OSSTMM

Crowd Sourcing Chapter Review

Quick Tips
Questions
Answers

Chapter 13 Software Testing and Acceptance

Perform Verification and Validation Testing
Software Qualification Testing
Qualification Testing Hierarchy
Identify Undocumented Functionality
Analyze Security Implications of Test Results
Classify and Track Security Errors

Bug Tracking
Defects
Errors
Bug Bar
Risk Scoring

Secure Test Data
Generate Test Data
Reuse of Production Data

Chapter Review

Quick Tips
Questions
Answers

Part VI Secure Software Lifecycle Management

Chapter 14 Secure Configuration and Version Control

Secure Configuration and Version Control
Define Strategy and Roadmap
Manage Security Within a Software Development Methodology
 Security in Adaptive Methodologies
 Security in Predictive Methodologies
Identify Security Standards and Frameworks

- Define and Develop Security Documentation
- Develop Security Metrics
- Decommission Software
 - End-of-Life Policies
 - Data Disposition
- Report Security Status
- Chapter Review
 - Quick Tips
 - Questions
 - Answers

Chapter 15 Software Risk Management

- Incorporate Integrated Risk Management
 - Regulations and Compliance
 - Legal
 - Standards and Guidelines
 - Risk Management
 - Terminology
 - Technical Risk vs. Business Risk
- Promote Security Culture in Software Development
 - Security Champions
 - Security Education and Guidance
- Implement Continuous Improvement
- Chapter Review
 - Quick Tips
 - Questions
 - Answers

Part VII Secure Software Deployment, Operations, Maintenance

Chapter 16 Secure Software Deployment

- Perform Operational Risk Analysis
 - Deployment Environment
 - Personnel Training
 - Safety Criticality

System Integration
Release Software Securely
 Secure Continuous Integration and Continuous Delivery Pipeline
 Secure Software Tool Chain
 Build Artifact Verification
 Securely Store and Manage Security Data
 Credentials
 Secrets
 Keys/Certificates
 Configurations
 Ensure Secure Installation
 Bootstrapping
 Least Privilege
 Environment Hardening
 Secure Activation
 Security Policy Implementation
 Secrets Injection
Perform Post-Deployment Security Testing
Chapter Review
 Quick Tips
 Questions
 Answers

Chapter 17 Secure Software Operations and Maintenance
 Obtain Security Approval to Operate
 Perform Information Security Continuous Monitoring
 Collect and Analyze Security Observable Data
 Threat Intel
 Intrusion Detection/Response
 Secure Configuration
 Regulation Changes
 Support Incident Response
 Root-Cause Analysis
 Incident Triage

Forensics
Perform Patch Management
Perform Vulnerability Management
Runtime Protection
Support Continuity of Operations
 Backup, Archiving, Retention
 Disaster Recovery
 Resiliency
Integrate Service Level Objectives and Service Level Agreements
Chapter Review
 Quick Tips
 Questions
 Answers

Part VIII Secure Software Supply Chain

Chapter 18 Software Supply Chain Risk Management
Implement Software Supply Chain Risk Management
Analyze Security of Third-Party Software
Verify Pedigree and Provenance
 Secure Transfer
 System Sharing/Interconnections
 Code Repository Security
 Build Environment Security
 Cryptographically Hashed, Digitally Signed Components
 Right to Audit
Chapter Review
 Quick Tips
 Questions
 Answers

Chapter 19 Supplier Security Requirements
Ensure Supplier Security Requirements in the Acquisition Process
Supplier Sourcing

Supplier Transitioning
Audit of Security Policy Compliance
Vulnerability/Incident Notification, Response,
Coordination, and Reporting
Maintenance and Support Structure
Security Track Record
Support Contractual Requirements
Intellectual Property
Legal Compliance
Chapter Review
 Quick Tips
 Questions
 Answers

Part IX Appendix and Glossary

Appendix About the Online Content
System Requirements
Your Total Seminars Training Hub Account
 Privacy Notice
Single User License Terms and Conditions
TotalTester Online
Technical Support

Glossary

Index

ACKNOWLEDGMENTS

We, the authors of *CSSLP Certification All-in-One Exam Guide, Third Edition*, have many individuals who we need to acknowledge—individuals without whom this effort would not have been successful.

The list needs to start with those folks at McGraw Hill who worked tirelessly with the project’s authors and led us successfully through the minefield that is a book schedule and who took our rough chapters and drawings and turned them into a final, professional product we can be proud of. We thank the good people from the acquisitions team, Wendy Rinaldi and Emily Walters. Wendy, our acquisitions editor, navigated a myriad of problems, put up with delays and setbacks, and made life easier for the author team. This project would not have been possible without her. And we thank the editorial and production team, Janet Walden and Thomas Somers; and at KGL, our project manager, Nitesh Sharma, and the composition and illustration teams. We also thank the technical editor, Brian Barta; the copy editor, Kim Wimpsett; the proofreader, Rachel Fogelberg; and the indexer, Ted Laux, for all their attention to detail that made this a finer work after they finished with it.

We would also like to thank Mary Ann Davidson, from Oracle Corporation, as she has provided inspiration as to the importance of getting software right and the real challenges of doing it in the real world, with business and technical considerations at every turn. She championed the cause of “an ounce of prevention is worth a pound of patching” while reminding us of the challenges of regression testing and the customer value of doing the right thing. The software development world needs more champions like her.

Dan Kaminsky, a true security pioneer, taken from us way too soon, stated that “The reality of most software development is that the consequences of failure are simply nonexistent.” May this book and practitioners heed these words and seek to make them history.

INTRODUCTION

Computer security is becoming increasingly important today as we are becoming more reliant upon computers and the number of security incidents is steadily increasing. Vulnerable software is one of the root causes of many security incidents, and given the increasingly complex nature of software, this is not an issue that will be solved in the near term. Reducing the number and severity of vulnerabilities is both possible and useful in software projects. The principles behind the CSSLP certification can provide a roadmap to this goal.

Why Focus on Software Development?

Software vulnerabilities are preventable. Reducing the number and severity of vulnerabilities in software is not a trivial task; it is one that is complex and difficult to execute. Years of experience across numerous software development firms have resulted in proven methods of improving the software development process. Using these principles, development teams can produce software that has fewer vulnerabilities, and those that are found are of lesser risk. This reduces the total cost of development over the entire development lifecycle. This also improves the overall enterprise security posture of the users of the software, reducing their costs as well. Reduced risk, reduced cost, improved customer relations, and the advantages of improving the development process make the hard tasks required worth undertaking.

The Role of CSSLP

Creating and managing the necessary processes to build a secure development lifecycle is a significant task. The CSSLP credential speaks to

the knowledge needed to make this possible. Software development is a team activity, and it is one that requires a series of processes in the enterprise. The tasks required to operate within a security-focused development environment require a workforce with an enhanced skillset. In addition to their individual skills in their areas of expertise, team members need to have an understanding of how a security-enhanced software development lifecycle process works. The body of knowledge for CSSLP covers these essential elements, and whether you are a designer, developer, tester, or program manager, the body of knowledge prepares you for operating in this environment.

How to Use This Book

This book covers everything you'll need to know for (ISC)²'s CSSLP exam. Each chapter covers specific objectives and details for the exam, as defined by (ISC)². We've done our best to arrange these objectives in the order set by (ISC)². Some topics may span objectives, so be sure to use the whole book for a complete reference.

Each chapter has several components designed to effectively communicate the information you'll need for the exam:

- Sidebars are designed to point out information, tips, and stories that will be helpful in your day-to-day responsibilities. In addition, they're just downright fun sometimes. Please note that although these entries provide real-world accounts of interesting pieces of information, they are sometimes used to reinforce testable material. Don't just discount them as simply "neat"—some of the circumstances and tools described in these sidebars may prove the difference in correctly answering a question or two on the exam.
- Exam Tips are exactly what they sound like. These are included to point out a focus area you need to concentrate on for the exam. No, they are not explicit test answers. Yes, they will help you focus your study.
- Notes provide interesting tidbits of information that are relevant to the discussion and point out extra information. Just as with the sidebars, don't discount them.

The Examination

Before we get to anything else, let us be frank: *This book will help you pass your test.* We've taken great pains to ensure that everything (ISC)² has asked you to know before taking the exam is covered in the book. Software development is a real task, and the information in this book needs to be included within the context of your experience in the development process. To get the value of the material in the book, it is important to combine it with the domain knowledge of software development processes.

Speaking of the test, these exam tips should help you:

- Be sure to pay close attention to the Exam Tips in the chapters. They are there for a reason. And retake the practice exams—both the end-of-chapter exams and the electronic exams. Practice will help, trust us.
- The exam is 125 questions, all multiple choice with four answers, and you are allowed to mark and skip questions for later review. Go through the entire exam, answering the ones you know beyond a shadow of a doubt. On the ones you're not sure about, *choose an answer anyway* and mark the question for further review (you don't want to fail the exam because you ran out of time and had a bunch of questions that didn't even have an answer chosen). At the end, go back and look at the ones you've marked. Change your answer only if you are absolutely, 100 percent sure about it.
- You will, with absolute certainty, see a couple of questions that will blow your mind. On every exam there are questions you will not recognize. When you see them, don't panic. Use deductive reasoning and make your best guess. Almost every single question on this exam can be whittled down to at least 50/50 odds on a guess. There is no penalty for guessing, so answer all questions.

Finally, dear reader, thank you for picking this book. We sincerely hope your exam goes well and wish you the absolute best in your upcoming career. Learn and use the material for good, and make better software.

Exam Objective Map

The following table has been constructed to allow you to cross-reference the

official exam objectives with the objectives as they are presented and covered in this book.

Official Exam Objective	Chapter	All-in-One Coverage	Page
Domain 1: Secure Software Concepts			
1.1 Core Concepts	1	Core Concepts	3
1.2 Security Design Principles	2	Security Design Principles	31
Domain 2: Secure Software Requirements			
2.1 Define Software Security Requirements	3	Define Software Security Requirements	55
2.2 Identify and Analyze Compliance Requirements	4	Identify and Analyze Compliance Requirements	65
2.3 Identify and Analyze Data Classification Requirements	4	Identify and Analyze Compliance Requirements	65
2.4 Identify and Analyze Privacy Requirements	4	Identify and Analyze Compliance Requirements	65
2.5 Develop Misuse and Abuse Cases	5	Misuse and Abuse Cases	93
2.6 Develop Security Requirement Traceability Matrix (STRM)	5	Misuse and Abuse Cases	93
2.7 Ensure Security Requirements Flow Down to Suppliers/Providers	5	Misuse and Abuse Cases	93
Domain 3: Secure Software Architecture and Design			
3.1 Perform Threat Modeling	6	Secure Software Architecture	105
3.2 Define the Security Architecture	6	Secure Software Architecture	105
3.3 Performing Secure Interface Design	7	Secure Software Design	133
3.4 Performing Architectural Risk Assessment	7	Secure Software Design	133
3.5 Model (Non-Functional) Security Properties and Constraints	7	Secure Software Design	133

3.6 Model and Classify Data	7	Secure Software Design	133
3.7 Evaluate and Select Reusable Secure Design	7	Secure Software Design	133
3.8 Perform Security Architecture and Design Review	7	Secure Software Design	133
3.9 Define Secure Operational Architecture (e.g., deployment topology, operational interfaces)	7	Secure Software Design	133
3.10 Use Secure Architecture and Design Principles, Patterns, and Tools	7	Secure Software Design	133

Domain 4: Secure Software Implementation

4.1 Adhere to Relevant Secure Coding Practices (e.g., standards, guidelines and regulations)	8	Secure Coding Practices	157
4.2 Analyze Code for Security Risks	9	Analyze Code for Security Risks	181
4.3 Implement Security Controls (e.g., watchdogs, File Integrity Monitoring (FIM), anti-malware)	10	Implement Security Controls	199
4.4 Address Security Risks (e.g., remediation, mitigation, transfer, accept)	10	Implement Security Controls	199
4.5 Securely Reuse Third-Party Code or Libraries (e.g., Software Composition Analysis (SCA))	10	Implement Security Controls	199
4.6 Securely Integrate Components	10	Implement Security Controls	199
4.7 Apply Security During the Build Process	10	Implement Security Controls	199

Domain 5: Secure Software Testing

5.1 Develop Security Test Cases	11	Security Test Cases	217
5.2 Develop Security Testing Strategy and Plan	12	Security Testing Strategy and Plan	229
5.3 Verify and Validate Documentation (e.g., installation and setup instructions, error messages, user guides, release notes)	13	Software Testing and Acceptance	241
5.4 Identify Undocumented Functionality	13	Software Testing and Acceptance	241
5.5 Analyze Security Implications of Test Results (e.g., impact on product management, prioritization, break build criteria)	13	Software Testing and Acceptance	241

5.6 Classify and Track Security Errors	13	Software Testing and Acceptance	241
5.7 Secure Test Data	13	Software Testing and Acceptance	241
5.8 Perform Verification and Validation Testing	13	Software Testing and Acceptance	241

Domain 6: Secure Software Lifecycle Management

6.1 Secure Configuration and Version Control (e.g., hardware, software, documentation, interfaces, patching)	14	Secure Configuration and Version Control	259
6.2 Define Strategy and Roadmap	14	Secure Configuration and Version Control	259
6.3 Manage Security Within a Software Development Methodology	14	Secure Configuration and Version Control	259
6.4 Identify Security Standards and Frameworks	14	Secure Configuration and Version Control	259
6.5 Define and Develop Security Documentation	14	Secure Configuration and Version Control	259
6.6 Develop Security Metrics (e.g., defects per line of code, criticality level, average remediation time, complexity)	14	Secure Configuration and Version Control	259
6.7 Decommission Software	14	Secure Configuration and Version Control	259
6.8 Report Security Status (e.g., reports, dashboards, feedback loops)	14	Secure Configuration and Version Control	259
6.9 Incorporate Integrated Risk Management (IRM)	15	Software Risk Management	273
6.10 Promote Security Culture in Software Development	15	Software Risk Management	273
6.11 Implement Continuous Improvement (e.g., retrospective, lessons learned)	15	Software Risk Management	273

Domain 7: Secure Software Deployment, Operations, Maintenance

7.1 Perform Operational Risk Analysis	16	Secure Software Deployment	285
7.2 Release Software Securely	16	Secure Software Deployment	285
7.3 Securely Store and Manage Security Data	16	Secure Software Deployment	285
7.4 Ensure Secure Installation	16	Secure Software Deployment	285

7.5 Perform Post-Deployment Security Testing	16	Secure Software Deployment	285
7.6 Obtain Security Approval to Operate (e.g., risk acceptance, sign-off at appropriate level)	17	Secure Software Operations and Maintenance	301
7.7 Perform Information Security Continuous Monitoring (ISCM)	17	Secure Software Operations and Maintenance	301
7.8 Support Incident Response	17	Secure Software Operations and Maintenance	301
7.9 Perform Patch Management (e.g., secure release, testing)	17	Secure Software Operations and Maintenance	301
7.10 Perform Vulnerability Management (e.g., scanning, tracking, triaging)	17	Secure Software Operations and Maintenance	301
7.11 Runtime Protection (e.g., Runtime Application Self-Protection (RASP), Web Application Firewall (WAF), Address Space Layout Randomization (ASLR))	17	Secure Software Operations and Maintenance	301
7.12 Support Continuity of Operations	17	Secure Software Operations and Maintenance	301
7.13 Integrate Service Level Objectives (SLO) and Service Level Agreements (SLA) (e.g., maintenance, performance, availability, qualified personnel)	17	Secure Software Operations and Maintenance	301

Domain 8: Secure Software Supply Chain

8.1 Implement Software Supply Chain Risk Management	18	Software Supply Chain Risk Management	317
8.2 Analyze Security of Third-Party Software	18	Software Supply Chain Risk Management	317
8.3 Verify Pedigree and Provenance	18	Software Supply Chain Risk Management	317
8.4 Ensure Supplier Security Requirements in the Acquisition Process	19	Supplier Security Requirements	327
8.5 Support contractual requirements (e.g., Intellectual Property (IP) ownership, code escrow, liability, warranty, End-User License Agreement (EULA), Service Level Agreements (SLA))	19	Supplier Security Requirements	327

CSSLP Version 3 (2020)

In the summer of 2020, (ISC)² updated the domains of the CSSLP exam to more accurately reflect the current practice of secure development. This action was taken as a result of an analysis of job task analysis data. There are minor changes in the knowledge, skills, and abilities associated with the CSSLP domains. The domain weighting also has changed, as illustrated next.

CSSLP Certification All-in-One Exam Guide, Third Edition takes these changes into account and adds detailed information across all domains. The length and time for the exam will be the same, but the content is updated to reflect the new domain weightings.

Domains	Weight
1. Secure Software Concepts	10%
2. Secure Software Requirements	14%
3. Secure Software Architecture and Design	14%
4. Secure Software Implementation	14%
5. Secure Software Testing	14%
6. Secure Software Lifecycle Management	11%
7. Secure Software Deployment, Operations, Maintenance	12%
8. Secure Software Supply Chain	11%
Total	100%

PART I

Secure Software Concepts

- **Chapter 1** Core Concepts
- **Chapter 2** Security Design Principles

Core Concepts

In this chapter you will

- Learn about confidentiality, integrity, and availability
 - Understand authentication and authorization
 - Explore accountability
 - Learn about nonrepudiation
-

Secure software development is intimately tied to the information security domain. For members of the software development team to develop secure software, a reasonable knowledge of security principles is required. The first knowledge domain area, Secure Software Concepts, comprises a collection of principles, tenets, and guidelines from the information security domain. Understanding these concepts as they apply to software development is a foundation of secure software development.

Security can be defined in many ways, depending upon the specific discipline that it is being viewed from. From an information and software development point of view, some specific attributes are commonly used to describe the actions associated with security: confidentiality, integrity, and availability (CIA). A second set of action-oriented elements, authentication, authorization, and accountability, provide a more complete description of the desired tasks associated with the information security activity. A final term, *nonrepudiation*, describes an act that one can accomplish when using the previous elements. An early design decision is determining what aspects of protection are required for data elements and how they will be employed.



EXAM TIP The term CIA is commonly used in the security industry to refer to confidentiality, integrity, and availability. Understanding the application of these three terms is important for the exam.

Confidentiality

Confidentiality is the concept of preventing the disclosure of information to unauthorized parties. Keeping secrets secret is the core concept of confidentiality. The identification of authorized parties makes the attainment of confidentiality dependent upon the concept of authorization, which is presented later in this chapter. There are numerous methods of keeping data confidential, including access controls and encryption. The technique employed to achieve confidentiality depends upon whether the data is at rest, in transit, or in use. Access controls are typically preferred for data in use and at rest, while encryption is common for data in transit and at rest.

Issues involving protecting data from unauthorized disclosure can be decomposed to confidentiality requirements. The policy will define confidentiality requirements in terms of who can exchange what information between what endpoints. The key elements to determine are who the authorized users are and for what specific data elements or types. It can then be assumed that any other users would be unauthorized. Based on this set of information, the following items can be enumerated:

- Who is authorized to see what specific data elements
- What mechanism should be employed to enforce confidentiality
- What are the business requirements with respect to data collection, transfer, storage, and use with respect to confidentiality

Implementing Confidentiality

Confidentiality is the protection of information from observation by unauthorized parties. This is done primarily through the use of encryption to make the data unreadable. Encryption acts by scrambling the data in a

manner that hides the true data values from observation. Encryption is a process that requires an algorithm and a key, and those who possess the correct keys can access the data. Confidentiality can be achieved by several means, both overt and covert. The typical mechanism is encryption, and it is crucial that best practices be employed in this aspect, for the typical failure of cryptographic protections falls into a few common categories. Encryption relies upon two elements, an algorithm and a key. Failure to use algorithms from approved cryptographic libraries, i.e., rolling your own crypto, almost always leads to failure. Also, failing to protect the key leads to failure.

The challenge to covert methods is simple: once found, they are no longer covert, and the secrecy value is summarily lost. Modern cryptographic processes are built upon a foundation of overt implementation of approved encryption algorithms, with steps taken to secure the key.



NOTE For cryptography to be successful, it must be properly implemented. Use only approved cryptographic libraries and functions. Never create your own implementations. Always protect the key, and never store it in code or a file where it can be stolen.

Integrity

Integrity is similar to confidentiality, except rather than protecting the data from unauthorized access, *integrity* refers to protecting the data from unauthorized alteration. Unauthorized alteration is a more fine-grained control than simply authorizing access. Users can be authorized to view information but not alter it, so integrity controls require an authorization scheme that controls update and delete operations. For some systems, protecting the data from observation by unauthorized parties is critical, whereas in other systems, it is important to protect the data from unauthorized alteration. Controlling alterations, including deletions, can be an essential element in a system's stability and reliability. Integrity can also play a role in the determination of authenticity.

Issues involving protecting data from unauthorized alteration can be

decomposed to integrity requirements. The policy will define integrity requirements in terms of who can alter which information elements between what endpoints. The key elements to determine are who is authorized to alter which specific data streams. It can then be assumed that any other users' alterations would be unauthorized. Based on this set of information, the following items can be enumerated:

- Who is authorized to alter which specific data elements
- What mechanism should be employed to detect errors and enforce integrity
- What are the business requirements with respect to data collection, transfer, storage, and use with respect to integrity

Because integrity can be expressed in the form of data errors, consideration should be given to requirements that can be used as a basis for monitoring error conditions.

Implementing Integrity

Integrity refers to protecting data from unauthorized alteration. Alterations can come in the form of changing a value or deleting a value. Integrity builds upon confidentiality, for if data is to be altered or deleted, then it is probably also visible to the user account. Users can be authorized to view information, but not alter it, so integrity controls require an authorization scheme that controls update and delete operations.

Protecting integrity can be done through the use of access control mechanisms. Individual specific levels of access with respect to read, write, and delete can be defined. Applying controls is just a piece of the integrity puzzle. The other aspect involves determining whether data has been altered or not. In some cases, this is just as important, if not more so, as the access control part. Determining whether the information has been altered or not requires some form of control other than simple access control.

If there is a requirement to verify the integrity of a data element throughout its lifecycle, the cryptographic function of hashing can be used. Hash functions can determine whether single or multiple bits of data change from the original form. The principle is simple: Take the original data, run it through a hash function, and record the result. At any subsequent time,

running the current value of the data through the same hash function will produce a current digest value. If the two digest values match, then the data is unchanged. This can be for something small, such as a few bytes, to an entire file system—in all cases, the digest size is fixed based on the hash algorithm.

The use of cryptographic methods to digitally sign an item, or in the case of finished code, code-signing, proved a method of ensuring that an item is free of unauthorized modifications, or has not changed since signing. The addition of the identity of the signing party helps users determine authenticity.

The design issue is, how does one implement this functionality? When are hash digests calculated? How are the digests stored and transmitted by a separate channel to the party checking? Adding the necessary elements to allow hash-based checking of integrity requires the designing of the necessary apparatus for parties to be able to verify integrity. In some cases, like in Microsoft Update Service, this entire mechanism is designed into the transfer mechanism so that it is invisible to the end user. The design phase is where these decisions need to be made so that the coding phase can properly implement the desired behavior.

Availability

Access to systems by authorized personnel can be expressed as the system's *availability*. Availability is concerned with two issues: ensuring systems are available for authorized users when they require those systems and denying access to unauthorized users at all other times. Policy elements associated with determining access can be translated into the availability requirements. Availability is an often-misunderstood attribute, but its value is determined by the criticality of the data and its purpose in the system. For systems such as e-mail or web browsing, temporary availability issues may not be an issue at all. For IT systems that are controlling large industrial plants, such as refineries, availability may be the most important attribute. The criticality of data and its use in the system are critical factors in determining a system's availability. The challenge in system definition and design is to determine the correct level of availability for the data elements of the system. Common availability issues include denying illegitimate access to systems as well as preventing availability attacks such as denial of service.



NOTE If a system prevents a user from seeing a document they are authorized to see, this is a security failure, specifically an availability failure. Calling it “security” is wrong; it is a failure, and in the case of data access it can lead to wrong decisions.

The objective of security is to apply the appropriate measures to achieve a desired risk profile for a system. One of the challenges in defining the appropriate control objectives for a system is classifying and determining the appropriate balance of the levels of confidentiality, integrity, and availability in a system across the data elements. Although the attributes are different, they are not necessarily contradictory. They all require resources, and determining the correct balance between them is a key challenge early in the requirements and design process.

Confidentiality, Integrity, and Availability Requirements

At the core of software is data, and one of the key policy elements that needs to be defined for the software development team is the required level of data protection. Defining the required specific level of protection in terms of confidentiality, integrity, and availability is a key security requirement. Per Federal Information Processing Standard (FIPS) 199 and 44 U.S.C., Sec. 3542, the terms confidentiality, integrity, and availability are defined as follows:

CONFIDENTIALITY

“Preserving authorized restrictions on information access and disclosure, including means for protecting personal privacy and proprietary information...”

A loss of confidentiality is the unauthorized disclosure of information.

INTEGRITY

“Guarding against improper information modification or

destruction, and includes ensuring information non-repudiation and authenticity...”

A loss of integrity is the unauthorized modification or destruction of information.

AVAILABILITY

“Ensuring timely and reliable access to and use of information...”

A loss of availability is the disruption of access to or use of information or an information system.

Authentication

Authentication is the process of determining the identity of a user. All processes in a computer system have an identity assigned to them so that a differentiation of security functionality can be employed. Authentication is a foundational element of security, as it provides the means to define the separation of users by allowing the differentiation between authorized and unauthorized users. In systems where all users share a single account, they share the authentication and identity associated with that account.

It is the job of authorization mechanisms to ensure that only valid users are permitted to perform specific allowed actions. Authentication, on the other hand, deals with verifying the identity of a subject. To help understand the difference, consider the example of an individual attempting to log in to a computer system or network. Authentication is the process used to verify to the computer system or network that the individual is who they claim to be. Authorization refers to the rules that determine what that individual can do on the computer system or network after being authenticated. Three general methods are used in authentication. To verify their identity, a user can provide

- Something you know
- Something you have
- Something about you (something that you are)

The most common authentication mechanism is to provide something that only you, the valid user, should know. The most common example of

something you know is the use of a userid (or username) and password. In theory, since you are not supposed to share your password with anybody else, only you should know your password, and thus by providing it you are proving to the system that you are who you claim to be. Unfortunately, for a variety of reasons, such as the fact that people tend to choose very poor and easily guessed passwords or share them, this technique to provide authentication is not as reliable as it should be. Other, more secure authentication mechanisms are consequently being developed and deployed.



NOTE Originally published by the U.S. government in one of the “rainbow series” of manuals on computer security, the categories of shared “secrets” are as follows:

- What users know (such as a password)
- What users have (such as tokens)
- What users are (static biometrics such as fingerprints or iris pattern)

Today, because of technological advances, new categories have emerged, patterned after subconscious behaviors and measurable attributes:

- What users do (dynamic biometrics such as typing patterns or gait)
- Where a user is (actual physical location)

Another common method to provide authentication involves the use of something that only valid users should have in their possession, commonly referred to as a *token*. A physical-world example of this is the simple lock and key. Only those individuals with the correct key will be able to open the lock and thus achieve admittance to your house, car, office, or whatever the lock was protecting. For computer systems, the token frequently holds a cryptographic element that identifies the user. The problem with tokens is that people can lose them, which means they can't log in to the system, and somebody else who finds the key may then be able to access the system, even though they are not authorized. To address the lost token problem, a combination of the something-you-know and something-you-have methods is used—requiring a password or personal identification number (PIN) in

addition to the token. The key is useless unless you know this code. An example of this is the automatic teller machine (ATM) card most of us carry. The card is associated with a PIN that only you should know. Knowing the PIN without having the card is useless, just as having the card without knowing the PIN will also not provide you access to your account. Properly configured tokens can provide high levels of security at an expense only slightly higher than passwords.

The third authentication method involves using something that is unique about the user. We are used to this concept from television police dramas, where a person's fingerprints or a sample of their DNA can be used to identify them. The field of authentication that uses something about you or something that you are is known as *biometrics*. A number of different mechanisms can be used to accomplish this form of authentication, such as a voice print, a retinal scan, or hand geometry. The downside to these methods is the requirement of additional hardware and the lack of specificity that can be achieved with other methods.



EXAM TIP Three general factors are used in authentication. To verify your identity, you can provide something you know, something you have, or something about you (something that you are). When two different factors of this list are used together, it is referred to as *two-factor authentication*.

Multifactor Authentication

Multifactor authentication (or multiple-factor authentication) is simply the combination of two or more types of authentication. Five broad categories of authentication can be used: what you are (for example, biometrics), what you have (for instance, tokens), what you know (passwords and other information), somewhere you are (location), and something you do (physical performance). Two-factor authentication combines any two of these before granting access. An example would be a card reader that then turns on a fingerprint scanner—if your fingerprint matches the one on file for the card, you are granted access. Three-factor authentication would combine all three types, such as a smart card reader that asks for a PIN before enabling a retina

scanner. If all three correspond to a valid user in the computer database, access is granted.



NOTE Whereas two-factor authentication combines any two methods, matching items such as a token with a biometric, three-factor authentication combines any three, such as a passcode, biometric, and a token.

Multifactor authentication methods greatly enhance security by making it difficult for an attacker to obtain all the correct materials for authentication. They also protect against the risk of stolen tokens, as the attacker must have the correct biometric, password, or both. More important, multifactor authentication enhances the security of biometric systems by protecting against a stolen biometric. Changing the token makes the biometric useless unless the attacker can steal the new token. It also reduces false positives by trying to match the supplied biometric with the one that is associated with the supplied token. This prevents the computer from seeking a match using the entire database of biometrics. Using multiple factors is one of the best ways to ensure proper authentication and access control.

Identity Management

Determining a user's identity requires an authentication process. Authentication is an identity verification process that attempts to determine whether users are who they say they are. Typically, a user enters some token to demonstrate to the system that they are who they purport to be. This requires a previous identification step, where the original record of the user is created. The identification step requires the following issues to be determined by requirements:

- Mechanism to be used for identity
- Management of identities, including reaffirmations

The mechanism to be used for identity sets the means to be used to authenticate users when services are requested. Identity is a step usually taken

only once—when a user is established in the authentication system and a shared secret is created. This shared secret is what will be used to confirm identity during authentication events.

Identity management (IDM) is the comprehensive set of services related to managing the use of identities as part of an access control solution. Strictly speaking, the identity process is one where a user establishes their identity. Authentication is the act of verifying the supplied credentials against the set established during the identity process. The term *identity management* refers to the set of policies, processes, and technologies for managing digital identity information. *Identity and access management* (IAM) is another term associated with the comprehensive set of policies, processes, and technologies for managing digital identity information.

Identity management is a set of processes associated with the identity lifecycle, including the provisioning, management, and deprovisioning of identities. The provisioning step involves the creation of a digital identity from an actual identity. The source of the actual identity can be a person, a process, an entity, or virtually anything. The identity process binds some form of secret to the digital identity so that at future times, the identity can be verified. The secret that is used to verify identity is an item deserving specific attention as part of the development process. Protecting the secret, yet making it usable, are foundational elements associated with the activity.

In a scalable system, management of identities and the associated activities needs to be automated. A large number of identity functions needs to be handled in a comprehensive fashion. Changes to identities, the addition and removal of roles, changes to rights and privileges associated with roles or identities—all of these items need to be done securely and logged appropriately. The complexity of the requirements makes the use of existing enterprise systems an attractive option when appropriate.



EXAM TIP Security controls are audited annually under Sarbanes-Oxley (SOX) section 404, and IAM controls are certainly security controls. Designing and building IAM controls to support this operational issue is a good business practice.

Identity management can be accomplished through third-party programs that enhance the operating system offerings in this area. The standard operating system implementation leaves much to be desired in the management of user-selected provisioning or changes to identity metadata. Many enterprises have third-party enterprise-class IDM systems that provide services such as password resets, password synchronization, single sign-on, and multiple identity methods.

Having automated password resets can free up significant help-desk time and provide faster service to users who have forgotten their password. Automated password reset systems require a reasonable set of challenges to verify that the person requesting the reset is authorized to do so. Then, the reset must occur in a way that does not expose the old password. E-mail resets via a uniform resource locator (URL) are one common method employed for the reset operation. Users can have multiple passwords on different systems, complicating the user activity. Password synchronization systems allow a user to synchronize a set of passwords across connected but different identity systems, making it easier for users to access the systems. Single sign-on is an industrial-strength version of synchronization. Users enter their credentials into one system, and it connects to the other systems, authenticating based on the entered credentials.

Identity Provider

The term *identity provider* (IdP) is used to denote a system or service that creates, maintains, and manages identity information. IdPs can range in scale and scope—from operating for a single system to operating across an enterprise. Additionally, they can be operated locally, distributed, or federated, depending on the specific solution. Multiple standards have been employed to achieve these services, including those built on the Security Assertion Markup Language (SAML), OpenID, and OAuth.



TIP The IdP creates, manages, and is responsible for authenticating identity.

Identity Attributes

How would you describe the elements of an identity? *Identity attributes* are the specific characteristics of an identity—name, department, location, login ID, identification number, e-mail address, and so on—that are used to accurately describe a specific entity. These elements are needed if one is to store identity information in some form of directory, such as an LDAP directory. The particulars of a schema need to be considered to include attributes for people, equipment (servers and devices), and services (apps and programs), as any of these can have an identity in a system. The details of schemas have already been taken care of via Active Directory, various IdPs, and so on, so this is not something that needs to be created; however, it does need to be understood.

Certificates

Certificate-based authentication is a means of proving identity via the presentation of a certificate. *Certificates* offer a method of establishing authenticity of specific objects such as an individual's public key or downloaded software. A digital certificate is a digital file that is sent as an attachment to a message and is used to verify that the message did indeed come from the entity it claims to have come from. Using a digital certificate is a verifiable means of establishing possession of an item (specifically, the certificate). When the certificate is held within a store that prevents tampering or extraction, this becomes a reliable means of identification, especially when combined with an additional factor such as something you know or a biometric.

Identity Tokens

An *Identity Token* is a security token that represents claims about the authentication of an entity by a specific authentication scheme. An *access token* is a form of identity token in physical form that identifies specific access rights and, in authentication, falls into the “something you have” factor. The advantages of token-based systems include the fact that any token can be deleted from the system without affecting any other token or the rest of the system. The tokens themselves can also be grouped in multiple ways to provide different access levels to different groups of people. Smart cards can

also be used to carry identification tokens. The primary drawback of token-based authentication is that only the token is being authenticated. Therefore, the theft of the token could grant anyone who possesses the token access to what the system protects.

The risk of theft of the token can be offset by the use of multifactor authentication. One of the ways that people have tried to achieve multifactor authentication is to add a biometric factor to the system. A less expensive alternative is to use hardware tokens in a challenge/response authentication process. In this way, the token functions as both a “something you have” and “something you know” authentication mechanism. Several variations on this type of device exist, but they all work on the same basic principles. The device has an LCD screen and may or may not have a numeric keypad. Devices without a keypad will display a password (often just a sequence of numbers) that changes at a constant interval, usually about every 60 seconds. When an individual attempts to log in to a system, they enter their own user ID number and then the number that is displayed on the LCD. These two numbers are either entered separately or concatenated. The user’s own ID number is secret, and this prevents someone from using a lost device. The system knows which device the user has and is synchronized with it so that it will know the number that should have been displayed. Since this number is constantly changing, a potential attacker who is able to see the sequence will not be able to use it later, since the code will have changed.

SSH Keys

SSH keys are access credentials used by the Secure Shell (SSH) protocol. They function like usernames and passwords, but SSH keys are primarily used for automated processes and services. SSH keys are also used in implementing single sign-on (SSO) systems used by system administrators. SSH keys are exchanged using public key cryptography, and the keys themselves are digital keys.

Smart Cards

Smart cards are devices that store cryptographic tokens associated with an identity. The form factor is commonly a physical card, credit card sized, that contains an embedded chip that has various electronic components to act as a

physical carrier of information.

Implementing Authentication

Authentication is the process of verifying that a user is who they claim to be and applying the correct values in the access control system. The level of required integration is high, from the storage systems that store the credentials and the access control information to the transparent handling of the information establishing or denying the validity of a credential match. When referring to authentication, one is referring to the process of verification of an identity. When one refers to the authentication system, one is typically referring to the underlying operating system aspect, not the third-party application that sits on top.

Authentication is, therefore, a validation that the user is presenting the known shared secret. This requires the establishment of a specific authentication method and a means of protecting the shared secret from disclosure. Other issues of authentication that need to be resolved include

- Method of authentication management
- Strength of authentication method

An authentication system may be as simple as a plaintext password system. An example of this would be basic authentication of a web-based system, where credentials are passed in the clear. For some uses, where no real security requirements exist, this provides sufficient control. Basic authentication provides no protection for credentials being passed between systems, so in reality, there is no security provided by this authentication method. For systems where more demanding security is needed, a more complicated system such as the Kerberos system may be employed with its encrypted secrets.

In all cases, authentication systems depend on some unique bit of information known only to the party being authenticated and the authentication system, a shared secret established during the identification phase. To protect this shared secret, in most cases, it is represented by a token such as a salted hash. This enables manipulation without risk of disclosure.

The concept of tokenization can be extended to other shared secrets, such as biometrics. A biometric can be converted to a specific representative code,

which can then be treated like any other secret. Another form of tokenization is in the use of a security token or smart card system.

In the end, the requirement for authentication depends upon the level of security needed, the scale of users, and the management of users, coupled with the specific uses of the authentication. For a given system, multiple methods may be used; for example, high-value customers may use two-factor authentication, whereas visitors may have only a simple single factor. System administrators may require stronger authentication based on their higher access levels. The specifics for all of these individual circumstances can be decomposed from policy-driven requirements.



EXAM TIP Most authentication involves one party authenticating the other, as in a bank authenticating an account holder. Mutual authentication involves both parties authenticating each other at the same time, an essential process to prevent some forms of man-in-the-middle attacks.

Authentication systems come in a variety of sizes and types. Several different elements can be used as secrets as part of the authentication process. Passwords, tokens, biometrics, smart cards—the list can be long. The types can be categorized as something you know, something you have, or something you are. The application of one or more of these factors simultaneously for identity verification is a standard process in virtually all computing systems.

The underlying mechanism has some best-practice safeguards that should be included in a system. Mechanisms such as an escalating time lock-out after a given number of successive failures, logging of all attempts (both successful and failed), and integration with the authorization system once authentication is successful are common protections. Password/token reset, account recovery, periodic changes, and password strength issues are just some of the myriad of functionalities that need to be encapsulated in an authentication system.



NOTE Passwords and other verification credentials, such as PINs, passphrases, token values, etc., are secrets and should never be accessible by anyone, including system administrators. Cryptography allows secrets to remain secret and still be used. If a system can e-mail you your password, it is not stored properly; disclosure should be impossible.

Numerous technologies are in use for authentication and authorization. Federated ID systems allow users to connect to systems through known systems. The ability to use your Facebook account to log in to another system is a useful convenience for many users. A known user experience (UX) interface and simple-to-use method for users to have a single sign-on environment, a federated ID system can use best-practice authentication systems. There are two main parties in these systems: a relying party (RP) and an identity provider (IdP). The user wants access to an RP and has credentials established on an IdP. As shown in [Figure 1-1](#), a trust relationship exists between the RP and the IdP, and through data transfers between these parties, access can be granted.

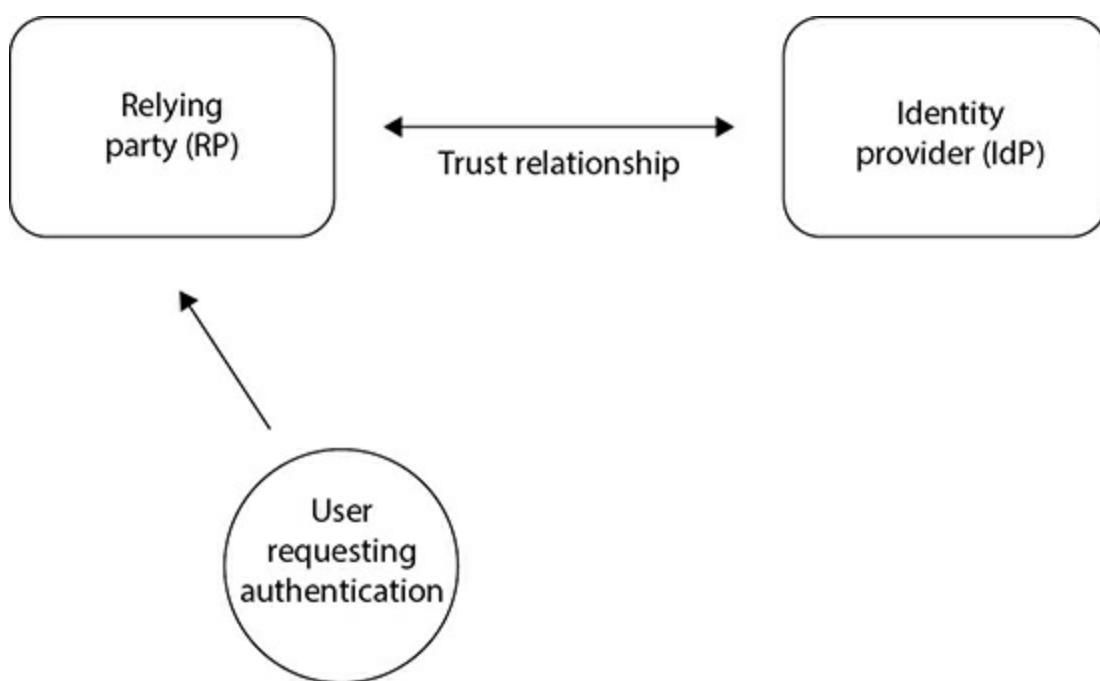


Figure 1-1 RP and IdP relationships

Two of the more prevalent systems are OpenID and OAuth. OpenID was created for federated authentication, specifically to allow a third party to authenticate your users for you by using accounts that users already have. The OpenID protocol enables websites or applications (consumers) to grant access to their own applications by using another service or application (provider) for authentication. This can be done without requiring users to maintain a separate account/profile with the consumers.

OAuth was created to eliminate the need for users to share their passwords with third-party applications. The OAuth protocol enables websites or applications (consumers) to access protected resources from a web service (service provider) via an application programming interface (API), without requiring users to disclose their service provider credentials to the consumers. [Figure 1-2](#) highlights some of the differences and similarities between OpenID and OAuth.

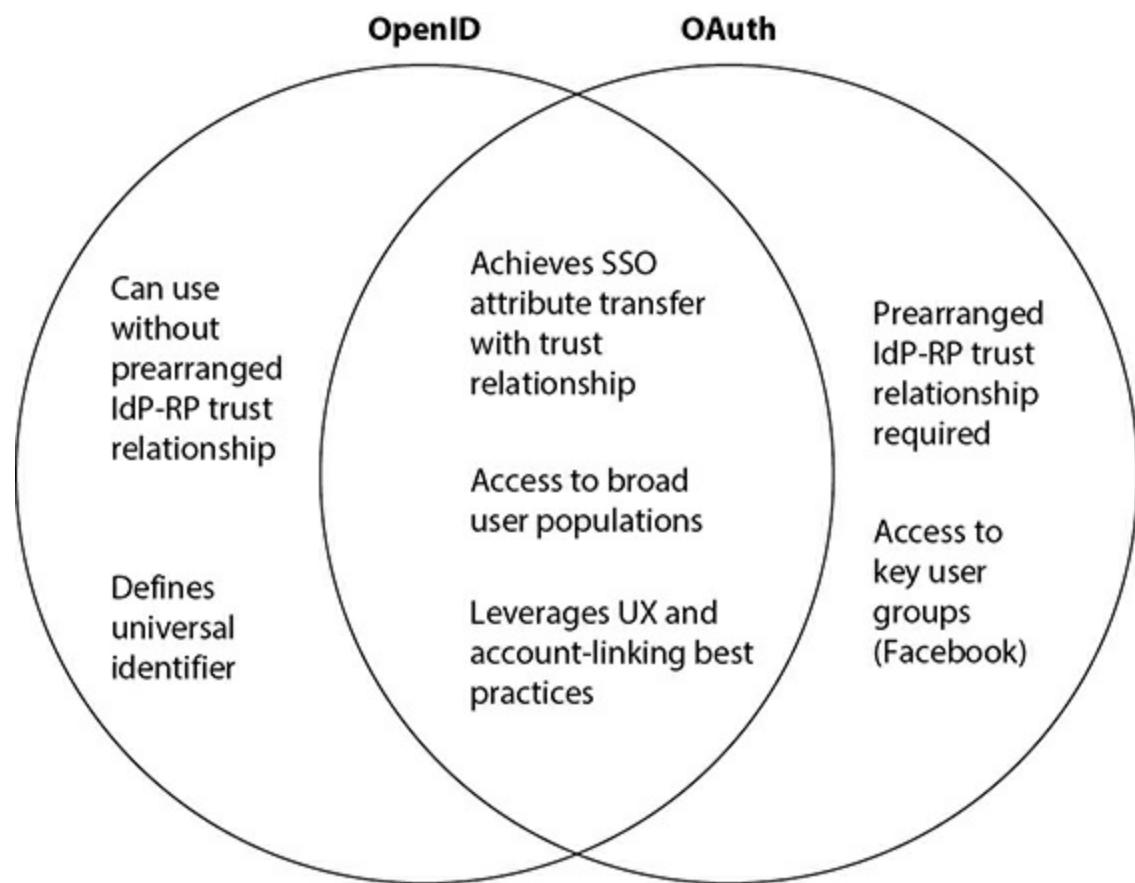


Figure 1-2 OpenID vs. OAuth

Both OpenID (for authentication) and OAuth (for authorization) accomplish many of the same things. Each protocol provides a different set of features, which are required by their primary objective, but essentially, they are interchangeable. At their core, both protocols have an assertion verification method. They differ in that OpenID is limited to the “this is who I am” assertion, while OAuth provides an “access token” that can be exchanged for any supported assertion via an API.

Credential Management

There are numerous methods of authentication, and each has its own set of credentials that require management. The identifying information that is provided by a user as part of their claim to be an authorized user is sensitive data and requires significant protection. The identifying information is frequently referred to as *credentials*. These credentials can be in the form of a passed secret, typically a password. Other common forms include digital strings that are held by hardware tokens or devices, biometrics, and certificates. Each of these forms has advantages and disadvantages.

Each set of credentials, regardless of the source, requires safekeeping on the part of the receiving entity. Managing these credentials includes tasks such as credential generation, storage, synchronization, reset, and revocation. Because of the sensitive nature of manipulating credentials, all of these activities should be logged.

X.509 Credentials

X.509 refers to a series of standards associated with the manipulation of certificates used to transfer asymmetric keys between parties in a verifiable manner. A digital certificate binds an individual’s identity to a public key, and it contains all the information a receiver needs to be assured of the identity of the public key owner. After a registration authority (RA) verifies an individual’s identity, the certificate authority (CA) generates the digital certificate. The digital certificate can contain the information necessary to facilitate authentication.

X.509 Digital Certificate Fields

The following fields are included within an X.509 digital certificate:

- **Version number** Identifies the version of the X.509 standard that was followed to create the certificate; indicates the format and fields that can be used.
- **Serial number** Provides a unique number identifying this one specific certificate issued by a particular CA.
- **Signature algorithm** Specifies the hashing and digital signature algorithms used to digitally sign the certificate.
- **Issuer** Identifies the CA that generated and digitally signed the certificate.
- **Validity** Specifies the dates through which the certificate is valid for use.
- **Subject** Specifies the owner of the certificate.
- **Public key** Identifies the public key being bound to the certified subject; also identifies the algorithm used to create the private/public key pair.
- **Certificate usage** Specifies the approved use of the certificate, which dictates intended use of this public key.
- **Extensions** Allow additional data to be encoded into the certificate to expand its functionality. Companies can customize the use of certificates within their environments by using these extensions. X.509 version 3 has expanded the extension possibilities.

Certificates are created and formatted based on the X.509 standard, which outlines the necessary fields of a certificate and the possible values that can be inserted into the fields. As of this writing, X.509 version 3 is the most current version of the standard. X.509 is a standard of the International Telecommunication Union (www.itu.int). The IETF's Public-Key Infrastructure (X.509), or PKIX, working group has adapted the X.509 standard to the more flexible organization of the Internet, as specified in RFC 3280, and is commonly referred to as PKIX for Public Key Infrastructure X.509.

The public key infrastructure (PKI) associated with certificates enables the passing and verification of these digital elements between firms. Because certificates are cryptographically signed, elements within them are protected from unauthorized alteration and can have their source verified. Building out a complete PKI infrastructure is a complex endeavor, requiring many different levels of protection to ensure that only authorized entities are permitted to make changes to the certification.

Setting up a functioning and secure PKI solution involves many parts, including certificate authorities, registration authorities, and certificate revocation mechanisms, meaning either certificate revocation lists (CRLs) or the Online Certificate Status Protocol (OCSP).

Figure 1-3 shows the actual values of the different certificate fields for a particular certificate in a web browser. The version of this certificate is V3 (X.509 v3), and the serial number is also listed—this number is unique for each certificate that is created by a specific CA. The CA used the SHA1 hashing algorithm to create the message digest value, and it then signed it using the CA’s private key, which used the RSA algorithm.

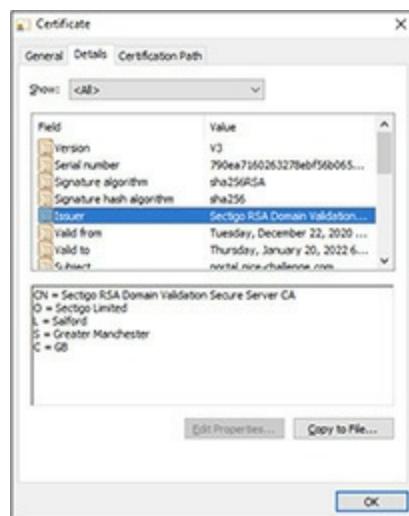


Figure 1-3 Digital certificate

X.509 certificates provide a wide range of benefits to any application that needs to work with public key cryptography. Certificates provide a standard means of passing keys, a standard that is accepted by virtually every provider and consumer of public keys. This makes X.509 a widely used and proven technology.

Single Sign-On

Single sign-on makes it possible for a user, after authentication, to have their credentials reused on other applications without the user re-entering the secret. To achieve this, it is necessary to store the credentials outside of the application and then reuse the credentials against another system. There are a number of ways that this can be accomplished, but two of the most popular and accepted methods for sharing authentication information are Kerberos and Security Assertion Markup Language. The OpenID protocol has proven to be a well-vetted and secure protocol for SSO. However, as with all technologies, security vulnerabilities can still occur due to misuse or misunderstanding of the technology.

The key concept with respect to SSO is federation. In a federated authentication system, users can log in to one site and access another or affiliated site without re-entering credentials. The primary objective of federation is user convenience. Authentication is all about trust, and federated trust is difficult to establish. SSO can be challenging to implement, and because of trust issues, it is not an authentication panacea. As in all risk-based transactions, a balance must be achieved between the objectives and the risks. SSO-based systems can create single-point-of-failure scenarios, so for certain high-risk implementations, their use is not recommended.

Authorization

After the authentication system identifies a user, the authorization system takes over and applies the predetermined access levels to the user.

Authorization is the process of applying access control rules to a user process and determining whether a particular user process can access an object. There are numerous forms of access control systems, which are covered later in the chapter. Three elements are used in the discussion of authorization: a requestor (sometimes referred to as the *subject*), the object, and the type or level of access to be granted. The authentication system identifies the subject to be one of a known set of subjects associated with a system. When a subject requests access to an object, be it a file, a program, an item of data, or any other resource, the authorization system makes the access determination as to grant or deny access. A third element is the type of access requested, with the common forms being read, write, create, delete, or the right to grant access

rights to other subjects.

Once a user is positively identified by the authentication system, an authorization determination still needs to be made. The best method of decomposing authorization follows a subject-object-activity model. A subject is a user that has been authenticated. An object is the item that a user wants to perform an action on—it can be a file, a system, a database entry, a web page, basically any resource whatsoever. The activity is the desired action that the subject wants to invoke with the object. For file systems, this can be read, write, execute, delete, and so on. With database entries, there are basically the same options.

At this point, the loop back to the requirements associated with CIA is complete. The instantiation of the CIA requirements is performed by the authorization system. Whether a simple access control mechanism or a more sophisticated granular control such as a database access lattice, the concept is the same. The authorization system implements the control of access.

Access Control Mechanisms

The term *access control* has been used to describe a variety of protection schemes. It sometimes refers to all security features used to prevent unauthorized access to a computer system or network—or even a network resource such as a printer. In this sense, it may be confused with authentication. More properly, access is the ability of a subject (such as an individual or a process running on a computer system) to interact with an object (such as a file or hardware device). Once the individual has verified their identity, access controls regulate what the individual can actually do on the system. Just because a person is granted entry to the system does not mean they should have access to all the data the system contains.



NOTE It may seem that access control and authentication are two ways to describe the same protection mechanism. This, however, is not the case. Authentication provides a way to verify to the computer who the user is. Once the user has been authenticated, the access controls decide what

operations the user can perform. The two go hand in hand but are not the same thing.

Access control systems exist to implement access control models. Different access control models are used based on the scale and scope of the elements of the subject-object-activity relationship.



NOTE Common access control models include mandatory access control (MAC), discretionary access control (DAC), role-based access control (RBAC), and rule-based access control (RBAC). These are covered in the next chapter.

Accountability (Auditing and Logging)

Accounting is a means of measuring activity. *Accountability* is the recording of actions and the users performing them. In IT systems, this can be done by logging crucial elements of activity as they occur. With respect to data elements, accounting is needed when activity is determined to be crucial to the degree that it may be audited at a later date and time. Management has a responsibility for ensuring work processes are occurring as designed. Should there be a disconnect between planned and actual operational performance metrics, then it is management's responsibility to initiate and ensure corrective actions are taken and effective. Auditing is management's lens to observe the operation in a nonpartisan manner. Auditing is the verification of what actually happened on a system. Security-level auditing can be performed at several levels, from an analysis of the logging function that logs specific activities of a system to the management verification of the existence and operation of specific controls on a system.

Auditing can be seen as a form of recording historical events in a system. Operating systems have the ability to create audit structures, typically in the form of logs that allow management to review activities at a later point in time. One of the key security decisions is the extent and depth of audit log creation. Auditing takes resources, so by default it is typically set to a

minimal level. It is up to a system operator to determine the correct level of auditing required based on a system's criticality. The system criticality is defined by the information criticality associated with the information manipulated or stored within it. Determination and establishment of audit functionality must occur prior to an incident, as the recording of the system's actions cannot be accomplished after the fact.

Audit logs are a kind of balancing act. They require resources to create, store, and review. The audit logs in and of themselves do not create security; it is only through the active use of the information contained within them that security functionality can be enabled and enhanced. As a general rule, all critical transactions should be logged, including when they occurred and which authorized user is associated with the event. Additional metadata that can support subsequent investigation of a problem is also frequently recorded.



NOTE A key element in audit logs is the employment of a monitoring, detection, and response process. Without mechanisms or processes to “trigger” alerts or notifications to admins based on specific logged events, the value of logging is diminished or isolated to a post-incident resource instead of contributing to an alerting or incident prevention resource.

The information system security policy decomposition for auditing activities should consider both risk-based and organizational characteristics. The risk-based issues can be examined as cases of three forms of audit-related risk (also known as *residual risk*):

- Inherent risk
- Detection risk
- Control risk

Inherent risks are those associated with the process and its inherent error rate, assuming no internal controls exist to handle the potential errors. Detection risk is the risk that an audit will not detect an issue that can result

in material error. Control risk is the risk that controls will not detect or prevent material errors in a timely fashion.

Organizational characteristics that can drive auditing include items such as organizational history, the business environment, and supervisory issues. When decomposing policies, the following organizational security elements should be explored for auditing:

- Roles and responsibilities
- Separation of duties
- Training and qualifications
- Change management
- Control management

Logging

An important element in any security system is the presence of security logs. Logs enable personnel to examine information from a wide variety of sources after the fact, providing information about what actions transpired, with which accounts, on which servers, and with what specific outcomes. Many compliance programs require some form of logging and log management. The challenges in designing log programs are what to log and where to store it.

What needs to be logged is a function of several criteria. First, numerous compliance programs—HIPAA, SOX, PCI DSS, EOC, and others—have logging requirements, and these need to be met. The next criterion is one associated with incident response. What information would investigators want or need to know to research failures and issues? This is a question for the development team—what is available that can be logged that would provide useful information for investigators, either to the cause of the issue or to the impact?

The “where to log it” question also has several options, each with advantages and disadvantages. Local logging can be simple and quick for the development team. But it has the disadvantage of being yet another log to secure and integrate into the enterprise log management system. Logs by themselves are not terribly useful. What makes individual logs useful is the combination of events across other logs, detailing the activities of a particular

user at a given point in time. This requires a coordination function, one that is supported by many third-party software vendors through their security information and event management (SIEM) tool offerings. These tools provide a rich analytical environment to sift through and find correlations in large datasets of security information.

To Log or Not to Log

One of the challenging questions is to what extent items should be logged. If all goes right and things never fail, there is still a need for some logging, if for no other reason than to administratively track what has occurred on the system. Logs provide a historical record of transactions through a system.

Logs can also provide information that can be critical in debugging problems encountered by the software. Logs generated as part of error processing routines can provide great insight not only into individual program issues but also into the overall system state at the time of the error.

When deciding what to log, there are two central questions. First, when would we need the information, and what information would be needed in that use case? This defines the core foundation of a log entry. As each log entry takes time and space, both to record and later to sift through when using, an economic decision on “does this deserve logging?” must occur.

The second foundational element to consider when logging is the security of the information being logged. Details, such as system state, user IDs, critical variable values, including things such as passwords—all of these are subject to misuse or abuse in the wrong hands and thus must be protected. Logs require security. The level of security is a key element to determine in the creation of logging requirements.

Syslog

Syslog is an Internet Engineering Task Force (IETF)—approved protocol for log messaging. It was designed and built around UNIX and provides a UNIX-centric format for sending log information across an Internet Protocol (IP) network. Although in its native form it uses User Datagram Protocol (UDP)

and transmits information in the clear, wrappers are available that provide Transport Layer Security (TLS)-based security and Transmission Control Protocol (TCP)-based communication guarantees. While syslog is the de facto standard for logging management in Linux and UNIX environments, there is no equivalent in the Microsoft sphere of influence. Microsoft systems log locally, and there are some Microsoft solutions for aggregating logs to a central server, but these solutions are not as mature as syslog. Part of the reason for this is the myriad of third-party logging and log management solutions that provide superior business-level analytical packages that are focused on log data.

Nonrepudiation

Nonrepudiation is the concept of preventing a subject from denying a previous action with an object in a system. When authentication, authorization, and auditing are properly configured, the ability to prevent repudiation by a specific subject with respect to an action and an object is ensured. In simple terms, there is a system in place to prevent a user from saying they did not do something; this system can prove, in fact, whether an event took place or not. Nonrepudiation is a general concept, so security requirements must specify the subject, objects, and events for which nonrepudiation is desired, as this will affect the level of audit logging required. If complete nonrepudiation is desired, then every action by every subject on every object must be logged, and this could be a very large log dataset.

Secure Development Lifecycle

The term *secure development lifecycle* (SDL) comes from adding security to a software development lifecycle to reduce the number of security bugs in the software being produced. There are a wide range of different software development lifecycle models in use today across software development firms. To create a secure development lifecycle model, all one has to do is add a series of process checks to enable the development process to include the necessary security elements in the development process. The elements that are added are the same, although the location and methodology of adding

the new elements to the process are dependent upon the original process.

Security vs. Quality

Quality has a long history in manufacturing and can be defined as fitness for use. Quality can also be seen as absence of defects. Although this may seem to be the same thing as security, it is not, but they are related. Software can be of high quality and free of defects and still not be secure. The converse is the important issue; if software is not of high quality and has defects, then it is not going to be secure. Because a significant percentage of software security issues in practice are due to basic mistakes where the designer or developer should have known better, software quality does play a role in the security of the final product. Ross Anderson, a renowned expert in security, has stated that investments in software quality will result in a reduction of security issues, whether the quality program targets security issues or not.

Figure 1-4 illustrates some of the common issues associated with software quality versus security issues. If the product has quality but lacks security, then the result is a set of vulnerabilities. If the software is secure but is lacking in quality, then undocumented features may exist that can result in improper or undesired behaviors. The objective is to have both quality and security in the final output of the SDL process.

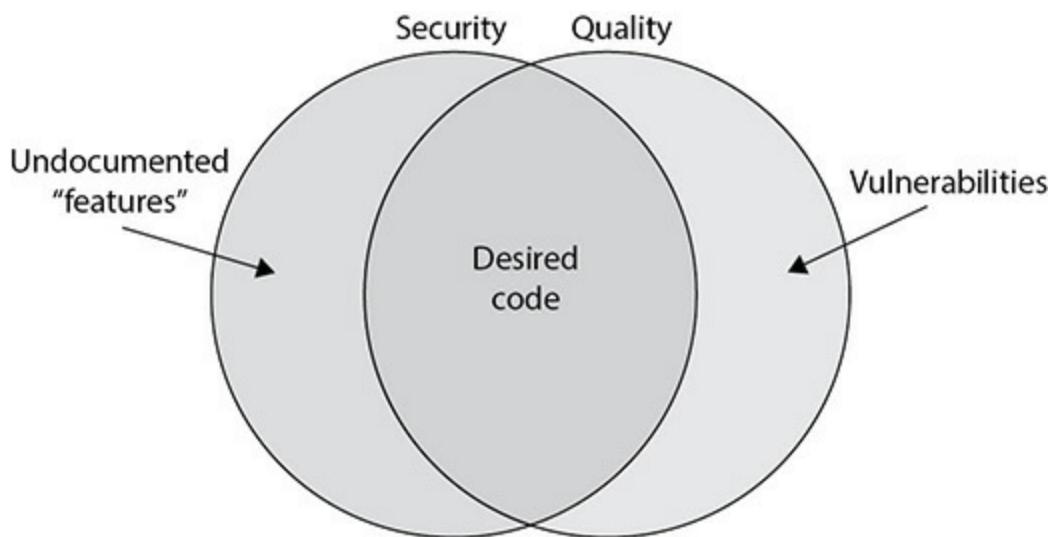


Figure 1-4 Software security vs. quality

Security Features != Secure Software

A common misconception is when someone confuses security features with secure software. Security features are elements of a program specifically designed to provide some aspect of security. Adding encryption, authentication, or other security features can improve the usability of software and thus are commonly sought after elements to a program. This is not what secure development is about. Secure software development is about ensuring that all elements of a software package are constructed in a fashion where they operate securely. Another way of looking at this is the idea that secure software does what it is designed to do and only what it is designed to do. Adding security features may make software more marketable from a features perspective, but this is not developing secure software. If the software is not developed to be secure, then even the security features cannot be relied upon to function as desired. An example of this was the case of the Debian Linux random-number bug, where a design flaw resulted in a flawed random-number function, which, in turn, resulted in cryptographic failures.

Secure Development Lifecycle Components

SDLs contain a set of common components that enable operationalization of security design principles. The first of these components is a current team-awareness and education program. Having a team that is qualified to do the task in an SDL environment includes security knowledge. The next component is the use of security gates as a point to check compliance with security requirements and objectives. These gates offer a chance to ensure that the security elements of the process are indeed being used and are functioning to achieve desired outcomes. Three sets of tools—bug tracking, threat modeling, and fuzzing—are used to perform security-specific tasks as part of the development process. The final element is a security review, where the results of the SDL process are reviewed to ensure that all of the required activities have been performed and completed to an appropriate level.

Software Team Awareness and Education

All team members should have appropriate training and refresher training

throughout their careers. This training should be focused on the roles and responsibilities associated with each team member. As the issues and trends associated with both development and security are ever changing, it is important for team members to stay current in their specific knowledge so that they can appropriately apply it in their work.

Security training can come in two forms: basic knowledge and advanced topics. Basic security knowledge, including how it is specifically employed and supported as part of the SDL effort, is an all-hands issue, and all team members need to have a functioning knowledge of this material. Advanced topics can range from new threats to tools, techniques, etc., and are typically aimed at a specific type of team member (i.e., designer, developer, tester, project manager). The key element of team awareness and education is to ensure that all members are properly equipped with the correct knowledge before they begin to engage in the development process.

Gates and Security Requirements

As part of the development process, periodic reviews are conducted. In an SDL, these are referred to as *gates*. The term *gate* is used to signify a condition that one must pass through. To pass the security gate, a review of the appropriate security requirements is conducted. Missing or incomplete elements can prevent the project from advancing to the next development phase until these elements or issues are addressed. This form of discipline, if conducted in a firm and uniform manner, results in eventual behavior by the development team where the gates are successfully negotiated as a part of normal business. This is the ultimate objective; the inclusion of security is a part of the business process.

Bug Tracking

Bug tracking is a basic part of software development. As code is developed, bugs are discovered. Bugs are elements of code that have issues that result in undesired behaviors. Sometimes, the behavior results in something that can be exploited, and this makes it a potential security bug. Bugs need to be fixed, and hence, they are tracked to determine their status. Some bugs may be obscure, impossible to exploit, and expensive to fix; thus, the best economic decision may be to leave them until the next major rewrite, saving

costs now on something that is not a problem. Tracking all of the bugs and keeping a log so that things can be fixed at appropriate times are part of managing code development.

Threat Modeling

Threat modeling is a design technique used to communicate information associated with a threat throughout the development team. The threat modeling effort begins at the start of the project and continues throughout the development effort. The purpose of threat modeling is to completely define and describe threats to the system under development. In addition, information as to how the threat will be mitigated can be recorded. Communicating this material among all members of the development team enables everyone to be on the same page with respect to understanding and responding to threats to the system. Threat modeling will be covered in detail in [Chapter 6](#).

Fuzzing

Fuzzing is a test technique where the tester applies a series of inputs to an interface in an automated fashion and examines the outputs for undesired behaviors. This technique is commonly used by hackers to discover unhandled input exceptions. The concept is fairly simple: For each and every input to a system, a test framework presents a variety of inputs and monitors the results of the system response. System crashes and unexpected behaviors are then examined for potential exploitation.

Security Reviews

Adding a series of security-related steps to the SDL can improve the software security. But as the old management axiom goes, you get what you measure, so it is important to have an audit function that verifies that the process is functioning as desired. Security reviews operate in this fashion. Sprinkled at key places, such as between stages in the SDL, the purpose of these reviews is not to test for security, but rather to examine the process and ensure that the security-related steps are being carried out and not being circumvented to expedite the process. Security reviews act as moments where the process can be checked to ensure that the security-related elements of the process are

functioning as designed. It may seem to be a paperwork drill, but it is more than that. Security reviews are mini-audit events where the efficacy of the security elements of the SDL process is being checked.

Mitigations

Not all risks are created equal. Some bugs pose a higher risk to the software and need fixing more than lesser-risk bugs. Using a system such as the DREAD model in the gating process of bug tracking informs us of the risk level ($\text{Risk} = \text{Impact} \times \text{Probability}$) associated with each bug so as to determine which bugs pose the greatest risk and therefore should be fixed first. The mitigation process itself defines how these bugs can be addressed. The DREAD model is covered in detail in [Chapter 13](#).

When bugs are discovered, there is a natural tendency to want to just fix the problem. But as in most things in life, the devil is in the details, and most bugs are not easily fixed with a simple change of the code. Why? Well, these errors are caught much earlier and rooted out before they become a “security bug.” When a security bug is determined to be present, four standard mitigation techniques are available to the development team:

- Do nothing
- Warn the user
- Remove the problem
- Fix the problem

The first, do nothing, is a tacit acknowledgment that not all bugs can be removed if the software is to ship. This is where the bug bar comes into play; if the bug poses no significant threat (i.e., is below the critical level of the bug bar), then it can be noted and fixed in a later release. Warning the user is in effect a cop-out. This pushes the problem on to the user and forces them to place a compensating control to protect the software. This may be all that is available until a patch can be deployed for serious bugs or until the next release for minor bugs. This does indicate a communication with the user base, which has been shown to be good business. Most users hate surprises, as in when they find out about bugs that they feel the software manufacturer should have warned them of in advance.

Removing the problem is a business decision, because it typically involves

disabling or removing a feature from a software release. This mitigation has been used a lot by software firms when the feature was determined not to be so important as to risk an entire release of the product. If a fix will take too long or involve too many resources, then this may be the best business recourse. The final mitigation, fixing the problem, is the most desired method and should be done whenever it is feasible given time and resource constraints. If possible, all security bugs should be fixed as soon as possible, for the cascading effect of multiple bugs cannot be ignored.

Chapter Review

In this chapter, you explored basic security concepts and terms. The constructs of confidentiality, integrity, and availability were introduced, together with the supporting elements of authentication, authorization, accountability, and nonrepudiation. With each of these concepts, examples of how they are implemented were covered. The chapter then covered the elements that create a secure development lifecycle. These elements include team awareness and education, gates and security requirements, bug tracking, threat modeling, fuzzing, security reviews, and mitigations.

A thorough understanding of these secure design principles is expected knowledge for CSSLP candidates.

Quick Tips

- Information assurance and information security place the security focus on the information and not on the hardware or software used to process it.
- The original elements of security were confidentiality, integrity, and availability—the CIA of security.
- Authentication, authorization, auditability, and nonrepudiation have been added to CIA to complete the characterization of operational security elements.

Questions

To further help you prepare for the CSSLP exam, answer the following

questions. The correct answers are provided in the following section.

1. If one desires nonrepudiation with respect to an event performed by a user, which of the following is/are required?
 - A. Authentication
 - B. Authorization
 - C. Auditing
 - D. All the above
2. Authentication can be based on what?
 - A. Something a user possesses
 - B. Something a user knows
 - C. Something measured on a user, such as a fingerprint
 - D. All of the above
3. What concept is the key element for implementing confidentiality, integrity, and availability?
 - A. Security definition
 - B. Authorized user
 - C. Password
 - D. Access control mechanism
4. Determining what a user is authorized to do on a system is a function of what?
 - A. Identity
 - B. Authentication
 - C. Access control
 - D. Availability
5. What is the concept of preventing a subject from denying a previous action with an object in a system?
 - A. Identity
 - B. Nonrepudiation
 - C. Authorization
 - D. Auditing

6. Issues involving protecting data from unauthorized alteration can be decomposed to what type of requirements?

 - A. Nonrepudiation
 - B. Access control
 - C. Integrity
 - D. Availability
7. The CIA of security includes what three attributes?

 - A. Confidentiality, integrity, authentication
 - B. Certificates, integrity, availability
 - C. Confidentiality, inspection, authentication
 - D. Confidentiality, integrity, availability
8. Logging is a key element of what?

 - A. Accountability
 - B. Access control
 - C. Integrity
 - D. Authorization
9. What is the fundamental approach to managing digital identity information called?

 - A. CIA
 - B. IdP
 - C. IAM
 - D. X.509
10. Which of the following is not a valid authentication factor or mechanism?

 - A. Something you have
 - B. Someone you know
 - C. Somewhere you are
 - D. All of the above

Answers

1. **D.** Nonrepudiation is the concept of preventing a subject from denying a previous action with an object in a system. When authentication, authorization, and auditing are properly configured, the ability to prevent repudiation by a specific subject with respect to an action and an object is ensured.
2. **D.** Authentication is commonly performed with passwords (something you know), tokens (something you have), and biometrics (such as fingerprints).
3. **B.** The definitions of confidentiality, integrity, and availability all have the dependency on an authorized user. This is why authentication is important.
4. **C.** Authentication provides a way to verify to the computer who the user is. Once the user has been authenticated, the access controls decide what operations the user can perform.
5. **B.** This is the definition of nonrepudiation.
6. **C.** Integrity. Integrity is the control over authorized alterations.
7. **D.** Don't forget that even though authentication was described at great length in this chapter, the A in the CIA of security represents availability, which refers to both the hardware and the data being accessible when the user wants them.
8. **A.** Accounting is a means of measuring activity. Accountability is the recording of actions and the users performing them. In IT systems, this can be done by logging crucial elements of activity as they occur.
9. **C.** Identity and access management (IAM) is the term associated with the comprehensive set of policies, processes, and technologies for managing digital identity information.
10. **B.** *Someone* you know is incorrect; *something* you know is a valid factor.

Security Design Principles

In this chapter you will

- Learn basic terminology associated with computer and information security
 - Discover the basic approaches to computer and information security
 - Examine security models used to implement security in systems
 - Explore the types of adversaries associated with software security
-
-

Secure software development is a key element in improving the cybersecurity landscape in the worldwide enterprise. Software is used by virtually every company, and your user base relies upon your secure coding efforts as part of their security posture, so secure development matters. This chapter will look at the classic system tenets of session, exception, and configuration management. We will also explore how the secure design tenets are embodied in software solutions. The common models used to describe complex security issues are presented, as these processes are involved in most software. The chapter closes with a discussion of the things that can go wrong, such as attacks, and how they impact software. Understanding these concepts as they apply to software development is a foundation of secure software development.

System Tenets

The creation of software systems involves the development of several foundational system elements within the overall system. Communication between components requires the management of a communication session, commonly called *session management*. When a program encounters an unexpected condition, an error can occur. Securely managing error conditions

is referred to as *exception management*. Software systems require configuration in production, and *configuration management* is a key element in the creation of secure systems.

Session Management

Software systems frequently require communications between program elements or between users and program elements. The control of the communication session between these elements is essential to prevent the hijacking of an authorized communication channel by an unauthorized party. Session management refers to the design and implementation of controls to ensure that communication channels are secured from unauthorized access and disruption of a communication. A common example is the Transmission Control Protocol (TCP) handshake that enables the sequential numbering of packets and allows for packet retransmission for missing packets; it also prevents the introduction of unauthorized packets and the hijacking of the TCP session. Session management requires additional work and has a level of overhead and hence may not be warranted in all communication channels. User Datagram Protocol (UDP), a connectionless/sessionless protocol, is an example of a communication channel that would not have session management and session-related overhead. An important decision early in the design process is determining when sessions need to be managed and when they do not. Understanding the use of the channel and its security needs should dictate the choice of whether session management is required.

Exception Management

There are times when a system encounters an unknown condition or is given input that results in an error. The process of handling these conditions is exception management. A remote resource may not respond, or there may be a communication error—whatever the cause of the error, it is important for the system to respond in an appropriate fashion. Several criteria are necessary for secure exception management. First, all exceptions must be detected and handled. Second, the system should be designed so as not fail to an insecure state. Last, all communications associated with the exception should not leak information.

For example, assume a system is connecting to a database to verify user

credentials. Should an error occur, as in the database is not available when the request is made, the system needs to properly handle the exception. The system should not inadvertently grant access in the event of an error. The system may need to log the error, along with information concerning what caused the error, but this information needs to be protected. Releasing the connection string to the database or passing the database credentials with the request would be a security failure.

Configuration Management

Dependable software in production requires the managed configuration of the functional connectivity associated with today's complex, integrated systems. Initialization parameters, connection strings, paths, keys, and other associated variables are typical examples of configuration items. As these elements can have significant effects upon the operation of a system, they are part of the system and need to be properly controlled for the system to remain secure. The identification and management of these elements is part of the security process associated with a system.

Management has a responsibility to maintain production systems in a secure state, and this requires that configurations be protected from unauthorized changes. This has resulted in the concept of configuration management, change control boards, and a host of workflow systems designed to control the configuration of a system. One important technique frequently employed is the separation of duties between production personnel and development/test personnel. This separation is one method to prevent the contamination of approved configurations in production.

Secure Design Tenets

Secure designs do not happen by accident. They are the product of deliberate architecture and deliberate plans and are structured upon a foundation of secure design principles. These principles have borne the test of time and have repeatedly proven their worth in a wide variety of security situations. The seminal work in this area, the application of secure design principles to computer systems, is Saltzer and Schroeder's 1975 article "The Protection of Information in Computer Systems."

Good Enough Security

Security is never an absolute, and there is no such thing as complete or absolute security. This is an important security principle, for it sets the stage for all of the security aspects associated with a system. There is a trade-off between security and other aspects of a system. Secure operation is a requirement for reliable operation, but under what conditions? Every system has some appropriate level of required security, and it is important to determine this level early in the design process. Just as one would not spend \$10,000 on a safe to protect a \$20 bill, a software designer should not use national security-grade encryption to secure publicly available information.

Least Privilege

One of the most fundamental approaches to security is *least privilege*. Least privilege means that a subject should have only the necessary rights and privileges to perform its current task with no additional rights and privileges. Limiting a subject's privileges limits the amount of harm that can be caused, thus limiting a system's exposure to damage. If a subject may require different levels of security for different tasks, it is better to switch security levels with specific tasks, rather than run all the time at a higher level of privilege.

Another issue that falls under the least privilege concept is the security context in which an application runs. All programs, scripts, and batch files run under the security context of a specific user on an operating system. They will execute with specific permissions as if they were that user. The infamous Sendmail exploit utilized this specific design issue. Sendmail needs root-level access to accomplish specific functions, and hence the entire program was run as root. If the program was compromised after it had entered the root access state, the attacker could obtain a root-level shell. The crux of this issue is that programs should execute only in the security context that is needed for that program to perform its duties successfully.

Separation of Duties

Another fundamental approach to security is *separation of duties*. Separation of duties ensures that for any given task, more than one individual needs to be involved. The critical path of tasks is split into multiple items, which are then

spread across more than a single party. By implementing a task in this manner, no single individual can abuse the system. A simple example might be a system where one individual is required to place an order and a separate person is needed to authorize the purchase.

This separation of duties must be designed into a system. Software components enforce separation of duties when they require multiple conditions to be met before a task is considered complete. These multiple conditions can then be managed separately to enforce the checks and balances required by the system.

Defense in Depth

Defense in depth is one of the oldest security principles. If one defense is good, multiple overlapping defenses are better. A castle has a moat, thick walls, restricted access points, high points for defense, multiple chokepoints inside, etc., working together as a whole series of defenses all aligned toward a single objective. Defense in depth is also known by the terms *layered security* (or defense) and *diversity defense*.

Software should utilize the same type of layered security architecture. There is no such thing as perfect security. No system is 100 percent secure, and there is nothing that is foolproof, so a single specific protection mechanism should never be solely relied upon for security. Every piece of software and every device can be compromised or bypassed in some way, including every encryption algorithm, given enough time and resources. The true goal of security is to make the cost of compromising a system greater in time and effort than it is worth to an adversary.

Diversity of defense is a related concept that complements the idea of various layers of security. The concept of layered security, illustrated in [Figure 2-1](#), is the application of multiple security defenses to craft an overlapping, more comprehensive solution. For the layers to be diverse, they should be dissimilar in nature so that if an adversary makes it past one layer, another layer may still be effective in maintaining the system in a secure state. Coupling encryption and access control provides multiple layers that are diverse in their protection nature, and yet both can provide confidentiality.

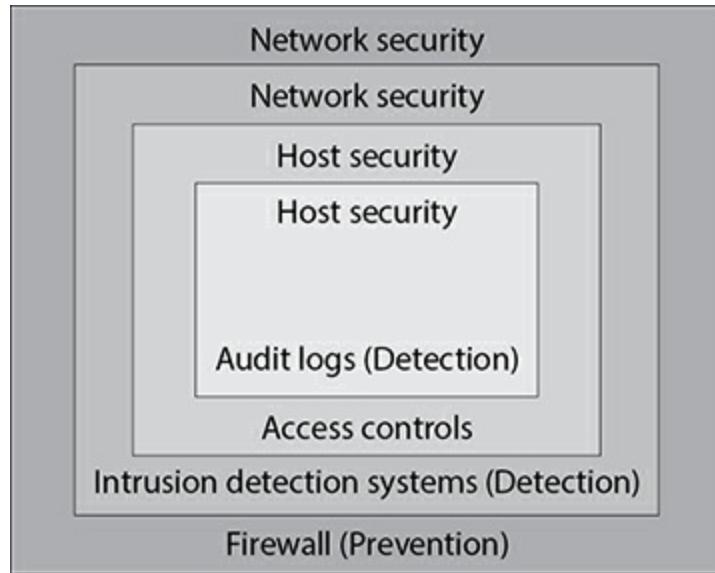


Figure 2-1 Layered security

Defense in depth provides security against many attack vectors. The fact that any given defense is never 100 percent effective is greatly assisted by a series of different defenses, multiplying their effectiveness. Defense in depth is a concept that can be applied in virtually every security function and instance.

Fail-Safe

As mentioned in the “Exception Management” section, all systems will experience failures. The *fail-safe* design principle is that when a system experiences a failure, it should fail to a safe state. One form of implementation is to use the concept of explicit deny. Any function that is not specifically authorized is denied by default. When a system enters a failure state, the attributes associated with security, confidentiality, integrity, and availability need to be appropriately maintained. Availability is the attribute that tends to cause the greatest design difficulties. Ensuring that the design includes elements to degrade gracefully and return to normal operation through the shortest path assists in maintaining the resilience of the system. During design, it is important to consider the path associated with potential failure modes and how this path can be moderated to maintain system stability and control under failure conditions.

Economy of Mechanism

The terms *security* and *complexity* are often at odds with each other. This is because the more complex something is, the harder it is to understand, and you cannot truly secure something if you do not understand it. Another reason complexity is a problem within security is that it usually allows too many opportunities for something to go wrong. If an application has 4,000 lines of code, there are a lot fewer places for buffer overflows, for example, than in an application of two million lines of code.

As with any other type of technology or problem in life, when something goes wrong with security mechanisms, a troubleshooting process is used to identify the actual issue. If the mechanism is overly complex, identifying the root of the problem can be overwhelming, if not nearly impossible. Security is already a complex issue because there are so many variables involved, so many types of attacks and vulnerabilities, so many different types of resources to secure, and so many different ways of securing them. You want your security processes and tools to be as simple and elegant as possible. They should be simple to troubleshoot, simple to use, and simple to administer.

Another application of the principle of keeping things simple concerns the number of services that you allow your system to run. Default installations of computer operating systems often leave many services running. The keep-it-simple principle tells us to eliminate those that we don't need. This is also a good idea from a security standpoint because it results in fewer applications that can be exploited and fewer services that the administrator is responsible for securing. The general rule of thumb should be to always eliminate all nonessential services and protocols. This, of course, leads to the question, how do you determine whether a service or protocol is essential or not? Ideally, you should know what your computer system or network is being used for, and thus you should be able to identify those elements that are essential and activate only them. For a variety of reasons, this is not as easy as it sounds. Alternatively, a stringent security approach that one can take is to assume that no service is necessary (which is obviously absurd) and activate services and ports only as they are requested. Whatever approach is taken, there is a never-ending struggle to try to strike a balance between providing functionality and maintaining security.

Complete Mediation

The principle of complete mediation states that when a subject's authorization is verified with respect to an object and an action, this verification occurs every time the subject requests access to an object. The system must be designed so that the authorization system is never circumvented, even with multiple, repeated accesses. This principle is the one involved in the use of the security kernel in operating systems, functionality that cannot be bypassed, and allows security management of all threads being processed by the operating system (OS). Modern operating systems and IT systems with properly configured authentication systems are difficult to compromise directly, with most routes to compromise involving the bypassing of a critical system such as authentication. With this in mind, it is important during design to examine potential bypass situations and prevent them from becoming instantiated.

Open Design

Another concept in security that should be discussed is the idea of security through obscurity. This approach has not been effective in the actual protection of the object. Security through obscurity may make someone work a little harder to accomplish a task, but it does not provide actual security. This approach has been used in software to hide objects, such as keys and passwords, buried in the source code. Reverse engineering and differential code analysis have proven effective at discovering these secrets, eliminating this form of "security."

The concept of open design states that the security of a system must be independent of the form of the security design. In essence, the algorithm that is used will be open and accessible, and the security must not be dependent upon the design, but rather on an element such as a key. Modern cryptography has employed this principle effectively; security depends upon the secrecy of the key, not the secrecy of the algorithm being employed.

Least Common Mechanism

The concept of *least common mechanism* refers to a design method designed to prevent the inadvertent sharing of information. Having multiple processes share common mechanisms leads to a potential information pathway between

users or processes. Having a mechanism that services a wide range of users or processes places a more significant burden on the design of that process to keep all pathways separate. When presented with a choice between a single process that operates on a range of supervisory and subordinate-level objects and/or a specialized process tailored to each, choosing the separate process is the better choice.

The concepts of least common mechanism and leveraging existing components can place a designer at a conflicting crossroad. One concept advocates reuse and the other separation. The choice is a case of determining the correct balance associated with the risk from each.

Psychological Acceptability

Users are a key part of a system and its security. Including a user in the security of a system requires that the security aspects be designed so that they are psychologically acceptable to the user. When a user is presented with a security system that appears to obstruct the user, the result will be the user working around the security aspects of the system. For instance, if a system prohibits the e-mailing of certain types of attachments, the user can encrypt the attachment, masking it from security, and perform the prohibited action anyway.

Ease of use tends to trump many functional aspects. The design of security in software systems needs to be transparent to the user, just like air— invisible, yet always there, serving the need. This places a burden on the designers; security is a critical functional element, yet one that should impose no burden on the user.

Weakest Link

The weakest link is the common point of failure for all systems. Every system by definition has a “weakest” link. Adversaries do not seek out the strongest defense to attempt a breach. A system can be considered only as strong as its *weakest link*. Expending additional resources to add to the security of a system is most productive when it is applied to the weakest link. Throughout the software lifecycle, it is important to understand the multitude of weaknesses associated with a system, including the weakest link. Including in the design a series of diverse defenses, in other words, defense in

depth, is critical to hardening a system against exploitation. Managing the security of a system requires understanding the vulnerabilities and defenses employed, including the relative strengths of each one, so that they can be properly addressed.

Leverage Existing Components

Component reuse has many business advantages, including increases in efficiency and security. As components are reused, fewer new components are introduced to the system; hence, the opportunity for additional vulnerabilities is reduced. This is a simplistic form of reducing the attack surface area of a system. The downside of massive reuse is associated with a monoculture environment, which is where a failure has a larger footprint because of all the places it is involved with.

Single Point of Failure

Just as multiple defenses are a key to a secure system, so, too, is a system design that is not susceptible to a *single point of failure*. A single point of failure is any aspect of a system that, if it fails, means the entire system fails. It is imperative for a secure system to not have any single points of failure. The design of a software system should be such that all points of failure are analyzed and a single failure does not result in system failure. Single points of failure can exist for any attribute, confidentiality, integrity, availability, etc., and may well be different for each attribute. Examining designs and implementations for single points of failure is important to prevent this form of catastrophic failure from being released in a product or system.

Security Models

Models are used to provide insight and explanation. Security models are used to understand the systems and processes developed to enforce security principles. Three key elements play a role in systems with respect to model implementation: people, processes, and technology. Addressing a single element of the three may provide benefits, but more effectiveness can be achieved through addressing multiple elements. Controls that rely on a single element, regardless of the element, are not as effective as controls that

address two or all three elements.

Access Control Models

The term *access control* has been used to describe a mechanism to ensure protection. Access controls define what actions a subject can perform on specific objects. Access controls assume that the identity of the user has been verified through an authentication process. There are a variety of different access control models that emphasize different aspects of a protection scheme. One of the most common mechanisms used is an *access control list* (ACL). An ACL is a list that contains the subjects that have access rights to a particular object. An ACL will identify not only the subject, but also the specific access that subject has for the object. Typical types of accesses include read, write, and execute. Several different models are discussed in security literature, including discretionary access control (DAC), mandatory access control (MAC), role-based access control (RBAC), and rule-based access control (RBAC).

MAC Model

A less frequently employed system for restricting access is *mandatory access control* (MAC). MAC has its roots in military control systems, and referring to the Orange Book, a Department of Defense document that at one time was the standard for describing what constituted a trusted computing system, we can find a definition for mandatory access controls, which are “a means of restricting access to objects based on the sensitivity (as represented by a label) of the information contained in the objects and the formal authorization (i.e., clearance) of subjects to access information of such sensitivity.”



EXAM TIP Mandatory access control restricts access based on the sensitivity of the information and whether the user has the authority to access that information.

In MAC systems, the owner or subject can't determine whether access is to be granted to another subject; it is the job of the operating system to

decide. In MAC, the security mechanism controls access to all objects, and individual subjects cannot change that access. This places the onus of determining security access upon the designers of a system, requiring that all object and subject relationships be defined before use in a system. SELinux, a specially hardened form of Linux based on MAC, was developed by the National Security Agency (NSA) to demonstrate the usefulness of this access model.

DAC Model

The most commonly employed model for access control is the *discretionary access control* (DAC) model. Both discretionary access control and mandatory access control are terms originally used by the military to describe two different approaches to controlling what access an individual had on a system. As defined by the Orange Book, discretionary access controls are “a means of restricting access to objects based on the identity of subjects and/or groups to which they belong. The controls are discretionary in the sense that a subject with certain access permission is capable of passing that permission (perhaps indirectly) on to any other subject.”

The DAC model is really rather simple. In systems that employ discretionary access controls, the owner of an object can decide which other subjects may have access to the object and what specific access they may have. The owner of a file can specify what permissions are granted to which users. Access control lists are the most common mechanism used to implement discretionary access control. The strength of DAC is its simplicity. The weakness is that it is discretionary, or, in other words, optional. One of the other weaknesses is that this requires the object owner to define access for all objects under its control, a task that can end up with tens of thousands of access control decisions. To facilitate the scaling of DAC, group-based models can be employed.

Role-Based Access Control

Access control lists can become long, cumbersome, and take time to administer properly. An access control mechanism that addresses the length and cost of ACLs is the *role-based access control* (RBAC). In this scheme, instead of each user being assigned specific access permissions for the objects associated with the computer system or network, users are assigned to

a set of roles that they may perform. A common example of roles would be developer, tester, production, manager, and executive. In this scheme, a user could be a developer and be in a single role or could be a manager over testers and be in two roles. The assignment of roles need not be exclusionary. The roles are, in turn, assigned the access permissions necessary to perform the tasks associated with them. Users will thus be granted permissions to objects in terms of the specific duties they must perform. An auditor, for instance, can be assigned read access only—allowing audits but preventing change. [Figure 2-2](#) illustrates how roles act as mediators between users and objects. When the number of users grows and permissions must be applied to thousands of objects, roles reduce the number of assignments required. In a firm with hundreds of employees and hundreds of thousands of objects, the number of assignments can approach the number of users times the number of objects, or more. Roles can reduce this by many orders of magnitude.

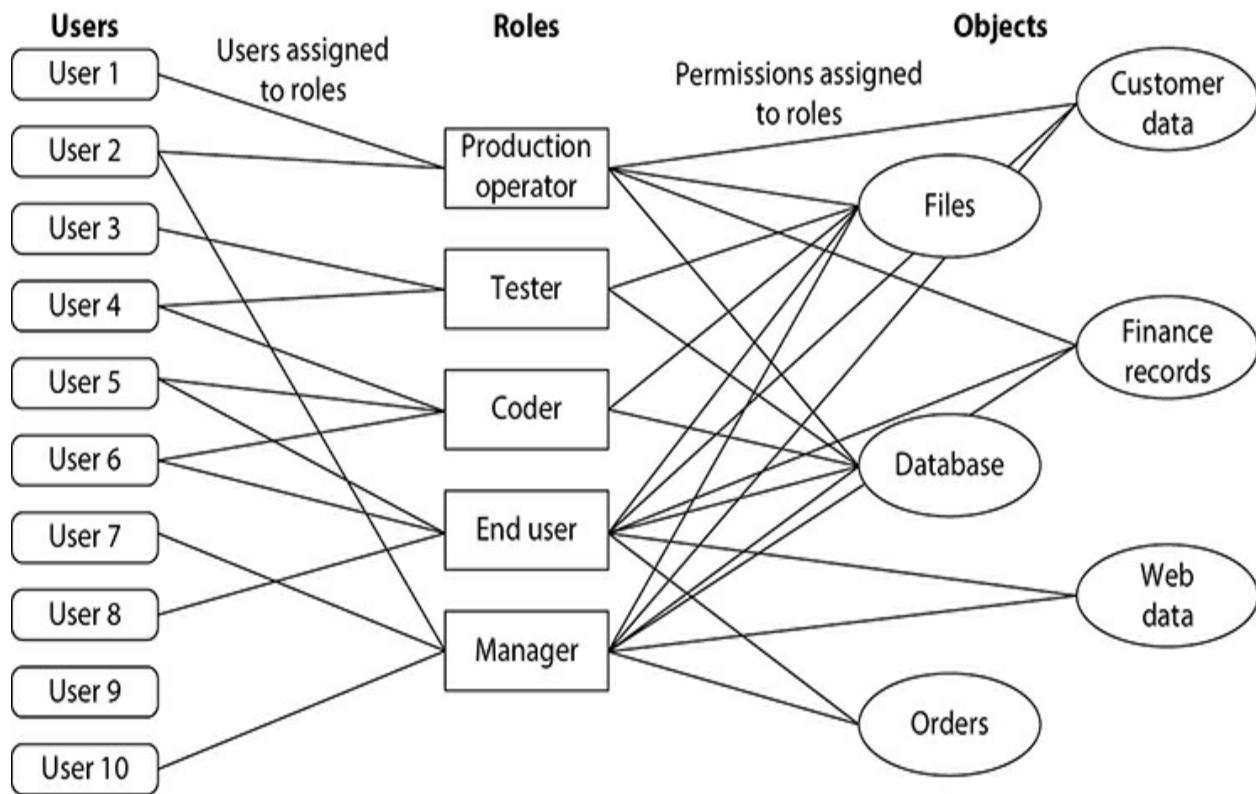


Figure 2-2 Using roles to mediate permission assignments

Rule-Based Access Control

A second use of the acronym RBAC is for *rule-based access control*. Rule-based access control systems are much less common than role-based access control, but they serve a niche. In rule-based access control, we again utilize elements such as access control lists to help determine whether access should be granted. In this case, a series of rules is contained in the access control list, and the determination of whether to grant access will be made based on these rules. An example of such a rule might be a rule that states that nonmanagement employees may not have access to the payroll file after hours or on weekends. Rule-based access control can actually be used in addition to, or as a method of, implementing other access control methods. For example, role-based access control may be used to limit access to files based on job assignment, and rule-based controls may be added to control time-of-day or network restrictions.

Access Control Matrix Model

The *access control matrix* model is a simplified form of access control notation where the allowed actions a subject is permitted with an object are listed in a matrix format. This is a general-purpose model, with no constraints on its formulation. The strength in this model is its simplicity in design, but this also leads to its major weakness: difficulty in implementation. Because it has no constraints, it can be difficult to implement in practice and does not scale well. As the number of subjects and objects increases, the intersections increase as the product of the two enumerations, leading to large numbers of ACL entries.

Attribute-Based Access Control

Attribute-based access control (ABAC) is a new access control schema based on the use of attributes associated with an identity. These can use any type of attributes (user attributes, resource attributes, environment attributes, and so on), such as location, time, activity being requested, and user credentials. An example would be a doctor getting one set of access for a specific patient versus a different patient. ABAC can be represented via the *eXtensible Access Control Markup Language (XACML)*, a standard that implements attribute- and policy-based access control schemes.

Bell-LaPadula Confidentiality Model

The *Bell-LaPadula model* is a confidentiality preserving model. The Bell-LaPadula security model employs both mandatory and discretionary access control mechanisms when implementing its two basic security principles. The first of these principles is called the *Simple Security Rule*, which states that no subject can read information from an object with a security classification higher than that possessed by the subject itself. This rule is also referred to as the *no-read-up* rule. This means that the system must have its access levels arranged in hierachal form, with defined higher and lower levels of access. Because the Bell-LaPadula model was designed to preserve confidentiality, it is focused on read and write access. Reading material higher than a subject's level is a form of unauthorized access.



NOTE The Simple Security Rule is just that: the most basic of security rules. It basically states that for you to see something, you have to be authorized to see it.

The second security principle enforced by the Bell-LaPadula security model is known as the **-property* (pronounced “star property”). This principle states that a subject can write to an object only if its security classification is less than or equal to the object’s security classification. This is also known as the *no-write-down* principle. This prevents the dissemination of information to users who do not have the appropriate level of access. This can be used to prevent data leakage, such as the publishing of bank balances, presumably protected information, to a public webpage.

Take-Grant Model

The *take-grant model* for access control is built upon graph theory. This model is conceptually very different from the other models, but has one distinct advantage: it can be used to definitively determine rights. This model is a theoretical model based on mathematical representation of the controls in the form of a directed graph, with the vertices being the subjects and objects. The edges between them represent the rights between the subject and objects. There are two unique rights to this model: take and grant. The representation

of the rights takes the form of $\{t, g, r, w\}$, where t is the take right, g is the grant right, r is the read right, and w is the write right. A set of four rules, one each for take, grant, create, and remove, forms part of the algebra associated with this mathematical model.

The take-grant model is not typically used in the implementation of a particular access control system. Its value lies in its ability to analyze an implementation and answer questions concerning whether a specific implementation is complete or might be capable of leaking information.

Multilevel Security Model

The *multilevel security model* is a descriptive model of security where separate groups are given labels and these groups act as containers, keeping information and processes separated based on the labels. These can be hierarchical in nature, in which some containers can be considered to be superior to or include lower containers. An example of multilevel security is the military classification scheme: Top Secret, Secret, and Confidential. A document can contain any set of these three, but the “container” assumes the label of the highest item contained. If a document contains any Top Secret information, then the entire document assumes the Top Secret level of protection. For purposes of maintenance, individual items in the document are typically marked with their applicable level, so that if information is “taken” from the document, the correct level can be chosen. Additional levels can be added to the system, such as NoForn, which restricts distribution from foreigners. There is also the use of keywords associated with Top Secret so that specific materials can be separated and not stored together.

Integrity Models

Integrity-based models are designed to protect the integrity of the information. For some types of information, integrity can be as important as, or even more important than, confidentiality. Public information, such as stock prices, is available to all, but the correctness of their value is crucial, leading to the need to ensure integrity.

Biba Integrity Model

In the Biba model, integrity levels are used to separate permissions. The

principle behind integrity levels is that data with a higher integrity level is believed to be more accurate or reliable than data with a lower integrity level. Integrity levels indicate the level of “trust” that can be placed in the accuracy of information based on the level specified.

The Biba model employs two rules to manage integrity efforts. The first rule is referred to as the *low-water-mark policy*, or *no-write-up* rule. This policy in many ways is the opposite of the *-property from the Bell-LaPadula model, in that it prevents subjects from writing to objects of a higher integrity level. The Biba model’s second rule states that the integrity level of a subject will be lowered if it acts on an object of a lower integrity level. The reason for this is that if the subject then uses data from that object, the highest the integrity level can be for a new object created from it is the same level of integrity of the original object. In other words, the level of trust you can place in data formed from data at a specific integrity level cannot be higher than the level of trust you have in the subject creating the new data object, and the level of trust you have in the subject can be only as high as the level of trust you had in the original data.

Clark-Wilson Model

The *Clark-Wilson security model* takes an entirely different approach than the Biba model, using transactions as the basis for its rules. It defines two levels of integrity: constrained data items (CDIs) and unconstrained data items (UDIs). CDI data is subject to integrity controls, while UDI data is not. The model then defines two types of processes: integrity verification processes (IVPs) and transformation processes (TPs). IVPs ensure that CDI data meets integrity constraints (to ensure the system is in a valid state). TPs are processes that change the state of data from one valid state to another. Data in this model cannot be modified directly by a user; it can only be changed by trusted TPs.

Using banking as an example, an object with a need for integrity would be an account balance. In the Clark-Wilson model, the account balance would be a CDI because its integrity is a critical function for the bank. Since the integrity of account balances is of extreme importance, changes to a person’s balance must be accomplished through the use of a TP. Ensuring that the balance is correct would be the duty of an IVP. Only certain employees of the bank should have the ability to modify an individual’s account, which can be controlled by limiting the number of individuals who have the authority to

execute TPs that result in account modification.

Information Flow Models

Another methodology in modeling security is built around the notion of information flows. Information in a system must be protected when at rest, in transit, and in use. Understanding how information flows through a system, the components that act upon it, and how it enters and leaves a system provides critical data on the necessary protection mechanisms. A series of models that explores aspects of data or information flow in a system assist in the understanding of the application of appropriate protection mechanisms.

Brewer-Nash Model (Chinese Wall)

The *Brewer-Nash model* is a model designed to enforce confidentiality in commercial enterprise operations. In a commercial enterprise, there are situations where different aspects of a business may have access to elements of information that cannot be shared with the other aspects. In a financial consulting firm, personnel in the research arm may become privy to information that would be considered “insider information.” This information cannot, by law or ethically, be shared with other customers. The common term used for this model is the Chinese Wall model.

Security is characterized by elements involving technology, people, and processes. The Brewer-Nash model is a model where elements associated with all three can be easily understood. Technology can be employed to prevent access to data by conflicting groups. People can be trained not to compromise the separation of information. Policies can be put in place to ensure that the technology and the actions of personnel are properly engaged to prevent compromise. Employing actions in all three domains provides a comprehensive implementation of a security model.

Data Flow Diagrams

The primary issue in security is the protection of information when stored, while in transit, and while being processed. Understanding how data moves through a system is essential in designing and implementing the security measures to ensure appropriate security functionality. *Data flow diagrams* (DFDs) are specifically designed to document the storage, movement, and processing of data in a system. Data flow diagrams are graphical in nature

and are constructed on a series of levels. The highest level, level 0, is a high-level contextual view of the data flow through the system. The next level, level 1, is created by expanding elements of the level 0 diagram. This level can be exploded further to a level 2 diagram, or lowest-level diagram of a system.

Use Case Models

While a DFD examines a system from the information flow perspective, the *use case model* examines the system from a functional perspective model. Requirements from the behavioral perspective provide a description of how the system utilizes data. Use cases are constructed to demonstrate how the system processes data for each of its defined functions. Use cases can be constructed for both normal and abnormal (misuse cases) to facilitate the full description of how the system operates. Use case modeling is a well-defined and mainstream method for system description and analysis. Combined with DFDs, use cases provide a comprehensive overview of how a system uses and manipulates data, facilitating a complete understanding of the security aspects of a system.



NOTE Use case modeling is important to secure programming, as it can represent security requirements as well as misuse cases. Understanding what should not happen (misuse case) is as important as understanding what should (use case).

Assurance Models

The Committee on National Security Systems has defined software assurance as the “level of confidence that software is free from vulnerabilities, either intentionally designed into the software or accidentally inserted at any time during its lifecycle, and that the software functions in the intended manner.” This shift in focus moves toward the preventive element of the operational security model and is driven by a management focus on system design and construction. The current software development methodology employed by many teams is focused on speed to market and functionality. More functions

and new versions can drive sales. This focus has led to a lesser position for security, with patching issues when found as the major methodology. This has proven less than satisfactory for many types of critical programs, and the government has led the effort to push for an assurance-based model of development.

Assurance cases, including misuse and abuse cases, are designed and used to construct a structured set of arguments and a corresponding body of evidence to satisfactorily demonstrate specific claims with respect to its security properties. An assurance case is structured like a legal case. An overall objective is defined. Specific elements of evidence are presented that demonstrate conclusively a boundary of outcomes that eliminates undesirable ones and preserves the desired ones. When sufficient evidence is presented to eliminate all undesired states or outcomes, the system is considered to be assured of the claim.

The Operational Model of Security

There are three primary actionable methods of managing security in production: prevention, detection, and response. The operational security model encompasses these three elements in a simple form for management efforts. Most effort is typically put on prevention efforts, because incidents that are prevented are eliminated from further concern. For the issues that are not prevented, the next step is detection. Some issues may escape prevention efforts, and if they escape detection efforts, then they can occur without any intervention on the part of security functions. Elements that are detected still need response efforts. The operational model of security, illustrated in [Figure 2-3](#), shows examples of elements in the model.

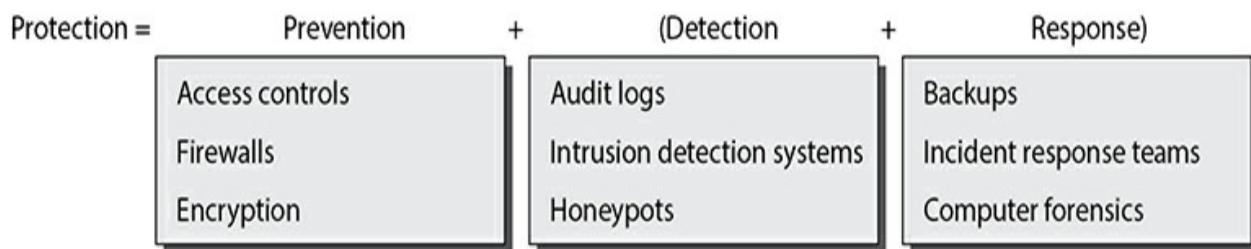


Figure 2-3 Operational model of security

The NIST CSF Model

In April 2018, the National Institute of Standards and Technology (NIST) released the latest version of its Cybersecurity Framework (CSF). This is a voluntary framework, consisting of standards, guidelines, and best practices, and is designed to manage cybersecurity-related risk in an organization. The CSF uses five main functions to categorize activities: Identify, Protect, Detect, Respond, Recover. The CSF is being adopted as a framework for information security activities at many firms and thus is an important foundational element in understanding how security is actually implemented. Understanding these concepts and terms can assist in cases where the project under development will be used in a CSF environment.

Adversaries

Security is the protection of elements from an adversary. Adversaries come in many shapes and sizes, with widely varying motives and capabilities. The destructive capability of an adversary depends upon many factors, including efforts to protect an element from damage. One of the most damaging adversaries is nature. Natural disasters range from narrow damage associated with storms, such as tornados, to large-scale issues from hurricanes and ice storms and the resulting outages in the form of power and network outages. The “saving grace” with natural disasters is the lack of motive or specific intent and hence the nonadaptability of an attack. Other classifications of adversaries are built around capabilities, specifically in the form of their adaptability and capability to achieve their objective despite security controls in place.

Adversary Type

Adversaries can be categorized by their skill level and assigned a type. This classification is useful when examining the levels of defenses needed and to what degree they can be effective. As the skill level of adversaries increases, the numbers in each category decline. While little to no specific training is required to practice at the level of a script kiddie, years of dedicated study are required to obtain, and even more to retain, the rank of an elite hacker. Fortunately, the number of elite hackers is small.

Script Kiddie

The term *script kiddie* is used to describe the most basic form of attacker. The term is considered derisive by most in the industry, as it describes a user who can use only published scripts to perform attacks. The specific knowledge one has when in this category can be virtually nonexistent, other than the skill to Google a script and then try it against a target. This category is seen to comprise as much as 80 to 85 percent of the attacking community. The good news is that the attack vectors are known, and typically there are specific defenses to prevent these attacks from achieving their goal. The bad news is there are so many of them that they create a level of background noise that must be addressed, requiring resources to manage.

Hacker

The term *hacker* has historically referred to a user who is an explorer, one who explores how a system operates and how to circumvent its limits. The term *cracker* has been used to refer to an individual who was a hacker, but with malicious intent. This level of attacker, by nature, has training in how systems operate and has the ability to manipulate systems to achieve outcomes that may or may not be desired or permitted. The actual skill level can vary, but the higher-skilled individuals have the ability to develop new scripts that can be employed by those of lesser skill levels. This group makes up 15 to 20 percent of the attacking population. This group is the key adversary, because between their skill and motivation, the damage they cause may be catastrophic to an organization.

Elite

The *elite* group of hackers is a small fraction, 1 to 3 percent, of the overall attacking population. The key distinguishing element in this group is truly skill based, with a skill level that would be considered impossible by most users. This group is completely underground, because one of the essential skills to enter this level is the skill to cover one's tracks to the point of making them virtually undetectable and untraceable. We know they exist, however, because of two factors. First, there are members at this skill level who operate on the side of the good guys, or white hats. Second, there are specific exploits, known as *zero-day exploits*, where the exploit precedes the "discovery" of the vulnerability. After a vulnerability is found and patched, when later analysis shows cases of long-term exploitation, the obvious answer is a set of highly skilled attackers that maintain an extremely low

profile. This group has the skills to make them almost defense proof, and unless you are an extremely sensitive industry, spending resources to defend against this group is not particularly efficient.

Adversary Groups

Adversaries can be analyzed from the perspective of adversary groups. The grouping of adversaries, based on skill and capability, provides a structure that can be used to analyze the effectiveness of defenses. The least capable form is an unstructured threat, a single actor from the outside. A highly structured threat, or nation-state attack, has significantly greater capability. The delineation of an attacker as either an insider or an outsider can also play a role in determining capability. An insider has an advantage in that they already have some form of legitimate access that can be exploited. This, coupled with their internal knowledge of systems and the value of data and its location, places insiders ahead of the pack on essential knowledge in the prosecution of an attack.

Unstructured Threat

Unstructured threats are those with little or no resources. Typically individuals or groups with limited skills, unstructured threats are limited in their ability to focus and pursue a target over time or with a diversity of methods. Most script kiddies act as solo attackers and lose interest in their current target when difficulties in the attack arise. Most unstructured threats pursue targets of opportunity rather than specific targets for motivational reasons. Because the skill level is low, their ability to use comprehensive attack methods is limited, and this form of threat is fairly easily detected and mediated. Because of the random nature of their attack patterns, searching any target for a given exploit, unstructured threats act as a baseline of attack noise, ever present, yet typically only an annoyance, not a serious threat to operations.

Structured Threat

When attackers become organized and develop a greater resource base, their abilities can grow substantially. The structured threat environment is indicative of a group that has a specific mission; has the resources to commit significant time and energy to an attack, at times for periods extending into

months; and has the ability to build a team with the varied skills necessary to exploit an enterprise and its multitude of technologies and components. These groups can employ tools and techniques that are more powerful than simple scripts floating on the Internet. They have the ability to develop specific code and employ and use botnets and other sophisticated elements to perpetrate their criminal activities.

The level of sophistication of modern botnets has demonstrated that structured threats can be real and have significant effects on an enterprise. There have been significant efforts on the part of law enforcement to shut down criminal enterprises utilizing botnets, and security efforts on the part of system owners are needed to ensure that their systems are secured and monitored for activity from this level of threat. Where an unstructured threat may penetrate a system, either by luck or fortune, their main goal is to perform an attack. Structured threats are characterized by their goal orientation. It is not enough to penetrate a system; they view penetration as merely a means to their end. And their end is to steal information that is of value and that may result in a loss for the firm under attack.

Highly Structured Threat

A highly structured threat is a structured threat with significantly greater resources. Criminal organizations have been found to employ banks of programmers that develop crimeware. The authors of the modern botnets are not single individuals, but rather structured programming teams producing a product. Organized crime has moved into the identity theft and information theft business, as it is much safer from prosecution than robbing banks and companies the old-fashioned way. The resource base behind several large criminal organizations involved in cybercrime enables them to work on security issues for years, with teams of programmers building and utilizing tools that can challenge even the strongest defenses.

Nation-State Threat

When a highly structured threat is employed by a nation-state, it assumes an even larger resource base and, to some degree, a level of protection. The use of information systems as a conduit for elements of espionage and information operations is a known reality of the modern technology-driven world. In the past few years, a new form of threat, the *advanced persistent*

threat (APT), has arisen. The APT is a blended attack composed of multiple methods, designed to penetrate a network's defenses and then live undetected as an insider in a system. The objective of the APT is attack specific, but rather than attempt to gather large amounts of information with the chance of being caught, the APT has been hallmark by judicious, limited, almost surgical efforts to carefully extract valuable information yet leave the system clean and undetected.

Nation-state-level threats will not be detected or deflected by ordinary defense mechanisms. The U.S. government has had to resort to separate networks and strong cryptographic controls to secure itself from this level of attack. The positive news is that nation-state attack vectors are not aimed at every firm, only a select few of typically government-related entities. Espionage is about stealing secrets, and fortunately, the monetization of the majority of these secrets for criminal purposes is currently limited to cases of identity theft, personal information theft, and simple bank account manipulation (wire fraud). The primary defense against the environment of attackers present today is best driven by aiming not at nation-state-level threats or highly structured threats, for they are few in number. Targeting the larger number of structured threat vectors is a more efficient methodology and one that can be shifted to more serious threats as the larger numbers are removed from the picture.

Insider vs. Outsider Threat

Users of computer systems have been divided into two groups: insiders and outsiders. Insiders are those who have some form of legitimate access to a system. Outsiders are those characterized as not having a legitimate form of access. For many years, efforts have been aimed at preventing outsiders from gaining access. This was built around a belief that "criminals" were outside the organization. Upon closer examination, however, it has been found that criminals exist inside the system as well, and the fact that a user has legitimate access overcomes one of the more difficult elements of a computer hack—to gain initial access. Insiders thus have the inside track for account privilege escalation, and they have additional knowledge of a system's strengths, weaknesses, and the location of the valuable information flows. A modern comprehensive security solution is one that monitors a system for both external and internal attacks.

Threat Landscape Shift

The threat landscape has for years been defined by the type of actor in the attack, the motivations being unique to each group. From individual hackers who are, in essence, explorers whose motivation is to explore how a system works to the hacktivist groups that have some message to proclaim (a common rationale for web defacement) to nation-states whose motivation is espionage based, the group defined the purpose and the activity to a great degree. Around 2005, a shift occurred in the attack landscape: the criminalization of cyberspace. This was a direct result of criminal groups developing methods of monetizing their efforts. This has led to an explosion in attack methods, as there is a market for exploits associated with vulnerabilities. This has driven actual research into the art of the attack, with criminal organizations funding attacks for the purpose of financial gain. The botnets being used today are sophisticated decentralized programs, using cryptographic and antivirus detection methods, and are designed to avoid detection. These botnets are used to gather credentials associated with online financial systems, building profiles of users including bank access details, stock account details, and other sensitive information.

This shift has a practical effect on all software—everything is now a target. In the past, unless one was a bank, large financial institution, or government agency, security was considered not as important as “who would attack me?” This dynamic has changed, as criminals are now targeting smaller businesses, and even individuals, as the crimes are nearly impossible to investigate and prosecute, and just as people used to think they could hide in all the bits on the Internet, the criminals are doing just that. And with millions of PCs under botnet control, their nets are wide, and even small collections from large numbers of users are profitable. The latest shift in direction is toward targeting individuals, making specific detection even more difficult.

Chapter Review

In this chapter, the basic tenets of software systems, session management, exception management, and configuration management were introduced and placed in context with operational security objectives. Secure design principles were described, including

- Good enough security
- Least privilege
- Separation of duties
- Defense in depth
- Fail-safe
- Economy of mechanism
- Complete mediation
- Open design
- Least common mechanism
- Psychological acceptability
- Weakest link
- Leverage existing components
- Single point of failure

A thorough understanding of these secure design principles is expected knowledge for CSSLP candidates.

Moving from security principles to security models, the chapter covered the principle of security depending upon people, processes, and technology. Access control models, including mandatory access controls, discretionary access controls, role-based access controls, rule-based access controls, and matrix access controls, were presented. The Bell-LaPadula, Biba, Clark-Wilson, and Brewer-Nash security models were described, presenting their role in designing secure software.

The types and characteristics of threats and adversaries were presented. Understanding the threat landscape is a key element in understanding software assurance.

Quick Tips

- Systems have a set of characteristics; session management, exception management, and configuration management provide the elements needed to secure a system in operation.
- A series of secure design principles describe the key characteristics associated with secure systems.

- Security models describe key aspects of system operations with respect to desired operational characteristics, including preservation of confidentiality and integrity.
- The Bell-LaPadula security model preserves confidentiality and includes the simple security rule, the *-property, and the concept of “no read up, no write down.”
- The Biba integrity model preserves integrity and includes the concept of “no write up and no read down.”
- Access control models are used to describe how authorization is implemented in practice.
- Understanding the threat environment educates the software development team on the security environment the system will face in production.

Questions

To further help you prepare for the CSSLP exam, answer the following questions. The correct answers are provided in the following section.

1. Which access control mechanism provides the owner of an object the opportunity to determine the access control permissions for other subjects?
 - A. Mandatory
 - B. Role-based
 - C. Discretionary
 - D. Token-based
2. What was described in the chapter as being essential in order to implement discretionary access controls?
 - A. Object owner-defined security access
 - B. Certificates
 - C. Labels
 - D. Security classifications
3. Complete mediation is an approach to security that includes what?
 - A. Protecting systems and networks by using defense in depth

- B. A security design that cannot be bypassed or circumvented
 - C. Using interlocking rings of trust to ensure protection to data elements
 - D. Using access control lists to enforce security rules
- 4. What is the fundamental approach to security in which an object has only the necessary rights and privileges to perform its task with no additional permissions?
 - A. Layered security
 - B. Least privilege
 - C. Role-based security
 - D. Clark-Wilson model
- 5. Which access control technique relies on a set of rules to determine whether access to an object will be granted or not?
 - A. Role-based access control
 - B. Object and rule instantiation access control
 - C. Rule-based access control
 - D. Discretionary access control
- 6. The security principle that ensures that no critical function can be executed by any single individual (by dividing the function into multiple tasks that can't all be executed by the same individual) is known as what?
 - A. Discretionary access control
 - B. Security through obscurity
 - C. Separation of duties
 - D. Implicit deny
- 7. What describes the ability of a subject to interact with an object?
 - A. Authentication
 - B. Access
 - C. Confidentiality
 - D. Mutual authentication

- 8.** Open design places the focus of security efforts on what?
 - A.** Open-source software components
 - B.** Hiding key elements (security through obscurity)
 - C.** Proprietary algorithms
 - D.** Producing a security mechanism in which its strength is independent of its design
- 9.** The security principle of fail-safe is related to what?
 - A.** Session management
 - B.** Exception management
 - C.** Least privilege
 - D.** Single point of failure
- 10.** Using the principle of keeping things simple is related to what?
 - A.** Layered security
 - B.** Simple Security Rule
 - C.** Economy of mechanism
 - D.** Implementing least privilege for access control

Answers

- 1.** **C.** This is the definition of discretionary access control.
- 2.** **A.** Object owners define access control in discretionary access control systems.
- 3.** **B.** This is the definition of complete mediation.
- 4.** **B.** This was the description supplied for least privilege. Layered security referred to using multiple layers of security (such as at the host and network layers) so that if an intruder penetrates one layer, they still will have to face additional security mechanisms before gaining access to sensitive information.
- 5.** **C.** This is a description of rule-based access control.
- 6.** **C.** This is a description of the separation of duties principle.

- 7.** **B.** This is the definition of access.
- 8.** **D.** Open design states that the security of a system must be independent from its design. In essence, the algorithm that is used will be open and accessible, and the security must not be dependent upon the design, but rather on an element such as a key.
- 9.** **B.** The principle of fail-safe states that when failure occurs, the system should remain in a secure state and not disclose information. Exception management is the operational tenet associated with fail-safe.
- 10.** **C.** The principle of economy of mechanism states that complexity should be limited to make security manageable; in other words, keep things simple.

PART II

Secure Software Requirements

- **Chapter 3** Define Software Security Requirements
- **Chapter 4** Identify and Analyze Compliance Requirements
- **Chapter 5** Misuse and Abuse Cases

Define Software Security Requirements

In this chapter you will

- Learn basic terminology associated with software requirements
 - Examine functional requirements used to implement security in systems
 - Understand the role of non-functional requirements such as operational and deployment requirements
-

Requirements are the blueprint by which software is designed, built, and tested. As one of the important foundational elements, it is critical to manage this portion of the software development lifecycle (SDLC) process properly. Requirements set the expectations for what is being built and how it is expected to operate. Developing and understanding the requirements early in the SDLC process are important because if one has to go back and add new requirements later in the process, it can cause significant issues, including rework, delays, and increased cost.

Functional Requirements

Functional requirements describe how the software is expected to function. They begin as business requirements and can come from several different places. The line of business that is going to use the software has some business functionality it wants to achieve with the new software. These business requirements are translated into functional requirements. The IT operations group may have standard requirements, such as deployment platform requirements, database requirements, disaster recovery/business

continuity planning (DR/BCP) requirements, infrastructure requirements, and more. The organization may have its own coding requirements in terms of good programming and maintainability standards. Security may have its own set of requirements. In the end, all of these business requirements must be translated into functional requirements that can be followed by designers, coders, testers, and more to ensure they are met as part of the SDLC process.

Role and User Definitions

Role and user definitions are the statements of who will be using what functionality of the software. At a high level, these will be in generic form, such as which groups of users are allowed to use the system. Subsequent refinements will detail specifics, such as which users are allowed which functionality as part of their job. The detailed listing of what users are involved in a system form part of the use-case definition. In computer science terms, users are referred to as *subjects*. This term is important to understand the subject-object-activity matrix presented later in this section.

Objects

Objects are items that users (subjects) interact with in the operation of a system. An object can be a file, a database record, a system, or a program element. Anything that can be accessed is an object. One method of controlling access is through the use of access control lists assigned to objects. As with subjects, objects form an important part of the subject-object-activity matrix. Specifically defining the objects and their function in a system is an important part of the SDLC. This ensures all members of the development team can properly use a common set of objects and control the interactions appropriately.

Activities/Actions

Activities or actions are the permitted events that a subject can perform on an associated object. The specific set of activities is defined by the object. A database record can be created, read, updated, or deleted. A file can be accessed, modified, deleted, etc. For each object in the system, all possible activities/actions should be defined and documented. Undocumented functionality has been the downfall of many a system when a user found an

activity that was not considered during design and construction, but still occurred, allowing functionality outside of the design parameters.

Subject-Object-Activity Matrix

Subjects represent the who, objects represent the what, and activities or actions represent the how of the subject-object-activity relationship. Understanding the activities that are permitted or denied in each subject-object combination is an important requirements exercise. To assist designers and developers in correctly defining these relationships, a matrix referred to as the *subject-object-activity matrix* is employed. For each subject, all of the objects are listed, along with the activities for each object. For each combination, the security requirement of the state is then defined. This results in a master list of allowable actions and another master list of denied actions. These lists are useful in creating appropriate use and misuse cases, respectively. The subject-object-activity matrix is a tool that permits concise communication about allowed system interactions.

Use Cases

Use cases are a powerful technique for determining functional requirements in developer-friendly terms. A use case is a specific example of an intended behavior of the system. Defining use cases allows a mechanism by which the intended behavior (functional requirement) of a system can be defined for both developers and testers. Use cases are not intended for all subject-object interactions, as the documentation requirement would exceed the utility. Use cases are not a substitute for documenting the specific requirements. Where use cases are helpful is in the description of complex or confusing or ambiguous situations associated with user interactions with the system. This facilitates the correct design of both the software and the test apparatus to cover what would otherwise be incomplete due to poorly articulated requirements.



EXAM TIP Use cases are constructed of actors representing users and

intended system behaviors, with the relationships between them depicted graphically.

Use-case modeling shows the intended system behavior (activity) for actors (users). This combination is referred to as a *use case* and is typically presented in a graphical format. Users are depicted as stick figures, and the intended system functions as ellipses. Use-case modeling requires the identification of the appropriate actors, whether person, role, or process (nonhuman system), as well as the desired system functions. The graphical nature enables the construction of complex business processes in a simple-to-understand form. When sequences of actions are important, another diagram can be added to explain this. [Figure 3-1](#) illustrates a use-case model for a portion of an online account system.

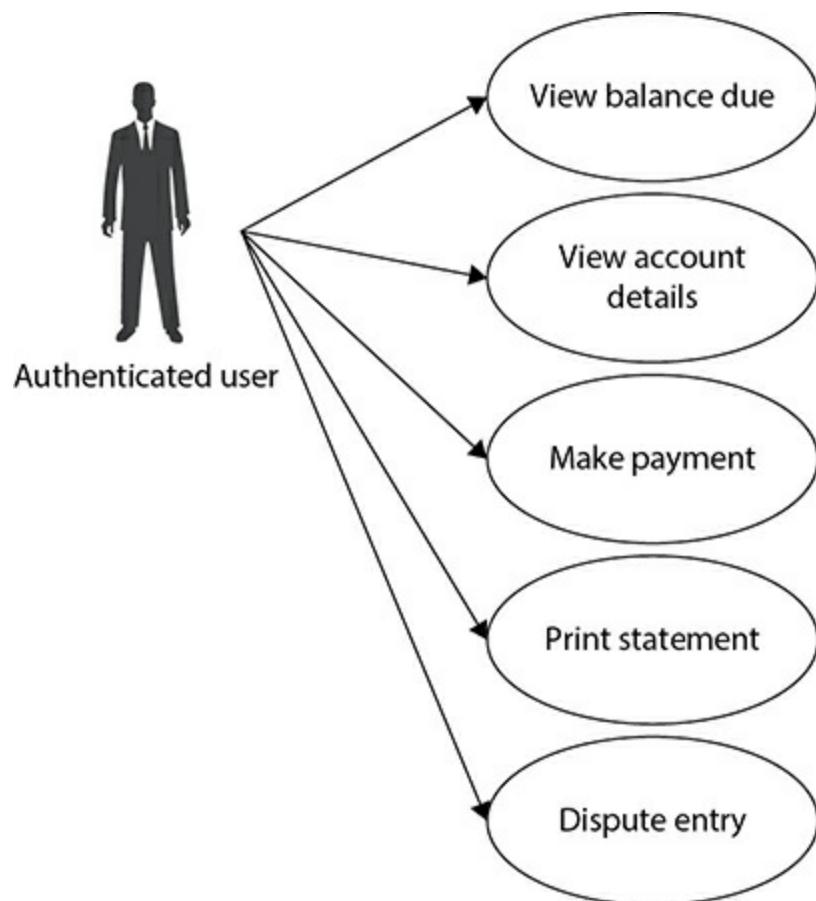


Figure 3-1 Use-case diagram

Sequencing and Timing

In today's multithreaded, concurrent operating model, it is possible for different systems to attempt to interact with the same object at the same time. It is also possible for events to occur out of sequence based on timing differences between different threads of a program. Sequence and timing issues such as race conditions and infinite loops influence both design and implementation of data activities. Understanding how and where these conditions can occur is important to members of the development team. In technical terms, what develops is known as a *race condition*, or from the attack point of view, a system is vulnerable to a time of check/time of use (TOC/TOU) attack.



EXAM TIP A TOC/TOU attack is one that takes advantage of a separation between the time a program checks a value and when it uses the value, allowing an unauthorized manipulation that can affect the outcome of a process.

Race conditions are software flaws that arise from different threads or processes with a dependence on an object or resource that affects another thread or process. A classic race condition is when one thread depends on a value (A) from another function that is actively being changed by a separate process. The first process cannot complete its work until the second process changes the value of A. If the second function is waiting for the first function to finish, a lock is created by the two processes and their interdependence. These conditions can be difficult to predict and find. Multiple unsynchronized threads, sometimes across multiple systems, create complex logic loops for seemingly simple atomic functions. Understanding and managing record locks is an essential element in a modern, diverse object programming environment.

Race conditions are defined by race windows, a period of opportunity when concurrent threads can compete in attempting to alter the same object. The first step to avoid race conditions is to identify the race windows. Then, once the windows are identified, the system can be designed so that they are

not called concurrently, a process known as *mutual exclusion*.

Another timing issue is the infinite loop. When program logic becomes complex—for instance, date processing for leap years—care should be taken to ensure that all conditions are covered and that error and other loop-breaking mechanisms do not allow the program to enter a state where the loop controls will fail. Failure to manage this exact property resulted in Microsoft Zune devices failing if they were turned on across the New Year following a leap year. The control logic entered a sequence where a loop would not be satisfied, resulting in the device crashing by entering an infinite loop and becoming nonresponsive.



EXAM TIP Complex conditional logic with unhandled states, even if rare or unexpected, can result in infinite loops. It is imperative that all conditions in each nested loop be handled in a positive fashion.

Secure Coding Standards

Secure coding standards are language-specific rules and recommended practices that provide for secure programming. It is one thing to describe sources of vulnerabilities and errors in programs; it is another matter to prescribe forms that, when implemented, will preclude the specific sets of vulnerabilities and exploitable conditions found in typical code.



NOTE Secure coding standards have been published by the Software Engineering Institute/CERT at Carnegie Mellon University for C, C++, and Java. Each of these standards includes rules and recommended practices for secure programming in the specific language.

Application programming can be considered a form of manufacturing. Requirements are turned into value-added product at the end of a series of

business processes. Controlling these processes and making them repeatable is one of the objectives of a secure development lifecycle. One of the tools an organization can use to achieve this objective is the adoption of an enterprise-specific set of secure coding standards.

Organizations should adopt the use of a secure application development framework as part of their secure development lifecycle process. Because secure coding guidelines have been published for most common languages, adoption of these practices is an important part of secure coding standards in an enterprise. Adapting and adopting industry best practices are also important elements in the secure development lifecycle.

One common problem in many programs results from poor error trapping and handling. This is a problem that can benefit from an enterprise rule where all exceptions and errors are trapped by the generating function and then handled in such a manner so as not to divulge internal information to external users.



EXAM TIP To prevent error conditions from cascading or propagating through a system, each function should practice complete error mitigation, including error trapping and complete handling, before returning to the calling routine.

Logging is another area that can benefit from secure coding standards. Standards can be deployed specifying what, where, and when issues should be logged. This serves two primary functions: it ensures appropriate levels of logging, and it simplifies the management of the logging infrastructure.

Operational and Deployment Requirements

Software is deployed in an enterprise environment where it is rarely completely on its own. Enterprises will have standards as to technology deployment, specifying platforms, operating systems (Linux, Microsoft Windows), specific types and versions of database servers, web servers, and other infrastructure components. These technology choices are part of the

enterprise overall strategy, and while the results may not be optimal for single instances, the overall collection is optimal for the enterprise.

Software in the enterprise rarely works all by itself without connections to other pieces of software. A new system may provide new functionality, but would do so touching existing systems, such as connections to users, parts databases, customer records, etc. One set of operational requirements is built around the idea that a new or expanded system must interact with the existing systems over existing channels and protocols. At a high level, this can be easily defined, but it is not until detailed specifications are published that much utility is derived from the effort.



NOTE A complete SDLC solution ensures systems are secure by design, secure by default, and secure in deployment. A system that is secure by design but deployed in an insecure configuration or method of deployment can render the security in the system worthless.

One of the elements of secure software development is that it is secure in deployment. Ensuring that systems are secure by design is commonly seen as the focus of an SDLC, but it is also important to ensure systems are secure when deployed. This includes elements such as secure by default and secure when deployed. Ensuring the default configuration maintains the security of an application if the system defaults are chosen, and since this is a common configuration and should be a functioning configuration, it should be secure.

Software will be deployed in the environment as best suits its maintainability, data access, and access to needed services. Ultimately, at the finest level of detail, the functional requirements that relate to system deployment will be detailed for use. An example is the use of a database and web server. Corporate standards, dictated by personnel and infrastructure services, will drive many of the selections. Although there are many different database servers and web servers in the marketplace, most enterprises have already selected an enterprise standard, sometimes by type of data or usage. Understanding and conforming to all the requisite infrastructure requirements are necessary to allow seamless interconnectivity between different systems.

Modern secure development practices include managing the deployment of the software. This is often done via a secure continuous integration (CI) and continuous delivery (CD) pipeline. This material is discussed in detail in [Chapter 16](#).

Connecting the Dots

Requirements are the foundational elements used in the development of any project. They come from many sources. This chapter looked at functional requirements and operational requirements through the lens of secure software design. In the first section of the book, threat modeling was covered, and one of the key outputs from the threat modeling process is a set of requirements to mitigate known and expected threats. The key to understanding requirements is that they represent all the knowledge one has with respect to building a project. The easiest sets of requirements are those that represent the features that a customer is asking for, but this is just the tip of the iceberg. Customers will never give you the requirement “it needs to work” because, of course, that is always implied. The challenge is to enumerate and document all of the related security, functional, and operational requirements that are not stated because they are “implied.”

The task of creating a good list of security requirements is challenging at first, as there are so many details, so many “sources of information,” that the sheer organization of it all is overwhelming. But as a team goes from project to project using a core list and refining it and adding to it, a more comprehensive set can be developed over time. The bottom line to the story is simple: if you want a development team to do something, it needs to be enumerated in the requirements for the project.

Chapter Review

This chapter examined the requirements associated with a system. The chapter began with a description of functional requirements and how they can come from numerous sources, including the business, the architecture group to ensure interoperability with the existing enterprise elements, and the security group to ensure adherence to security and compliance issues. The concepts of users and roles, together with subjects and objects and the

allowed activities or actions, were presented. This chapter also covered the development of a subject-object-activity matrix defining permitted activities.

Use cases were presented as a means of communicating business requirements and security requirements across the SDLC. These tools can be powerful in communicating ambiguous requirements and in ensuring that specific types of security issues are addressed. Other security concerns, such as sequence and timing issues, infinite loops, and race conditions, were discussed. The use of enterprise-wide secure coding standards to enforce conformity across the development processes was presented. This is the first foundational element in defining an enterprise methodology that assists in security and maintainability and assists all members of the development team in understanding how things work.

Operational and deployment requirements are those that ensure the system functions as designed when deployed.

Quick Tips

- Functional requirements are those that describe how the software is expected to function.
- Business requirements must be translated into functional requirements that can be followed by designers, coders, testers, and more to ensure they are met as part of the SDLC process.
- Role and user definitions are the statements of who will be using what functionality of the software.
- Objects are items that users (subjects) interact with in the operation of a system. An object can be a file, a database record, a system, or a program element.
- Activities or actions are the legal events that a subject can perform on an associated object.
- The subject-object-activity matrix is a tool that permits concise communication about allowed system interactions.
- A use case is a specific example of an intended behavior of the system.
- Sequence and timing issues, such as race conditions and infinite loops, influence both design and implementation of data activities.
- Secure coding standards are language-specific rules and recommended

practices that provide for secure programming.

- A complete SDLC solution ensures systems are secure by design, secure by default, and secure in deployment.

Questions

To further help you prepare for the CSSLP exam, answer the following questions. The correct answers are provided in the following section.

1. An activity designed to clarify requirements through the modeling of expected behaviors of a system is called what?
 - A. Functional requirement decomposition
 - B. Requirement traceability matrix
 - C. Threat modeling
 - D. Use-case modeling
2. Business requirements are translated into _____ for the development team to act upon.
 - A. Programming rules
 - B. Data lifecycle elements
 - C. Functional requirements
 - D. Data flow diagrams
3. Enterprise secure coding standards ensure what?
 - A. Certain types of vulnerabilities are precluded
 - B. Code is error free
 - C. Code is efficient
 - D. Security functionality is complete
4. Functional requirements include all of the following except what?
 - A. Determining specific architecture details
 - B. Deployment platform considerations
 - C. DR/BCP requirements
 - D. Security requirements
5. Access control lists are assigned to _____ as part of a security

scheme.

- A. Users
 - B. Roles
 - C. Objects
 - D. Activities
6. To prevent error conditions from propagating through a system, each function should do what?
- A. Log all abnormal conditions
 - B. Include error trapping and handling
 - C. Clear all global variables upon completion
 - D. Notify users of errors before continuing
7. Corporate standards, driven by defined infrastructure services, will drive what?
- A. Deployment environment requirements
 - B. Database requirements
 - C. Web server requirements
 - D. Data storage requirements
8. Complex conditional logic can result in _____ for unhandled states.
- A. Infinite loops
 - B. Race conditions
 - C. Memory leaks
 - D. Input vulnerabilities
9. Use cases should be constructed for what?
- A. All requirements
 - B. All requirements that have security concerns
 - C. Business requirements that are poorly defined
 - D. Implementation features that need testing
10. To assist designers and developers in correctly defining the relationships between users and the desired functions on objects, a _____ can

be employed.

- A. Functional requirements matrix
- B. Requirements traceability matrix
- C. Use case
- D. Subject-object-activity matrix

Answers

1. D. Defining use cases provides a mechanism by which the intended behavior (functional requirement) of a system can be defined for both developers and testers.
2. C. Functional requirements begin as business requirements and can come from several different places.
3. D. Secure coding standards are language-specific rules and recommended practices that provide for secure programming.
4. A. The specific architecture details come from requirements but are not specified directly as functional requirements.
5. C. Access control lists are associated with users, objects, and activities, but are assigned to objects.
6. B. To prevent error conditions from cascading or propagating through a system, each function should practice complete error mitigation, including error trapping and complete handling, before returning to the calling routine.
7. A. Deployment environment requirements include issues such as corporate standards for databases, web services, data storage, and more.
8. A. Complex conditional logic with unhandled states, even if rare or unexpected, can result in infinite loops.
9. C. Use cases are specifically well suited for business requirements that are not well defined.
10. D. To assist designers and developers in correctly defining the relationships between users (subjects), objects, and activities, a matrix referred to as the subject-object-activity matrix is employed.

Identify and Analyze Compliance Requirements

In this chapter you will

- Learn about regulations and compliance requirements
 - Examine data classification requirements
 - Explore privacy requirements
-
-

Software development methodologies have been in existence for decades, with new versions being developed to capitalize on advances in teamwork and group functionality. While security is not itself a development methodology, it has been shown by many groups and firms that security functionality can be added to a development lifecycle, creating a secure development lifecycle. While this does not guarantee a secure output, including security in the process used to develop software has been shown to dramatically reduce the defects that cause security bugs.

Regulations and Compliance

Regulations and compliance drive many activities in an enterprise. The primary reason behind this is the simple fact that failure to comply with rules and regulations can lead to direct, and in some cases substantial, financial penalties. Compliance failures can carry additional costs, as in increased scrutiny, greater regulation in the future, and bad publicity. Since software is a major driver of many business processes, a CSSLP needs to understand the basis behind various rules and regulations and how they affect the enterprise in the context of their own development efforts. This enables decision-making as part of the software development process that is in concert with

these issues and enables the enterprise to remain compliant.

Much has been said about how compliance is not the same as security. In a sense, this is true, for one can be compliant and still be insecure. When viewed from a risk management point of view, security is an exercise in risk management, and so are compliance and other hazards. Add it all together, and you get an “all hazards” approach, which is popular in many industries, as senior management is responsible for all hazards and the residual risk from all risk sources.

Regulations can come from several sources, including industry and trade groups and government agencies. The penalties for noncompliance can vary as well, sometimes based on the severity of the violation and other times based on political factors. The factors determining which systems are included in regulation and the level of regulation also vary based on situational factors. Typically, these factors and rules are published significantly in advance of instantiation to allow firms time to plan enterprise controls and optimize risk management options. Although not all firms will be affected by all sets of regulations, it is also not uncommon for a firm to have multiple sets of regulations across different aspects of an enterprise, even overlapping on some elements. This can add to the difficulty of managing compliance, as different regulations can have different levels of protection requirements.

Many development efforts may have multiple regulatory impacts, and mapping the different requirements to the individual data flows that they each affect is important. For instance, if an application involves medical information and payment information, different elements may be subject to regulations such as Payment Card Industry Data Security Standard (PCI DSS) and Health Insurance Portability and Accountability Act (HIPAA). These and other common regulatory requirements are covered later in this chapter.



NOTE For a CSSLP, it is important to understand the various sources of security requirements, as they need to be taken into account when executing software development. It is also important to not mistake security

functionality for the objective of secure software development. Security functions driven by requirements are important, but the objective of a secure development lifecycle process is to reduce the number and severity of vulnerabilities in software.

Security Standards

Standards are a defined level of activity that can be measured and monitored for compliance by a third party. Standards serve a function by defining a level of activity that allows different organizations to interact in a known and meaningful way. Standards also facilitate comparisons between organizations. The process of security in an enterprise is enhanced through the use of standards that enable activities associated with best practices. There are a wide range of sources of standards, including standards bodies, both international and national, and industry and trade groups.

Security standards serve a role in promoting interoperability. In software design and development, there will be many cases where modules from different sources will be interconnected. In the case of web services, the WS-security standard provides a means of secure communication between web services.

ISO

ISO is the International Organization for Standardization, a group that develops and publishes international standards. The United States has an active relationship to ISO through the activities of the U.S. National Committee, the International Electrotechnical Commission (IEC), and the American National Standards Institute (ANSI). ISO has published a variety of standards covering the information security arena. To ensure that these standards remain relevant with respect to ever-changing technology and threats, ISO standards are on a five-year review cycle.

The relevant area of the standards catalog are under JTC 1 – Information Technology, specifically subcommittees 7 (Software and Systems Engineering) and 27 (IT Security Techniques). Depending upon the specific topic, other subcommittees may also have useful standards (see www.iso.org/iso/home/store/catalogue_tc/catalogue_tc_browse.htm?comid=45020).

Prominent ISO Standards

The list of ISO standards is long, covering many topics, but some of the more important ones for CSSLPs to understand are as follows:

ISO/IEC 25010:2011	Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models.
ISO/IEC 10746	Information Technology – Open distributed processing. Multipart series standard.
ISO/IEC/IEEE 12207:2017	Systems and Software Engineering – Software life cycle processes
ISO/IEC 14143	Information Technology – Software measurement – Functional size measurement. Multipart series standard.
ISO/IEC/IEEE 15026-1:2019	Systems and software engineering – Systems and software assurance.
ISO/IEC/IEEE 15288:2015	Systems and Software Engineering – System life cycle processes
ISO/IEC 15408	Information technology – Security techniques – Evaluation criteria for IT security (Common Criteria)
ISO/IEC 21827	Information Technology – Security techniques – Systems Security Engineering – Capability Maturity Model (SSE-CMM)
ISO/IEC 27000:2018	Information Security Management System (ISMS) Overview and Vocabulary
ISO/IEC 27002:2013	Code of Practice for Information Security Management
ISO/IEC 27003:2017	Information security management systems — Guidance
ISO/IEC 27004:2016	Information Security Management – Monitoring, measurement, analysis and evaluation
ISO/IEC 27005:2018	Information Security Risk Management

ISO 2700X Series

The ISO 2700X series of standards does for information security what the ISO 900X series does for quality management. This series defines the relevant vocabulary, a code of practice, management system implementation guidance, metrics, and risk management principles. The ISO/IEC 2700X series of information security management standards is a growing family with more than 20 standards currently in place. Broad in scope, covering more than just privacy, confidentiality, or technical security issues, this family of standards is designed to be applicable to all shapes and sizes of organizations.

ISO/IEC 15408 Common Criteria

The Common Criteria is a framework where security functional and assurance requirements can be specified in precise terms, allowing vendors to implement and/or make claims about the security attributes of their products. Testing laboratories can evaluate the products to determine if they actually meet the claims stated using the Common Criteria framework. The Common Criteria provide a measure of assurance that specific objectives are present in a given product.

The Common Criteria use specific terminology to describe activity associated with the framework. The Target of Evaluation (TOE) is the product or system that is being evaluated. The Security Target (ST) is the security properties associated with a TOE. The Protection Profile (PP) is a set of security requirements associated with a class of products; i.e., firewalls have PPs and operating systems have PPs, but these may differ. PPs help streamline the comparison of products within product classes.

The output of the Common Criteria process is an evaluation assurance level (EAL), a set of seven levels, from 1, the most basic, through 7, the most comprehensive. The higher the EAL value, the higher the degree of assurance that a TOE meets the claims. Higher EAL does not indicate greater security.

ISO/IEC 15408 (Common Criteria) Evaluation Assurance Levels

The following table illustrates the levels of assurance associated with specific evaluation assurance levels correlated with the Common Criteria:

Evaluation Assurance Level (EAL)	TOE Assurance
EAL 1	Functionally tested
EAL 2	Structurally tested
EAL 3	Methodically tested and checked
EAL 4	Methodically designed, tested, and reviewed
EAL 5	Semiformally designed and tested
EAL 6	Semiformally verified, designed, and tested
EAL 7	Formally verified, designed, and tested

ISO/IEC 9126: Software Engineering – Product Quality

Product quality is an international standard for the evaluation of software quality. This four-part standard addresses some of the critical issues that adversely affect the outcome of a software development project. The standard provides a framework that defines a quality model for the software product. The standard addresses internal metrics that measure the quality of the software and external metrics that measure the software results during operation. Quality-of-use metrics are included to examine the software in particular scenarios.

ISO/IEC 9126 Quality Characteristics

ISO 9126 defines six quality characteristics that can be used to measure the quality of software:

- Functionality
- Reliability
- Usability
- Efficiency
- Maintainability
- Portability

ISO/IEC/IEEE 12207: Systems and Software Engineering – Software Life Cycle Processes

This international standard establishes a set of processes covering the lifecycle of the software. Each process has a defined set of activities, tasks, and outcomes associated with it. The standard acts to provide a common structure so all parties associated with the software development effort can communicate through a common vocabulary.

ISO/IEC 33001:2015 Information Technology – Process Assessment

Process assessment is also known as SPICE. SPICE originally stood for Software Process Improvement and Capability Evaluation, but international concerns over the term *evaluation* has resulted in the substitution of the term *determination* (SPICD). ISO 33001 is a set of technical standards documents for the computer software development process. The standard was derived from ISO/IEC 12207 and ISO 15504, the process lifecycle standard, and from maturity models like the CMM. ISO 33001 is used for process capability determination and process improvement efforts related to software development.

ISO 33001 defines a capability level on the following scale:

Level	Name
5	Optimizing process
4	Predictable process
3	Established process
2	Managed process
1	Performed process
0	Incomplete process

The ISO 33001 series consists of a series of documents. These documents contain a reference model and sets of process attributes and capability levels.

NIST

The National Institute of Standards and Technology is a federal agency that is charged with working with industry to develop technology, measurements, and standards that align with the interests of the U.S. economy. The Computer Security Division is the element of NIST that is charged with computer security issues, including those necessary for compliance with the Federal Information Security Management Act of 2002 (FISMA) and its successors. NIST develops and publishes several relevant document types associated with information security. The two main types of documents are Federal Information Processing Standards (FIPS) and the Special Publication (SP) 800 series from the NIST Information Technology Laboratory (ITL). The ITL's Computer Security Division also publishes security bulletins. Security bulletins are published on an average of six times a year, presenting an in-depth discussion of a single topic of significant interest to the information systems community. NIST also publishes Interagency or Internal Reports (NISTIRs) that describe research of a technical nature.

Federal Information Processing Standards

The Federal Information Processing Standards (FIPS) are mandatory sets of requirements on federal agencies and specific contractors. Although limited in number, they are wide sweeping in authority and scope. Older FIPS had sections describing a waiver process, but since the passage of FISMA, all aspects of FIPS are now mandatory, and the waiver process is no longer applicable.

NIST SP 800 Series

The more common set of NIST publications utilized by industry is the 800 series of Special Publications. These documents are designed to communicate the results of relevant research and guidelines associated with securing information systems. The 800 series has documents ranging from describing cryptographic protocols, to security requirements associated with a wide range of system elements, to risk management framework elements associated with information security governance.

Industry

SAFECODE is an industry-backed organization that is committed to increasing

communication between firms on the topic of software assurance. This group was formed by members who voluntarily share their practices, which together form a best practice solution. SAFECode is dedicated to communicating best practices that have been used successfully by member firms. A sampling of SAFECode's publications include

- Software Assurance: An Overview of Current Industry Best Practices
 - Fundamental Practices for Secure Software Development, 3rd Edition
 - Overview of Software Integrity Controls
 - Security Engineering Training
-



NOTE The users' stories for agile can be a valuable resource for CSSLP agile developers to explore. See "SAFECode Releases Software Security Guidance for Agile Practitioners." This paper provides practical software security guidance to agile practitioners in the form of security-focused stories and security tasks they can easily integrate into their agile-based development environments. You can find it at www.safecode.org/guidance-for-agile-practitioners/.

One of the strengths of SAFECode's publications is that they are not geared just for large firms, but are applicable across a wide array of corporate sizes, from large to small.

The Open Web Application Security Project (OWASP) is a worldwide free and open community focused on improving the security of application software. OWASP publishes a series of Top Ten vulnerability lists highlighting the current state-of-the-art threats facing web application developers. OWASP maintains a website (www.owasp.org) with significant resources to help firms build better software and eliminate these common and pervasive problems.

Prominent NIST Publications

The list of NIST security publications is long, covering many topics, but some of the more important ones are as follows:

FIPS 200	Minimum Security Requirements for Federal Information and Information Systems
FIPS 199	Standards for Security Categorization of Federal Information and Information Systems
FIPS 197	Advanced Encryption Standard (AES)
FIPS 186-3	Digital Signature Standard (DSS)
FIPS 190-4	Secure Hash Standard (SHS)
FIPS 140 series	Security Requirements for Cryptographic Modules
SP 800-152	A Profile for U.S. Federal Cryptographic Key Management Systems (CKMS)
SP 800-107	Recommendation for Applications Using Approved Hash Algorithms
SP 800-100	Information Security Handbook: A Guide for Managers
SP 800-63	Digital Identity Guidelines (a 4 volume set of documents)
SP 800-53	Security and Privacy Controls for Information Systems and Organizations
SP 800-30	Guide for Conducting Risk Assessments
SP 800-12	An Introduction to Computer Security

FISMA

The Federal Information Security Management Act of 2002 (FISMA) is a federal law that requires each federal agency to implement an agency-wide information security program. The National Institute of Standards and Technology (NIST) was designated the agency to develop implementation guidelines and did so through the publication of a risk management framework (RMF) for compliance. The initial compliance framework included the following set of objectives, which were scored on an annual basis by the Inspector General's office:

- Inventory of systems
- Categorize information and systems according to risk level
- Security controls
- Certification and accreditation of systems (including risk assessment and system security plans)
- Training

As the FISMA program has matured over the past two decades, NIST added the Information Security Automation Program and the Security Content Automation Protocol (SCAP). Currently, all accredited systems are supposed to have a set of monitored security controls to provide a level of continuous monitoring. FISMA is mandated for federal agencies and, by extension, contractors that implement and operate federal information systems. Like all security programs, the effectiveness of FISMA is directly related to the level of seriousness placed on it by senior management. When viewed as a checklist that is for compliance purposes, its effectiveness is significantly lower than in agencies that embrace the power of controls and continuous monitoring as a means to reduce system-wide risk.

Currently, NIST has responded with a series of publications detailing a security lifecycle built around a risk management framework. Detailed in NIST SP 800-39, a six-step process to create an RMF is designed to produce a structured, yet flexible, methodology of managing the risk associated with information systems. The six steps are

- 1.** Categorize information systems
- 2.** Select security controls
- 3.** Implement security controls
- 4.** Assess security controls
- 5.** Authorize information systems
- 6.** Monitor security controls

Each of these steps has a separate NIST Special Publication to detail the specifics. This is a process-based methodology of achieving desired security levels in an enterprise. CSSLPs will need to integrate their development work

into this framework in organizations that operate under an RMF.

Sarbanes-Oxley

The Sarbanes-Oxley Act of 2002 was a reaction to several major accounting and corporate scandals, costing investors billions and shaking public confidence in the stock markets. Although composed of many parts, the primary element concerned with information security is Section 404, which mandates a specific level of internal control measures. In simple terms, the information systems used for financial accounting must have some form of security control over integrity so that all may have confidence in the numbers being reported by the system. Criticized by many for its costs, it is nonetheless the current law, and financial reporting systems must comply.

Gramm-Leach-Bliley

The Financial Modernization Act of 1999, also known as the Gramm-Leach-Bliley Act (GLBA), contains elements designed to protect consumers' personal financial information (PFI). From a software perspective, it is important to understand that the act specifies rules as to the collection, processing, storage, and disposal of PFI. The three primary rules worth noting are

- The Financial Privacy Rule, which governs the collection and disclosure of PFI, including companies that are nonfinancial in nature
- The Safeguards Rule, which applies to financial institutions and covers the design, implementation, and maintenance of safeguards deployed to protect PFI
- The Pretexting Protections, which addresses the use of pretexting (falsely pretending) to obtain PFI

HIPAA and HITECH

While GLBA deals with PFI, the Healthcare Insurance Portability and Accountability Act (HIPAA) deals with personal health information (PHI). PHI contains information that can have significant value to criminal organizations. Enacted in 1996, the privacy provisions of HIPAA were not

prepared for the industry movement to electronic records. The Health Information Technology for Economic and Clinical Health Act (HITECH Act) is part of the American Recovery and Reinvestment Act of 2009 (ARRA) and is designed to enhance privacy provisions of electronic personal health information records.

Payment Card Industry Data Security Standard

PCI stands for Payment Card Industry, an industry group established to create, manage, and enforce regulations associated with the securing of cardholder data. There are three main standards: the Data Security Standard (PCI DSS), the Payment Application Data Security Standard (PA DSS), and the PIN Transaction Security (PTS). Each of these is designed to provide a basic level of protection for cardholder data.

The PCI DSS is the governing document that details the contractual requirements for members that accept and process bank cards. This standard includes requirements for security management, policies and procedures, network architecture, software design, and other critical protective measures for all systems associated with the processing and storing of cardholder data. Arranged in six groups of control objectives, 12 high-level requirements are detailed. Under each of these requirements are a significant number of subrequirements and testing procedures that are used to determine a baseline security foundation.

The PA DSS standard is a set of requirements used by software vendors to validate that a payment application is compliant with the requirements associated with PCI DSS. This document describes requirements in a manner consistent with software activity, not the firm's. This is relevant, as software vendors do not necessarily have to comply with PCI DSS, but when creating applications designed to handle cardholder data, compliance with PA DSS signals that the software is properly designed. Use of PA DSS alone is not sufficient, as there are non-software-associated requirements associated with cardholder data requirements in PCI DSS that are still necessary to be compliant.

One of the most important elements of the cardholder data is the PIN, and security aspects associated with the PIN are governed by the PTS standard. The majority of this standard applies to hardware devices known as PIN entry devices (PEDs).

PCI standards are contractual requirements and can carry severe financial penalties for failing to comply. If a firm accepts payment cards, stores payment card data, or makes products associated with payment cards, then there are PCI standards to follow. These are not optional, nor are they without significant detail, making them a significant compliance effort. And because of the financial penalties, their importance tends to be near the head of the line in the risk management arena.

Other Regulations

There are a myriad of lesser known, but equally important, regulations. Authentication for banking over the Internet is governed by rules from the Federal Financial Institutions Examination Council (FFIEC). Current FFIEC regulations state that authentication must be multifactor in nature at a minimum. Any systems designed for use in this environment must include this as a requirement.

Legal Issues

Legal issues frame a wide range of behaviors and work environments. This comes from the concept that when disputes between parties arise, the legal system is a method of resolving these disputes. Over time, a body of laws and regulations has been created to govern activities, providing a roadmap for behavior between parties.

Intellectual Property

Intellectual property is a legal term that recognizes that creations from the mind can be and are property to which exclusive control can be granted to the creator. A variety of different legal mechanisms can be used to protect the exclusive control rights. The association of legal mechanism to the property is typically determined by the type of property. The common forms of legal protection are patents, copyrights, trademarks, trade secrets, and warranties.

Patents

A patent is an exclusive right granted by a government to the inventor for a specified period of time. Patents are used to protect the inventor's rights in

exchange for a disclosure of the invention. Patent law can differ between countries. In the United States, the requirements of an invention is that it represent something new, useful, and nonobvious. It can be a process, a machine, an article of manufacture, or a composition of matter. Patents for software and designs have drawn considerable attention in recent years as to whether the ideas are nonobvious and “new.” Patents allow an inventor time to recoup their investment in the creation of an invention. They give their owners the right to prevent others from using a claimed invention, even if the other party claims they independently developed a similar item and there was no copying involved. Patent applications are highly specialized legal documents requiring significant resources to achieve success. For patent protection to occur, patents must be applied for prior to disclosure of the invention, with the specifics differing by country.

Software Patents

There is intense debate over the extent to which software patents should be granted, if at all. In the United States, patent law excludes issuing patents to abstract ideas, and this has been used to deny some patents involving software. In Europe, computer programs as such are typically excluded from patentability. There is some overlapping protection for software in the form of copyrights, which are covered in the next section. Patents can cover the underlying algorithms and methods embodied in the software. They can also protect the function that the software is intended to serve. These protections are independent of the particular language or specific coding.

Copyrights

A copyright is a form of intellectual property protection applied to any expressible form of an idea or information that is substantive and discrete. Copyrights are designed to give the creator of an original work exclusive rights to it, usually for a limited time. Copyrights apply to a wide range of creative, intellectual, or artistic items. The rights given include the right to be credited for the work, to determine who may adapt the work to other forms, who may perform the work, who may financially benefit from it, and other related rights. Copyrights are governed internationally through the Berne

Convention, which requires its signatories to recognize the copyright of works of authors from other signatory countries in the same manner as it recognizes the copyright of its own authors. For copyright to be enforceable, an application must be made to the copyright office detailing what is being submitted as original work and desiring protection. Unlike patents, this filing is relatively straightforward and affordable even by individuals.

Software Copyrights

Patent protection and copyright protection constitute two different means of legal protection that may cover the same subject matter, such as computer programs, since each of these two means of protection serves its own purpose. Using copyright, software is protected as works of literature under the Berne Convention. Copyright protection allows the creator of a program to prevent another entity from copying it.

Copyright law prohibits the direct copying of some or all of a particular version of a given piece of software, but it does not prevent other developers from independently writing their own versions. A common practice in the industry is to publish interface specifications so that programs can correctly interface with specified functions; this places specific limitations on input and output specifications and would not result in copyright violations.

Trademarks

A trademark is a recognizable quality associated with a product or firm. The nature of the trademark is to build a brand association, and hence, copying by others is prohibited. Trademarks can be either common law–based or registered. Registering a trademark with the government provides significantly more legal protection and recovery options. Internationally, trademarks are managed through the World Intellectual Property Organization, using protocols developed in Madrid, referred to as the Madrid System.

Names are commonly trademarked to protect a brand image. In this vein, [Amazon.com](#) is trademarked, as it is used to project the image of the firm. Common terms or simply descriptive terms are not eligible for trademark

protection. In fact, trademark holders must protect their trademarks from general generic use not aligned with their products, as they can lose a trademark that becomes a generic term.

Trade Secrets

Trade secrets offer the ultimate in time-based protection for intellectual property. A trade secret is just that—a secret. Trade secrets are protected by a variety of laws, with the requirement that a firm keep a secret a secret, or at least make a reasonable attempt to do so. The most famous trade secrets typically revolve around food and taste, such as Coca-Cola's recipe or Kentucky Fried Chicken's recipe. Should someone manage to steal the recipes, they could then attempt to sell them to a competitor, but such attempts fail, as no respectable corporation would subject itself to the legal ramifications of attempting to circumvent legal protections for intellectual property. One issue with trade secrets is that should someone independently discover the same formula, then the original trade secret holder has no recourse.

Trade secrets are difficult to use in software, as the distribution of software, even compiled, provides the end user with access to much information. There are limited cases where cryptographic algorithms or seeds may be considered trade secrets, as they are not passed to clients and can be protected. There is a limited amount of related protection under the reverse-engineering provisions of the U.S. Digital Millennium Copyright Act, where reverse-engineering of security safeguards is prohibited.

Warranties

Warranties represent an implied or contractually specified promise that a product will perform as expected. When you buy computer hardware, the warranty will specify that for some given period of time the hardware will perform to a level of technical specification and, should it fail to do so, will outline the vendor's responsibilities. The warranty typically does not guarantee that the hardware will perform the tasks the user bought it for—merely that it will work at some specified technical level. Warranty is necessary for fitness for use, but is not sufficient.

With respect to software, the technical specification, i.e., the program performs as expected, is typically considered by the end user to be fitness for

use on the end user's problem. This is not what a vendor will guarantee; in fact, most software licenses specifically dismiss this measure, claiming the software is licensed using phrases such as "as-is" and "no warranty as to use" or "no vendor responsibility with respect to any failures resulting from use."

Data Classification

Data can be classified in several different manners, each with a level of significance for security. Data can be classified as to its state, its use, or its importance from a security perspective. As these are overlapping schemes, it is important to understand all aspects of data classification before determining how data should be handled as part of the development process. Data classification is a risk management tool, with the objective to reduce the costs associated with protecting data. One of the tenets of security is to match the level of protection and cost of security to the value of the asset under protection. Data classification is one of the tools that are used to align protection and asset value in the enterprise.

Data classification can be simple or fairly complex, depending on the size and scope of the enterprise. A small enterprise may be sufficiently covered with a simple strategy, whereas a large enterprise may have a wide range of data protection needs. In large enterprises, it may be desirable to actually determine separate, differing protection needs based on data attributes such as confidentiality, integrity, and availability. These attributes may be expanded to include specific compliance requirements.

One way of looking at data-related security is to take a data-centric view of the security process. Always examining what protections are needed for the data across the entire lifecycle can reveal weaknesses in enterprise protection schemes.

Data States

Data can be considered a static item that exists in a particular state at any given time. For the purposes of development and security, these states are

- At rest, or being stored
- Being created

- Being transmitted from one location to another
- Being changed or deleted

In addition, one should consider where the data is currently residing:

- On permanent media (hard drive, CD/DVD/optical disc)
- On remote media (USB, cloud/hosted storage)
- In RAM on a machine

When considering data states, it is easy to expand this idea to the information lifecycle model (ILM), which includes generation, retention, and disposal.

Data Usage

Data can also be classified as to how it is going to be used in a system. This is meant to align the data with how it is used in the business and provide clues as to how it should be shared, if appropriate. The classifications include

- **Internal data** initialized in the application, used in an internal representation, or computed within the application itself
- **Input data** read into a system and possibly stored in an internal representation or used in a computation and stored
- **Output data** written to an output destination following processing

In addition, data can be considered security sensitive, marked as containing personally identifiable information (PII) or to be hidden. These categories include

- **Security-sensitive data** A subset of data of high value to an attacker
- **PII data** that contains PII elements
- **Hidden data** that should be concealed to protect it from unauthorized disclosure using obfuscation techniques

Data Risk Impact

Data can be classified as to the specific risk associated with the loss of the

data. This classification is typically labeled high, medium, and low, although additional caveats of PII or compliance may be added to include elements that have PII or compliance issues if disclosed.

Data that is labeled high risk is data that if disclosed or lost could result in severe or catastrophic adverse effect on assets, operations, or people. The definition of severe will vary from firm to firm in financial terms, as what is severe for a small firm may be of no consequence to a multinational firm.

Data that is labeled medium risk is data that if disclosed would have serious consequences. Low-risk data is data that has limited, if any, consequences if lost or disclosed. Each firm, as part of its data management plan, needs to determine the appropriate definitions of severe, serious, and limited, both from a dollar loss point of view and from an operational impact and people impact point of view.

Additional labels, such as PII, compliance-related, or for official use only, can be used to alert the development team as to specialized requirements associated with data elements.

Data Lifecycle

Data in the enterprise has a lifecycle. It can be created, used, stored, and even destroyed. Although data storage devices have come down in price, the total cost of storing data in a system is still a significant resource issue. Data that is stored must also be managed from a backup and business continuity/disaster recovery perspective. Managing the data lifecycle is a data owner's responsibility. Ensuring the correct sets of data are properly retained is a business function, one that is best defined by the data owner.

Generation

Data can be generated in the enterprise in many ways. It can be generated in a system as a result of operations, or it can be generated as a function of some input. Regardless of the path to generation, data that is generated has to be managed at this point—is it going to be persistent or not? If the data is going to be persistent, then it needs to be classified and have the appropriate protection and destruction policies assigned. If it is not going to be persistent—that is, it is some form of temporary display or calculation—then these steps are not necessary.

Data Ownership

Data does not really belong to people in the enterprise; it is actually the property of the enterprise or company itself. That said, the enterprise has limited methods of acting except through the actions of its employees, contractors, and other agents. For practical reasons, data will be assigned to people in a form of an ownership or stewardship role. Ownership is a business-driven issue, because the driving factors behind the data ownership responsibilities are business reasons.

Data Owner

Data owners act in the interests of the enterprise when managing certain aspects of data. The data owner is the party who determines who has specific levels of access associated with specific data elements: who can read, who can write, who can change, delete, and so on. The owner is not to be confused with the custodian, the person who actually has the responsibility for making the change. A good example is in the case of database records. The owner of the data for the master chart of accounts in the accounting system may be the chief financial officer (CFO), but the ability to directly change it may reside with a database administrator (DBA).



EXAM TIP Data owners are responsible for defining data classification, defining authorized users and access criteria, defining the appropriate security controls, and making sure the controls are implemented and operational.

This brings forth the issue of data custodians, or people who have the ability to directly interact with the data. Data owners define the requirements, while data custodians are responsible for implementing the desired actions.

Data Custodian

Data custodians support the business use of the data in the enterprise. As such, they are responsible for ensuring that the processes safely transport, manipulate, and store the data. Data custodians are aware of the data

management policies issued by the data owners and are responsible for ensuring that during operations these rules and regulations are followed.



EXAM TIP Data custodians are responsible for maintaining defined security controls, managing authorized users and access controls, and performing operational tasks such as backups and data retention and disposal.

Data custodians may not require access to read the data elements. They do need appropriate access to apply policies to the data elements. Without appropriate segregation of data controls to ensure custodians can only manage the data without actually reading the data, confidentiality is exposed to a larger set of people, a situation that may or may not be desired.

Labeling

Because data can exist in the enterprise for an extended period of time, it is important to label the data in a manner that can ensure it is properly handled. For data in the enterprise, the use of metadata fields, which is data about the data, can be used to support data labeling. The metadata can be used to support the protection of the data by providing a means to ensure a label describing the importance of the data is coupled with it.

Sensitivity

Data can have different levels of sensitivity within an organization. Payroll data can be sensitive, with employees having restricted access. But some employees, based on position, may need specific access to this type of data. A manager has a business reason to see and interact with salary and performance data for people under his or her direct management, but not others. HR personnel have business reasons to access data such as this, although in these cases the access may not be just by job title or position, but also by current job task. Understanding and properly managing sensitive data can prevent issues should it become public knowledge or disclosed. The commonsense approach is built around business purpose—if someone has a

legitimate business purpose, they should have appropriate access. If not, then they should not have access. The challenge is in defining the circumstances and building the procedures and systems to manage data according to sensitivity. Fortunately, the range of sensitive data is typically limited in most organizations.

Impact

The impact that data can have when improperly handled is a much wider concern than sensitivity. Virtually all data in the enterprise can, and should, be classified by impact. Data can be classified by the impact the organization would suffer in the event of data loss, disclosure, or alteration. Impact is a business-driven function, and although highly qualitative in nature, if the levels of high, medium, and low impact are clearly defined, then the application of the impact designation is fairly straightforward:

- Typically, three levels are used: high, medium (or moderate), and low.
- Separate levels of impact may be defined by data element for each attribute. For example, a specific data element could have high impact for confidentiality and high for integrity, but low for availability.



EXAM TIP NIST FIPS 199 and SP 800-18 provide a framework for classifying data based on impacts across the three standard dimensions: confidentiality, integrity, and availability.

The first step in impact analysis is defining the levels of high, medium, and low. The idea behind the high level is to set the bar high enough that only a reasonably small number of data elements are included. The exception to this is when the levels are set with some specific criteria associated with people. The differentiators for the separation of high, medium, and low can be based on impact to people, impact on customers, and financial impact.

Table 4-1 shows a typical breakdown of activity.

Impact	Personnel	Customer	Financial (Specific \$ May Vary)
High	Death or maiming	Severe impact to current and future customer relations	Loss of \$1 million or more
Medium	Severe injury, loss of functionality	Significant impact to current and future customer relations	Loss of \$100,000 or more
Low	Minor injury	Minor impact to current and future customer relations	Loss of less than \$100,000

Table 4-1 Summary of Impact Level Definitions

Each organization needs to define its own financial limits—a dollar loss that would be catastrophic to some organizations is a rounding error to others. The same issue revolves around customer-related issues—what is severe in some industries is insignificant in others. Each organization needs to completely define each of the levels summarized in [Table 4-1](#) for its own use.

Privacy

Privacy is the principle of controlling information about one's self: who it is shared with, for what purpose, and how it is used and transferred to other parties. Control over one's information is an issue that frequently involves making a choice. To buy something over the Internet, you need to enter a credit card or other payment method into a website. If you want the item delivered to your house, you need to provide an address, typically your home address. While it may seem that the answer to many privacy issues is simple anonymization, and with the proper technology it could be done, the practical reality requires a certain level of traceable sharing. To obtain certain goods, a user must consent to share their information. The issues with privacy then become one of data disposition—what happens to the data after it is used as needed for the immediate transaction.

If the data is stored for future orders, safeguards are needed. In the case of credit card information, regulations such as PCI DSS dictate the requirements for safeguarding such data. Data can also be used to test systems. However, the use of customer data for system testing can place the customer data at

risk. In this instance, anonymization can work. Proper test data management includes an anonymization step to erase connection to meaningful customer information before use in a test environment.

Privacy and Software Development

Privacy may seem like an abstract issue for CSSLP, but the ramifications associated with software development and privacy are significant. Gone are the days of collecting and storing any data, in any form, and in any way. There are a myriad of privacy rules and regulations, and development teams need to be aware of the general issues so that they can properly apply their skills to meeting the specific requirements of a project. If a project collects personal data or stores it and there are no specific requirements with respect to privacy, then the team should know to raise the questions of which rules and regulations are likely to apply.

Privacy Policy

The privacy policy is the high-level document that describes the principles associated with the collection, storage, use, and transfer of personal information within the scope of business. A privacy policy will detail the firm's responsibility to safeguard information. A business needs to collect certain amounts of personal information in the course of regular business. A business still has a responsibility to properly secure the information from disclosure to unauthorized parties. A business may have partners with which it needs to share elements of personal information in the course of business. A firm may also choose to share the information with other parties as a revenue stream. The privacy policy acts as a guide to the employees as to their responsibilities associated with customer information.

A customer-facing privacy policy, commonly referred to as *privacy disclosure statement*, is provided to customers to inform them of how data is protected, used, and disposed of in the course of business. In the financial sector, the Gramm-Leach-Bliley Act mandates that firms provide clear and accurate information as to how customer information is shared.

Personally Identifiable Information

Information that can be used to specifically identify an individual is referred to as *personally identifiable information*. PII is viewed as a technical term, but it has its roots in legal terms. One of the primary challenges associated with PII is the effect of data aggregation. Obtaining several pieces from different sources, a record can be constructed that permits the identification of a specific individual. Recognizing this, the U.S. government defines PII using the following from an Office of Management and Budget (OMB) Memorandum:

Information which can be used to distinguish or trace an individual's identity, such as their name, social security number, biometric records, etc., alone, or when combined with other personal or identifying information which is linked or linkable to a specific individual, such as date and place of birth, mother's maiden name, etc.

Common PII Elements

The following items are commonly used to identify a specific individual and are, hence, considered PII:

- Full name (if not common)
- National identification number (i.e., SSN)
- IP address (in some cases)
- Home address
- Motor vehicle registration plate number
- Driver's license or state ID number
- Face, fingerprints, or handwriting
- Credit card and bank account numbers
- Date of birth
- Birthplace
- Genetic information

To identify an individual, only a small subset may be needed. A study by Carnegie Mellon University found that nearly 90 percent of the U.S. population could be uniquely identified with only gender, date of birth, and ZIP code.

Personal Health Information

Personal health information, also sometimes called *protected health information*, is the set of data elements associated with an individual's healthcare that can also be used to identify a specific individual. These elements can include, but are not limited to, PII elements, demographic data, medical test data, biometric measurements, and medical history information. This data can have significant risk factors to an individual should it fall into the possession of unauthorized personnel. For this reason, as well as general privacy concerns, PHI is protected by a series of statutes, including HIPAA and the HITECH Act.



NOTE PHI and associated medical data are sought after by cybercriminals because they contain both insurance information and financial responsibility information, including credit cards, both of which can be used in fraud. In addition, there is sufficient PII for an identity to be stolen, making health records a highly valued source of information for cybercriminals.

Breach Notifications

When security fails to secure information and information is lost to parties outside of authorized users, a breach is said to have occurred. Data breaches trigger a series of events. First is the internal incident response issue—what happened, how it happened, what systems/data were lost, and other questions that need to be answered. In a separate vein, customers whose data was lost deserve to be informed. The state of California was the first to address this issue with SB 1386, a data disclosure law that requires

a state agency, or a person or business that conducts business in California, that owns or licenses computerized data that includes personal information, as defined, to disclose in specified ways, any breach of the security of the data, as defined, to any resident of California whose unencrypted personal information was, or is reasonably believed to have been, acquired by an unauthorized person.

Two key elements of the law are “unencrypted personal information” and “reasonably believed to have been acquired by an unauthorized party.” Encrypting data can alleviate many issues associated with breaches. “Reasonably believed” means that certainty as to loss is not necessary, thus increasing the span of reportable issues. Since its start in July 2003, other states have followed with similar measures. Although no federal measure exists, virtually every state and U.S. territory is covered by a state disclosure law.

General Data Protection Regulation

Two factors led to what can only be seen as a complete rewrite of EU data protection regulations. In light of the Snowden revelations, the EU began a new round of examining data protection when shared with the United States and others. This brought Safe Harbor provisions into the spotlight as the EU wanted to renegotiate stronger protections. Then, the European Court of Justice invalidated the Safe Harbor provisions. This led the way to the passage of the General Data Protection Regulation (GDPR), which went into effect in May 2018.



NOTE The Safe Harbor principles that once allowed the harmonization of U.S. and EU privacy rules no longer are sufficient. GDPR has made these methods obsolete, and GDPR rules must be followed for customers in the EU.

The GDPR ushers in a brand-new world with respect to data protection

and privacy. With global trade being important to all countries, the ability to transfer data, including personal data, between parties becomes important. Enshrined in the Charter of Fundamental Rights of the EU is the fundamental right to the protection of personal data, including when such data elements are transferred outside the EU. Recognizing that, the new set of regulations is more expansive and restrictive, making the Safe Harbor provisions obsolete. For all firms that want to trade with the EU, there is now a set of privacy regulations that will require specific programs to address the requirements.

The GDPR brings many changes, one being the appointment of a data protection officer (DPO). This role may be filled by an employee or a third-party service provider (for example, consulting or law firm), and it must be a direct report to the highest management level. The DPO should operate with significant independence, and provisions in the GDPR restrict control over the DPO by management.

GDPR Considerations

The GDPR requires significant consideration, including the following:

- Assessing personal data flows from the EU to the United States to define the scale and scope of the cross-border privacy-compliance challenge
- Assessing readiness to meet model clauses, remediate gaps, and organize audit artifacts of compliance with the clauses
- Updating privacy programs to ensure they are capable of passing an EU regulator audit
- Conducting EU data-breach notification stress tests
- Monitoring changes in EU support for model contracts and binding corporate rules

The GDPR specifies requirements regarding consent, and they are significantly more robust than previous regulations. Consent requirements are also delineated for specific circumstances:

- Informed/affirmative consent to data processing. Specifically, “a

statement or a clear affirmative action” from the data subject must be “freely given, specific, informed and unambiguous.”

- Explicit consent to process special categories of data. Explicit consent is required for “special categories” of data, such as genetic data, biometric data, and data concerning sexual orientation.
- Explicit parental consent for children’s personal data.
- Consent must be specific to each data-processing operation, and the data subject can withdraw consent at any time.

The GDPR provides protections for new individual rights, and these may force firms to adopt new policies to address these requirements. The rights include the Right to Information, Right to Access, Right to Rectification, Right to Restrict Processing, Right to Object, Right to Erasure, and Right to Data Portability. Each of these rights is clearly defined with technical specifics in the GDPR. The GDPR also recognizes the risks of international data transfer to other parties and has added specific requirements that data protection issues be addressed by means of appropriate safeguards, including binding corporate rules (BCRs); model contract clauses (MCCs), also known as standard contractual clauses (SCCs); and legally binding documents. These instruments must be enforceable between public authorities or bodies, as well as all who handle data.

The differences in approach between the United States and the EU with respect to data protection led the EU to issue expressions of concern about the adequacy of data protection in the United States, a move that could have paved the way to the blocking of data transfers. This has forced U.S. and other international companies to adapt their privacy protections to at least align with the GDPR for EU customers.

GDPR Personal Data Elements

Under GDPR, personal data is defined as any information relating to an identified or identifiable natural person. This includes the following if they are capable of being linked back to the data subject:

- Online identifiers
- IP addresses

- Cookies

This also includes indirect information, including physical, physiological, genetic, mental, economic, cultural, or social identities that can be traced back to a specific individual.

GDPR demands that individuals must have full access to information on how their data is processed, and this information should be available in a clear and understandable way. When companies obtain data from an individual, some of the issues that must be made clear to the individual whose data is being collected are

- The identity and contact details of the organization behind the data request (who is asking)
- The purpose of acquiring the data and how it will be used (why they are asking)
- The period for which the data will be stored (how long they will keep it)
- Whether the data will be transferred internationally (where else it can go)
- The individual's right to access, rectify, or erase the data (right to be forgotten)
- The individual's right to withdraw consent at any time (even after collection)
- The individual's right to lodge a complaint

Several of these elements can be difficult to operationally implement unless they are planned into the software itself. One of the common elements in the press is the right to be forgotten. GDPR mandates that individuals must be able to withdraw consent at any time and have a right to be forgotten. Included in this right is that if data is no longer required for the reasons for which it was collected, it must be erased.

California Consumer Privacy Act 2018 (AB 375)

In June 2018, California passed a sweeping privacy bill that holds many

similarities to GDPR. Passed in response to the threat of a ballot initiative, AB 375 mandates several key elements.

While individuals must opt out of sharing, they

- Have a right to know how personal data is being used
- Have a right to disclosure and objection relating to who data is being sold to
- Have a right to know who data has been provided to
- Experience no discrimination if they object to data being sold
- Have a right to access the data being held

The full ramifications of AB 375 are not yet known as the bill only became law in mid-2020 and it may take years to fully understand the implications. Privacy laws are going to multiply in the future.

Privacy-Enhancing Technologies

One principal connection between information security and privacy is that without information security, you cannot have privacy. If privacy is defined as the ability to control information about oneself, then the aspects of confidentiality, integrity, and availability from information security become critical elements of privacy. Just as technology has enabled many privacy-impacting issues, technology also offers the means in many cases to protect privacy. An application or tool that assists in such protection is called a *privacy-enhancing technology* (PET).

Encryption is at the top of the list of PETs for protecting privacy and anonymity. One of the driving factors behind Phil Zimmerman's invention of PGP was the desire to enable people living in repressive cultures to communicate safely and freely. Encryption can keep secrets secret, and it's a prime choice for protecting information at any stage in its lifecycle. The development of Tor routing to permit anonymous communications, coupled with high-assurance, low-cost cryptography, has made many web interactions securable and safe from eavesdropping.

Other PETs include small application programs called *cookie cutters* that are designed to prevent the transfer of cookies between browsers and web servers. Some cookie cutters block all cookies, while others can be

configured to selectively block certain cookies. Some cookie cutters also block the sending of HTTP headers that might reveal personal information but might not be necessary to access a website, as well as block banner ads, pop-up windows, animated graphics, or other unwanted web elements. Some related PET tools are designed specifically to look for invisible images that set cookies (called *web beacons* or *web bugs*). Other PETs are available to PC users, including encryption programs that allow users to encrypt and protect their own data, even on USB keys.

Data Minimization

Data minimization is one of the most powerful privacy-enhancing technologies. In a nutshell, it involves not keeping what you don't need. Limiting the collection of personal information to that which is directly relevant and necessary to accomplish a specified purpose still allows the transactions to be accomplished, but it also reduces risk from future breaches and disclosures by not keeping “excess” data. In the EU, privacy rules are built around the idea that individuals own the rights to the reuse of their data, and unless they grant it to a company, the right to store and reuse the data beyond the immediate transaction is prohibited. This serves several purposes, but one important outcome is that when a breach/disclosure event occurs, the reach of the PII loss is limited.

While you may need to have a reasonable amount of PII to process and ship an order, once that process has concluded, do you need the data? There may be a need for a reasonable period for returns, warranty claims, and so on, but once that period has passed, destroying unneeded PII removes it from the chance of disclosure.

Data Masking

Data masking involves the hiding of data by substituting altered values. A mirror version of a database is created, and data modification techniques such as character shuffling, encryption, and word or character substitution are applied to change the data. Another form is to physically redact elements by substituting a symbol such as * or x. This is seen on credit card receipts, where the majority of the digits are removed in this fashion. Data masking makes reverse engineering or detection impossible.



NOTE Data masking hides personal or sensitive data but does not render it unusable.

Tokenization

Tokenization is the use of a random value to take the place of a data element that has traceable meaning. A good example of this is when you have a credit card approval, you do not need to keep a record of the card number, the cardholder's name, or any of the sensitive data concerning the card verification code (CVC) because the transaction agent returns an approval code, which is a unique token to that transaction. You can store this approval code, the token, in your system, and if there comes a time you need to reference the original transaction, this token provides you with complete traceability to it and yet, if disclosed to an outside party, reveals nothing.

Tokens are used all the time in data transmission systems involving commerce because they protect the sensitive information from being reused or shared, yet they maintain the desired nonrepudiation characteristics of the event. Tokenization is not an encryption step because encrypted data can be decrypted. By substituting a nonrelated random value, tokenization breaks the ability for any outside entity to "reverse" the action because there is no connection.



NOTE Tokenization assigns a random value that can be reversed or traced back to the original data.

Anonymization

Data anonymization is the process of protecting private or sensitive information by removing identifiers that connect the stored data to an individual. Separating the PII elements such as names, Social Security

numbers, and addresses from the remaining data through a data anonymization process retains the usefulness of the data but keeps the connection to the source anonymous. Data anonymization is easier said than done, because data exists in many places in many forms. This permits data aggregators to collect multiple instances and then, through algorithms and pattern matching, de-anonymize the data through multiple cross-references against multiple sources.

Pseudo-anonymization

Pseudo-anonymization is a de-identification method that replaces private identifiers with fake identifiers or pseudonyms (for example, replacing the value of the name identifier “Mark Sands” with “John Doe”). Not all uniquely identifying fields are changed because some, such as date of birth, may need to be preserved to maintain statistical accuracy. Noise can be added to some fields to remove direct connections, while maintaining the approximate value; for example, randomly adding or subtracting three days to/from the actual date of birth preserves the age but de-identifies the original record. Pseudonymization preserves statistical accuracy and data integrity, allowing the modified data to be used for training, development, testing, and analytics while protecting data privacy.

Chapter Review

This chapter began with an examination of the regulatory and compliance elements that are commonly associated with secure development. Examples of these elements include security standards such as those put out by ISO and NIST, laws and regulations such as FISMA, Sarbanes-Oxley, Gramm-Leach-Bliley, HIPAA, and HITECH. Contractual schemes such as PCI DSS were also covered. The chapter then discussed legal issues and intellectual property concerns.

The chapter then pivoted to data classification, examining how data and its descriptors and uses have a material effect on the security of a software solution. The chapter closed with an examination of privacy, the laws and regulations that are associated with privacy, and technologies used to enhance privacy.

Quick Tips

- Regulations and compliance elements can result in specific requirements as part of software development.
- Common sources of compliance elements can be from security standards, government documents, government regulations concerning financial elements, and contractual standards.
- Protection of intellectual property is an element of concern during the requirements process.
- Classification of data and handling rules are important elements that will drive specific requirements.
- Data ownership, labeling, data types, and data lifecycles are important requirement drivers.
- Privacy implications have impacts on software requirements.

Questions

To further help you prepare for the CSSLP exam, answer the following questions. The correct answers are provided in the following section.

1. What party determines which users or groups should have access to specific data elements?
 - A. Data custodian
 - B. Data manager
 - C. System administrator
 - D. Data owner
2. Which of the following would not be considered structured data?
 - A. Excel spreadsheet of parts prices
 - B. Oracle database of customer orders
 - C. XML file of parts and descriptions
 - D. Log file of VPN failures
3. Which of the following is not a stage of the data lifecycle?
 - A. Retention

- B. Disposal
 - C. Sharing
 - D. Generation
- 4. What party is responsible for defining data classification?
 - A. Data custodian
 - B. Senior manager (CIO)
 - C. Security management
 - D. Data owner
- 5. What is the primary governing law for federal computer systems?
 - A. NIST
 - B. Sarbanes-Oxley
 - C. FISMA
 - D. Gramm-Leach-Bliley
- 6. Which of the following is a security standard associated with the collection, processing, and storing of credit card data?
 - A. Gramm-Leach-Bliley
 - B. PCI DSS
 - C. HIPAA
 - D. HITECH
- 7. When using customer data as test data for production testing, what process is used to ensure privacy?
 - A. Data anonymization
 - B. Delinking
 - C. Safe Harbor principles
 - D. Data disambiguation
- 8. Which of the following is not a common PII element?
 - A. Full name
 - B. Order number
 - C. IP address

- D.** Date of birth
- 9.** Which of the following is an important element in preventing a data breach when backup tapes are lost in transit?
- A.** Service level agreements with a backup storage company
 - B.** Use of split tapes to separate records
 - C.** Proprietary backup systems
 - D.** Data encryption
- 10.** To interface data sharing between U.S. and European firms, what should be invoked?
- A.** GDPR principles
 - B.** Safe Harbor principles
 - C.** Onward transfer protocol
 - D.** Data protection regulation

Answers

- 1.** **D.** The data owner is the party who determines who has specific levels of access associated with specific data elements.
- 2.** **A.** Microsoft Office files are considered unstructured data.
- 3.** **C.** The stages of the lifecycle are generation, retention, and disposal.
- 4.** **D.** Data owners are responsible for defining data classification.
- 5.** **C.** The Federal Information Security Management Act of 2002 (FISMA) is a federal law that requires each federal agency to implement an agency-wide information security program.
- 6.** **B.** The PCI DSS is the governing document that details the contractual requirements for members that accept and process bank cards.
- 7.** **A.** Anonymizing the data, stripping it of customer PII, is part of the test data management process.
- 8.** **B.** Order numbers cannot be correlated to other PII elements, making them non-PII.
- 9.** **D.** Encrypted data is no longer useful data, but simply ones and zeros.

- 10. A.** GDPR principles apply to all EU data. The Safe Harbor principles that allowed the harmonization of U.S. and EU privacy rules no longer are sufficient.

Misuse and Abuse Cases

In this chapter you will

- Learn how to develop misuse and abuse cases
 - Understand the security requirements traceability matrix (RTM)
 - See how to ensure that security requirements flow down to suppliers/providers
-
-

Use cases have been used for years to contextualize how software is to be used, conveying requirements with respect to function to developers. Misuse cases provide the opposite case, a set of requirements that should specifically not be allowed. If a use case conveys what should happen when an authorized user requests authorized functions, the misuse case is when a user attempts to do something that should be prohibited.

Misuse/Abuse Cases

Misuse or abuse cases can be considered a form of use case illustrating specifically prohibited actions. Although one could consider the situation that anything not specifically allowed should be denied, making this redundant, misuse cases still serve a valuable role in communicating requirements to developers and testers. [Figure 5-1](#) illustrates a series of misuse cases associated with the online account management system.

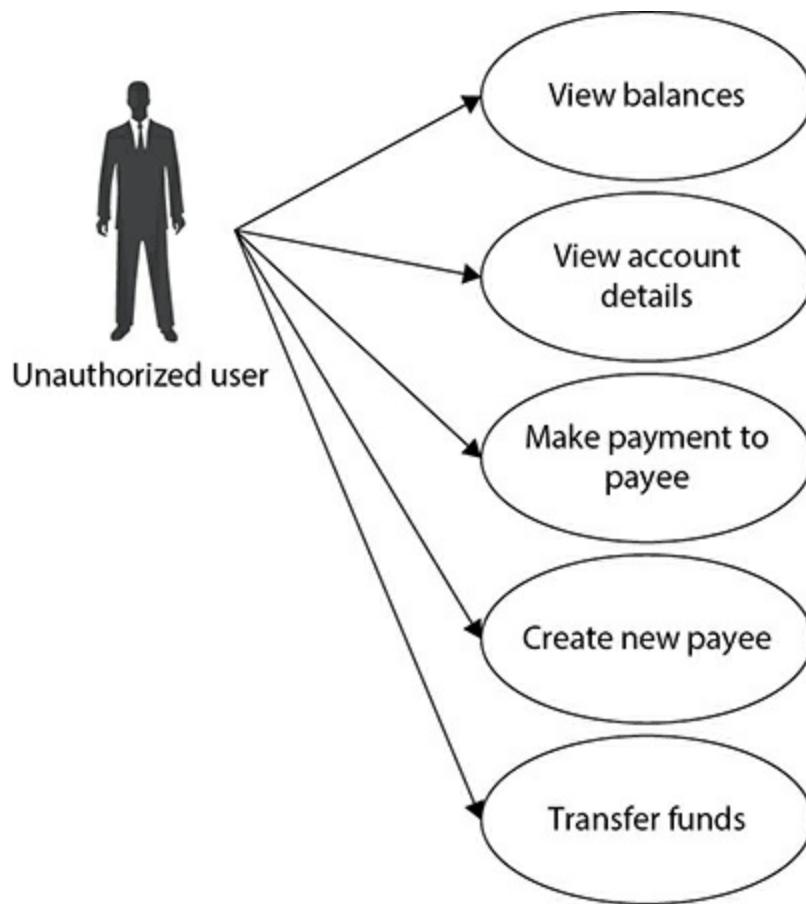


Figure 5-1 Misuse-case diagram

In this diagram, the actor is now labeled as unauthorized user. This is different from the previous authenticated user, as this misuse actor may indeed be authenticated. The misuse actor could be another customer or an internal worker with some form of access required to manage the system. Through brainstorming exercises, the development team has discovered the possibility for someone with significant privilege—that is, a system administrator—to have the ability to create a new payee on an account. This would enable them to put themselves or a proxy into the automatic bill-pay system. This would not be an authorized transaction, and to mitigate such activity, a design of an out-of-band mechanism—that is, e-mail the user for permission—makes it significantly more difficult for this activity to be carried out, as the misuser must now also have e-mail access to the other user's information to approve the new payee. What this misuse case specifically does is draw attention to ensuring that authenticated but not authorized users do not have the ability to interact in specific ways. As use

cases also drive testing, this misuse case ensures that these issues are also tested as another form of defense.

Misuse cases are used early in the development process, during the requirements phase, to decide and document how the software should react to improper use. The simplest, most practical method for creating misuse cases is usually through a process of informed brainstorming. This is frequently done as part of the development of a threat model for the software and is done by a team involving both developers and security personnel acting to find ways to abuse the software functionality.

Misuse cases can present commonly known attack scenarios and are designed to facilitate communication among designers, developers, and testers to ensure that potential security holes are managed in a proactive manner. Misuse cases can examine a system from an attacker's point of view, whether the attacker is an inside threat or an outside one. Properly constructed misuse cases can trigger specific test scenarios to ensure known weaknesses have been recognized and dealt with appropriately before deployment.



NOTE SAFECode has made significant contributions to development and distribution use cases. It has published a useful document describing "Practical Security Stories and Security Tasks for Agile Development Environments," available for use and free download at https://safecode.org/publication/SAFECode_Agile_Dev_Security0712.pdf.

Misuse cases can be used to help document the types of nonfunctional or quality requirements, such as reliability, resiliency, maintainability, testability, and so on. Security failures occur whenever software performs a function that is not authorized, whether by bad input (hostile attacker) or environmental conditions (errors or unseen conditions). While security threats many times stem directly from genuinely hostile agents, the reliability requirements may be elicited and analyzed as threats caused by things that are not necessarily intelligent. Such causes include human error, storms, design errors (e.g., software bugs), and interference or noise across communication

links. All of these conditions can cause software crashes and other types of failure. Misuse cases provide a means to model these conditions and test them during the course of software development.

It is not difficult to think of misuse cases for other functional requirements such as user interfaces. For instance, what if a novice operator becomes confused by the user interface and enters commands that are incorrect for an instance. Something as simple as a number off by an extra zero, 100 in place of 10. Usability is one of the hallmarks of use cases and has a role in misuse cases as well. When examining the threat model and the implications of threats to function, misuse cases provide a means of communication this information to the entire development team.

In the end, there is a functional diagram describing how the software functions and how it should not function. In this diagram use cases and misuse cases can be connected. When a misuse case intersects with a use case, taking the example of encryption being used to validate a user, this use case is said to mitigate the misuse case. It is important for all misuse cases to be properly mitigated, else they represent a threat and potential risk to the system.

Requirements Traceability Matrix

The requirements traceability matrix (RTM) is a grid that assists the development team in tracking and managing requirements and implementation details. The RTM assists in the documentation of the relationships between security requirements, controls, and test/verification efforts. [Table 5-1](#) illustrates a sample RTM. An RTM allows the automation of many requirements, providing the team to gather sets of requirements from centralized systems. The security requirements could be brought in en masse from a database based on an assessment of what systems and users will be involved in the software.

Requirement ID Number	Requirement Description	Requirement Source	Test Objective(s)	Verification Method(s)	Use Cases
A unique identifier	Description of each requirement that is to be verified	Source of the requirement	Individual test objective to illustrate compliance	Method used to verify the test objective	

Table 5-1 Sample Requirements Traceability Matrix

Software with only internal users will have a different set of requirements from that of a customer interface across the Web. Many development teams will have predefined sets of requirements for infrastructure, security, data sources, and the like, and using the RTM to promulgate them to the development teams will go a long way in ensuring critical requirements are not overlooked.

An RTM acts as a management tool and documentation system. By listing all the requirements and how they can be validated, it provides project managers with the information they need to ensure all requirements are appropriately managed and that none is missed. The RTM can assist with use-case construction and ensure that elements are covered in testing.

Software Acquisition

Software is not always created as a series of greenfield exercises, but rather, it is typically created by combining existing elements, building systems by connecting separate modules. Not all software elements will be created by the development team. Acquisition of software components has security implications, and those are covered in detail in Chapter 20. But acquisition is an important component that has connections throughout the lifecycle, so what follows is a brief overview of how this topic fits into the CSSLP discussion.

Definitions and Terminology

Acquisition has its own set of terms used throughout this technical/legal

discipline, but a couple of them stand out in the secure software environment. The first and most prevalent is commercial off-the-shelf (COTS) software. This term describes an element that is readily available for purchase and integration into a system. A counterpart to this is government off-the-shelf (GOTS) software. This term refers to software that is specifically developed for government use. GOTS tends to be more specialized and have higher costs per unit, as the base is significantly smaller.

Build vs. Buy Decision

Software acquisition can be accomplished in two manners, either by building it or buying it. This results in a build vs. buy decision. In today's modular world of software, the line between build and buy is blurred, as some elements may be built and some purchased. Many of today's applications involve integrating elements such as databases, business logic, communication elements, and user interfaces. Some elements, such as database software, are best purchased, whereas mission-critical core activities involving proprietary business information are generally best developed in-house. One of the key elements in successful integration is the degree of fit between software and the existing business processes, ensuring requirements include both the business process perspective and generic features and functions.

Outsourcing

Software development is an expensive undertaking. The process to develop good software is complex, the skill levels needed can be high, and every aspect seems to lead to higher costs. These cost structures, plus the easily transported nature of software, makes outsourcing of development a real possibility. Wages for developers vary across the globe, and highly skilled programmers in other countries can be used for a fraction of the cost of local talent. In the late 1990s, there was a widespread movement to offshore development efforts. A lot was learned in the early days of outsourcing. Much of the total cost of development was in elements other than the coders, and much of these costs could not be lowered by shipping development to a cheaper group of coders based on geography.

The geographic separation leads to greater management challenges and

costs. Having developers separate from the business team adds to the complexity, learning curves, and cost. The level of tacit knowledge and emergent understanding that is common on development teams becomes more challenging when part of the team is separated by geography. So, in the end, outsourcing can make sense, but just like build vs. buy decisions, it is critical to understand the details—their costs and benefits.

Contractual Terms and Service Level Agreements

Contractual terms and service level agreements are used to establish expectations with respect to future performance. Contractual terms when purchasing software should include references to security controls or standards that are expected to be implemented. Specific ISO standards or NIST standards that are desired by a supplier should be included in these mechanisms to ensure clear communication of expectations. Service level agreements can include acceptance criteria that software is expected to pass prior to integration.

Requirements Flow Down to Suppliers/Providers

Software is seldom created without connections to suppliers or providers. Software is built using libraries that inherit dependencies and can bring both good functionality and bad (bugs and vulnerabilities). In recent times a movement called a *software bill of materials* (SBOM) has emerged, where software vendors provide documentation of what is in their software. Viewed like a nutrition label for software, the SBOM can provide downstream users with an awareness of potential exposures due to third-party code embedded in a product.

The driving set of requirements for this type of information comes from the supply chain documentation and purchasing rules. As companies become more risk aware of software vulnerabilities and dependencies, they are incorporating language into their purchasing contracts that bring requirements with respect to many aspects of software development. The Open Web Application Security Project (OWASP) has some model contract information covering the elements that need to be determined as part of a contract negotiation. This information begins with the philosophy behind the requirements, as listed in [Table 5-2](#).

Security Decisions Will Be Based on Risk	Decisions about security will be made jointly by both Client and Developer based on a firm understanding of the risks involved.
Security Activities Will Be Balanced	Security effort will be roughly evenly distributed across the entire software development lifecycle.
Security Activities Will Be Integrated	All the activities and documentation discussed herein can and should be integrated into Developer's software development lifecycle and not kept separate from the rest of the project. Nothing in this Annex implies any particular software development process.
Vulnerabilities Are Expected	All software has bugs, and some of those will create security issues. Both Client and Developer will strive to identify vulnerabilities as early as possible in the lifecycle.
Security Information Will Be Fully Disclosed	All security-relevant information will be shared between Client and Developer immediately and completely.
Only Useful Security Documentation Is Required	Security documentation does not need to be extensive in order to clearly describe security design, risk analysis, or issues.

Table 5-2 OWASP Secure Software Contract Annex

Operationalizing the elements in [Table 5-2](#) requires that many subjects are agreed to in detail, including specific security requirements such as connections to external systems, input validation and encoding, authentication and session management, access control, logging, error handling, secure configuration, encryption, availability, libraries, testing procedures, and bug remediation processes, as well as general security issues such as source code control including revision control and code escrow. In the end, this list can become long and is highly dependent upon a comprehensive secure lifecycle development methodology. Just as a developing firm may have these questions for code elements they outsource, they should expect it from their customers.

Testing requirements and code bases can be very sensitive contractual issues as it can expose the source code, which in most cases is sensitive intellectual property. Having a third-party certify testing, both static and

dynamic, may suffice, but the devil is always in the details on who gets to see what with respect to the code base. But in the world of connected supply chains and the risk from cascading vulnerabilities through a software supply chain, one should expect greater detail and scrutiny in this regard in the future, not less.

Chapter Review

In this chapter, we examined misuse/abuse cases and how they are used to improve security as a set of security requirements. The concept of a requirements traceability matrix was introduced to assist in managing security requirements. The chapter concluded with thoughts on how security requirements will flow through a supply chain for software acquisition, and how that will impact development teams.

Quick Tips

- Misuse or abuse cases can be considered a form of use case illustrating specifically prohibited actions.
- The requirements traceability matrix is a grid that allows users to track and manage requirements and implementation details.
- Supply chains have begun using software security requirements in contractual language, making them enforceable parts of a business transaction. This has deliverable implications for development teams.

Questions

To further help you prepare for the CSSLP exam, answer the following questions. The correct answers are provided in the following section.

1. Presenting a known attack methodology to the development team to ensure appropriate mitigation can be done via what?
 - A. Use case
 - B. Misuse case
 - C. Security requirement
 - D. Business requirement

2. What is the name of a grid to assist the development team in tracking and managing requirements and implementation details?

 - A. Functional requirements matrix
 - B. Subject-object-activity matrix
 - C. Use case
 - D. Requirements traceability matrix
3. Supply chains commonly include which of the following as part a set of requirements?

 - A. Testing regimes
 - B. Source code control
 - C. Encryption
 - D. All of the above
4. Misuse cases represent what?

 - A. Improper conditions being handled by the software
 - B. Bug tracking mechanisms
 - C. Testing failures
 - D. Errors in the software development process that manifest themselves in the final product
5. What is a software bill of materials?

 - A. A method of version control
 - B. A listing of the included libraries and modules in software
 - C. A set of requirements concerning modules
 - D. A means of communicating use and misuse cases
6. Which of the following is not a use of misuse cases?

 - A. Documenting functional requirements
 - B. Alerting developers to actions that may require mitigation
 - C. Testing common failure modes of software
 - D. Communicating threat issues to developers
7. The RTM tracks which of the following?

- A. Requirement description, failure modes, verification method(s), use cases
 - B. Requirement source, test objective(s), verification method(s), known bugs
 - C. Requirement description, test objective(s), verification method(s), use cases
 - D. Requirement source, test objective(s), failure modes, use cases
8. Software acceptance testing may be an issue in which of the following elements?
- A. Supply chain contracts
 - B. Requirements traceability matrix
 - C. Creation of use cases
 - D. All of the above
9. Which of the following is not likely to be in a software contract?
- A. Security activities will be integrated.
 - B. Disclosure of included libraries and third-party modules used.
 - C. All vulnerabilities will be remediated.
 - D. Security information will be fully disclosed.
10. Misuse cases are typically not used for which of the following?
- A. Attack cases covered by models such as MITRE ATT&CK
 - B. Resiliency issues
 - C. Document software bugs
 - D. Environmental failure modes (no attacker involved)

Answers

1. B. Misuse cases can present commonly known attack scenarios and are designed to facilitate communication among designers, developers, and testers to ensure that potential security holes are managed in a proactive manner.
2. D. The requirements traceability matrix is a grid that allows users to

track and manage requirements and implementation details.

3. D. Common supply chain contract requirements include specific security requirements such as connections to external systems, input validation and encoding, authentication and session management, access control, logging, error handling, secure configuration, encryption, availability, libraries, testing procedures, and bug remediation processes, as well as general security issues such as source code control including revision control and code escrow.
4. A. Misuse cases represent improper conditions that should be appropriately handled by software including things such as erroneous inputs.
5. B. A software bill of materials is a listing of all embedded libraries and third-party modules in software to allow users to understand embedded risks.
6. A. Misuse cases document failure cases, while use cases document functional requirements.
7. C. The requirements traceability matrix tracks requirement description, test objective(s), verification method(s), use cases.
8. D. Software acceptance testing methods and results may be items in software contracts, they are connected to the RTM and use cases.
9. C. All vulnerabilities will be remediated is false on two points: All vulnerabilities would include unknown vulnerabilities, and even among the known vulnerabilities, some may not be fixed for a multitude of reasons.
10. C. Misuse cases are not used to document software bugs. Bugs are documented via bug tracking, not misuse cases.

PART III

Secure Software Architecture and Design

- **Chapter 6** Secure Software Architecture
- **Chapter 7** Secure Software Design

Secure Software Architecture

In this chapter you will

- Explore threat modeling
 - Define the security architecture
-
-

Software architectures are the elements employed to achieve the requirements of the system both now and into the future. To create an architecture, one needs information from the requirements and from the environment the system will operate in over its lifetime. To explore the environment and the risk from threats, a set of tools called, collectively, *threat modeling* is used. To address the functional and nonfunctional requirements directly, a wide array of technologies and techniques can be employed. Making effective use of the options requires that the entire team be on the same page with respect to many architectural elements; hence, the architectural plan is published to the team to act as a guiding document.

Perform Threat Modeling

Threat modeling is a process used to identify and document all of the threats to a system. Part of the description will include the mitigating actions that resolve the exposure. This information is communicated across all members of the development team and acts as a living document that is kept current as the software progresses through the development process. Performing threat modeling from the beginning of the development process helps highlight the issues that need to be resolved early in the process, when it is easier to resolve them. Threat modeling is an entire team effort, with everyone having a role in the complete documentation of the threats and issues associated with the software.

Threat Model Development

The threat model development process is a team-based process. The threat model development occurs across several phases. The first is defining the security objectives for the system. Following that is the system decomposition, then the threat identification, followed by mitigation analysis. The final step is the validation of the model.

Identify Security Objectives

As part of the requirements process, one of the essential elements is the determination of the security objectives associated with the system under development. These requirements can come from a number of sources, including legal, contractual, and corporate standards and objectives. Both security and privacy elements can be considered. Documenting an understanding of the business rationale for obtaining and storing data; how it will be used; and the related laws, regulations, and corporate standards for such behaviors will assist in later stages in the understanding of what is required. Leaving the decision of what data to protect and how to protect it to a development team is a mistake, as they may not have the breadth or depth of knowledge with respect to these requirements to catch all of the possible associated threats. Laws, regulations, contractual requirements, and even corporate standards can be complex and byzantine, yet if they are to be properly covered, they need to be enumerated as a list of requirements for the development effort.

System Decomposition

Once the security objectives are defined for a system, designers need to ensure they are covered in the actual design of the system. Numerous modeling systems are used in the design of a system. Unified Modeling Language (UML), use cases, misuse cases, data flow diagrams (DFDs), and other methods have been used to define a system. For the purposes of threat modeling, since the target is typically the information being manipulated, the DFD model is the best choice to use in documenting the threats.

Beginning in the design phase, as part of a system decomposition exercise, the designers can use DFDs to document the flow of data in the system. When using DFDs, it is important to identify and include all processes, data stores, and data flows between elements. Special attention should be given to

trust boundaries, where data crosses from one zone of trust to another. In large, complex systems, breaking down the DFD into scenario-based pieces may make documentation easier. Every assumption and every dependency should be enumerated and described. This is the baseline of the document. As the development process progresses, any changes to the items in the threat model should be documented and updated.

Trust boundaries are key elements, as they represent points where an attacker can interject into the system. Inside a trust boundary, items share the same privileges, rights, access, and identifiers. Machine boundaries are obvious trust boundaries, but inside a system, if privilege changes, this is a trust boundary, too.

DFD Elements for Threat Modeling

The following are examples of elements to identify as part of the threat modeling process:

External Entities

- Users (by type)
- Other systems

Data Stores

- Files
- Database
- Registry
- Shared memory
- Queues/stack

Trust Boundaries

- Users
- File systems
- Process boundaries

Data Flows

- Function calls
- Remote procedure calls (RPCs)
- Network traffic

Threat Identification

Once the system is well documented, then the process of identifying threats begins. Threat identification can be done in a variety of manners, and in most cases, several methods will be used. An experienced security person can provide a series of known threats that are common across applications, although this list needs to be tailored to the application under development.

During the DFD documentation process, as security objectives are documented, the question of what issues need to be protected against is answered, and this information describes threats. The use of the STRIDE method can be repeated across processes, data flows, trust boundaries—anywhere you want to understand the security implications of that portion of the application under development. Using each of the elements of STRIDE, the threat model is documented as to what happens as a result of each element of the STRIDE acronym. Not every STRIDE element will warrant documenting at every location, but this exercise will uncover many important security considerations.

As the threats are enumerated and documented, try to avoid becoming distracted by things that the application cannot control. For instance, if the system administrator decides to violate the system (machine) trust, this is an operating system issue, not an application issue. Likewise, with elements such as “someone steals the hard drive,” other than encrypting essential data at rest, this is an outside-the-application issue.

By this point, a lot of data will be associated with the threat model. There are tools to assist in the collection and manipulation of the threat model data, and it is wise to use an automated tool to manage the documentation. All of the described elements up to this point need to be numbered, categorized, and managed in some fashion. The next step is an examination of mitigations that can be employed to address the threats.

STRIDE

The acronym STRIDE is used to denote the following types of threats:

Threat	Security Property
Spoofing	Authentication
Tampering	Integrity
Repudiation	Nonrepudiation
Information disclosure	Confidentiality
Denial of service	Availability
Elevation of privilege	Authorization

Mitigation Analysis

Each threat requires mitigation. Mitigation can be of four types: redesign to eliminate vulnerability, apply a standard mitigation, invent a new mitigation, or accept the vulnerability. If redesign is an option, and it may be in the early parts of the development process, this is the best method, as it removes the risk. The next best method is to apply a standard mitigation. Standard mitigations are common security functions, such as access control lists (ACLs), and have several advantages. First, they are well understood and are known to be effective. Second, in many cases, they will apply across a large number of specific vulnerability points, making them an economical choice. Developing new mitigation methods is riskier and more costly, as it will require more extensive testing. Accepting the vulnerabilities is the least desired solution, but may on occasion be the only option.

The threat model is a dynamic document that changes with the development effort; so, too, do the mitigations. For a threat that is documented late in the development cycle, elements such as redesign may not be practical for this release. But the information can still be documented, and on the next release of the software, redesign may be an option that improves the security.

In picking the appropriate mitigation, one needs to understand the nature of the threat and how it is manifested upon the system. One tool that can assist in this effort is an attack tree model. An attack tree is a graphical representation of an attack, beginning with the attack objective as the root

node. From this node, a hierarchical tree-like structure of necessary conditions is listed. [Figure 6-1](#) illustrates a simple attack tree model for an authorization cookie.

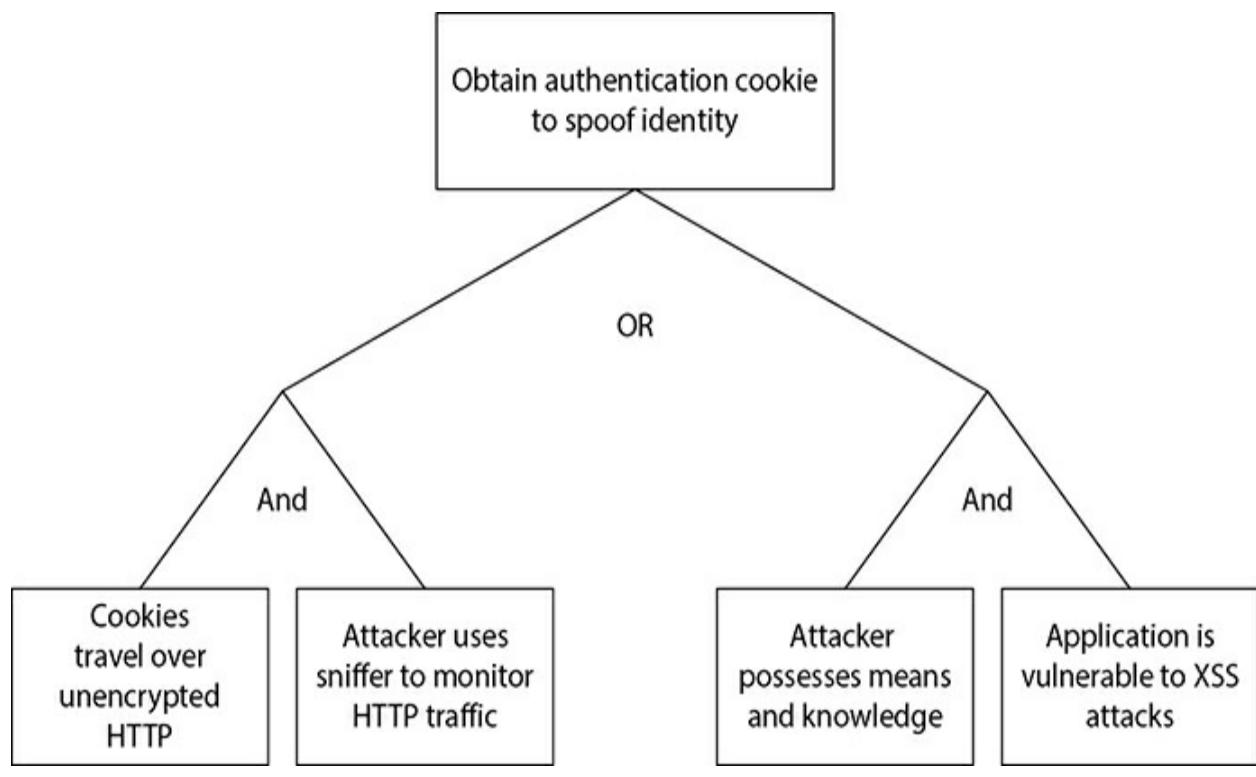


Figure 6-1 Attack tree

When describing a threat, it is also useful to gauge the priority one should place on it. Some attacks are clearly more serious than others, and risk management has dealt with this using probability and impact. Probability represents the likelihood that the attack will have success. This is not necessarily a numerical probability, but typically is described using nominal values of high, medium, and low. Impact represents the loss or risk faced by a successful attack, and it, too, is frequently scored as high, medium, or low. To convert these to a numerical score, you can apply numbers (high = 3, medium = 2, low = 1) to each and multiply the two. The resulting scores will be between 9 and 1. This provides a means of differentiating threats, with those scoring higher clearly more important to address.

Another method is referred to as DREAD, with DREAD being an acronym for damage potential, reproducibility, exploitability, affected users,

and discoverability. To use DREAD, one can assign a value from 0 (nothing) to 5 (moderate) to 10 (severe) for each of the elements. Then the total risk can be obtained by summing and dividing by 5, which will yield a score from 0 to 10. Another method is to set discoverability to 10, assuming discovery when doing the analysis. DREAD can be mapped into the probability impact model by taking the following factors into account: probability (reproducibility + exploitability + discoverability) and impact (damage potential + affected users).

Threat Model Validation

As the software development process moves through the phases of the SDL, at gates between the phases, an opportunity to examine materials presents itself. The current version of the threat model should be validated at each of these opportunities. The purpose of the validation phase is to assess the quality of the threats and mitigations. For the threats, it is important to ensure that they describe the attack and the impact in detail relevant to the context of the application. For mitigations, it is important that they are associated with a threat and are completely described in terms relevant to the context of the application.

Any and all dependencies should be documented. Documenting both what other code bases you are relying on and the security functions associated with that code is important. Ensuring all dependencies are documented is essential at each phase of the secure development lifecycle (SDL). Any assumptions that are made as part of the development process are also important to document.

Attack Surface Evaluation

The attack surface of software is the code within the system that can be accessed by unauthorized parties. This is not just the code itself but can also include a wide range of resources associated with the code, including user input fields, protocols, interfaces, resource files, and services. One measure of the attack surface is the sheer number or weighted number of accessible items. Weighting by severity may change some decision points, but the bottom line is simple. The more elements that can be attacked, the greater the risk. The attack surface of software does not represent the quality of the code —it does not mean there are flaws; it is merely a measure of how many

features/items are available for attack.

A long-standing security practice is not to enable functionality that is not used or needed. By turning off unnecessary functionality, there is a smaller attack surface, and there are fewer security risks. Understanding the attack surface throughout the development cycle is an important security function during the development process. Defining the attack surface and documenting changes helps enable better security decision-making with respect to functionality of the software.

Software is attacked via a series of common mechanisms, regardless of what the software is or what it does. Software is attacked all the time, and in some remarkably common ways. For instance, weak ACLs are a common attack point, regardless of the operating system. Attacks against the operating system can be used against many applications, including software that is otherwise secure. Attack surface evaluation is a means of measuring and determining the risks associated with the implications of design and development.

Attack Surface Measurement

To understand a product's attack surface, you need to measure the number of ways it can be “accessed.” Each software product may be different, but they will also share elements. Like many other security items, the first time a team builds a list, it will be more difficult. But, over time, the incremental examination will detail more items, making future lists easier to develop. In this light, here is a published list from Microsoft on attack surface elements associated with Windows. Although this list may not be totally applicable, it does provide a starting point and acts as a guide of the types of elements associated with an attack surface.

- Open sockets
- Open RPC endpoints
- Open named pipes
- Services
- Services running by default
- Services running as SYSTEM
- Active web handlers (ASP files, HTR files, and so on)

- Active Internet Server Application Programming Interface (ISAPI) filters
- Dynamic webpages (ASP and such)
- Executable virtual directories
- Enabled accounts
- Enabled accounts in admin group
- Null sessions to pipes and shares
- Guest account enabled
- Weak ACLs in the file system
- Weak ACLs in the registry
- Weak ACLs on shares

Another source of information is in the history of known vulnerabilities associated with previous developments. Determining the root cause of old vulnerabilities is good for fixing them and preventing future occurrences, and it is also valuable information that can be used in determining the attack surface.

The list of features that form the attack surface is the same list that attackers use to attack a specific piece of software. The items on the list are not necessarily vulnerabilities, but they are items that attackers will attempt to compromise. These elements are at the base of all vulnerabilities, so while a particular vulnerability may or may not exist for a given attack surface element, there will be one under each vulnerability that is uncovered. This makes measuring the attack surface a solid estimation of the potential vulnerability surface.

It is important to note that each software product is different, and hence, comparing attack surface scores from different products has no validity. But counting and measuring the attack surface provides baseline information from which security decisions can be made. Different elements may have different scoring values, as a service running as a system is riskier than a nonprivileged service. Services that are run by default are riskier than those on the same level running only on demand.

Attack Surface Minimization

The attack surface is merely a representation of the potential vulnerability surface associated with the software. Once a baseline is known, the next step is to examine ways that this metric can be lowered. This can be done by turning off elements not needed by most users, as this reduces the default profile surface. Services that are used can be on or off by default, only on when needed, run as system, or without privilege. The lower the privilege, the less a service is running, so these factors can reduce the attack surface.

Attack surface minimization is not necessarily an “on or off” proposition. Some elements may be off for most users, yet on for specific users under appropriate circumstances. This can make certain features usable, yet not necessarily exploitable. Reducing the exposure of elements to untrusted users can improve the security posture of the code. Minimization is a form of least privilege, and the application of it works in the same manner. If the application is network aware, restricting access to a set number of endpoints or a restricted IP range reduces the surface.

The attack surface should be calculated throughout the development process. Work done to reduce the attack surface should be documented throughout the process. This will assist in lowering the default surface level at deployment, but the information can also be provided to the customer so they can make informed decisions as they determine specific deployment options. As with all changes, the earlier in the development process a change is made, the lesser the impact will be to surrounding elements.

Attack surface minimization should be considered with design efforts. Determining the design baseline and listing the elements and details assist the development team in achieving these objectives. As the process continues and decisions are made that affect the surface, the documentation of the current attack surface area helps everyone understand the security implications of their specific decisions. Although security may not be easy to measure directly, the use of the attack surface as a surrogate has shown solid results across numerous firms.

Threat Intelligence

A major tool for defenders who are hunting attackers is threat intelligence. Threat intelligence is the actionable information about malicious actors, their tools, infrastructure, and methods. Incident response is a game of resource management. No firm has the resources to protect everything against all

threats or investigate all possible hostile actions; attempting to do so would result in wasted efforts. A key decision is where to apply incident response resources in response to an incident. Threat intelligence combined with the concept of the kill chain (the attacker's most likely path) means you can prioritize actions against most meaningful threats.

Software developers do not need the real-time situation awareness that threat intelligence strives to provide. Yet, the risks that threat intelligence detail are typically the result of software developer actions. Still, knowing what attackers are doing could help focus efforts. The vast majority of attacks on software are reliant upon a few vulnerabilities. Per the Verizon Data Breach Investigative Report (VDBIR), as few as ten vulnerabilities accounted for 97 percent of all attacks against companies in 2014, with greater than 99 percent of the attacks exploiting vulnerabilities older than a year. Understanding what is being attacked, and how, is important information for the development team. How did this get past the security processes including threat modeling and risk testing efforts? As developers include third-party, open source, and commercial libraries into their projects, these sources must be watched for newly discovered vulnerabilities. Again, this function of threat intelligence can feed important information to the development team.

Threat Hunting

Threat hunting is an iterative process of proactively searching out threats inside the network. Several different models can be employed for threat hunting, but one of the most effective is based on creating a hypothesis and then examining that hypothesis. This act provides a level of scope to the hunt —rather than looking for anything in a sea of mostly normal, one is looking for specific items. A typical hypothesis would be something like “an adversary is using stolen credentials to mimic authorized users during nonworking hours.” This hypothesis is concise and can be tested by examining a set of logs for specific activities during nonworking hours.

The objective of threat hunting is to use current knowledge of what adversaries are doing to firms and check to see if that is happening on your network. This can increase detection of malicious activity beyond the typical incident-type triggers. A complete explanation of threat hunting can be found in the whitepaper “A Practical Model for Conducting Cyber Threat Hunting,” by Dan Gunter and Marc Seitz (<https://www.sans.org/reading-room/whitepapers/cybersecurity/practical-model-conducting-cyber-threat-hunting>)

[room/whitepapers/threathunting/practical-model-conducting-cyber-threat-hunting-38710](#)).

Define the Security Architecture

A modern enterprise is never going to be singular in its architectural form. IT systems have grown over time through an accretive process where the new “system” is designed to meet requirements and then joins the other systems in the enterprise. Cross-integration between architectures allows data reuse and significantly increases the overall utility of the enterprise architecture as a whole. As new services and opportunities are presented to the IT enterprise, the need to fully integrate, as opposed to rebuilding existing data services, is both a cost- and risk-reducing proposition. Going forward, enterprise accretion will continue, with the addition of new capability and the retirement of no longer used or needed capabilities.

Architectures happen whether planned or not. Getting the true benefit of a security architecture requires that it be a planned element to support the desired security outcomes. This is important in both new greenfield projects and updates and expansions of existing systems. Reviewing the architecture with respect to security goals and communicating the design to the team provides guidance for the developers when choices need to be made concerning options and technologies to employ. The key elements are a solid thoughtful design and communication of that to the team.

Security Control Identification and Prioritization

Security controls are the mechanisms used to achieve security objectives in systems. Whether simple like permissions and access control or more complex like those associated with open design or complete mediation, controls are defined as the measures taken to detect, prevent, or mitigate the risks associated with the threats a system faces. Controls are also sometimes referred to as *countermeasures* or *safeguards*. They can be associated with several types of actions: administrative, technical, or physical. For each of these classes of controls there are four types of controls: preventive, detective, corrective, and compensating.



EXAM TIP Controls offer the best method of managing risk in a software environment. Understanding the risk environment and applying security controls to manage the risk is within the domain of the development team.

Types of Controls

Controls can be classified based on the types of actions they perform. Three classes of controls exist:

- Administrative
- Technical
- Physical

For each of these classes, there are four types of controls:

- Preventive (deterrent)
- Detective
- Corrective (recovery)
- Compensating

Preventive

Preventive controls are used to prevent the vulnerability from being exploited. Preventive controls are one of the primary control mechanisms used in the deployment of security functionality. They are proactive in nature and provide the widest level of risk mitigation per control. Examples of preventive controls include separation of duties, adequate documentation, physical controls over assets, and authorization mechanisms.

Detective

When the preventive controls fail, a vulnerability can be exploited. At this point, detective controls, or controls that can detect the presence of an attack,

are employed. Detective controls act after the fact. Typical detective controls include elements such as logs, audits, and inventories.

Corrective

Corrective controls correct a system after a vulnerability is exploited and an impact has occurred. Because impacts may have multiple aspects, a corrective control acts on some aspects to reduce the total impact. Corrective controls are also after the fact and are typically targeted toward the system under attack rather than the attack vector. Backups are a common form of a corrective control, for they are useful only after an attack has occurred and serve to make recovery more efficient.

Compensating

Compensating controls are designed to act when a primary set of controls has failed. Compensating controls typically occur after the fact, as they are employed as a form of defense in depth. Separation of duties might be a primary control to prevent fraud, and a financial review of accounting reports can serve as an after-the-fact compensating control.

Controls Framework

Controls are not implemented in a vacuum or as individual isolated items. Controls work to reduce risk as part of a complete system. Managing risk across an enterprise is a complex endeavor, and the use of a framework to organize the individual risk controls assists in organizing the design of a comprehensive set. [Figure 6-2](#) illustrates the relationships between the risk elements and forms the basis for a controls framework.

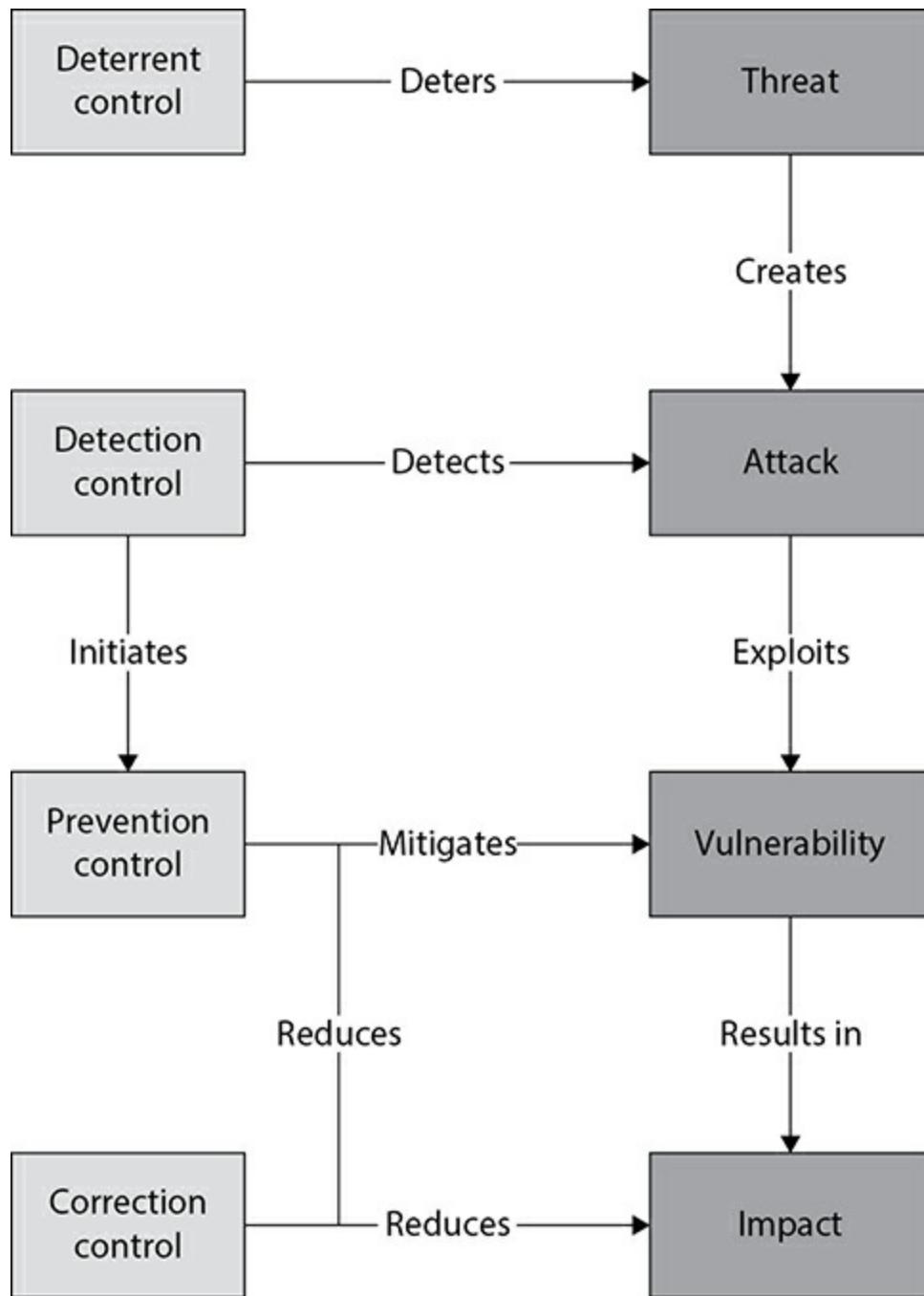


Figure 6-2 Controls framework

Distributed Computing

With the rise of affordable computing solutions that were smaller in size, yet capable of significant processing, came the distribution of computing out of the center and into the business. The availability of networking and storage

options furthered the case for distributing the processing closer to users across the enterprise. There are several architectural forms of distributed computing and supporting elements.

Client-Server

Client-server architectures are defined by two types of machines working in concert. Servers are more capable of processing and storage, serving numerous users through applications. The client machines are also capable of processing and storage, but this is typically limited to single-user work. Clients are typically described as thin or fat clients, depending on the level of processing present on the client. Thin clients perform the majority of the processing on the server, while fat clients take on a significant amount of the processing themselves.

Another characteristic of the client-server architecture is the level of communication between the servers and clients. Because processing is distributed, communications are needed to facilitate the distributed processing and storage. This distribution of processing and storage across multiple machines increases the need for security. One method of describing the multimachine architecture model is called the *n-tier model* (see [Figure 6-3](#)). The *n* refers to the number levels of applications doing the work. A three-tier model can have a client talking to an intermediate server performing business logic and then to a database server level to manage storage. Separating an architecture into a series of layers provides for a separation of business functionality in a manner that facilitates integration and security. When implementing distributed processing and storage, security becomes an important concern.

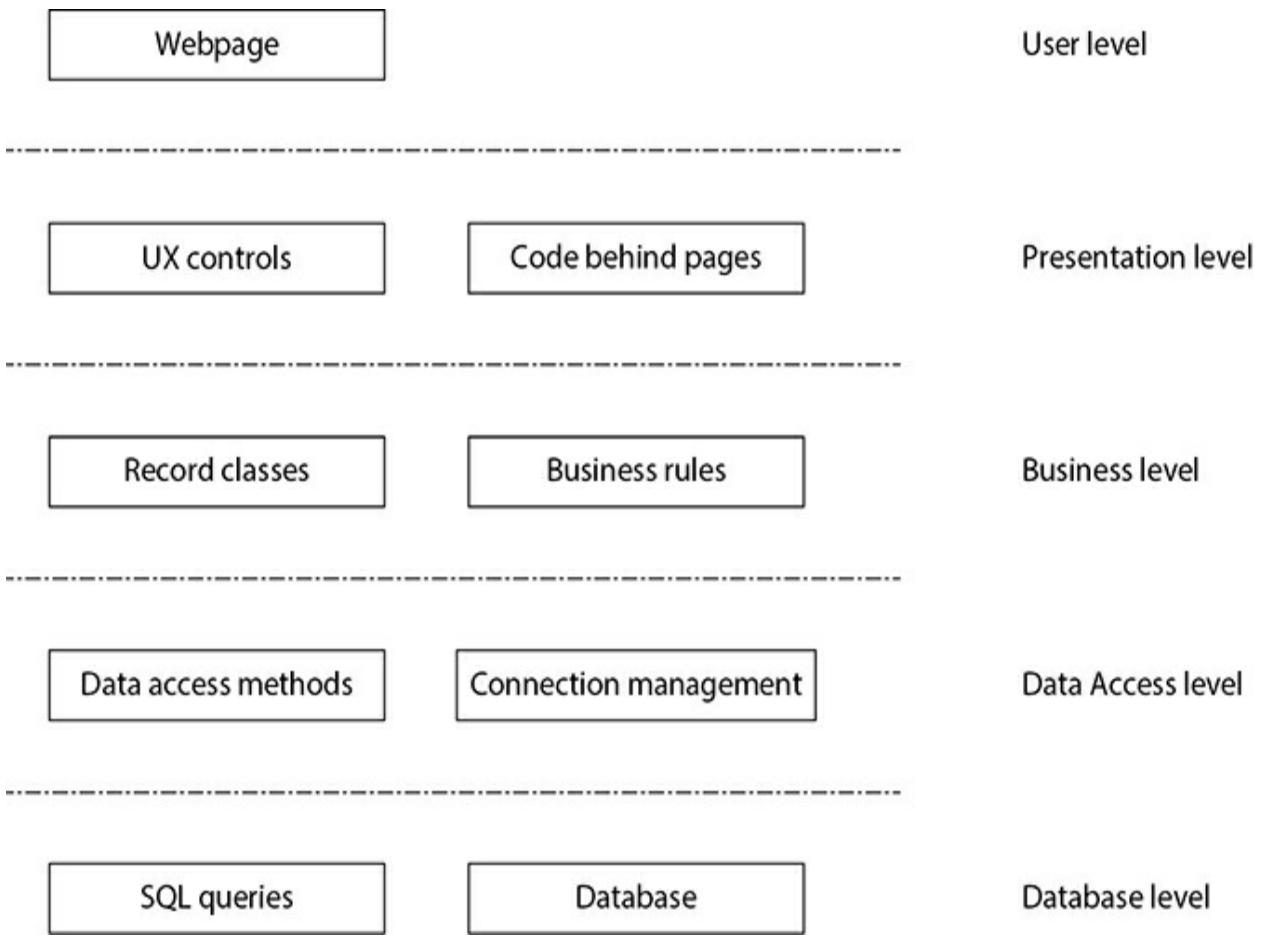


Figure 6-3 N-tier architecture

Cloud computing can be considered an extreme case of the client-server model. The same elements of an n-tier model can be implemented in clouds, software as a service (SaaS), platform as a service (PaaS), and information as a service (IaaS) models.

Peer-to-Peer

Peer-to-peer architectures are characterized by sets of machines that act as peers. While the client-server model implies a separation of duties and power, peer-to-peer models are characterized by the member devices sharing the work. In peer-to-peer sharing, both parties are at equivalent levels of processing. This type of network is more commonly found in the transfer of information, file sharing, and other communication-based systems. A common utilization of the peer-to-peer model is in file sharing, where machines directly share files without an intermediate storage hub.

Message Queuing

Moving information from one processing system to another can be done in a variety of ways. One of the challenges is managing throughput and guaranteeing delivery. Message queuing technology solves this problem through the use of an intermediate server that mediates transmission and delivery of information between processes. In large enterprise systems with multiple data paths, message queuing can solve many data transfer issues, such as point-to-multipoint data flows. Message queuing can be constructed to manage guaranteed delivery, logging, and security of the data flows.

Service-Oriented Architecture

Service-oriented architecture (SOA) is a distributed architecture with several specific characteristics. These characteristics include the following:

- Platform neutrality
- Interoperability
- Modularity and reusability
- Abstracted business functionality
- Contract-based interfaces
- Discoverability

SOAs can be implemented with several different technologies, including common object model (COM), common object request broker architecture (CORBA), and web services (WS). Most SOA implementations use eXtensible Markup Language (XML) as the messaging methodology of choice, although this brings additional issues with regard to security. The XML messages can be secured either through XML encryption or transport over secure channels (SSL/TLS).

Services are core units of functionality that are self-contained and designed to perform a specific action. When implemented in web services, SOA uses a set of technologies that are unique to web services. Simple Object Access Protocol (SOAP) and Representational State Transfer (REST) are two common protocols utilized for messaging in the enterprise service bus.

Enterprise Service Bus

Enterprise service bus (ESB) is a name given to a specific form of SOA architecture where all the communications between producers and consumers of the data take place. An ESB solution is designed to monitor and control the routing of messages between services in the system. Frequently, this messaging is done via a form of message queuing services that keeps everything aligned with the requirements of the system. The ESB acts as a form of abstraction layer for the interprocess communication services.

The ESB can provide a range of services to the enterprise applications that are served by the system. The ESB can be configured to

- Perform protocol conversions and handle translation and transformation of communications
- Handle defined events
- Perform message queuing and mapping of data flows

The key characteristic of the ESB is the use of a bus-based architecture as a means of managing communications between processes. [Figure 6-4](#) illustrates the bus nature of communications between producers and consumers of information. The actual implementation of the bus can be performed in a variety of protocols, including message queue technologies.

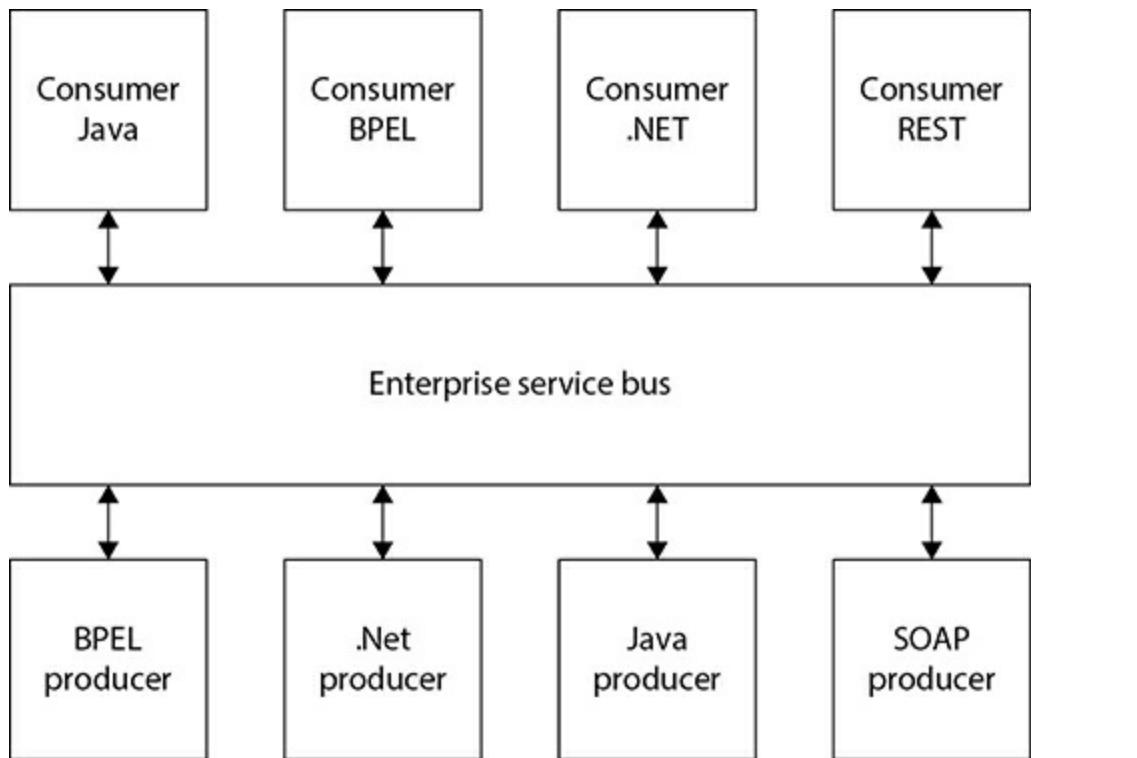


Figure 6-4 Enterprise service bus

The ESB acts as the conduit between all types of protocols. Each connector can operate through an adapter enabling the cross-communication between different protocols. An ESB allows XML, EDI, WSDL, REST, DCOM, CORBA, and others to communicate with each other in a seamless fashion.

Web Services

Web services are a means of communication between elements over the Internet. The term *web services* is a descriptor of a wide range of different means of communications. Web services are characterized by a machine-readable description of the interface. This machine-readable format is referred to as Web Services Description Language (WSDL). WSDL is an XML-based interface description language that is used for describing the functionality offered by a web service, including how the service can be called, what parameters it expects, and what data structures it returns.

Web services originally were designed with SOAP, but a movement now favors REST as the means of communication. The advantage of SOAP is that

it can use XML as the basis of communication. There have been performance concerns over using XML, however, as well as concerns over the complexity in implementation.

REST

Representational State Transfer (REST) is an architecture method used to define web services that can operate at the scale of the Internet. REST-compliant systems allow text-based access of web resources in a stateless manner.

JSON

JavaScript Object Notation (JSON) is another method of data interchange that is commonly employed in web services. Although it was patterned on JavaScript, JSON is language independent and can be employed in virtually any language. It is built on two universal data structures:

- A collection of name-value pairs. This form is typically realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.
- An ordered list of values. This form is realized as an array, vector, list, or sequence.

Virtually all modern programming languages support these structures. This has made JSON a common data interchange format.

Universal Description, Discovery, and Interface

Universal Description, Discovery, and Interface (UDDI) is a universal platform-independent method for enterprises to dynamically discover and invoke web services. Using XML, UDDI was designed as a protocol-based registry through which services worldwide can list themselves on the Internet. UDDI includes a mechanism to register and locate web service applications. It was originally proposed as a core web service standard but was never widely adopted and is only really found inside organizations, not across the entire Internet as originally planned.

W3C Web Service Definition

A web service is a software system designed to support interoperable machine-to-machine interaction over a network. Using WSDL, it has an interface described in a machine-processable format. Other systems can interact with the web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other web-related standards, such as JSON.

The W3C, under the Web Services Architecture, identifies two major classes of web services:

- REST-compliant web services, in which the primary purpose of the service is to manipulate XML representations of web resources using a uniform set of “stateless” operations
- Arbitrary web services, in which the service may expose an arbitrary set of operations

Rich Internet Applications

Rich Internet applications (RIAs) are a form of architecture that use the Web as a transfer mechanism and the client as a processing device, typically for display formatting control functions. An example of an RIA is Facebook, or any of the other social media sites. The objective of an RIA is to create an application with the characteristics of a desktop application but delivered across the Internet.

RIAs are created using a variety of frameworks, including Adobe Flash, Java, and Microsoft Silverlight. With the introduction of HTML5, the future appears to be one dominated by HTML5/JavaScript-based RIAs. RIAs can accommodate a wide range of functionality, from complex business interfaces, to games, to learning platforms. If it can be done on a desktop, it can be done in an RIA.

Just because the RIA does the majority of the processing on a server or back-end platform, however, does not mean that security can be ignored. In fact, the opposite is true. Client-side exploits and remote code execution-type

attacks can exploit the architecture of an RIA.

Client-Side Exploits or Threats

In all client-server and peer-to-peer operations, one universal truth remains. Never trust input without validation. Systems that ignore this are subject to client-side attacks. Even though one can design a system where they have control over the clients, there is always the risk that the client can become corrupted, whether by malware, a disgruntled user, or simple misconfiguration. Client-based architectures, such as RIAs, are specifically susceptible to client-side threats.

Remote Code Execution

Remote code execution is a term used to describe the process of triggering arbitrary code execution on a machine from another machine across a network connection. This can be a serious attack, for when executed successfully, the arbitrary code operates under the security credentials of the process that is infected. This is a consequence of the architectural decision in which there is not a distinction between code and data. This makes it possible for malicious input (arbitrary code) to be executed within the security context of the running process.

Pervasive/Ubiqitous Computing

With the advent of cost-effective microprocessors, computers have become integrated into many devices and systems, with the purpose of improved control. With the addition of the Internet and low-cost connectivity becoming possible between devices, this trend has accelerated into interconnecting the devices, sharing information between them and further enabling remote control of functions. With the addition of handheld computers built around mobile phones, the era of pervasive and ubiquitous computing has begun. Users, connecting with their phones, have the ability to interact with a wide range of devices and services, with the common expression of “I have an app for that” becoming everyday vernacular.

The implications of always-on, hyper-connectedness, and cross-platform/system integration are many—some positive and some challenging. The driving factor behind connecting everything to everything is a combination of convenience and efficiency. The challenges come from the

intrinsic properties of highly complex systems, including emergent issues of safety, security, and stability. One of the new challenges in this area is in the Internet of Things (IoT), a set of Internet-connected (networked) devices that operate at Internet scale. These systems are composed of large numbers of items, which are typically low cost and seldom have significant security capabilities on their own.

The importance of some basic security properties, defense in depth, fail to a secure state, and complete mediation, becomes more important in the complex systems being created through pervasive computing. Each element needs to become self-sufficient and self-reliant for security, safety, and stability aspects of their own operations. The system as a whole needs to operate securely. This drives requirements through the software development lifecycle (SDLC) and makes them increasingly important.

Wireless

Wireless communications have gone from a rare instance when wiring was difficult to a norm for many devices. Driven by low-cost technology, wireless communications are becoming ubiquitous in the industrialized world. Wireless computer networks can be created with several different protocols, from cellular-based systems to 802.11 Wi-Fi to 802.15 Zigbee, Bluetooth, and Wi-Max, each with differing capacity, distance, and routing characteristics.

Wireless networking frees the deployment from the hassles of wires, making the movement of machines, the routing through walls and between buildings, etc., easier. Wireless allows mobile devices to be connected to network resources while remaining mobile. With the advantages, however, come risks. Wired networks can control device connections through the control over the physical connection points. With wireless, this is a different proposition. With wired networks, communications between devices are not available to others, as the signal does not necessarily pass others. In wireless networks, anyone within the signal range can see the signal and monitor the communications from all devices.

When designing applications where information is going to be transmitted across a network, one needs to consider the fact that the data may be transmitted across an unsecure network, such as a wireless network. The implication is simple—the developers need to take responsibility for the security of the data when transmitted; expecting the network to take care of

security is an opportunity for failure during deployment.

Location-Based

With the rise of mobile, handheld computing, one of the new “killer apps” is the use of location-based data as part of the application process. Whether marketing (i.e., telling users when they are close to a store that has sales) or security (i.e., only allowing certain functionality when you are in certain locations), the ability to make application decisions based on user location has advantages.



NOTE *Geofencing* describes the triggering of an action as a mobile device crosses a boundary. This can be a specific boundary per GPS or a distance from a point. This has utility in linking mobile device location and desired activities into an automated system of notification or action.

Location-based data can be very valuable in an application. But a user’s location can also be abused, and protecting sensitive data such as location, both in apps and in downstream processes, is important. Not all sensitive data is easily identified as such. Sometimes, it is necessary to examine how data elements could be abused in order to come to the correct security consideration for an element.

Constant Connectivity

The combination of hyper-connectivity, ubiquitous computing, and affordable computing, coupled with Internet-based software such as social media, has created a situation where there is virtually constant connectivity. Whether it is individual users or corporate machines connected via the Internet to other machines and devices, the result is the same: Everything is always connected.

Radio Frequency Identification

Radio frequency identification (RFID) is a radio frequency, noncontact means of transferring data between two parties. Using a transmitter/receiver

and a device called a *tag*, the radio equipment can read the tag. Widely used for contactless inventory, the range can be a few meters for simple ones to hundreds of meters for battery-powered ones that act as transponders. RFID tags can be used to track things, with the added advantage that because the tag is RF-based, it does not need to be in the line of sight of the reader. When a reader sends a signal to a tag, the tag responds with a digital number, enabling individual serialization. Tags can be small (the size of dust particles) and cheap (costing just pennies) and come in a wide range of form factors. Picking the correct RFID tag involves planning the deployment and environment as well as the desired options.

RFID tags can be either active or passive, depending upon whether the tag has a source of power. Powered tags can offer greater range for detection and reading, but at an increased cost. Because of the use of RF, the different frequency bands used have differing regulatory requirements by country. Numerous standards have emerged covering both technical details of the system and its use. Several International Organization for Standardization (ISO) standards exist, and there are several industry-supported groups concerned with standardization.

U.S. citizens became acutely aware of RFID tags when they began showing up as a security and convenience factor in passports. Designed to be readable for only short distances, shielding had to be added after researchers showed that they could be read from several meters away. Another widespread adoption is in Walmart Corporation's supply-chain effort, using RFID-based Electronic Product Code (EPC) tags in all of its major supply chains.

Near-Field Communication

Near-field communication (NFC) is a protocol and set of standards for communication via radio frequency energy over very short distances. Limited in distance to a few inches, the communication occurs while a user typically touches one device to another. Designed to offer contactless communications over short distances, and with no setup for the user, this technology has caught on in mobile devices and payment systems. NFC is a low-speed connection, but with proper setup, it has been used to bootstrap higher-bandwidth transfers. The Android Beam process uses NFC to initiate a Bluetooth connection between devices. This enables the higher transfer speeds of Bluetooth, but with the security associated with the close

proximity.

Sensor Networks

Sensor networks are connections of distributed autonomous sensors designed to monitor some measurable condition. These networks serve to collect data on physical and environmental processes, such as rainfall, weather, communication efficiency, and more. Each network is designed for a purpose, with the choice of sensor type, location, and communication method being chosen to fit the situation. The majority of sensor networks are now being deployed using wireless communications. This is partly due to the ubiquity and relatively low cost of wireless solutions.

The individual elements of a sensor network are referred to as *nodes*. A node is typically a measuring device as well as a communication platform. The actual network architecture is dependent upon the communication technology and the business objectives of the network.

Embedded

Embedded systems are dedicated systems where the hardware and software are coupled together to perform a specific purpose. As opposed to general-purpose computers, such as servers and PCs, which can perform a wide range of activities, an embedded system is designed to solve a specific problem. Embedded systems are created to perform a specific task, one where time-sensitive constraints are common. They exist in a wide range of electronics, from watches to audio/video players to control systems for factories and infrastructure to vehicles. Embedded systems can be found virtually everywhere.

Cloud Architectures

Cloud computing is a relatively new term in the computer field used to describe an architecture of scalable services that are automatically provisioned in response to demand. Although the term is new, the operational concept is not and has been in use for decades. Cloud computing is marked by the following characteristics:

- On-demand self-service

- Broad network access
- Resource pooling
- Rapid elasticity
- Measured service

Customers can unilaterally provision and de-provision their level of service as needed. Scaling can increase and decrease on demand, with resource utilization being monitored and measured. Cloud computing can be economical because of the resource pooling and sharing across multiple customers with differing scale needs at any given time. Cloud computing is ideally suited for small and medium-sized businesses, as it alleviates the technical side of building out infrastructures and resolves many scaling issues.

Cloud-based computing is also taking hold in large enterprises, because by adopting the methodologies used in cloud computing, the large enterprises can garner the advantages. When the scale permits, large enterprises can run their own cloud-based instances, offering SaaS, PaaS, and IaaS capabilities in-house.

The National Institute of Standards and Technology (NIST) document of cloud computing, NIST Special Publication 800-145, “The NIST Definition of Cloud Computing,” defines four deployment models:

- Private cloud
- Public cloud
- Community cloud
- Hybrid cloud

The private and public clouds are exactly as they sound, serving either a single entity or multiple entities. The community cloud differs from a public cloud in that its membership is defined by a community of shared concerns. A hybrid cloud is an environment composed of more than one of the previously mentioned characteristics, with them remaining separate but bound by some form of common technology.

Software as a Service

Software as a service is a type of cloud computing where the software runs in

the cloud on external hardware and the user derives the benefit through a browser or browser-like interface. Moving IT deliverables to an “as a service” deployment methodology has gained tremendous traction because of the convenience factors. The SaaS model allows for virtually no contact distribution, instant update and deployment methods, and the ability to manage software interactions with other packages. Pricing can take advantage of economies of scale and low costs for user provisioning. Rather than a firm needing to stand up servers and back-end processes, all they need are clients. SaaS ties nicely into the cloud and PaaS and IaaS movements, providing compelling business capabilities, especially for small and medium-sized businesses.

One of the advantages is that the consumer does not manage or control the underlying cloud infrastructure. This includes the required network, servers, operating systems, storage, or even individual application capabilities. All that a consumer would need to configure would be a limited set of user-specific application configuration settings.

Data security is still a legitimate issue in SaaS, and software planned for this type of deployment needs to consider the ramifications of the vendor holding and protecting client data. This can be a serious issue, and significant planning and detailed attention need to be placed on this aspect of the development cycle. Failure to plan and protect can lead to market failures, either through nonacceptance or customer losses and claims in the event of a breach.

Platform as a Service

Platform as a service is a form of cloud computing that offers a complete platform as a solution to a computing need. This is a service model of computing where the client subscribes to a service, which in this case can include multiple elements. The platform may include infrastructure elements (IaaS) and software elements (SaaS). PaaS can exist as a selected collection of elements into a designed solution stack for a specific problem. This may include apps, databases, web services, storage, and other items that are offered as a service.

As with SaaS, one of the advantages is that the consumer does not manage or control the underlying cloud infrastructure. This includes the required network, servers, operating systems, storage, or even individual application capabilities. As the scale of the platform increases, so does the operational

savings from this aspect. All that a consumer would need to configure would be a limited set of user-specific application configuration settings, a task that can be modified to scale across the platform in a single action.

PaaS has also been used to support the development effort by offering all the desired software development components as a solution stack, accessible via a web browser. Creating a stack of approved software applications that are kept appropriately updated can add value to the development effort as well as other business environments. Integration into testing platforms and change management systems, in addition to simple development environments, simplifies the development process.

Infrastructure as a Service

Infrastructure as a service is a form of cloud computing that offers a complete platform as a provisioning solution to a computing need. A typical computing environment consists of networks connecting storage, processing, and other computing resources into a functional environment. IaaS provides the consumer of the service with the capability to manage the provisioning of these elements in response to needs. The consumer can deploy and operate arbitrary software across a cloud-based platform without worrying about the specifics of the infrastructure. IaaS is a partial solution in the continuum from SaaS to PaaS, with the consumer retaining some control over the platform configurations within an IaaS environment.

Shared Responsibility Model

The cloud can be a unique environment with both the customer and vendor having roles and responsibilities in implementation activities. Cloud service providers have created a shared security responsibility model that details the responsibilities held by the cloud provider and those retained by the customer. This means both security teams have specific responsibilities for security, and both parties need policies and procedures to ensure coverage. Specifically, the cloud customer will have responsibilities as defined in the cloud terms of service, and the customer hence needs to take the necessary protection efforts as they move applications, data, containers, and workloads to the cloud. Defining the line between cloud customer and cloud service provider is a key element of cloud terms of service, and customers need to be aware of what is and what isn't covered by the cloud service provider.

Understanding the range of your responsibilities as the cloud customer and those of your cloud providers is imperative for managing the risk associated with cloud environments. [Table 6-1](#) illustrates how shared responsibilities are split across the various “as a service” models.

		On prem	IaaS	PaaS
Elements specific to customer's business—customer responsibility	Application user access management	C	C	C
	Application-specific data assets	C	C	C
	Application-specific logic/code	C	C	C
Workload-based responsibilities—shared based on model	Application/platform software	C	C	P
	OS and networking	C	C	P
	VM/Container/server instances	C	C	P
Infrastructure level responsibilities—tied to owner of the infrastructure elements	Virtualization platform	C	P	P
	Physical hosts	C	P	P
	Physical network	C	P	P
	Physical datacenter environment	C	P	P

Table 6-1 Division of Responsibilities Between Customer (C) and Cloud Provider (P)

Mobile Applications

Mobile applications are software applications designed to run on mobile devices, such as phones and tablets. Becoming nearly ubiquitous for numerous purposes, there is a commonly used phrase, “I have an app for that,” to describe this form of computing architecture. Mobile apps are designed and developed with the native constraints of the mobile device in mind. Limited processing power, limited memory, and limited input/output capability, yet always on and always with a user (convenience and persistence to the user), offer a unique computing environment.

The typical mobile application environment includes an element known as the *app store*. App stores are repositories designed to mediate the distribution

of software to the mobile devices. There are numerous forms of app stores, from private app stores set up within enterprises to commercial app stores run by the mobile device manufacturers (Apple, Nokia, BlackBerry, Google) to commercial stores such as Amazon. The secure development of mobile apps is an interesting issue, as mobile devices are becoming common interfaces to a network and can pose a connection or client-side risk. Mobile apps tend to have the potential to access a significant quantity of information stored on the device.

Hardware Platform Concerns

Hardware elements can bring their own software security concerns. When interfacing with elements such as the Trusted Platform Module or hardware security module, understanding of the technology, its advantages, and its limitations is important.

Trusted Platform Module

The Trusted Platform Module (TPM) is a hardware solution on the motherboard, one that assists with key generation and storage as well as random number generation. When the encryption keys are stored in the TPM, they are not accessible via normal software channels and are physically separated from the hard drive or other encrypted data locations. This makes the TPM a more secure solution than keeping the keys in the machine's normal storage. A TPM acts as a secure cryptoprocessor. It is a hardware solution that assists with key generation and secure, encrypted storage.

Hardware Security Module

A hardware security module (HSM) is a device used to manage or store encryption keys. It can also assist in cryptographic operations such as encryption, hashing, or the application of digital signatures. HSMs typically are peripheral devices connected via USB or a network connection. HSMs have tamper-protection mechanisms to prevent physical access to the secrets they protect. Because of their dedicated design, they can offer significant performance advantages over general-purpose computers when it comes to cryptographic operations. When an enterprise has significant levels of cryptographic operations, HSMs can provide throughput efficiencies. Storing private keys anywhere on a networked system is a recipe for loss. HSMs are

designed to allow the use of keys without exposing them to the wide range of host-based threats.

Side Channel

Attacks that use some byproduct of a system are typically called *side channel attacks*. The term comes from the cryptographic world, where it represents an attack against the implementation of a cryptosystem, rather than the strength of the algorithm itself (e.g., cold booting). There are different types of side channel attacks, including timing attacks, power attacks, data remanence attacks, and electromagnetic attacks. Attacks against the human element, also called *social engineering attacks*, may fit the general description of a side channel attack, but they are usually considered separately and are covered in the next section.

Timing and power attacks examine elements such as power used or time to achieve some function to make determinations about what is happening. Although these seem far-fetched, they have been used successfully to reveal information about what is happening inside a program. Electromagnetic attacks were famous in the era of cathode ray tube (CRT) monitors, as devices were constructed that could read the magnetic patterns of a CRT from a distance, reproducing what was on the screen. A modern equivalent is the acoustic attack, where the computer's own microphone is used to record keystrokes and then decode them based on the different sounds each key makes.

The data remanence attack has been in the headlines lately, where researchers have cooled RAM in a machine to very cold temperatures, allowing them time to get key values out of the RAM even after the power was turned off. Some types of malware are known to scrape the memory of systems in search of key elements, such as keys and other secret values. Modern efforts such as address space layout randomization (ASLR) are designed to defeat this, but as in all tech “wars,” both sides keep improving the game. The current ASLR scheme used in Windows is already beginning to show signs of age and will probably be enhanced in the next version.

Cognitive Computing

Artificial intelligence (AI) and machine learning (ML) are the new darling children of computer science, solving problems previously unaddressable by

standard algorithmic means. Software development is a manufacturing process with many known qualities, and one of those is an error rate. Most consensus estimates that there is one vulnerability per 10,000 lines of code. This makes a 10-million-line program a grab-bag of potential problems. Best practices can catch many of these, but as programs get larger and more complex (a high-end electric vehicle with self-driving features can be 100 million lines of code or more), this is a problem that needs new solutions. DARPA created a grand challenge with the task for a computer program to create a computer program, and not a simple program, but a complex offense and defense solution to a capture-the-flag event. The solution, while not perfect, significantly advanced the state of the art of using AI and ML in both coding and system design.

The future is already here in several development tools. AI is already being employed in source code analysis, reducing false alarms and improving the recognition of critical errors. AI is improving fuzzers by crafting more relevant data sets for testing.

Control Systems

Control systems are specialized computer systems used for the automated control of equipment. These systems are referred to by many names, including industrial control systems and operational technology (OT). A wide range of types of equipment and supporting software fall into this category, from programmable logic controllers (PLCs) to remote terminal units (RTUs). These devices are commonly referred to as *supervisory control and data acquisition* (SCADA) systems when used in the collective form. Control system equipment can be viewed as a form of embedded system, because they are integrated into a physical environment for the sole purpose of providing computer control in that environment.

Chapter Review

In this chapter, you examined the concepts and practices employed in threat modeling and the creation of a threat model for the system. This chapter also covered the different forms of architectures used in computer systems and the security implications of each. Distributed computing is a form of separating the processing and storage across multiple systems. Forms of distributed

computing include client-server and peer-to-peer architectures. Message queuing technologies can be used as a supporting technology. Service-oriented architectures are platform-neutral, modular applications that have contract-based interfaces. A key component of an SOA is the enterprise service bus. SOAs can be supported by SOAP and REST protocols. Web services are a form of SOA that uses WSDL for provisioning and the Internet for communication channels.

Rich Internet applications can mimic a desktop application look and feel but add a concern over client-side exploits and remote code execution threats. The combination of constant connectivity, hyper-connectedness, mobile devices, and affordable computing, coupled with apps such as Google search, the World Wide Web, and social media, has led to a state of pervasive computing. Wireless networks allow much easier networking for elements such as mobile devices. Mobile and wireless devices are enhanced with technologies such as location-based services, RFID, NFC, and sensor networks, and using mobile applications can provide new and enhanced services. The chapter concluded with a discussion of cloud computing, presenting SaaS, PaaS, and IaaS.

Quick Tips

- System architectures define options of technology and methodology to the entire dev team to assist them in working toward a solution where the parts work together and resolve requirements.
- Threat modeling is a major tool for understanding threats and mitigation actions in the development process.
- Client-server architectures are characterized by a distributed application structure that partitions operations between the providers of a resource or service, called servers, and the requesters, called clients.
- A common utilization of the peer-to-peer model is in file sharing, where machines directly share files without an intermediate storage hub.
- Service-oriented architectures are distributed, modular applications that are platform neutral and have automated interfaces.
- SOAs can involve SOAP, XML, and REST protocols.

- Web services are a form of SOA that use WSDL for interface definitions.
- Rich Internet applications replicate desktop functionality in a web-based form.
- Clients are susceptible to exploitation and remote code injections against the server.
- Radio frequency identification (RFID) is a radio frequency, noncontact means of transferring data between two parties.
- Near-field communication is a protocol and set of standards for communication via radio frequency energy over very short distances.
- Cloud computing is a relatively new term in the computer field used to describe an architecture of scalable services that are automatically provisioned in response to demand.
- Software as a service is a type of cloud computing where the software runs in the cloud on external hardware, and the user derives the benefit through a browser or browser-like interface.
- Platform as a service is a form of cloud computing that offers a complete platform as a solution to a computing need.
- Infrastructure as a service is a form of cloud computing that offers a complete platform as a provisioning solution to a computing need.

Questions

To further help you prepare for the CSSLP exam, answer the following questions. The correct answers are provided in the following section.

1. On which platform can a customer deploy and operate arbitrary software across a cloud-based platform without worrying about the specifics of the environment?
 - A. Infrastructure as a service
 - B. Platform as a service
 - C. Software as a service
 - D. Architecture as a service
2. _____ is a selected collection of elements into a designed solution

stack for a specific problem.

- A. Infrastructure as a service
 - B. Platform as a service
 - C. Software as a service
 - D. Architecture as a service
3. _____ is an architecture that can mimic desktop applications in usability and function.
- A. RIA
 - B. NFC
 - C. REST
 - D. SOAP
4. What is the architectural element that can act as a communication conduit between protocols?
- A. REST
 - B. XML
 - C. ESB
 - D. WSDL
5. Platform-neutral, interoperable, and modular with contract-based interfaces describes what?
- A. SOA
 - B. XML
 - C. WSDL
 - D. ESB
6. Thin clients are examples of what?
- A. Distributed computing
 - B. Message queuing
 - C. Peer-to-peer
 - D. Client server
7. SOA is connected to all of the following except what?

- A. CORBA
 - B. SOAP
 - C. REST
 - D. RIA
8. What is one of the major risks associated with the client server architecture?
- A. Client-side exploits
 - B. Scalability
 - C. Confidentiality
 - D. Stability
9. The term STRIDE stands for what?
- A. Spoofing, tampering, repudiation, information disclosure, denial of service, and elevation of privilege
 - B. Spoofing, tampering, reproducibility, information disclosure, denial of service, and elevation of privilege
 - C. Spoofing, tampering, reproducibility, information disclosure, discoverability, and elevation of privilege
 - D. Spoofing, tampering, repudiation, information disclosure, discoverability, and elevation of privilege
10. Which of the following describes the purpose of threat modeling?
- A. Enumerate threats to the software
 - B. Define the correct and secure data flows in a program
 - C. Communicate testing requirements to the test team
 - D. Communicate threat and mitigation information across the development team

Answers

1. A. IaaS is a form of cloud computing that offers a complete platform as a provisioning solution to a computing need.
2. B. PaaS can exist as a selected collection of elements into a designed

solution stack for a specific problem. This may include apps, databases, web services, storage, and other items that are offered as a service.

3. A. Rich Internet applications (RIAs) are a form of architecture using the Web as a transfer mechanism and the client as a processing device, typically for display formatting control functions.
4. C. An ESB allows XML, EDI, WSDL, REST, DCOM, CORBA, and others to communicate with each other in a seamless fashion.
5. A. SOA characteristics include platform neutrality, interoperability, modularity and reusability, abstracted business functionality, contract-based interfaces, and discoverability.
6. D. Thin clients perform the majority of the processing on the server.
7. D. Rich Internet applications (RIAs) are not necessarily connected to SOAs.
8. A. Client-side exploits are attacks against the client side of the client server architecture.
9. A. The term STRIDE refers to sources of threats: spoofing, tampering, repudiation, information disclosure, denial of service, and elevation of privilege.
10. D. Threat modeling is a tool used to communicate information about threats and the mitigation procedures to all members of the development team.

Secure Software Design

In this chapter you will

- Learn about performing secure interface design
 - Learn about performing architectural risk assessments
 - Learn to model (nonfunctional) security properties and constraints
 - Learn to model and classify data
 - Learn to evaluate and select reusable secure designs
 - Learn to perform security architecture and design reviews
 - Define secure operational architectures
 - Learn to use secure architecture and design principles, patterns, and tools
-
-

Security implementation begins with requirements and becomes built in if designed in as part of the design phase of the secure development lifecycle (SDL). Designing in the security requirements enables the coding and implementation phases to create a more secure product. Minimization of vulnerabilities is the first objective, with the development of layered defenses for those that remain being the second objective. Designing an application is the beginning of implementing security into the final application. Using the information uncovered in the requirements phase, designers create the blueprint developers used to arrive at the final product. It is during this phase that the foundational elements to build the proper security functionality into the application are initiated. To determine which security elements are needed in the application, designers can use the information from the attack surface analysis and the threat model to determine the “what” and “where” elements. Knowledge of secure design principles can provide the “how” elements. Using this information in a comprehensive plan can provide developers with a targeted foundation that will greatly assist in creating a

secure application.

Performing Secure Interface Design

One of the key security interactions for every module of a computer system occurs at the interface. The interface is the boundary between inputs and outputs. Having proper valid inputs is key to every module operating correctly, and proper interface designs assist in ensuring this is the case. A top cause of vulnerabilities is improper input validation, and this can be laid clearly at the foot of interface design. All data movement through a system is subject to modification, and when unauthorized or incorrect, it can result in system failures. Understanding the role of each interface in protecting a module from these issues is critical to proper system construction.

Logging

Applications operate in a larger enterprise environment. It is advantageous to use enterprise resources for the management of security. During the design phase, it is incumbent upon the design team to determine the level and method of integration with enterprise resources. Logging provides the data, but the design team needs to plan how this data can be accessed by the security team once the application is in operation. There are a variety of ways that access to the data can be handled—management consoles, log interfaces, in-band versus out-of-band management—each with advantages and limitations. In most cases, a combination of these techniques should be utilized.

In-band management offers more simplicity in terms of communications. But it has limitations in that it is exposed to the same threats as the communication channel itself. Denial-of-service attacks may result in loss of the ability to control a system. If an adversary sniffs the traffic and determines how the management channel works, then the security of the management channel becomes another problem. Out-of-band management interfaces use a separate communication channel, which may or may not solve the previous communication issues. Properly provisioned, out-of-band management allows management even under conditions where the app itself is under attack. The disadvantage to this approach is that separate, secure communication channels need to be designed.

Creating a separate management interface to allow users to manage security reporting is an additional functionality that requires resources to create. The advantage is that the interface can be designed to make the best presentation of the material associated with the application. The limitation is that this information is in isolation and is not naturally integrated with other pertinent security information. Sending the information to the enterprise operational security management system—typically a security information and event management system—has many advantages. The security operations personnel are already familiar with the system, and information can be cross-correlated with other operational information.

The important aspect of interfacing the security functionality with the enterprise is in the deliberate design to meet security objectives. Ensuring a solid design for successful integration into the enterprise security operations will enable the customer to minimize the risks of using the application.

An important element in any security system is the presence of security logs. Logs enable personnel to examine information from a wide variety of sources after the fact, providing information about what actions transpired, with which accounts, on which servers, and with what specific outcomes. Many compliance programs require some form of logging and log management. The challenges in designing log programs are what to log and where (and for how long) to store it.

What needs to be logged is a function of several criteria. First, numerous compliance programs—HIPAA, SOX, PCI DSS, EOC, and others—have logging requirements, and these need to be met. The next criterion is one associated with incident response. What information would investigators want or need to know to research failures and issues? This is a question for the development team—what is available that can be logged that would provide useful information for investigators, either to the cause of the issue or impact?

The “where to log it” question also has several options, each with advantages and disadvantages. Local logging can be simple and quick for the development team. But it has the disadvantage of being yet another log to secure and integrate into the enterprise log management system. Logs by themselves are not terribly useful. What makes individual logs useful is the combination of events across other logs, detailing the activities of a particular user at a given point in time. This requires a coordination function, one that is supported by many third-party software vendors through their security

information and event management (SIEM) tool offerings. These tools provide a rich analytical environment to sift through and find correlations in large datasets of security information.

Protocol Design Choices

Application programming interfaces (APIs) are a set of definitions and protocols for building and integrating software components. APIs can be thought of as contracts that represent the agreement between program modules on how they communicate between each other. Documentation spells out the specifics of the interface, and this allows external parties the ability to interface with a module in an intended fashion. APIs can act as a bridge between a core in-house program and external functions, allowing the changing of external functions through the selection of a different API. This supports rapid development and deployment of new functionality associated with core processing functionality.

API selection is an architectural and design question, as connections between software modules can increase the attack surface area and associated risk with a program. But APIs also provide a documented and established manner of managing the risk, because they can be chosen and documented and, if necessary, changed in the future—without major disruption to the underlying software project. As APIs are at their core an interface, there can be a wide range of interface technologies available for choice. This decision, as all design and architecture decisions, should be based on overall risk to the system.

Performing Architectural Risk Assessment

Performing an architectural risk assessment is at its core a risk management exercise. Architectural risk assessments identify flaws in the software architecture and extend this to risk associated with the information systems. Using a standard risk assessment methodology, an architectural risk assessment is the starting point for risk assessment associated with the software to be developed. Flaws that are discovered to expose information assets to risk are prioritized based on their impact to the system. For risks above a threshold value, mitigations are developed and implemented as part of the software development process. Understanding the risks associated with

the software under development, at an early stage, provides a wide range of remedies including design-based remedies, and the fixes do not impact the delivery schedule. Conversely, risks found late in the process that require patches and other measures almost always result in additional cost and schedule impacts.

Model (Nonfunctional) Security Properties and Constraints

Designing a software project involves meeting a wide range of requirements. Requirements that specify specific functionalities that the program is supposed to perform with the data are called *functional requirements*. But it takes more than just functional requirements to make a secure program. There are additional requirements that are used to define system attributes such as but not limited to: security, reliability, resilience, performance, maintainability, scalability, and usability. The requirements serve as constraints or restrictions on the design of the system. These are called *nonfunctional requirements*. Nonfunctional requirements are just as important as functional ones, and failures associated with compliance-related requirements can cause significant legal issues.

Model and Classify Data

Data is the reason for software; it is the material that software processes, displays, and stores. Not all data is created equal; it comes in a variety of different types. Understanding the different types of data, and the associated limitations, can have an impact on program design and execution.

Types of Data

Data can come in many forms, and it can be separated into two main types: structured and unstructured. Databases hold a lot of enterprise data, yet many studies have shown that the largest quantity of information is unstructured data in elements such as office documents, spreadsheets, and e-mails. The type of data can play a role in determining the appropriate method of securing it.

Structured

Structured data has defined structures and is managed via those structures. The most common form of structured data is that stored in databases and arranged in tables that relate to the specific structures. Other forms of structured data include formatted file structures, Extensible Markup Language (XML) data, JSON, and certain types of text files, such as log files. The structure allows a parser to go through the data, sort, and search.



EXAM TIP Structured data is identifiable because it is organized in a structure, not because of how it is stored. To determine if the data is structured data, examine it for structure (relationships between specific data elements), not the storage mechanism. Unstructured data can be stored in structured storage mechanisms such as databases.

Unstructured

Unstructured data is the rest of the data in a system. Although it may be structured per some application such as Microsoft Word, the structure is irregular and not easily parsed and searched. It is also more difficult to modify outside the originating application. Unstructured data makes up the vast majority of data in most firms, but its unstructured nature makes it more difficult to navigate and manage. A good example of this is examining the number that represents the sales totals for the previous quarter. This can be found in databases, in the sales application system, in Word documents and PDFs describing the previous quarter's performance, and in e-mails between senior executives. When searching for specific data items, some of these sources are easily navigated (the structured ones, such as financial scorecards), while it is virtually impossible to find items in e-mails and word processing or PDF documents without the use of enterprise data archival and discovery tools.

Evaluate and Select Reusable Secure Design

As a manufacturing paradigm, the overall concept of reusability has probably been around since the time of Julius Caesar. But the concept of code reuse has gained traction in the software industry only in the past 20 years. At the machine level, code reuse has been a fundamental principle of compilers since the 1950s. In fact, reusing fundamental machine functions instead of individually writing a machine language program each time an operation had to be performed was the innovation that essentially ushered in modern high-level programming languages. Moreover, in that respect, standard template libraries (STLs) are still a fundamental part of compiler languages like C++.

In simple terms, at the application level, code reuse is nothing more than the construction of new software from existing components. Those components might have been created for another project, or they might have been common functions that were created and stuck in a library with the intention of reusing them in a range of products. From a cost-benefit standpoint, reuse is a logical way to approach the problem of producing code. Having standard, already written functions, templates, or procedures available saves time and thereby reduces cost. It also ensures quality control and a standard level of capability for the code that is produced from reusable parts. The problem, however, is that a single reused module that has been compromised can introduce risks into a wide range of subsequent applications. Therefore, in the conventional supply chain risk management sense, it is absolutely critical to be able to define the specific situations and uses where a given reusable module or reuse process can be safely and legitimately applied.

Creating a Practical Reuse Plan

The general aim of the reuse process is to develop a secure library of reusable components, much like a physical parts inventory, which will both leverage production and assure quality. The reused code modules have to be assured correct to avoid duplicating prior errors in logic and syntax. That is the reason why reuse is supported by basic software design principles such as modularity, information hiding, and decoupling. Those principles ensure that the code can be validated to be both properly functioning and secure. However, to guarantee that security, a range of practical risk management activities has to be adopted and followed.

Because reusable code usually originates from other, sometimes unknown

sources, utilizing it can be risky. Therefore, the terms of its use have to be clearly stated in any contract involving reuse, along with a basis for negotiated liability for loss or damage should problems arise. The strategy and decision criteria are normally laid out as a result of a strategic planning activity. That activity guides code reuse for the project. It is usually part of the overall risk management plan. The strategy ensures that the types of reuse that will be utilized are well defined and that reuse planning is conducted accordingly. Planning stipulations typically include a requirement to state whether the reuse process will involve open-source code, other reusable code, and/or value-added products or services. The reuse planning process supports decision-makers by helping them to evaluate the advantages of reusable code as well as how to ensure any reusable software against risk.

Once risks within the project space and to the reusable objects are laid out, prioritized, and documented, the organization conducts a risk management security testing process with all relevant stakeholders. That includes software developers, asset managers, domain experts, and users. The aim of this process is to validate the strategy that will effectively guide the reuse process. Once the security testing is complete and the results accepted, the description of all identified risks and their proposed mitigations is passed along to the appropriate development managers for application to the reuse process. Moreover, since risks appear constantly, this is an iterative process that is conducted in a cycle suitable to the risk environment.

Credential Management

There are numerous methods of authentication, and each has its own set of credentials that require management. The identifying information that is provided by a user as part of their claim to be an authorized user is sensitive data and requires significant protection. The identifying information is frequently referred to as *credentials*. These credentials can be in the form of a passed secret, typically a password. Other common forms include digital strings that are held by hardware tokens or devices, biometrics, and certificates. Each of these forms has advantages and disadvantages.

Each set of credentials, regardless of the source, requires safekeeping on the part of the receiving entity. Managing these credentials includes tasks such as credential generation, storage, synchronization, reset, and revocation. Because of the sensitive nature of manipulating credentials, all of these

activities should be logged.

X.509 Credentials

X.509 refers to a series of standards associated with the manipulation of certificates used to transfer asymmetric keys between parties in a verifiable manner. A digital certificate binds an individual's identity to a public key, and it contains all the information a receiver needs to be assured of the identity of the public key owner. After a registration authority (RA) verifies an individual's identity, the certificate authority (CA) generates the digital certificate. The digital certificate can contain the information necessary to facilitate authentication.

X.509 Digital Certificate Fields

The following fields are included within an X.509 digital certificate:

- **Version number** Identifies the version of the X.509 standard that was followed to create the certificate; indicates the format and fields that can be used.
- **Serial number** Provides a unique number identifying this one specific certificate issued by a particular CA.
- **Signature algorithm** Specifies the hashing and digital signature algorithms used to digitally sign the certificate.
- **Issuer** Identifies the CA that generated and digitally signed the certificate.
- **Validity** Specifies the dates through which the certificate is valid for use.
- **Subject** Specifies the owner of the certificate.
- **Public key** Identifies the public key being bound to the certified subject; also identifies the algorithm used to create the private/public key pair.
- **Certificate usage** Specifies the approved use of the certificate, which dictates intended use of this public key.
- **Extensions** Allow additional data to be encoded into the certificate to expand its functionality. Companies can customize the use of

certificates within their environments by using these extensions. X.509 version 3 has expanded the extension possibilities.

Certificates are created and formatted based on the X.509 standard, which outlines the necessary fields of a certificate and the possible values that can be inserted into the fields. As of this writing, X.509 version 3 is the most current version of the standard. X.509 is a standard of the International Telecommunication Union (www.itu.int). The IETF's Public-Key Infrastructure (X.509), or PKIX, working group has adapted the X.509 standard to the more flexible organization of the Internet, as specified in RFC 3280, and is commonly referred to as PKIX for Public Key Infrastructure X.509.

The public key infrastructure (PKI) associated with certificates enables the passing and verification of these digital elements between firms. Because certificates are cryptographically signed, elements within them are protected from unauthorized alteration and can have their source verified. Building out a complete PKI infrastructure is a complex endeavor, requiring many different levels of protection to ensure that only authorized entities are permitted to make changes to the certification.

Setting up a functioning and secure PKI solution involves many parts, including certificate authorities, registration authorities, and certificate revocation mechanisms, either certificate revocation lists (CRLs) or Online Certificate Status Protocol (OCSP).

[Figure 7-1](#) shows the actual values of the different certificate fields for a particular certificate in Internet Explorer. The version of this certificate is V3 (X.509 v3), and the serial number is also listed—this number is unique for each certificate that is created by a specific CA. The CA used the SHA1 hashing algorithm to create the message digest value, and it then signed it using the CA's private key, which used the RSA algorithm.



Figure 7-1 Digital certificate

X.509 certificates provide a wide range of benefits to any application that needs to work with public key cryptography. Certificates provide a standard means of passing keys, a standard that is accepted by virtually every provider and consumer of public keys. This makes X.509 a widely used and proven technology.

Single Sign-On

Single sign-on (SSO) makes it possible for a user, after authentication, to have their credentials reused on other applications without the user re-entering the secret. To achieve this, it is necessary to store the credentials outside of the application and then reuse the credentials against another system. There are a number of ways that this can be accomplished, but two of the most popular and accepted methods for sharing authentication information are Kerberos and Security Assertion Markup Language (SAML). The OpenID protocol has proven to be a well-vetted and secure protocol for SSO. However, as with all technologies, security vulnerabilities can still occur due to misuse or misunderstanding of the technology.

The key concept with respect to SSO is federation. In a federated authentication system, users can log in to one site and access another or affiliated site without re-entering credentials. The primary objective of federation is user convenience. Authentication is all about trust, and federated trust is difficult to establish. SSO can be challenging to implement, and

because of trust issues, it is not an authentication panacea. As in all risk-based transactions, a balance must be achieved between the objectives and the risks. SSO-based systems can create single-point-of-failure scenarios, so for certain high-risk implementations, their use is not recommended.

Flow Control

In information processing systems, information flows between nodes, between processes, and between applications. The movement of information across a system or series of systems has security consequences. Sensitive information must be protected, with access provided to authorized parties and protected from unauthorized ones. The movement of information must be channeled correctly and protected along the way. There are technologies, firewalls, proxies, and queues that can be utilized to facilitate proper information transfer.

Firewalls

Firewalls act as policy enforcement devices, determining whether to pass or block communications based on a variety of factors. Network-level firewalls operate using the information associated with networking to determine who can communicate with whom. Next-generation firewalls provide significantly greater granularity in communication decisions. Firewalls operate on a packet level and can be either stateless or stateful. Basic network firewalls operate on a packet-by-packet basis and use addressing information to make decisions. In doing so, they are stateless, not carrying information from packet to packet as part of the decision process. Advanced firewalls can analyze multiple packets and utilize information from the protocols being carried to make more granular decisions. Did the packet come in response to a request from inside the network? Is the packet carrying information across web channels, port 80, using authorized or unauthorized applications? This level of stateful packet inspection, although difficult to scale, can be useful in providing significant levels of communication protection.

Firewalls are basically devices that, at the end of the day, are supposed to allow the desired communications and block undesired communications. Malicious attempts to manipulate a system via communication channels can be detected and blocked using a firewall. Firewalls can work with intrusion detection systems, acting as the enforcer in response to another system's

inputs. One of the limitations of firewalls is governed by network architecture. When numerous paths exist for traffic to flow between points, determining where to place devices such as firewalls becomes increasingly difficult and at times nearly impossible. Again, as with all things security, balance becomes a guiding principle.

Proxies

Proxies are similar to firewalls in that they can mediate traffic flows. They differ in that they act as middlemen, somewhat like a post-office box. Traffic from untrusted sources is terminated at a proxy, where the traffic is received and to some degree processed. If the traffic meets the correct rules, it can then be forwarded on to the intended system. Proxies come in a wide range of capabilities, from simple to complex, both in their rule-processing capabilities and additional functionalities. One of these functionalities is caching—a temporary local storage of web information that is frequently used and seldom changed, like images. In this role, a proxy acts as a security device and a performance-enhancing device.

Application Firewalls

Application firewalls are becoming more popular, acting as application-specific gateways between users, and potential users, and web-based applications. Acting as a firewall proxy, web application firewalls can monitor traffic in both directions, client to server and server to client, watching for anomalies. Web application firewalls act as guards against both malicious intruders and misbehaving applications. Should an outsider attempt to perform actions that are not authorized to an application, the web application firewall can block the requests from getting to the application. Should the application experience some failure, resulting in, say, large-scale data transfers when only small data transfers are the norm, again, the web application firewall can block the data from leaving the enterprise.



EXAM TIP One of the requirements of the PCI Data Security Standard is for web applications either to have a web application firewall between the

server and users or to perform application code reviews.

Data Loss Prevention

Data is the asset that security ultimately strives to protect. There may be secondary assets, such as equipment, controls, and applications, but these all are in place to protect the data in an organization. Data loss prevention (DLP) technologies exist as a last line of defense. DLP solutions act by screening traffic, looking for traffic that meets profile parameters. The profile may be size of transfer, may be destination, or might be specific data elements that are protected. If any of these elements is detected, then a data exfiltration event is in progress, and the connection is terminated.

Simple in theory but complex in implementation, DLP is a valuable tool in a defense-in-depth environment. One of the challenges has to do with detection location. DLP technology needs to be in the actual netflow path involved in the data transfer. In simple networks, this is easy; in large enterprises, this can be challenging. In enterprises with numerous external connections, it can be complex and expensive. The second challenge is visibility into the data itself. Attackers use encryption to prevent the data streams from being detected. The whole process gets more complicated with the move of services and data into the cloud.

Virtualization

It is now common for both servers and workstations to have a virtualization layer between the hardware and the operating system. This virtualization layer provides many benefits, allowing multiple operating systems to operate concurrently on the same hardware. Virtualization offers many advantages in the form of operational flexibility. It also offers some security advantages. If a browser surfing the Web downloads harmful content, the virtual machine can be deleted at the end of the session, preventing the spread of any malware to the other operating systems. The major providers of virtualization software are VMware, Microsoft, Oracle, and Xen.

Virtualization can provide many benefits to an organization, and these benefits are causing the rapid move to virtualization across many enterprises. These benefits include

- Reduced cost of servers resulting from server consolidation
- Improved operational efficiencies from administrative ease of certain tasks
- Improved portability and isolation of applications, data, and platforms
- Operational agility to scale environments, i.e., cloud computing

Virtual machines (VMs) are becoming a mainstream platform in many enterprises because of their advantages. Understanding the ramifications of a VM environment on an application can be important for a development team if there is any chance that the application would ever be deployed in one.

Trusted Computing

Trusted computing (TC) is a term used to describe technology developed and promoted by the Trusted Computing Group. This technology is designed to ensure that the computer behaves in a consistent and expected manner. One of the key elements in the TC effort is the Trusted Platform Module (TPM), a hardware interface for security operations.

TCB

The trusted computing base (TCB) of a computer system is the set of all hardware, firmware, and/or software components that are critical to its security. The concept of a TCB has its roots in the early 1970s, when computer security researchers examined systems and found that much of the system could misbehave and not result in security incidents. With the basis of security being defined in an organization's security policy, one risk is that given certain strict interpretations of security, a consultant could find or justify anything as affecting security.

With this concern in the open, the principal issue from days gone by is that of privilege escalation. If any element of the computer system has the ability to effect an increase in privilege without it being authorized, then this would be a violation of the security, and this part of the system would be part of the TCB. The idea of a TCB is not just theoretical conjecture, but indeed creates the foundation for security principles such as complete mediation.

TPM

The Trusted Platform Module is a hardware implementation of a set of cryptographic functions on a computer's motherboard. The intent of the TPM is to provide a base level of security that is deeper than the operating system and virtually tamperproof from the software side of the machine. The TPM can hold an encryption key that is not accessible to the system except through the TPM chip. This assists in securing the system, but has also drawn controversy from some quarters concerned that the methodology could be used to secure the machine from its owner. There also are concerns that the TPM chip could be used to regulate what software runs on the system.

[Figure 7-2](#) illustrates the several different features in the TPM hardware available for use by the computing platform. It has a series of cryptographic functions: a hash generator, an RSA key generator, and a cryptographically appropriate random number generator, which work together with an encryption/decryption signature engine to perform the basic cryptographic functions securely on the silicon. The chip also features storage areas with a manufacturer's key, the endorsement key, and a series of other keys and operational registers.

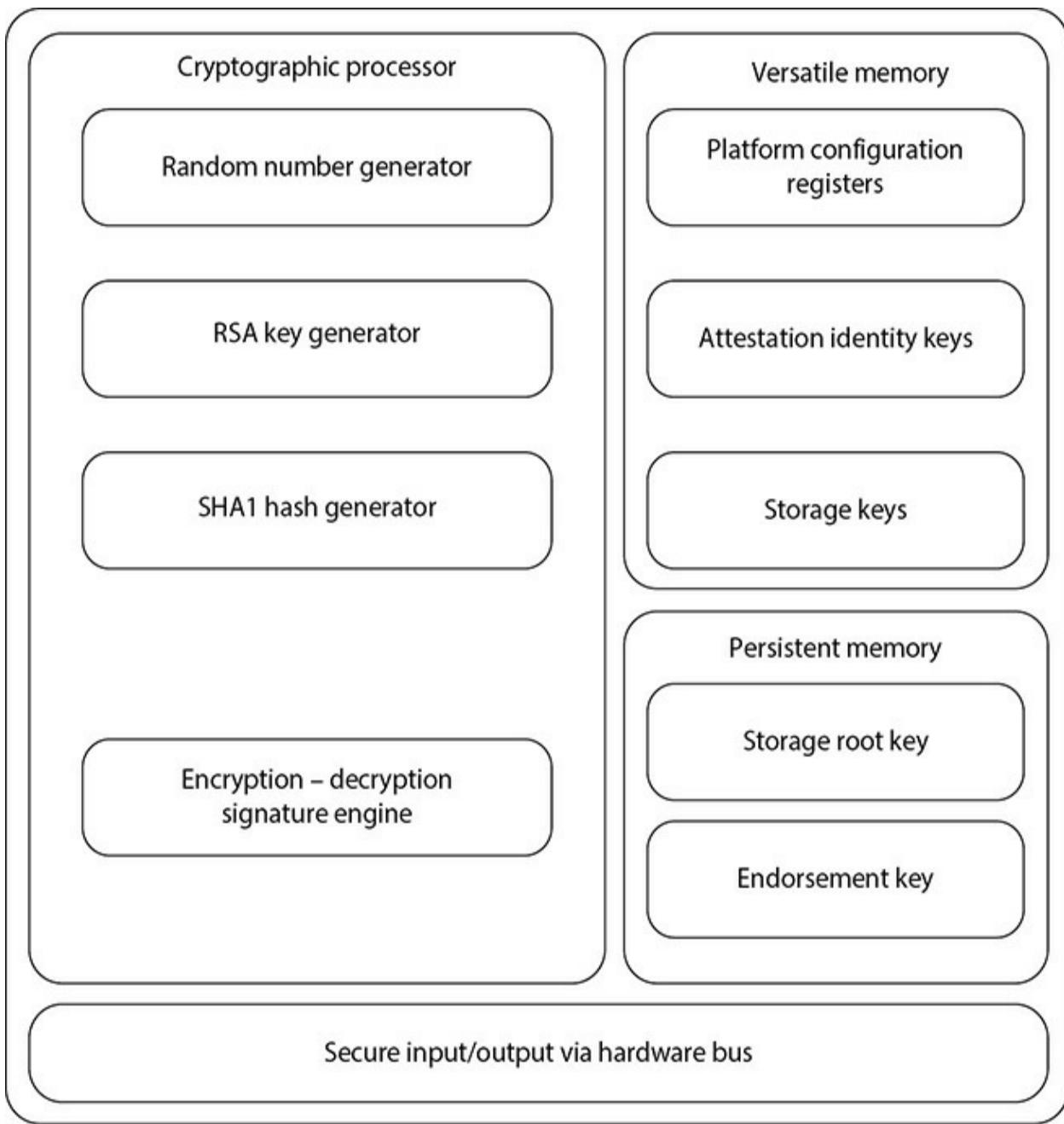


Figure 7-2 TPM hardware functions

Database Security

Databases are technologies used to store and manipulate data. Relational databases store data in tables and have a wide array of tools that can be used to access, manipulate, and store data. Databases have a variety of security mechanisms to assist in creating the appropriate level of security. This

includes elements for confidentiality, integrity, and availability. The details of designing a database environment for security are beyond the scope of the CSSLP practitioner, but it is still important to understand the capabilities.

Encryption can be employed to provide a level of confidentiality protection for the data being stored in a database. Data structures, such as views, can be created, giving different parties different levels of access to the data stored in the database. Programmatic structures called *stored procedures* can be created to limit access to only specific elements based on predefined rules. Backup and replication strategies can be employed to provide near-perfect availability and redundancy for critical systems. Taken together, the protections afforded the data in a modern database can be comprehensive and valuable. The key is in defining the types and levels of protection required based on risk.

Encryption

Data stored in a database is a lucrative target for attackers. Just like the vault in a bank, it is where the valuable material is stored, so gaining the correct level of access, for instance, administrative rights, can be an attacker's dream and a defender's nightmare. Encrypting data at rest is a preventative control mechanism that can be employed virtually anywhere the data is at rest, including databases. The encryption can be managed via native database management system functions, or it can be done using cryptographic resources external to the database.



EXAM TIP Primary keys are used to index and join tables and, as such, cannot be obfuscated or encrypted. This is a good reason not to use personally identifiable information (PII) and personal health information (PHI) as keys in a database structure.

Numerous factors need to be considered when creating a database encryption strategy. These include, but are not limited to, the following:

- What is the level of risk classification associated with the data?

- What is the usage pattern of the data—how is it protected in transit and in use?
- What is the differing classification across elements of the data—are some more sensitive than others?
- How is encryption being handled in the enterprise for other projects?
- What are the available encryption options to the development team?

In a given data record, not all of the information is typically of the same level of sensitivity. If only a few columns out of many are sensitive, then data segregation may provide a means of protecting smaller chunks with greater efficiency. Determining the detailed, data element by data element requirements for protection can provide assistance in determining the correct protection strategy.



EXAM TIP Regulations such as GLBA, HIPAA, and PCI DSS can impose protection requirements around certain data elements, such as PII and PHI. It is important for members of the design and development team to understand this to avoid operational issues later in the software lifecycle.

Triggers

Triggers are specific database activities that are automatically executed in response to specific database events. Triggers are a useful tool, as they can automate a lot of interesting items. Changes to a record can trigger a script; adding a record can trigger a script. Tag any database task and assign a script —this allows a lot of flexibility. Need to log something and include business logic? Triggers can provide the flexibility to automate anything in a database.

Views

Views are programmatically designed extracts of data in a series of tables. Tables can contain all the data, and a view can provide a subset of the information based on some set of business rules. A table could contain a record that provides all the details about a customer: addresses, names, credit card information, etc. Some of this information should be protected—PII and

credit card information, for instance. A view can provide data to a shipping routine for only the ship-to columns and not provide the protected information. Using a view as opposed to a table protects specific information as it is not possible to disclose what isn't there.

Privilege Management

Databases have their own internal access control mechanism, which are similar to ACL-based controls to file systems. Designing the security system for data records, users, and roles requires the same types of processes as designing file system access control mechanisms. The two access control mechanisms can be interconnected, with the database system responding to the enterprise authorization systems, typically through roles defined in the database system.

Concurrency

In many computer systems there are numerous activities handled in a parallel fashion. Programs can have multiple threads, which can independently access objects, and a means of thread safety must be employed to prevent thread concurrency errors. This makes multithreaded programs more difficult to construct as these mechanisms must be included because threads have no control over other threads and issues where they both read or write information from common sources can cause timing issues. Databases have the same problem, especially when scale becomes involved. If you have a large database system that is concurrently addressing transactions for literally thousands of concurrent customers, how do you manage things like concurrent inventory? How do you manage updates for common elements in multiple other transactions that are concurrently happening in a safe and predictable manner? Database management systems can handle these messy details for you automatically through database management system (DBMS) concurrency controls. These systems are designed to avoid the following types of problems:

- **Lost updates** These can occur when multiple transactions select the same row and update the row based on the value selected, and only one transaction is committed.
- **Dirty read** This is an uncommitted dependency issue that occurs

when a second transaction selects a row that is updated by another transaction but not yet committed.

- **Nonrepeatable reads** This is an error exemplified by a second transaction trying to access the same row several times and reading different data each time.
- **Incorrect summary issues** These can occur when one transaction uses a summary of all the values of all the instances of a repeated data item, and a second transaction updates a few instances of those specific data items, resulting in the summary no longer reflecting the correct result.

DBMSs offer different concurrency control protocols, and each method has different costs and benefits between the amount of concurrency they allow and the amount of overhead that they impose. Selecting the appropriate controls is part of the DBMS design architecture.

Programming Language Environment

Software developers use a programming language to encode the specific set of operations in what is referred to as *source code*. The programming language used for development is seldom the language used in the actual instantiation of the code on the target computer. The source code is converted to the operational code through compilers, interpreters, or a combination of both. The choice of the development language is typically based on a number of criteria, the specific requirements of the application, the skills of the development team, and a host of other issues.

Compilers offer one set of advantages, and interpreters others. Systems built in a hybrid mode use elements of both. Compiled languages involve two subprocesses: compiling and linking. The compiling process converts the source code into a set of processor-specific codes. Linking involves the connecting of various program elements, including libraries, dependency files, and resources. Linking comes in two forms: static and dynamic. Static linking copies all the requirements into the final executable, offering faster execution and ease of distribution. Static linking can lead to bloated file sizes.

Dynamic linking involves placing the names and relative locations of dependencies in the code, with these being resolved at runtime when all elements are loaded into memory. Dynamic linking can create a smaller file

but does create risk from hijacked dependent programs.

Another factor influencing program language choice is type safety. Type safety refers to the management of data types by a program, where strongly typed languages enforce data types and consistency in processing of types. Some languages are strongly typed and require type match or errors are generated. Other languages do not have these constraints. As in all trade-offs, there are performance costs, and these are considered as well. At the end of the day, most programming language decisions are based more on what the programming team is proficient in than other factors for the majority of projects.

Interpreters use an intermediary program that results in the execution of the source code on a target machine. Interpreters provide slower execution, but faster change between revisions, as there is no need for recompiling and relinking. The source code is actually converted by the interpreter into an executable form in a line-by-line fashion at runtime.

A hybrid solution takes advantage of both compiled and interpreted languages. The source code is compiled into an intermediate stage that can be interpreted at runtime. The two major hybrid systems are Java and Microsoft .NET. In Java, the intermediate system is known as the Java Virtual Machine (JVM), and in the .NET environment, the intermediate system is the common language runtime (CLR).

CLR

Microsoft's .NET language system has a wide range of languages in the portfolio. Each of these languages is compiled into what is known as common intermediate language (CIL), also known as Microsoft Intermediate Language (MSIL). One of the advantages of the .NET system is that a given application can be constructed using multiple languages that are compiled into CIL code that is executed using the just-in-time compiler. This compiler, the common language runtime (CLR), executes the CIL on the target machine. The .NET system operates what is known as *managed code*, an environment that can make certain guarantees about what the code can do. The CLR can insert traps, garbage collection, type safety, index checking, sandboxing, and more. This provides a highly functional and stable execution environment.

JVM

In Java environments, the Java language source code is compiled to an intermediate stage known as *byte code*. This byte code is similar to processor instruction codes but is not executable directly. The target machine has a JVM that executes the byte code. The Java architecture is referred to as the Java Runtime Environment (JRE), which is composed of the JVM and a set of standard class libraries, the Java Class Library. Together, these elements provide for the managed execution of Java on the target machine.

Compiler Switches

Compiler switches enable the development team to control how the compiler handles certain aspects of program construction. A wide range of options are available, manipulating elements such as memory, stack protection, and exception handling. These flags enable the development team to force certain specific behaviors using the compiler. The /GS flag enables a security cookie on the stack to prevent stack-based overflow attacks. The /SAFEH switch enables a safe exception handling table option that can be checked at runtime. The designation of the compiler switch options to be used in a development effort should be one of the elements defined by the security team and published as security requirements for use in the SDL process.

Sandboxing

Sandboxing is a term for the execution of computer code in an environment designed to isolate the code from direct contact with the target system. Sandboxes are used to execute untrusted code, code from guests, and unverified programs. They work as a form of virtual machine and can mediate a wide range of system interactions, from memory access to network access, access to other programs, the file system, and devices. The level of protection offered by a sandbox depends upon the level of isolation and mediation offered.

Managed vs. Unmanaged Code

Managed code is executed in an intermediate system that can provide a wide range of controls. .NET and Java are examples of managed code, a system with a whole host of protection mechanisms. Sandboxing, garbage collection, index checking, type safe, memory management, and multiplatform

capability—these elements provide a lot of benefit to managed code-based systems. Unmanaged code is executed directly on the target operating system. Unmanaged code is always compiled to a specific target system. Unmanaged code can have significant performance advantages. In unmanaged code, memory allocation, type safety, garbage collection, etc., need to be taken care of by the developer. This makes unmanaged code prone to memory leaks such as buffer overruns and pointer overrides and increases the risk.

Operating System Controls and Services

Operating systems are the collection of software that acts between the application program and the computer hardware resources. Operating systems exist for all platforms, from mainframes to PCs to mobile devices. They provide a functional interface to all the services enabled by the hardware. Operating systems create the environment where the applications execute, providing them the resources necessary to function. There are numerous different types of operating systems, each geared for a specific purpose. Systems created for multiple users have operating systems designed for managing multiple user processes, keeping them all separate and managing priorities. Real-time and embedded systems are designed to be simpler and leaner, and their operating systems enable those environments.

Secure Backup and Restoration Planning

Backups are an essential part of modern software. Both maintaining the code base, including configuration settings as well as the data that the system operates on is essential if backups are to be truly functional. In some cases, the data backup and restoration planning is handled by another system, such as a database. Securing the configuration settings for software in deployment is a serious matter, and planning needs to include how these settings are protected, backed up, and restored if necessary. In all cases, the secure system design process requires an understanding, via planning and communication, across the design team of the necessary elements associated with backing up and restoring all elements of the software and the data. More detailed information on these elements is in [Chapter 17](#).

Secure Data Retention, Retrieval, and Destruction

Data that is going to be persistent in the system, or stored, must have a series of items defined: who is the data owner, what is the purpose of storing the data, what levels of protection will be needed, and how long will it need to be stored? These are just some of the myriad of questions that need answers.

Data that is retained in an enterprise becomes subject to the full treatment of data owner and data custodian responsibilities. The protection schemes need to be designed, not just for the primary storage, but for alternative forms of storage as well, such as backups, copies for disaster recovery (DR) sites, and legal hold archives.

Important elements to consider in both security and retention are system logs. Data in log files can contain sensitive information, thus necessitating protection, and appropriate retention schemes need to be devised. Log files are important elements in legal hold, e-discovery, and many compliance elements. Proper log data security and planning of lifecycle elements are important for CSSLPs to consider throughout the development process.

Data destruction serves two primary purposes in the enterprise. First, it serves to conserve resources by ceasing to spend resources on data retention for elements that have no further business purpose. Second, it can serve to limit data-based liability in specific legal situations. The length of storage requirements is set by two factors: business purpose and compliance. Once data has reached its end of life as defined by all of these requirements, it is the data custodian's task to ensure it is appropriately disposed of from all applicable sources. Alternative sources, such as backups, copies for DR sites, data warehouse history, and other copies, need to be managed. Legal hold data is managed separately and is not subject to normal disposal procedures.

Perform Security Architecture and Design Review

Secure systems do not just happen; they are architected and engineered into existence. The objective is to secure both the current state and future growth states, and this requires initial efforts and reviews. Depending upon the scope of the system and its components, a wide range of security methodologies are available. From guidelines developed and published by the National Institute

of Standards and Technology, to the 20 Common Security Controls, to OWASP guidance for securing web applications, there are plenty of sources for good guidance. And for specialized data sources, including financial, healthcare, and governmental data, there are additional guides. What is important is that the correct sources are used, a security architecture is established, and the system is regularly reviewed to ensure that it is being followed.

Define Secure Operational Architecture

Defining the secure operational architecture for a system involves many different elements, all with a unifying goal: create a systematic approach to a system with respect to managing risks. The security architecture includes the systems, processes, people, and tools used to identify, prevent, and mitigate conditions that result in risk. This entire premise is designed and built around the operational deployment of the system and takes into account system topologies, communication methods, and interfaces that will provide information as to normal or abnormal conditions.

There are many different enterprise security architecture frameworks available for use. The following are the most commonly employed:

- **SABSA** A series of integrated frameworks, models, methods, and processes, used independently or as a holistic integrated enterprise solution. See <https://sabsa.org>.
- **Open Group Library** A wide range of architectural standards, guides, and information for establishing a secure enterprise architecture. See <https://publications.opengroup.org/?publicationid=12380>.
- **Open Security Architecture (OSA)** A set of tools, controls, and patterns to assist in creating a secure architecture environment. See <https://www.opensecurityarchitecture.org/cms/index.php>.

All three of these can be used together in a supporting manner, as each set has different levels of focus. A newer form of operational environment is the DevSecOps movement, a secure version of the Agile DevOps methods. One of the better references to see how to deploy software at the “speed of operations” while maintaining a secure enterprise is the US DoD Enterprise

DevSecOps Reference Design v1.0 document (https://dodcio.defense.gov/Portals/0/Documents/DoD%20Enterprise%20DevSecOps%20Reference%20Design%20v1.0_Public%20Release.pdf). The use of DevOps or DevSecOps does not exclude the use of the previously mentioned architectural guidance documents.

Use Secure Architecture and Design Principles, Patterns, and Tools

The use of a security architecture is not an academic exercise. It is a regular part of the operations involved in the deployment and operation of a software solution. Designing a security architecture is just the first step; the design must be followed by policies and procedures that use tools and methods to follow the architecture and make it a living real thing in the enterprise. Part of the design process for the architecture is the consideration of the necessary policies, patterns, tools, and procedures, and the integration of these into the development process.

OWASP has published a development guide to assist developers in actually operationalizing the principles of security. Although the OWASP Security Design Principles have been created to help developers build highly secure web applications, they apply to other software projects as well. These security design principles include elements such as clarifying assets, understanding attackers, paying attention to CIA and security principles, and implementing an architecture to support these efforts including both operational and procedural components.

Chapter Review

This chapter examined the elements of a secure design process by looking at how to perform a secure interface design followed by an architectural risk assessment and the modeling of nonfunctional security properties. The chapter then explored how to model and classify data, followed by a detailed examination of the tasks associated with evaluating and selecting a reusable secure design.

The final topics of the chapter were performing security architecture and design reviews, defining a secure operational architecture, and using secure

architecture and design principles, patterns, and tools.

Quick Tips

- A key security interaction for every module of a computer system occurs at the interface.
- Application programming interfaces are a set of definitions and protocols for building and integrating software components.
- Architectural risk assessments identify flaws in the software architecture and extend this to risk associated with the information systems.
- Requirements that specify specific functionalities that the program is supposed to perform with the data are called functional requirements.
- Important nonfunctional requirements include but are not limited to security, reliability, resilience, performance, maintainability, scalability, and usability requirements.
- Understanding the different types of data, and the associated limitations, can have an impact on program design and execution.
- In simple terms, at the application level, code reuse is nothing more than the construction of new software from existing components.
- Security architectures, with their designs, patterns, and tools, can assist in ensuring security is designed into a system.

Questions

To further help you prepare for the CSSLP exam, answer the following questions. The correct answers are provided in the following section.

1. Reusing components to reduce risk is an example of what?
 - A. Leverage existing components
 - B. Separation of duties
 - C. Weakest link
 - D. Least common mechanism
2. Security management interfaces in an application are best served by what?

- A. Logging exception data
 - B. Integration into existing enterprise security management systems
 - C. Exception management systems
 - D. Separate management interface within the application
3. Out-of-band management interfaces solve which of the following problems?
- A. Require separate communication channel
 - B. Require less bandwidth
 - C. Reduce development risks
 - D. Reduce operational risks
4. Which of the following are not elements associated with certificates?
- A. RA
 - B. OSCP
 - C. CA
 - D. CLR
5. What is a device that moderates traffic and includes caching of content?
- A. Proxy
 - B. Application firewall
 - C. Firewall
 - D. DLP
6. What is one of the biggest challenges in deploying DLP technologies?
- A. Access control lists
 - B. Network proxies
 - C. Network speeds
 - D. Network architecture
7. What is an example of a hybrid system with both compiling and interpreting?
- A. JVM
 - B. C++

- C. SQL
 - D. TCB
8. The process of isolating the executing code from direct contact with the resources of the target system is referred to as what?
- A. Trusted computing
 - B. MSIL
 - C. Managed code
 - D. Sandboxing
9. What is an advantage of unmanaged code?
- A. Performance
 - B. Security
 - C. Library functions
 - D. Portability
10. _____ is a series of standards associated with the manipulation of certificates used to transfer asymmetric keys between parties in a verifiable manner.
- A. X.509
 - B. PKIX
 - C. OSCP
 - D. CRL

Answers

- 1. A. Leveraging existing components reduces risk by using proven elements.
- 2. B. Integration into the enterprise security management system provides many operational benefits to an organization through increased operational efficiencies.
- 3. D. Out-of-band management interfaces are less prone to interference from denial-of-service attacks against an application, reducing operational risk from loss of management control.

- 4.** **D.** The common language runtime is a Microsoft-specific hybrid language environment.
- 5.** **A.** Proxies can cache content for multiple systems in an environment to improve performance.
- 6.** **D.** Network architectures can result in multiple paths out of a network, making DLP placement difficult.
- 7.** **A.** The Java Virtual Machine (JVM) interprets byte code into code for a system.
- 8.** **D.** Sandboxing is the technology used to isolate untrusted code from system resources.
- 9.** **A.** Unmanaged code can have a performance advantage over managed code.
- 10.** **A.** X.509 describes the infrastructure of using certificates for key transfer.

PART IV

Secure Software Implementation

- [**Chapter 8**](#) Secure Coding Practices
- [**Chapter 9**](#) Analyze Code for Security Risks
- [**Chapter 10**](#) Implement Security Controls

Secure Coding Practices

In this chapter you will

- Learn to adhere to relevant secure coding practices
 - Explore common implementation practices
-

Secure coding practices are a key element in the development of a secure development lifecycle. Building upon the foundations of requirements, and the variety of implementation models, this chapter examines specific coding practices that can be employed to achieve higher levels of secure code. Understanding the key relevant secure coding practices and the common implementation processes is an essential part of creating a secure development process. To achieve these goals, the software development team needs to have a detailed understanding of the specific threats and vulnerabilities in the software under development.

In [Chapter 1](#) we introduced the concept and ideas behind an SLD methodology. In [Chapter 3](#) we covered the use of secure coding standards as part of a specific development process. In [Chapter 4](#) we introduced a wide range of standards employed in software development and the role of regulation and compliance in secure development endeavors. These concepts do not stop at the end of design but are carried through the development process, and they play a critical role during the coding process. The coding process is where the conditions behind the design become built into the software product. There are many known issues that have been discovered over the years, and it is critical that current development not fail due to the known patterns of vulnerabilities.

Declarative vs. Imperative Security

Security can be instantiated in two different ways in code: in the container itself or in the content of the container. Declarative programming is when programming specifies the what, but not the how, with respect to the tasks to be accomplished. An example is SQL, where the “what” is described and the SQL engine manages the “how.” Thus, declarative security refers to defining security relations with respect to the container. Using a container-based approach to instantiating security creates a solution that is more flexible, with security rules that are configured as part of the deployment and not the code itself. Security is managed by the operational personnel, not the development team.

Imperative programming, also called *programmatic security*, is the opposite case, where the security implementation is embedded into the code itself. This can enable a much greater granularity in the approach to security. This type of fine-grained security, under programmatic control, can be used to enforce complex business rules that would not be possible under an all-or-nothing container-based approach. This is an advantage for specific conditions, but it tends to make code less portable or reusable because of the specific business logic that is built into the program.

The choice of declarative or imperative security functions, or even a mix of both, is a design-level decision. Once the system is designed with a particular methodology, then the SDL can build suitable protections based on the design. This is one of the elements that requires an early design decision, as many other elements are dependent upon it.

Bootstrapping

Bootstrapping refers to the self-sustaining startup process that occurs when a computer starts or a program is initiated. When a computer system is started, an orchestrated set of activities is begun that includes power-on self-test (POST) routines, boot loaders, and operating system initialization activities. Securing a startup sequence is a challenge—malicious software is known to interrupt the bootstrapping process and insert its own hooks into the operating system.

When coding an application that relies upon system elements, such as environment variables like path, care must be taken to ensure that values are

not being changed outside the control of the application. Using configuration files to manage startup elements and keeping them under application control can help in securing the startup and operational aspects of the application.

Cryptographic Agility

Cryptography is a complex issue, and one that changes over time as weaknesses in algorithms are discovered. When an algorithm is known to have failed, as in the case of Data Encryption Standard (DES), MD5, RC2, and a host of others, there needs to be a mechanism to efficiently replace it in software. History has shown that the cryptographic algorithms we depend upon today will be deprecated in the future. Cryptography can be used to protect confidentiality and integrity of data when at rest, in transit (communication), or even in some cases when being acted upon. This is achieved through careful selection of proper algorithms and proper implementation.

Cryptographic agility is the ability to manage the specifics of cryptographic functions that are embodied in code without recompiling, typically through a configuration file. Most often, this is as simple as switching from an insecure to a more secure algorithm. The challenge is in doing this without replacing the code itself.

Producing cryptographically agile code is not as simple as it seems. The objective is to create software that can be reconfigured on the fly via configuration files. There are a couple of ways of doing this, and they involve using library calls for cryptographic functions. The library calls are then abstracted in a manner by which assignments are managed via a configuration file. This enables the ability to change algorithms via a configuration file change and a program restart.

Cryptographic agility can also assist in the international problem of approved cryptography. In some cases, certain cryptographic algorithms are not permitted to be exported to or used in a particular country. Rather than creating different source-code versions for each country, agility can allow the code to be managed via configurations.

Cryptographic agility functionality is a design-level decision. Once the decision is made with respect to whether cryptographic agility is included or not, then the SDL can build suitable protections based on the design. This is one of the elements that requires an early design decision, as many other

elements are dependent upon it.



EXAM TIP When communications between elements involve sessions—unique communication channels tied to transactions or users—it is important to secure the session to prevent failures that can cascade into unauthorized activity. Session management requires sufficient security provisions to guard against attacks such as brute-force, man-in-the-middle, hijacking, replay, and prediction attacks.

Handling Configuration Parameters

Configuration parameters can change the behavior of an application. Securing configuration parameters is an important issue when configuration can change programmatic behaviors. Managing the security of configuration parameters can be critical. To determine the criticality of configuration parameters, one needs to analyze what application functionality is subject to alteration. The risk can be virtually none for parameters of no significance to extremely high if critical functions such as cryptographic functions can be changed or disabled.

Securing critical data such as configuration files is not a subject to be taken lightly. As in all risk-based security issues, the level of protection should be commensurate with the risk of exposure. When designing configuration setups, it is important to recognize the level of protection needed. The simplest levels include having the file in a directory protected by the access control list (ACL); the extreme end would include encrypting the sensitive data that is stored in the configuration file.

Configuration data can also be passed to an application by a calling application. This can occur in a variety of ways—for example, as part of a URL string or as a direct memory injection—based on information provided by the target application. Testing should explore the use of URLs, cookies, temp files, and other settings to validate the correct handling of configuration data.

Memory Management

Memory management is a crucial aspect of code security. Memory is used to hold the operational code, data, variables, and working space. Memory management is a complex issue because of the dynamic nature of the usage of memory across a single program, multiple programs, and the operating system. The allocation and management of memory are the responsibility of both the operating systems and the application. In managed code applications, the combination of managed code and the intermediate code execution engine takes care of memory management, and type safety makes the tasking easier. Memory management is one of the principal strengths of managed code. Another advantage of managed code is the automatic lifetime control over all resources. Because the code runs in a sandbox environment, the runtime engine maintains control over all resources.

In unmanaged code situations, the responsibility for memory management is shared between the operating system and the application, with the task being even more difficult because of the issues associated with variable type mismatch. In unmanaged code, virtually all operations associated with resources and memory are the responsibility of the developer, including garbage collection, thread pooling, memory overflows, and more. As in all situations, complexity is the enemy of security.

Type-Safe Practice

Type safety is the extent to which a programming language prevents errors resulting from different data types in a program. Type safety can be enforced either statically at compile time or dynamically at runtime to prevent errors. Type safety is linked to memory safety. Type-safe code will not inadvertently access arbitrary locations of memory outside the expected memory range. Type safety defines all variables, and this typing defines the memory lengths. One of the results of this definition is that type-safe programming resolves many memory-related issues automatically.

Locality

Locality is a principle that given a memory reference by a program, subsequent memory accesses are often predictable and are in close proximity to previous references. Buffer overflows are a significant issue associated

with memory management and malicious code. There are various memory attacks that take advantage of the locality principle. There are also defenses against memory corruption based on locality attacks. Address space layout randomization (ASLR) is a specific memory management technique developed by Microsoft to defend against locality attacks.

Error Handling

No application is perfect, and given enough time, they will all experience failure. How an application detects and handles failures is important. Some errors are user driven; some can be unexpected consequences or programmatic errors. The challenge is in how the application responds when an error occurs. This is referred to as *error handling*. The specific coding aspect of error handling is referred to as *exception management*.

Applications will have errors. Proper programming practices are that code should test for and detect a wide range of potential errors and handle these situations. This process of catching errors is referred to as exception management and is a key element in handling data in a program. When the data being handled is incorrect, and can be detected as wrong, this handling of the error close to the source is advantageous in that details can be logged and managed with less ambiguity as to the source or cause of the error.

Interface Coding

Application programming interfaces (APIs) define how software components are connected to and interacted with. Modern software development is done in a modular fashion, using APIs to connect the functionality of the various modules. APIs are significant in that they represent entry points into software. The attack surface analysis and threat model should identify the APIs that could be attacked and the mitigation plans to limit the risk. Third-party APIs that are being included as part of the application should also be examined, and errors or issues should be mitigated as part of the SDL process. Older, weak, and deprecated APIs should be identified and not allowed into the final application.

On all interface inputs into your application, it is important to have the appropriate level of authentication. It is also important to audit the external

interactions for any privileged operations performed via an interface.

Primary Mitigations

There are a set of primary mitigations that have been established over time as proven best practices. As a Certified Secure Software Lifecycle Professional (CSSLP), you should have these standard tools in your toolbox. An understanding of each, along with where and how it can be applied, is essential knowledge for all members of the development team. These will usually be employed through the use of the threat report. The standard best-practice mitigations are as follows:

- Lock down your environment.
- Establish and maintain control over all of your inputs.
- Establish and maintain control over all of your outputs.
- Assume that external components can be subverted and your code can be read by anyone.
- Use libraries and frameworks that make it easier to avoid introducing weaknesses.
- Use industry-accepted security features instead of inventing your own.
- Integrate security into the entire software development lifecycle.
- Use a broad mix of methods to comprehensively find and prevent weaknesses.

Defensive coding is not a black art; it is merely applying the materials detailed in the threat report. Attack surface reduction, an understanding of common coding vulnerabilities, and standard mitigations are the foundational elements of defensive coding. Additional items in the defensive coding toolkit include code analysis, code review, versioning, cryptographic agility, memory management, exception handling, interface coding, and managed code.



EXAM TIP *Concurrency* is the process of two or more threads in a program executing concurrently. Concurrency can be an issue when these threads access a common object, creating a shared object property. Should they change the state of the shared object, the conditions for a race condition apply. Controlling concurrency is one method of controlling for race conditions.

Learning from Past Mistakes

Software engineering is not a new thing. Nor are security issues. One of the best sources of information regarding failures comes from real-world implementation errors in the industry. When company ABC makes the news that it has to remediate a security issue, such as a back door in a product left by the development team, this should be a wake-up call to all teams in all companies.

Errors are going to happen. Mistakes and omissions occur. But repeating problems once they are known is a lot harder to explain to customers and management, especially when these errors are of significant impact and expensive to remediate, both for the software firm and the customer. Learning from others and adding their failures to your own list of failures to avoid is a good business practice.

Part of the role of the security team is keeping the list of security requirements up to date for projects. Examining errors from other companies and updating your own set of security requirements to prevent your firm from falling into known pitfalls will save time and money in the long run.

Secure Design Principles

Implementing secure design principles requires specific design elements to be employed. Using the information from the attack surface analysis and the threat model, designers can pinpoint the places where specific design elements can be employed to achieve the desired security objectives. Security controls can be chosen based on the design to implement the desired levels of protection per security requirements. This can include controls to ensure confidentiality, integrity, and availability.

Good Enough Security

When designing in security aspects of an application, care should be exercised to ensure that the security elements are in response to the actual risk associated with the potential vulnerability. Designing excess security elements is a waste of resources. Under-protecting the application increases the risk. The challenge is in determining the correct level of security functionality. Elements of the threat model and attack surface analysis can provide guidance as to the level of risk. Documenting the level and purpose of security functions will assist the development team in creating an appropriate and desired level of security.

Achieve Application Security with Secure Design Principles

The following are secure design principles that are employed to achieve application security:

- Good enough security
- Least privilege
- Separation of duties
- Defense in depth
- Fail safe
- Economy of mechanism
- Complete mediation
- Open design
- Least common mechanism
- Psychological acceptability
- Weakest link
- Leverage existing components
- Single point of failure

These are security principles that are important to follow when developing any software, because they are always an issue with respect to how risk develops in a system.

Least Privilege

One of the most fundamental approaches to security is least privilege. Least privilege should be utilized whenever possible as a preventative measure. Embracing designs with least privilege creates a natural set of defenses should unexpected errors happen. Least privilege, by definition, is sufficient privilege, and this should be a design standard. Excess privilege represents potentially excess risk when vulnerabilities are exploited.

Separation of Duties

Separation of duties must be designed into a system. Software components can be designed to enforce separation of duties when they require multiple conditions to be met before a task is considered complete. These multiple conditions can then be managed separately to enforce the checks and balances required by the system. In designing the system, designers also impact the method of operation of the system.

As with all other design choices, the details are recorded as part of the threat model. This acts as the communication method between all members of the development effort. Designing in operational elements such as separation of duties still requires additional work to happen through the development process, and the threat model can communicate the expectations of later development activities in this regard.

Defense in Depth

Defense in depth is one of the oldest security principles. If one defense is good, multiple overlapping defenses are better. The threat model document will contain information where overlapping defenses should be implemented. Designing in layers as part of the security architecture can work to mitigate a wider range of threats in a more efficient manner.



NOTE “It’s not that I’m assuming your code will fail. It’s that I’m assuming all code can fail, including mine. You are not a special case.” —

Dan Kaminsky, security guru

Because every piece of software can be compromised or bypassed in some way, it is incumbent on the design team to recognize this and create defenses that can mitigate specific threats. Although all software, including the mitigating defenses, can fail, the end result is to raise the difficulty level and limit the risk associated with individual failures.

Designing a series of layered defense elements across an application provides for an efficient defense. For layers to be effective, they should be dissimilar in nature so that if an adversary makes it past one layer, a separate layer may still be effective in maintaining the system in a secure state. An example is the coupling of encryption and access control methods to provide multiple layers that are diverse in their protection nature and yet both can provide confidentiality.

Fail Safe

As mentioned in the exception management section, all systems will experience failures. The fail-safe design principle refers to the fact that when a system experiences a failure, it should fail to a safe state. When designing elements of an application, one should consider what happens when a particular element fails. When a system enters a failure state, the attributes associated with security, confidentiality, integrity, and availability need to be appropriately maintained.

Failure is something that every system will experience. One of the design elements that should be considered is how the individual failures affect overall operations. Ensuring that the design includes elements to degrade gracefully and return to normal operation through the shortest path assists in maintaining the resilience of the system. For example, if a system cannot complete a connection to a database, when the attempt times out, how does the system react? Does it automatically retry, and if so, is there a mechanism to prevent a lockup when the failure continually repeats?

Economy of Mechanism

The terms *security* and *complexity* are often at odds with each other. This is because the more complex something is, the harder it is to understand, and

you cannot truly secure something if you do not understand it. During the design phase of the project, it is important to emphasize simplicity. Smaller and simpler is easier to secure. Designs that are easy to understand, easy to implement, and well documented will lead to more secure applications.

If an application is going to do a specific type of function—for example, gather standard information from a database—and this function repeats throughout the system, then designing a standard method and reusing it improves system stability. As systems tend to grow and evolve over time, it is important to design in extensibility. Applications should be designed to be simple to troubleshoot, simple to expand, simple to use, and simple to administer.

Complete Mediation

Systems should be designed so that if the authorization system is ever circumvented, the damage is limited to immediate requests. Whenever sensitive operations are to be performed, it is important to perform authorization checks. Assuming that permissions are appropriate is just that—an assumption—and failures can occur. For instance, if a routine is to permit managers to change a key value in a database, then assuming that the party calling the routine is a manager can result in failures. Verifying that the party has the requisite authorization as part of the function is an example of complete mediation. Designing this level of security into a system should be a standard design practice for all applications.

Open Design

Part of the software development process includes multiple parties doing different tasks that in the end create a functioning piece of software. Over time, the software will be updated and improved, which is another round of the development process. For all of this work to be properly coordinated, open communication needs to occur. Having designs that are open and understandable will prevent activities later in the development process that will weaken or bypass security efforts.

Least Common Mechanism

The concept of least common mechanism is constructed to prevent

inadvertent security failures. Designing a system where multiple processes share a common mechanism can lead to a potential information pathway between users or processes. The concepts of least common mechanism and leverage existing components can place a designer at a conflicting crossroad. One concept advocates reuse and the other separation. The choice is a case of determining the correct balance associated with the risk from each.

Take a system where users can access or modify database records based on their user credentials. Having a single interface that handles all requests can lead to inadvertent weaknesses. If reading is considered one level of security and modification of records a more privileged activity, then combining them into a single routine exposes the high-privilege action to a potential low-privilege account. Thus, separating mechanisms based on security levels can be an important design tool.

Psychological Acceptability

Users are a key part of a system and its security. Including a user in the security of a system requires that the security aspects be designed so that they are psychologically acceptable to the user. When a user is presented with a security system that appears to obstruct the user, the result will be the user working around these security aspects. Applications communicate with users all the time. Care and effort needs to be applied to ensure that an application is usable in normal operation. This same idea of usability needs to be extended to security functions. Designers should understand how the application will be used in the enterprise and what users will expect of it.

When users are presented with cryptic messages, such as the ubiquitous “Contact your system administrator” error message, expecting them to do anything that will help resolve an issue is unrealistic. Just as care and consideration are taken to ensure normal operations are comprehensible to the user, so, too, should a similar effort be made for abnormal circumstances.

Weakest Link

Every system, by definition, has a “weakest” link. Adversaries do not seek out the strongest defense to attempt a breach; they seek out any weakness they can exploit. Overall, a system can be considered only as strong as its weakest link. When designing an application, it is important to consider both

the local and system views with respect to weaknesses. Designing a system using the security tenets described in this section will go a long way in preventing local failures from becoming system failures. This limits the effect of the weakest link to local effects.

Leverage Existing Components

Modern software development includes extensive reuse of components. From component libraries to common functions across multiple components, there is significant opportunity to reduce development costs through reuse. This can also simplify a system through the reuse of known elements. The downside of massive reuse is associated with a monoculture environment, which is where a failure has a larger footprint because of all the places where it is involved.

During the design phase, decisions should be made as to the appropriate level of reuse. For some complex functions, such as in cryptography, reuse is the preferred path. In other cases, where the lineage of a component cannot be established, then the risk of use may outweigh the benefit. In addition, the inclusion of previous code, sometimes referred to as *legacy code*, can reduce development efforts and risk.



EXAM TIP The use of legacy code in current projects does not exempt that code from security reviews. All code should receive the same scrutiny, especially legacy code that may have been developed prior to the adoption of SDL processes.

Single Point of Failure

The design of a software system should be such that all points of failure are analyzed and that a single failure does not result in system failure. Examining designs and implementations for single points of failure is important to prevent this form of catastrophic failure from being released in a product or system. Single points of failure can exist for any attribute, confidentiality, integrity, availability, etc., and may well be different for each attribute.

During the design phase, failure scenarios should be examined with an eye for single points of failure that could cascade into an entire system failure.

Interconnectivity

Communication between components implies a pathway that requires securing. Managing the communication channel involves several specific activities. Session management is the management of the communication channel itself on a conversation basis. Exception management is the management of errors and the recording of critical information for troubleshooting efforts. Configuration management is the application of control principles to the actual operational configuration of an application system. These elements work together to manage interconnectivity of the elements that comprise the overall application.

Session Management

Software systems frequently require communications between program elements or between users and program elements. The control of the communication session between these elements is essential to prevent the hijacking of an authorized communication channel by an unauthorized party. Session management is the use of controls to secure a channel on a conversation-by-conversation basis. This allows multiple parties to use the same communication method without interfering with each other or disclosing information across parties.

Designing session management into a system can be as simple as using Hypertext Transfer Protocol Secure (HTTPS) for a communication protocol between components or, when that is not appropriate, replicating the essential elements. Individual communications should be separately encrypted so cross-party disclosures do not occur. The major design consideration is where to employ these methods. Overemployment can make a system unnecessarily complex; underemployment leaves a system open to hijacking.

Exception Management

Exception management is the programmatic response to the occurrence of an exception during the operation of a program. Properly coded for, exceptions

are handled by special functions in code referred to as exception handlers. Exception handlers can be designed to specifically address known exceptions and handle them according to pre-established business rules.

There are some broad classes of exceptions that are routinely trapped and handled by software. Arithmetic overflows are a prime example. Properly coded for, trapped, and handled with business logic, this type of error can be handled inside software itself. Determining appropriate recovery values from arithmetic errors is something that the application is well positioned to do, and something that the operating system is not.

Part of the development of an application should be an examination of the ways in which the application could fail, and also the correct ways to address those failures. This is a means of defensive programming, for if the exceptions are not trapped and handled by the application, they will be handled by the operating system. The operating system (OS) does not have the embedded knowledge necessary to properly handle the exceptions.

No application is perfect, and given enough time, they will all experience failure. How an application detects and handles failures is important. Some errors are user driven; some can be unexpected consequences or programmatic errors. The challenge is in how the application responds when an error occurs. This is referred to as *error handling*. The specific coding aspect of error handling is referred to as *exception management*.

When errors are detected and processed by an application, it is important for the correct processes to be initiated. If logging of critical information is a proper course of action, one must take care not to expose sensitive information such as personally identifiable information (PII) in the log entries. If information is being sent to the screen or terminal, then again, one must take care as to what is displayed. Disclosing paths, locations, passwords, userids, or any of a myriad of other information that would be useful to an adversary should be avoided. To prevent error conditions from cascading or propagating through a system, each function should practice complete error mitigation, including error trapping and complete handling, before returning to the calling routine.

Exceptions are typically not security issues—however, unhandled exceptions can become security issues. If the application properly handles an exception, then ultimately through logging of the condition and later correction by the development team, rare, random issues can be detected and fixed over the course of versions. Exceptions that are unhandled by the

application or left to the OS to handle are the ones where issues such as privilege escalation typically occur.

Configuration Management

Dependable software in production requires the managed configuration of the functional connectivity associated with today's complex, integrated systems. Initialization parameters, connection strings, paths, keys, and other associated variables are typical examples of configuration items. The identification and management of these elements are part of the security process associated with a system.

During the design phase, the configuration aspect of the application needs to be considered from a security point of view. Although configuration files are well understood and have been utilized by many systems, consideration needs to be given to what would happen if an adversary was able to modify these files. Securing the configuration of a system by securing these files is a simple and essential element in design.

Cryptographic Failures

Failures in the application of cryptography can result in failed protection for data and programs. Several attacks fall into this category: hard-coded credentials, missing encryption of sensitive data, use of a broken or risky cryptographic algorithm, download of code without integrity check, and use of a one-way hash without a salt. Using industry-accepted cryptographic libraries and not creating your own will assist in avoiding this type of failure. Ensuring cryptography is used both properly and from approved libraries is a necessity to avoid common cryptographic failures. Even with strong cryptography, hard-coded credentials that are reverse-engineered out of software result in complete failure of the otherwise-secure algorithm and subsequent failure of protection.

Hard-Coded Credentials

Hard-coding passwords, keys, and other sensitive data into programs has several serious drawbacks. First, it makes them difficult to change. Yes, a program update can change them, but this is a messy way of managing secret

data. But most importantly, they will not stay secret. With some simple techniques, hackers can reverse-engineer code and, through a series of analysis steps, determine the location and value of the secret key. This has happened to some large firms with serious consequences in a very public forum. This is easy to check for during code walk-throughs and should never be allowed in code.

Missing Encryption of Sensitive Data

This may seem to be a simple issue—how can one miss encrypting sensitive information?—yet it happens all the time. There are several causes, the first being ignorance on the part of the development team. Some items are obviously sensitive, but some may not be so obvious. The data owner is responsible for documenting the sensitivity of data and its protection requirements. When this step fails, it is hard to blame the development team.

Other cases of missing protection can also arise, typically as part of program operations. Are backups protected? Are log files protected? Backups and log files are two common places that secrets can become exposed if not protected. Error-reporting mechanisms can also handle sensitive data, and again, if not encrypted, is it exposed to risk of loss? The answer to all of these questions is yes, and many an enterprise has learned the hard way after the loss occurs that a simple encryption step would have prevented a breach and subsequent notification actions.



EXAM TIP To maintain the security of sensitive data, a common practice is *tokenization*. Tokenization is the replacement of sensitive data with data that has no external connection to the sensitive data. In the case of a credit card transaction, for example, the credit card number and expiration date are considered sensitive and are not to be stored, so restaurants typically print only the last few digits with XXXXs for the rest, creating a token for the data, but not disclosing the data.

Use of a Broken or Risky Cryptographic Algorithm

Cryptography is one of the more difficult technical challenges of modern times. Despite a lot of effort, there are surprisingly few secure cryptographic algorithms. The rise of computing power has caused many of the older algorithms to fail under massive number-crunching attacks, attacks that used to take significant resources but are managed today on a desktop. Data Encryption Standard (DES), the gold standard for decades, is now considered obsolete, as are many other common cryptographic functions.

Even worse is when a development team decides to create their own encryption methodology. This has been tried by many teams and always ends up with the system being exploited as the algorithm is broken by hackers. This forces a redesign/re-engineering effort after the software is deployed, which is an expensive solution to a problem that should never have occurred in the first place. The solution is simple—always use approved cryptographic libraries.



EXAM TIP Only approved cryptographic libraries should be used for encryption. In addition, attention must be paid to algorithms and key lengths. At the time of writing, RSA keys should be greater than 2,048 bits.

A common mode of cryptographic failure revolves around the random number function. The pseudo-random function that is built into most libraries may appear random and have statistically random properties, but it is not sufficiently random for cryptographic use. Cryptographically sufficient random number functions are available in approved cryptographic libraries and should be used for all cryptographic random calculations.

Hash functions have been falling to a series of attacks. MD-5 and SHA-1 are no longer considered secure. Others will continue to fall, which has led to the SHA-3 series being developed by the National Institute of Standards and Technology (NIST). Until the new hash functions are deployed, SHA-256, SHA-384, and SHA-512 are still available, with the number signifying the bit length of the digest. This brings up a design consideration. Even if the current design is to use SHA-256, it would be wise when planning data structures to plan for longer hash values, up to 512 bits, so that if the SHA function needs to be upgraded in the future, then the data structure will support the upgrade.

Download of Code Without Integrity Check

The Internet has become the medium of choice for distributing software, updates, data, and most digital content. This raises a series of concerns; how does one know the digital content is correct and from the correct source?

There are known instances of malware being attached to downloaded code and then being passed off as legitimate. Hash values can be used to verify the integrity of a file that is being downloaded. For reasons of integrity, whether to guard against malicious code or just accidental errors that will later affect production, all downloaded code should have its integrity verified before installation and use.

This requires designing in a checking mechanism, as integrity codes will need to be made available and a mechanism to verify them established. Simply attaching the hash values to the download is not sufficient, as this mechanism can be replicated by hackers who can recompute hash values after modifying an update. The hash values need to be made available in a manner that lets the user know they are from a valid source.

Some download methods, such as Adobe Update and Windows Update, perform the hash check automatically as part of the update process. Using the vendor's update methodology can help quite a bit, but verify before trusting. Contact the vendor and verify the safeguards are in place before trusting automatic update mechanisms.

Use of a One-Way Hash Without a Salt

Hashing is a common function used to secure data, such as passwords, from exposure to unauthorized parties. As hash values are impossible to reverse, the only solution is to try all possible inputs and look for a matching hash value. Hash tags worked well until the creation of rainbow tables. Rainbow tables exist for all possible combinations of passwords up to 14 characters, making the hash value a simple lookup field to get the original password from the table. The solution to this is simply using a technique called *salting the hash*. A salt value is concatenated to the password, or other value being hashed, effectively increasing its length beyond that of a rainbow table.

Salting a hash also solves a second problem. If the salt also contains an element from the username, then the issue of identical passwords between different accounts will no longer yield the same hash value. If two items have the same hash, the inputs are considered to be identical. By increasing the

length of the input with a salt value, you solve rainbow table lookups. By making part of the salt specific to a user ID, you solve the issue of identical passwords being shown by identical hash values.

Input Validation Failures

Probably the most important defensive mechanism that can be employed is input validation. Considering all inputs to be hostile until properly validated can mitigate many attacks based on common vulnerabilities. This is a challenge, as the validation efforts need to occur after all parsers have completed manipulating input streams, a common function in web-based applications using Unicode and other international character sets.

Input validation is especially well suited for the following vulnerabilities: buffer overflow, reliance on untrusted inputs in a security decision, cross-site scripting (XSS), cross-site request forgery (CSRF), path traversal, and incorrect calculation of buffer size.

Input validation may seem suitable for various injection attacks, but given the complexity of the input and ramifications from legal but improper input streams, this method falls short for most injection attacks. What can work is a form of recognition and whitelisting approach, where the input is validated and then parsed into a standard structure that is then executed. This restricts the attack surface to not only legal, but also expected, inputs.

Output validation is just as important in many cases as input validations. If querying a database for a username and password match, the expected forms of the output of the match function should be either one match or none. If using record count to indicate the level of match, a common practice, then a value other than 0 or 1 would be an error. Defensive coding using output validation would not act on values greater than 1, as these are clearly an error and should be treated as a failure.

Buffer Overflow

The most famous of all input validation failures is the incorrect calculation of buffer size, or buffer overflow attack. This attack comes when the input data is larger than the memory space allocated, overwriting other crucial elements. If there's one item that could be labeled as the "Most Wanted" in coding security, it would be the buffer overflow. The Computer Emergency

Response Team Coordination Center (CERT/CC) at Carnegie Mellon University estimates that nearly half of all exploits of computer programs stem historically from some form of buffer overflow. Finding a vaccine to buffer overflows would stamp out 50 percent of these security-related incidents by type and probably 90 percent by volume. The Morris finger worm in 1988 was an exploit of an overflow, as were recent big-name events such as Code Red and Slammer. The generic classification of buffer overflows includes many variants, such as static buffer overruns, indexing errors, format string bugs, Unicode and ANSI buffer size mismatches, and heap overruns.

The concept behind these vulnerabilities is relatively simple. The input buffer that is used to hold program input is overwritten with data that is larger than the buffer can hold. The root cause of this vulnerability is a mixture of two things: poor programming practice and programming language weaknesses. Programming languages such as C were designed for space and performance constraints. Many functions in C, like `gets()`, are unsafe in that they will permit unsafe operations, such as unbounded string manipulation into fixed buffer locations. The C language also permits direct memory access via pointers, a functionality that provides a lot of programming power but carries with it the burden of proper safeguards being provided by the programmer.

The first line of defense is to write solid code. Regardless of the language used or the source of outside input, prudent programming practice is to treat all input from outside a function as hostile. Validate all inputs as if they were hostile and an attempt to force a buffer overflow. Accept the notion that although during development, everyone may be on the same team, be conscientious, and be compliant with design rules, future maintainers may not be as robust. Designing prevention into functions is a foundational defense against this type of vulnerability.

There is good news in the buffer overflow category—significant attention has been paid to this type of vulnerability, and although it is the largest contributor to past vulnerabilities, its presence is significantly reduced in newly discovered vulnerabilities.

Canonical Form

In today's computing environment, a wide range of character sets is used.

Unicode allows multilanguage support. Character code sets allow multilanguage capability. Various encoding schemes, such as hex encoding, are supported to allow diverse inputs. The net result of all these input methods is that there are numerous ways to create the same input to a program. Canonicalization is the process by which application programs manipulate strings to a base form, creating a foundational representation of the input. The definition of canonical form is the simplest or standard form. Input can be encoded for a variety of reasons, sometimes for transport, sometimes to deal with legacy or older system compatibility, sometimes because of other protocols involved.

Character Encoding

Characters can be encoded in ASCII, Unicode, hex, UTF-8, or even combinations of these. So if the attacker desires to obfuscate his response, several things can happen. By URL hex encoding URL strings, it may be possible to circumvent filter security systems and IDs. The following

`http://www.myweb.com/cgi?file=/etc/passwd`

can become

`http://www.myweb.com/cgi?
file=%2F%65%74%63%2F%70%61%73%73%77%64`

Double encoding can complicate the matter even further. This Round 1 decoding

`scripts/..%255c./winnt`

becomes

`scripts/..%5c./winnt (%25 = "%" Character)`

This Round 2 decoding

`scripts/..%5c./winnt`

becomes

```
scripts/...\\winnt
```

The bottom line is simple: Know that encoding can be used, and plan for it when designing input verification mechanisms. Expect encoded transmissions to be used to attempt to bypass security mechanisms. Watch out for unique encoding schemes, such as language-dependent character sets that can have similar characters, bypassing security checking but parsing into something hostile.

Canonicalization errors arise from the fact that inputs to a web application may be processed by multiple applications, such as the web server, application server, and database server, each with its own parsers to resolve appropriate canonicalization issues. Where this is an issue relates to the form of the input string at the time of error checking. If the error checking routine occurs prior to resolution to the canonical form, then issues may be missed. The string representing `/...`, used in directory traversal attacks, can be obscured by encoding and, hence, missed by a character string match before an application parser manipulates it to canonical form. The bottom line is simple: Input streams may not be what they seem.

A Rose Is a Rose Is a R%6fse

All of the following can be equivalent filenames in Microsoft Windows:

- C:\\test\\Longfilename.dat
- ...\\Longfilename.dat
- Longfi~1.dat
- Longfilename.dat::\$DATA

Names are resolved to canonical form before use.

Missing Defense Functions

Common defense mechanisms such as authentication and authorization can only be effective when they are invoked as part of a protection scheme. Ensuring that the appropriate defensive mechanisms are employed on any

activity that crosses a trust boundary will mitigate many common attacks. This is effective against vulnerabilities such as missing authentication for critical functions, missing authorization, unrestricted upload of file with dangerous type, incorrect authorization, incorrect permission assignment for critical resource, execution with unnecessary privileges, improper restriction of excessive authentication attempts, URL redirection to untrusted site (“open redirect”), and uncontrolled format string. Ensuring that the basics of security such as authentication and authorization are uniformly applied across an application is essential to good practice. Having a ticket to a football game may get you into the stadium, but to get to the good seats, one must show their ticket again. Multiple checks aligned with the importance of the asset under protection are simply applying the fundamentals of security.

Output Validation Failures

The output from one routine becomes the input for some other system. Just as input validation is important, output validation is important. Ever seen a Visa card receipt on the Internet for 100 times the cost of dinner and the horror story about fixing it? It’s the same for electric bills, water bills, etc. With many customers now directly connecting their accounts, an error in one system can propagate into others. The solution is simple, especially for customer-facing systems. Have a sanity check before sending the output. If your system is normally billing in less than 100-dollar amounts, then 10 times or 100 times amounts should be flagged and not automatically sent to the next system. This will resolve mistakes that were not caught in some input system and prevent errors further down the line.

General Programming Failures

Programming is treated by many as an art, when it has progressed far from that form. Today’s modern programming is a complex engineering-type evolution with rules and guidelines to prevent failures. The use of a style guide that restricts certain functions for safety and security reasons is seen as handcuffs by many, but also as prudent by professionals. For each dangerous function, there is a manner by which it can be tamed, typically by substitution of a safe version. Buffer overflows due to functions that do not validate input size are a common example of such dangerous functions. An example is

`strcpy()` in the C/C++ language. This function does not validate input length, leaving it up to the programmer to manage independently. The companion function `strncpy()` does the check, and although it takes longer, it still takes less time than separate validation. This is just one example of the use of potentially dangerous functions, one of the Top 25. Another source of programming errors is the inclusion of old code or code obtained from another source. Without running these source code elements through the same software development lifecycle (SDLC) processes, one is stuck with any potential and unchecked vulnerabilities in the code. The inclusion of functionality from untrusted control sphere error is just this, using code that has not been validated. Although we may choose to ignore our own legacy code inside the enterprise, many a major defect has come from older code bases and direct adoption, even of internally generated code.

All source code should be tested using static test tools that can screen code for a wide variety of issues. From examination for obsolete or disallowed libraries and functions, to common weakness patterns, to errors like off by one or failure to properly initialize, the list of vulnerabilities a static code scanner can find is long. Ensuring that these common errors are cleared prior to each build is an essential mitigation step.

Sequencing and Timing

In today's multithreaded, concurrent operating model, it is possible for different systems to attempt to interact with the same object at the same time. It is also possible for events to occur out of sequence based on timing differences between different threads of a program. Sequence and timing issues such as race conditions and infinite loops influence both design and implementation of data activities. Understanding how and where these conditions can occur is important to members of the development team. In technical terms, what develops is known as a *race condition*, or from the attack point of view, a system is vulnerable to a time of check/time of use (TOC/TOU) attack.



EXAM TIP A TOC/TOU attack is one that takes advantage of the

separation between the time a program checks a value and when it uses the value, allowing an unauthorized manipulation that can affect the outcome of a process.

Race conditions are software flaws that arise from different threads or processes with a dependence on an object or resource that affects another thread or process. A classic race condition is when one thread depends on a value A from another function that is actively being changed by a separate process. The first process cannot complete its work until the second process changes the value of A. If the second function is waiting for the first function to finish, a lock is created by the two processes and their interdependence. These conditions can be difficult to predict and find. Multiple unsynchronized threads, sometimes across multiple systems, create complex logic loops for seemingly simple atomic functions. Understanding and managing record locks is an essential element in a modern, diverse object programming environment.

Race conditions are defined by race windows, a period of opportunity when concurrent threads can compete in attempting to alter the same object. The first step to avoid race conditions is to identify the race windows. Then, once the windows are identified, the system can be designed so that they are not called concurrently, a process known as *mutual exclusion*.

Another timing issue is the infinite loop. When program logic becomes complex—for instance, date processing for leap years—care should be taken to ensure that all conditions are covered and that error and other loop-breaking mechanisms do not allow the program to enter a state where the loop controls will fail. Failure to manage this exact property resulted in Microsoft Zune devices failing if they were turned on across the New Year following a leap year. The control logic entered a sequence where a loop would not be satisfied, resulting in the device crashing by entering an infinite loop and becoming nonresponsive.



EXAM TIP Complex conditional logic with unhandled states, even if rare or unexpected, can result in infinite loops. It is imperative that all conditions in each nested loop be handled in a positive fashion.

Technology Solutions

The processor industry has had its share of security moments with issues such as the Spectre and Meltdown vulnerabilities in Intel CPUs, chipset flaws, and similar flaws on AMD chips. Not as well-known is the series of technological innovations that the processor industry has pursued including microarchitecture security extensions (e.g., Software Guard Extensions [SGX], Advanced Micro Devices [AMD] Secure Memory Encryption [SME]/Secure Encrypted Virtualization [SEV], and ARM TrustZone). Intel's Software Guard Extensions (Intel SGX) provide hardware-based memory encryption to isolate specific application code and data in memory. It also allows user-level code to allocate private regions of memory, called *enclaves*, which are designed to be protected from processes running at higher privilege levels. These are security features that can be used to prevent a wide range of attacks against critical code and can be built in at the time of coding. AMD SME also provides for encrypting memory locations, protecting code and data to the originating process only. These protections add complexity and reduce efficiency, but the impacts are very manageable. While not a technique that would typically be employed on every project, critical functions that operate at privilege, these options greatly enhance security and reduce risks.

Chapter Review

This chapter opened with an analysis of the differences between declarative and programmatic security. An examination of bootstrapping, cryptographic agility, and secure handling of configuration parameters followed. Memory management and the related issues of type-safe practices and locality were presented. Error handling, including exception management, was presented as an important element in defensive coding. The security implications of the interface coding associated with APIs were presented. The chapter closed with an examination of the primary mitigations that are used in defensive coding.

Quick Tips

- Declarative security refers to defining security relations with respect to

the container.

- Programmatic security is where the security implementation is embedded into the code itself.
- Cryptographic agility is the ability to manage the specifics of cryptographic function that are embodied in code without recompiling, typically through a configuration file.
- Securing configuration parameters is an important issue when configuration can change programmatic behaviors.
- Memory management is a crucial aspect of code security.
- In managed code applications, the combination of managed code and the intermediate code execution engine takes care of memory management, and type safety makes the tasking easier.
- In unmanaged code situations, the responsibility for memory management is shared between the operating system and the application, with the task being even more difficult because of the issues associated with variable type mismatch.
- Type-safe code will not inadvertently access arbitrary locations of memory outside the expected memory range.
- Locality is a principle that, given a memory reference by a program, subsequent memory accesses are often predictable and are in close proximity to previous references.
- Exception management is the programmatic response to the occurrence of an exception during the operation of a program.
- APIs are significant in that they represent entry points into software.
- A set of primary mitigations have been established over time as proven best practices.
- Technological solutions such as Intel's Software Guard Extensions (SGX), Advanced Micro Devices (AMD) Secure Memory Encryption (SME)/Secure Encrypted Virtualization (SEV), and ARM TrustZone can provide hardware-based protection of data and code in memory, using encryption to tie the memory to the originating process and stopping attackers from interjecting into the code stream at execution.

Questions

To further help you prepare for the CSSLP exam, answer the following questions. The correct answers are provided in the following section.

1. To prevent error conditions from propagating through a system, what should each function do?
 - A. Log all abnormal conditions
 - B. Include error trapping and handling
 - C. Clear all global variables upon completion
 - D. Notify users of errors before continuing
2. Race conditions can be determined and controlled via what?
 - A. Multithreading
 - B. Mutual exclusion
 - C. Race windows
 - D. Atomic actions
3. To deal with international distribution issues associated with cryptography, a clean method is via what?
 - A. Versioning
 - B. Use of international cryptography for all versions
 - C. Use of approved cryptographic libraries only
 - D. Use of cryptographic agility
4. Address space layout randomization (ASLR) is a specific memory management technique used to defend against what?
 - A. Locality attacks
 - B. Stack overflows
 - C. Cryptographically agile attacks
 - D. Buffer overflows
5. What mechanism can be employed to handle data issues that lead to out-of-range calculations?
 - A. Regular expressions
 - B. Vetted functions

- C. Library calls
 - D. Exception handling
6. All of the following are concerns of memory management except what?
- A. Garbage collection
 - B. Buffer overflows
 - C. Memory allocation
 - D. Exception management
7. What do type-safe practices enable?
- A. Memory management safety
 - B. Smaller programs
 - C. Avoidance of exceptions in managed code
 - D. Easier-to-maintain code
8. Elements of defensive coding include all of the following except what?
- A. Custom cryptographic functions to avoid algorithm disclosure
 - B. Exception handling to avoid program termination
 - C. Interface coding efforts to avoid API-facing attacks
 - D. Cryptographic agility to make cryptographic functions stronger
9. The HasRows() function of the following can lead to what type of failure in use?

```
string query = String.Format( "SELECT COUNT(*) FROM Users WHERE " +
    "username='{0}' AND password='{1}'", txtUser.Text, txtPassword.Text );  
  
SqlCommand cmd = new SqlCommand(query, con);
conn.Open();
SqlDataReader reader = cmd.ExecuteReader();
try{
    if(reader.HasRows())
        IssueAuthenticationTicket();
    else
        TryAgain();
}
```

- A. Defense-in-depth failure

- B. Security through obscurity
 - C. Output validation
 - D. Reliance on untrusted inputs for a security decision
- 10. When validating input before use, one needs to be aware of the following issues except what?
 - A. Session management
 - B. Encoding methods
 - C. Canonical form
 - D. Language-based character sets

Answers

- 1. **B.** To prevent error conditions from cascading or propagating through a system, each function should practice complete error mitigation, including error trapping and complete handling, before returning to the calling routine.
- 2. **C.** Race conditions are defined by race windows, a period of opportunity when concurrent threads can compete in attempting to alter the same object. They are caused by multithreading and are resolved through atomic actions under mutual exclusion conditions. The key is in detecting when they occur.
- 3. **D.** Cryptographic agility can also assist in the international problem of approved cryptography. In some cases, certain cryptographic algorithms are not permitted to be exported to or used in a particular country. Rather than creating different source-code versions for each country, agility can allow the code to be managed via configurations.
- 4. **A.** Address space layout randomization (ASLR) is a specific memory management technique used to randomize memory locations by breaking up the locality problem.
- 5. **D.** Exception handling can use business logic when elements are out of bounds, forcing overflow-type exceptions, for instance.
- 6. **D.** Exception management is not directly related to memory management.

- 7.** **A.** Type-safe languages have greatly reduced memory management issues due to the removal of one of the memory management challenges.
- 8.** **A.** Custom cryptographic functions are always a bad idea and frequently lead to failure.
- 9.** **C.** Output validation assures that a function's output matches expectations before use as an input in another function or decision. It would be better to look for a single row, as multiple rows indicate a failure.
- 10.** **A.** Session management may be important in an application, but when it comes to input validation, canonical form, character-encoding schemes, and language-specific character sets are more important.

Analyze Code for Security Risks

In this chapter you will

- Examine secure code reuse
 - Explore vulnerability databases/lists including Open Web Application Security Project Top 10 and the Common Weakness Enumeration
 - Learn about static application security testing
 - Learn about dynamic application security testing
 - Explore manual code review
 - Learn to look for malicious code
 - Examine interactive application security testing
 - Learn about runtime application security protection
 - Explore common vulnerabilities and countermeasures
-

Designing an application is the beginning of implementing security into the final application. Using the information uncovered in the requirements phase, designers create the blueprint developers use to arrive at the final product. It is during this phase that the foundational elements to build the proper security functionality into the application are initiated. To determine which security elements are needed in the application, designers can use the information from the attack surface analysis and the threat model to determine the “what” and “where” elements. Knowledge of secure design principles can provide the “how” elements. Using this information in a comprehensive plan can provide developers with a targeted foundation that will greatly assist in creating a secure application.

Code Analysis (Static and Dynamic)

Code analysis is a term used to describe the processes to inspect code for weaknesses and vulnerabilities. It can be divided into two forms: static and dynamic. *Static analysis* involves examining the code without executing it. *Dynamic analysis* involves the execution of the code as part of the testing. Both static and dynamic analyses are typically done with tools, which are much better at the detailed analysis steps needed for any but the smallest code samples.

Code analysis can be performed at virtually any level of development, from unit level to subsystem to system to complete application. The higher the level, the greater the test space and more complex the analysis. When the analysis is done by teams of humans reading the code, typically at the smaller unit level, it is referred to as *code reviews*. Code analysis should be done at every level of development, because the sooner that weaknesses and vulnerabilities are discovered, the easier they are to fix. Issues found in design are cheaper to fix than those found in coding, which are cheaper than those found in final testing, and all of these are cheaper than fixing errors once the software has been deployed.

Static Application Security Testing

Static application security testing (SAST), or *static code analysis*, is when code is examined for security issues without being executed. This analysis can be performed on both source and object code bases. The term *source code* is typically used to designate the high-level language code, although technically source code is the original code base in any form, from high language to machine code. Static analysis can be performed by humans or tools, with humans limited to the high-level language, while tools can be used against virtually any form of code base.

Static code analysis is frequently performed using automated tools. These tools are given a variety of names but are commonly called *source code analyzers*. Sometimes, extra phrases, such as binary scanners or byte code scanners, are used to differentiate the tools. Static tools use a variety of mechanisms to search for weaknesses and vulnerabilities. Automated tools can provide advantages when checking syntax, verifying function/library calls are from approved sources, and examining rules and semantics associated with logic and calls. They can catch elements a human might overlook.

One of the original tools for analyzing C code was Lint. Lint led to the development of testing apparatus for virtually every language, and as a reference to their heritage, these tools are called *lint tools*. The word *linting* is used to describe using these tools to detect issues.

Static testing has some specific advantages including the following:

- Automating static testing results in higher code coverage than manual audits.
- Static testing can be integrated into the existing development environment and performed at multiple points of the software development cycle.
- Static testing can be customized to local rulesets, enforcing local environmental rules such as naming conventions and function restrictions.
- Modern static testing tools can find highly complex vulnerabilities during the first stages of development, which can be resolved quickly.
- Results from static testing are specific to code lines, making remediation faster.

Static testing isn't the perfect solution in the world of testing for security. The following are some common drawbacks or limitations associated with static code testing:

- Static testing doesn't test the application in the real environment. Vulnerabilities associated with application or conditional logic as well as insecure configuration are not covered in the test and lead to missed items.
- Deploying the technology at scale can be a challenge and may have issues with the development environment.
- Static testing is not easy to employ in code where one does not have the source, and most companies are not willing to provide data for source code analysis.
- Developers must deal with many false positives leading to hesitancy in using the technology.
- Dynamically typed languages can pose challenges; static code tools

need to semantically process a myriad of code elements including those that might be written in different programming languages.

Dynamic Application Security Testing

Dynamic application security testing (DAST), or *dynamic analysis*, is performed while the software is executed, on either a target or emulated system. The system is fed specific test inputs designed to produce specific forms of behaviors. Dynamic analysis can be particularly important on systems such as embedded systems, where a high degree of operational autonomy is expected. As a case in point, the failure to perform adequate testing of software on the Ariane rocket program led to the loss of an Ariane V booster during takeoff. Subsequent analysis showed that if proper testing had been performed, the error conditions could have been detected and corrected without the loss of the flight vehicle.

Dynamic analysis requires specialized automation to perform specific testing. There are dynamic test suites designed to monitor operations for programs that have high degrees of parallel functions. There are thread-checking routines to ensure multicore processors and software are managing threads correctly. There are programs designed to detect race conditions and memory addressing errors.

Dynamic application security testing analyzes the code by executing the application, and this leads to several distinct advantages over static testing:

- Dynamic analysis can be less expensive and less complex to implement when compared to static testing.
- Because it operates with full application knowledge, it leads to less false positives.
- It can support a variety of languages in an integrated development environment as it is operating on functioning code, not reading the code base.
- Dynamic analysis allows the identification of runtime issues, such as race conditions as well as conditions that result from the interaction with the system environment, such as authentication and authorization issues.

Like static analysis, dynamic analysis is not a complete solution and

suffers from some limitations such as the following:

- Dynamic tools have no access to company or internal coding standards and implementation of them, thus missing elements such as forbidden functions.
 - Dynamic tools can have difficulty pinpointing the exact location of an error, as function stacks and reference calls can obscure the actual code issue location.
 - Dynamic analysis relies upon functioning code, forcing its use later into the development cycle and postponing error remediation to later stages.
-



EXAM TIP It is important to be able to differentiate static and dynamic testing based on what each covers. Each has advantages, so understand how to pick the correct one for a situation. In practice, both are used, but a question with a situational context will favor one or the other. Be able to differentiate based on context of the situation.

Interactive Application Security Testing

Interactive application security testing (IAST) uses instrumentation to assess the performance conditions of an application and detect vulnerabilities. The use of sensors that have access to the code base, the data flow, the control flow, and environmental conditions such as configuration and integration with the operating system is part of the integrated development environment instantiation of IAST. This level of integration leads to a broader coverage of the application. These are some of the advantages of IAST:

- Integration into the IDE enables earlier detection of vulnerabilities.
- Because of the detailed information available to the sensors, IAST can pinpoint sources of error accurately.
- Because of the integration in the IDE, IAST can be part of agile development and the continuous integration/continuous deployment

(CI/CD) pipelines.

IAST does have limitations including the following:

- IAST tools can be more costly to implement and operate.
- IAST tools can affect the speed of operation of the application in the IDE, missing issues that can be a result of timing.
- As IAST is relatively new, the technology is not as mature as SAST and DAST, leading to areas of poor coverage.

Runtime Application Self-Protection

Runtime application self-protection (RASP) is the latest technology in the escalation of application security versus hackers battlespace. Rather than being a testing environment strategy, RASP is deployed when the application is run in real time. RASP uses runtime instrumentation that is built into the application to provide alerts or security actions in the event of unexpected or known bad operational behaviors. One can think of RASP as a monitor that watches the application and the environment and acts upon conditions that are outside the normal conditions for the application. RASP can respond to unanticipated inputs, preventing their ingestion by the application. RASP can also add significant complexity to the application and the environment and thus is generally reserved for only critical applications.



NOTE SAST and DAST are employed during code building and are testing platforms used as part of the development process. IAST acts at the testing phase of the development cycle. RASP does not examine the code or application itself; instead, it acts as a monitor during operation or production.

Code/Peer Review

Code reviews are a team-based activity where members of the development team inspect code. The premise behind peer-based code review is simple.

Many eyes can discover what one does not see. This concept is not without flaws, however, and humans have limited abilities to parse into multilayer obfuscated code. But herein lies the rub—the objective of most programming efforts is to produce clean, highly legible code that works now, but also, when it is updated later, the new developer can understand what is happening, how it works, and how to modify it appropriately. Code walk-throughs can be used to promote maintainability and sustainability of the code base. This makes the primary mission of code review both finding potential weaknesses or vulnerabilities and assisting developers in the production of clean, understandable code.

The process of the review is simple. The author of the code explains to the team, step-by-step, line-by-line, how the code works. The rest of the team can look for errors that each has experienced in the past and observe coding style, level of comments, etc. Having to present your code to the team and actually explain how it works leads developers to make cleaner, more defendable code to the group. This then has the benefits of the code being more maintainable in the long run. By explaining how it works, this also helps others on the team understand how it works and provides for backups if a developer leaves the team and someone else is arbitrarily assigned to modify the code.

Code walk-throughs are ideal times for checking for and ensuring mitigation against certain types of errors. Lists of common defects, such as the SANS Top 25 and the OWASP Top 10, can be checked. The list of previous errors experienced by the firm can be checked, because if it happened once, it is best not to repeat those issues. Unauthorized code elements, including Easter eggs and logic bombs, are much harder to include in code if the entire team sees all the code. [Table 9-1](#) lists some errors and how they can be caught with walk-throughs.

Error Mechanism	Review Tasking
Inefficient code	If code is obfuscated or overly complex, simplify it.
SANS Top 25 OWASP Top 10 Previously discovered errors	All code should be reviewed to specifically prevent recurrence of previous errors. Common error patterns from SANS and OWASP lists should be checked as well.
Errors and exception handling	All functions, all procedures, and all components should fully check and handle exceptions.
Injection flaws	Input validation checks.
Cryptographic calls	Approved libraries, good random numbers.
Unsafe and deprecated function calls	Use only approved functions.
Privilege levels	Least privilege violations.
Logging	Ensure proper logging of errors and conditions.
Secure key information	Handling of keys, PII, and other sensitive data.

Table 9-1 Issues for Code Reviews

Another advantage of code reviews is in the development of junior members of the development team. Code walk-throughs can be educational, both to the presenter and to those in attendance. Members of the team automatically become familiar with aspects of a project that they are not directly involved in coding, so if they are ever assigned a maintenance task, the total code base belongs to the entire team, not different pieces to different coders. Treating the review as a team event, with learning and in a nonhostile manner, produces a stronger development team as well.

Code Review Objectives

Code review has a multitude of objectives. At the most basic level it is a form of syntax and style checking. This can remove certain classes of errors. It can also enforce business rules with respect to programming style, identifying illegal function calls and lack of required elements such as error checking.

Code review can find areas of high entropy, indicating things like stored secrets in code bases (think: hard-coded passwords) or in memory where secrets can be vulnerable if left beyond their immediate use. These are all tedious things that humans can check for, but given the size of code bases, human manual checking just doesn't scale.

Code review can also identify dead code sections or malicious code sections. If backdoors or logic bombs are hidden in the code base, automated testing tools are one of the better ways to detect and find them before they enter production.

Additional Sources of Vulnerability Information

Code development has had the issues of errors, bugs, and vulnerabilities since the first code was designed and implemented decades ago. With this long history, one would think there is a master list of all the vulnerabilities that can be created by developers. And to a degree there is, but it's not one but several lists. The first and most important list should be one that every software team has, which is an internal list of errors that have been created by the team in the past. This list is important for several reasons. First, errors are not random events; they are caused by coding patterns, and if a developer has a particular form or habit in how they create code or if it creates a vulnerability at one time, it is likely to happen again. Habits are hard to change. So, checking for repeat issues is easy and wise. Repeat errors are also frustrating to the entire development chain, all the way to the customer.

Already mentioned are the SANS Top 25 list and the OWASP Top 10 list. Both of these lists cover the vast majority of common errors, as the distribution of errors is nonlinear, and after these lists, the remaining issues tend to be very rare. The ultimate lists are the MITRE common weakness enumeration (CWE) and common vulnerability enumerations (CVEs), although these lists are very long and not sorted by frequency of occurrence.

- **SANS Top 25 programming errors** <https://www.sans.org/top25-software-errors/>
- **OWASP Top 10 list** <https://owasp.org/www-project-top-ten/>
- **MITRE CWE** <https://cwe.mitre.org/>

- **MITRE CVE** <https://cve.mitre.org/>

CWE/SANS Top 25 Vulnerability Categories

Begun by MITRE and supported by the U.S. Department of Homeland Security, the CWE/SANS Top 25 list is the result of collaboration between many top software security experts worldwide. This list represents the most widespread and critical errors that can lead to serious vulnerabilities in software. They are often easy to find and easy to exploit. Left unmitigated, they are easy targets for attackers and can result in widespread damage to software, data, and even enterprise security.

The Top 25 list can be used in many ways. It is useful as a tool for development teams to provide education and awareness about the kinds of vulnerabilities that plague the software industry. The list can be used in software procurement as a specification of elements that need to be mitigated in purchased software. Although the list has not been updated since 2011, it is still highly relevant. One could argue over the relative position on the list, but at the end of the day, all the common vulnerabilities that can be exploited need to be fixed.

The Top 25 list can serve many roles in the secure development process. For programmers, the list can be used as a checklist of reminders, as a source for a custom “Top N” list that incorporates internal historical data. The data can also be used to create a master list of mitigations, which, when applied, will reduce occurrence and severity of the vulnerabilities. Testers can use the list to build a test suite that can be used to ensure that the issues identified are tested for before shipping.

CWE/SANS Top 25—2011 (Current)

1. CWE-89 SQL Injection
2. CWE-78 OS Command Injection
3. CWE-120 Buffer Overflow
4. CWE-79 Cross-Site Scripting (XSS)

5. CWE-306 Missing Authentication for Critical Function
6. CWE-862 Missing Authorization
7. CWE-798 Hard-Coded Credentials
8. CWE-311 Missing Encryption of Sensitive Data
9. CWE-434 Unrestricted Upload of File with Dangerous Type
10. CWE-807 Reliance on Untrusted Inputs in a Security Decision
11. CWE-250 Execution with Unnecessary Privileges
12. CWE-352 Cross-Site Request Forgery (CSRF)
13. CWE-22 Path Traversal
14. CWE-494 Download of Code Without Integrity Check
15. CWE-863 Incorrect Authorization
16. CWE-829 Inclusion of Functionality from Untrusted Control Sphere
17. CWE-732 Incorrect Permission Assignment for Critical Resource
18. CWE-676 Use of Potentially Dangerous Function
19. CWE-327 Use of a Broken or Risky Cryptographic Algorithm
20. CWE-131 Incorrect Calculation of Buffer Size
21. CWE-307 Improper Restriction of Excessive Authentication Attempts
22. CWE-601 URL Redirection to Untrusted Site (“Open Redirect”)
23. CWE-134 Uncontrolled Format String
24. CWE-190 Integer Overflow or Wraparound
25. CWE-759 Use of a One-Way Hash Without a Salt

OWASP Vulnerability Categories

The Open Web Application Security Project (OWASP) is an open

community dedicated to finding and fighting the causes of insecure web application software. All of the OWASP tools, documents, forums, and chapters are free and open to anyone interested in improving web application security, and they are available at www.owasp.org.

OWASP has published several significant publications associated with building more secure web applications. Its main treatise, “A Guide to Building Secure Web Applications and Web Services,” provides detailed information on a wide range of vulnerabilities and how to avoid them. Another commonly used item from OWASP is its Top 10 list of web application vulnerabilities.

OWASP Top 10—2013 (Current)

- A1 – Injection
- A2 – Broken Authentication and Session Management
- A3 – Cross-Site Scripting (XSS)
- A4 – Insecure Direct Object References
- A5 – Security Misconfiguration
- A6 – Sensitive Data Exposure
- A7 – Missing Function-Level Access Control
- A8 – Cross-Site Request Forgery (CSRF)
- A9 – Using Known Vulnerable Components
- A10 – Unvalidated Redirects and Forwards

Common Vulnerabilities and Countermeasures

The list of Top 25 and the list of Top 10 web application vulnerabilities overlap. All of the Top 10 items are in the Top 25. This is not unexpected, as web application programming is a subset of programming as a whole discipline. To examine the best countermeasure strategy, it is easier to group the vulnerabilities into like causes and apply countermeasures that address

several specific issues at once.

Injection Attacks

Injection attacks are some of the most common and severe that are currently being seen in software. These attacks include SQL injection, OS command injection, integer overflow or wraparound, path traversal, cross-site scripting (XSS), and cross-site request forgery (CSRF). Injection-type attacks can also be used against Lightweight Directory Access Protocol (LDAP), Extensible Markup Language (XML), and other common protocols.

Injection attacks can be difficult to decode on the fly. As in many cases, the inputs go through a series of parsers that change the form of the input before use. In these cases, it is better to have previously approved lists of options and let the user select the option based on a master list as opposed to being defined by input streams. Using user input in any direct fashion can result in unintended behaviors when malicious users enter code specifically designed to cause problems. Cleansing or correcting user input streams is difficult, if not impossible, in some situations, and the prudent course is to never allow users to directly define elements of programmatic behavior.

SQL Injections

Databases are one of the primary methods used to store data, especially large quantities of user data. Access to and manipulation of the data is done using Structured Query Language (SQL) statements. The SQL injection attack is performed by an attacker inputting a specific string to manipulate the SQL statement to do something other than that intended by the programmer or designer. This is a form of improper input validation that results in unintended behavior. The defense is easy, but it requires that the SQL statements be constructed in a manner that protects them from manipulation as a result of user input.

The best method to avoid SQL injection is to design database SQL access in a manner that does not allow the SQL statements to be manipulated by users. The safest method is to use stored procedures for all access, with user input being in the form of variables to the stored procedure. The stored procedure can validate the input and ensure that the SQL statements are not manipulated. Another method is to use parameterized queries.

The primary mitigation for SQL injection is developer awareness. SQL

injection vulnerabilities can be designed out of a project, and where exposure is unavoidable, input validation can greatly mitigate the issue. SQL injection can be easily tested for and caught as part of a normal test cycle. Failure to catch inputs susceptible to SQL injection is a testing failure. As with all known common attack vectors, SQL injection should be included in test plans.

SQL Injection Attack Methodology

The SQL injection attack has several steps:

1. Test input values to see if SQL is accessible and can be manipulated.
2. Experiment with SQL inputs, using error messages to enumerate the database and provide exploitation information.
3. Craft a SQL exploit input to achieve the exploitation goal.

Even if the SQL errors are suppressed, a structured form of attack referred to as *blind SQL injection* can use Boolean-based SQL statements rather effectively.

SQL Injection Example

Situation: A web form has a login input consisting of a username and password. The web form uses this information to query a database to determine whether the information is correct.

The attacker assumes that the SQL statement for logging in has the following form:

```
SELECT * FROM tblUsers WHERE username = '<inputfield1>' AND password = '<inputfield2>'
```

For the username, the attacker enters **admin' --**.

For the password, the user enters **A**.

These values are then crafted into the SQL statement, producing the

following:

```
SELECT * FROM tblUsers WHERE username = 'admin' --' AND  
password = 'A'
```

The key element in the attack is the double dash. In SQL, this tells the parser that the remainder of the line is a comment. This changes the SQL statement to the following:

```
SELECT * FROM tblUsers WHERE username = 'admin'
```

The password field and any other elements after the first field are ignored. While this may not work in all cases, there are numerous variations that can be used to manipulate the SQL statement.

Command Injections

A command injection attack is similar to the SQL injection attack, but rather than trying to influence a dynamic SQL input, the target is a dynamic command-generation element. When a program needs to perform a function that is normally handled by the operating system, it is common practice to use the operating system to perform the action. To craft the specific operation, it is common to use some form of user input in the command to be executed. Using user-supplied input that is not properly validated can lead to serious consequences.



NOTE The ; command separator that allows a second command and the use of the Linux mail command to send a file.

There are two common methods of using user-supplied input in command injection attacks. The first is where the end-user input is used as an argument in the command to be executed. This can have several interesting consequences, from actions on unintended files to additional commands that are appended to the arguments. The second form of this attack is where the user input includes the command to be executed. This can be even riskier, as

unvalidated or improperly validated input strings can result in disastrous consequences.



NOTE Here's an example of a command injection attack: when a program asks for an input file (file1) and you want to get a copy of file2 from the system, you get it via e-mail! Just use the following input:

```
file1.txt;mail tester@test.com < <insert absolute or relative path>/file2.txt.
```

The primary mitigation for command injection vulnerability is developer awareness. Command injection vulnerabilities can be designed out of a project, and where exposure is unavoidable, input validation can greatly mitigate the issue. Command injection can be easily tested for and caught as part of a normal test cycle. Failure to catch inputs susceptible to command injection is a testing failure. As with all known common attack vectors, command injection should be included in test plans.

Integer Overflow

Computer programs store numbers in variables of a defined size. For integers, these can be 8, 16, 32, and 64 bits, and in either signed or unsigned forms. This restricts the size of numbers that can be stored in the variable. When a value is larger than allowed, a variety of errors can ensue. In some cases, the values simply wrap around; in others, it just sticks as the maximum value. These can be processor and language dependent. In many cases, including the C language, overflows can result in undefined behavior.



NOTE A 32-bit integer can be either signed or unsigned. A 32-bit unsigned integer can hold numbers from 0 to 4,294,967,295, while a 32-bit signed integer holds numbers between -2,147,483,648 and 2,147,483,647.

Integer overflows can occur in the course of arithmetic operations. Using a web application that dispenses licenses to users, we can see how this can be manipulated. Once the user enters the application, there are three values: the number of licenses, a place to enter the number desired, and the number of remaining licenses. Assuming the program uses 32-bit signed variables and that user input checks verify that all the inputs are the correct size, how can there be an overflow? Let N = number of licenses held, R = the number requested, and B = the balance after R is dispensed. After verifying that R is a legitimate unsigned 32 int, the program performs the following: $B = N - R$. The intent is to check to see if B is < 0 , which would indicate that sufficient licenses did not exist and disallow that transaction. But if the value of $N - R$ does not fit in an int32, then the calculation will overflow, as the internal operation is to calculate $N - R$, put the value in a register, and then move to the location of B . The calculation of $N - R$ is the problem.

Overflows can be resolved in a variety of language-specific methods. The use of the checked directive in C#, for instance, turns on exception handling that allows for the trapping and management of overflows before the problem is exposed. Integer overflows can be specifically tested for, using both boundary values and values that will force internal errors as described earlier. These cases need to be designed and built into the test plan as part of the regular test plan development.

Path Traversal

Known by several names, including dot-dot-slash, directory traversal, directory climbing, and backtracking attacks, the path traversal attack attempts to access files and directories that are stored outside the web root folder. By using “`..`” notation in the path to a file, it is possible to traverse across the directory structure to access a specific file in a specific location. This file system navigation methodology takes advantage of the way that the system is designed. To mask the “`..`” characters in the input stream, the characters can be encoded, i.e., `%2e%2e%2f`.

Virtually every web application has a need for local resources, image file scripts, configurations, etc. To prevent a directory traversal attack, the key is to not use user input when accessing a local resource. Although it may require additional coding, matching the user input to a specific resource and then using a hard-coded path and resource to prevent the attack is the strongest defense.

Cross-Site Scripting

Cross-site scripting (XSS) is one of the most common web attack methodologies. The cause of the vulnerability is weak user input validation. The attack works because a user includes a script in their input and this script is not mitigated, but instead is rendered as part of the web process. There are several different types of XSS attacks, which are distinguished by the effect of the script.

A nonpersistent XSS attack is one where the injected script is not persisted or stored, but rather is immediately executed and passed back via the web server. A persistent XSS attack is one where the script is permanently stored on the web server or some back-end storage. This allows the script to be used against others who log in to the system. A document object model (DOM)-based XSS attack is one where the script is executed in the browser via the DOM process as opposed to the web server.

Cross-site scripting attacks can result in a wide range of consequences, and in some cases, the list can be anything that a clever scripter can devise. Common uses that have been seen in the wild include the following:

- Theft of authentication information from a web application
- Session hijacking
- Deploy hostile content
- Change user settings, including future users
- Impersonate a user
- Phish or steal sensitive information

Controls to defend against XSS attacks include the use of anti-XSS libraries to strip scripts from the input sequences. There are a variety of other mitigating factors, including limiting types of uploads and screening size of uploads, whitelisting inputs, etc., but attempting to remove scripts from inputs can be a tricky task. Well-designed anti-XSS input library functions have proven to be the best defense.

Cross-site scripting vulnerabilities are easily tested for and should be part of the test plan for every application. Testing a variety of encoded and unencoded inputs for scripting vulnerability is an essential test element.

Cross-Site Request Forgery

Cross-site request forgery attacks utilize unintended behaviors that are proper in defined use but are performed under circumstances outside the authorized use. This is an example of a confused deputy problem, a class of problems where one entity mistakenly performs an action on behalf of another. A CSRF attack relies upon several conditions to be effective. It is performed against sites that have an authenticated user, and it exploits the site's trust in a previous authentication event. Then, by tricking a user's browser to send an HTTP request to the target site, the trust is exploited. Assume your bank allows you to log in and perform financial transactions but does not validate the authentication for each subsequent transaction. If a user is logged in and has not closed their browser, then an action in another browser tab could send a hidden request to the bank resulting in a transaction that appears to be authorized, but in fact was not done by the user.

There are many different mitigation techniques that can be employed, from limiting authentication times to cookie expiration to managing some specific elements of a web page like header checking. The strongest method is the use of random CSRF tokens in form submissions. Subsequent requests cannot work, as the token was not set in advance. Testing for CSRF takes a bit more planning than other injection-type attacks, but this, too, can be accomplished as part of the design process.

Chapter Review

In this chapter, we examined the core concepts of code analysis and testing. Beginning with static testing and dynamic testing and then expanding to interactive testing and RASP, the methods employed to thoroughly test software as part of the development process were covered. The chapter also examined the methods of peer review and the objectives of the code review process. The chapter closed with an examination of the sources of vulnerability/error information.

Quick Tips

- Code should be inspected during development for weaknesses and vulnerabilities.
- Static code analysis is performed without executing the code.

- Dynamic code analysis involves examining the code under production conditions.
- Code walk-throughs are team events designed to find errors using human-led inspection of source code.
- Interactive application security testing examines the code during operation in the IDE as part of the development process.
- Runtime application self-protection is the set of technologies that can be employed during the production operation of code. RASP works with the built-in instrumentation in the application to alert on abnormal operational conditions that might indicate improper activity.
- Previous errors should always be screened for in future development to prevent repeat error occurrences.
- Information sources for common errors include SANS Top 25 list, OWASP Top 10 list, and the MITRE CWE and CVE projects.

Questions

To further help you prepare for the CSSLP exam, answer the following questions. The correct answers are provided in the following section.

1. Verifying that code can perform in a particular manner under production conditions is a task managed by what?
 - Static analysis
 - Dynamic analysis
 - Production testing
 - Code walk-throughs
2. At what level should code analysis be performed?
 - Unit level
 - Subsystem level
 - System level
 - All levels
3. Static analysis can be used to check for what?
 - Approved function/library calls, examining rules and semantics

- associated with logic, and thread performance management
- B. Syntax, approved function/library calls, and race conditions
 - C. Syntax, approved function/library calls, and memory management
 - D. Syntax, approved function/library calls, and examining rules and semantics associated with logic and calls
4. DAST has the following advantages over manual code review except what?
- A. Detection of unsafe or deprecated function calls
 - B. Support of a variety of languages
 - C. Speed of analysis
 - D. Integration into the IDE
5. RASP does not perform which of the following functions?
- A. Connect to instrumentation during program operation
 - B. Monitor environmental condition under which the application operates
 - C. Examine code for race conditions
 - D. Alert operators when code has malfunctions
6. Which of the following is the best method of finding race conditions?
- A. Code walk-throughs
 - B. SAST
 - C. IAST
 - D. DAST
7. What is the most important source of error information to employ when checking code?
- A. Previous errors in the code base(s)
 - B. SANS Top 25 list of programming errors
 - C. OWASP Top 10 list of application errors
 - D. MITRE CWE database
8. Advantages of static code analysis include all of the following except?
- A. It can be automated.

- B. It can cover large code bases for more issues more thoroughly than manual code review.
 - C. It can identify complex issues early in the development cycle.
 - D. Dynamically typed languages can pose challenges in implementation.
9. False positives are an issue with which testing methodology?
- A. Static code testing
 - B. Interactive application security testing
 - C. Dynamic code testing
 - D. All of the above
10. Which testing methodology can improve maintainability of the code base?
- A. Code walk-throughs
 - B. Static application security testing (SAST)
 - C. Dynamic application security testing (DAST)
 - D. Runtime application self-protection (RASP)

Answers

- 1. B. Testing of code while executing in a production-like environment is referred to as dynamic analysis.
- 2. D. Code analysis should be performed at all levels of development.
- 3. D. Static code analysis cannot test runtime issues such as thread performance, memory management, or race conditions.
- 4. A. DAST analysis can be challenged to identify unsafe or deprecated function calls. This is better done with code walk-throughs.
- 5. C. RASP does not examine the code base.
- 6. D. Race conditions are best found using DAST as they happen when code executes.
- 7. A. Because repeat errors are common, and very damaging, the list of previous problems is critical to prevent repeat errors.

- 8.** **D.** Dynamically typed languages can pose challenges; static code tools need to semantically process a myriad of code elements including those that might be written in different programming languages.
- 9.** **D.** All testing methods have false positives and require analysis before accepting the results.
- 10.** **A.** Code walk-throughs can be used to promote maintainability and sustainability of the code base.

CHAPTER 10

Implement Security Controls

In this chapter you will

- Learn methods of addressing security risks
 - Implement security controls
 - Apply security during the build process
 - Use anti-tampering techniques
 - Securely integrate components
 - Securely reuse third-party code or libraries
 - Understand the significance of systems-of-systems integration
-
-

Security controls are the measures taken to detect, prevent, and/or mitigate risks associated with a threat against a specific target such as an application or the information flow. Software is a collection of routines to perform a specific algorithm, changing inputs to outputs in a specific manner. Security controls are extra elements that are added to help ensure that the code does what it is supposed to do and only what it is supposed to do, regardless of any environmental interferences.

Security Risks

Once risks are identified, management has a series of options to deal with risks. The first option is *remediation*, or simply to fix the problem. Fixing the problem involves understanding the true cause of the vulnerability and correcting the issue that led to it. When available, remediation is the preferred option, as it solves the problem irrespective of external changes. The second method involves removing the risk element from the problem. This is referred to as *mitigation*, because although the event happens, the risk associated with it is attenuated or blocked. This can be done in a couple of

ways: If the problem is associated with a particular feature, then removing the feature may remove the problem. If the problem is associated with a particular standard—that is, confidentiality associated with cleartext protocols—removing the communication or protocol may not make sense and fixing the protocol is probably not possible; what remains is some form of compensating control. Adding encryption to the communication channel and removing the cleartext disclosure issue is a form of removing the problem. In both of these cases, there is an opportunity for some form of residual risk, and for that, management needs to make a similar decision. Can we live with the residual risk? If the answer is no, then you need to repeat the corrective process, removing additional levels of risk from the residual risk.

Other options exist for dealing with the risk. *Transferring* the risk to another party is an option. This can be in the form of a warning to a user, shifting the risk to the user or to a third party. In the case of financial fraud associated with online credit cards, the merchant is typically protected, and the bank card issuer ends up covering the fraud loss. This cost is ultimately borne by the customer, but only after aggregating and distributing the cost across all customers in the form of increased interest rates. It is also possible in some cases to purchase insurance to cover risk, in essence, shifting the financial impact to a third party.

The last option is to do nothing. This is, in essence, *accepting* the risk and the consequences associated with the impact should the risk materialize. This is a perfectly viable option, but it is best only when it is selected on purpose with an understanding of the ramifications. Ignoring risk also leads to this result, but it does so without the comprehensive understanding of the potential costs.



EXAM TIP There are four options when dealing with risks: remediation, mitigation, transfer, and accept.

Implement Security Controls

There are some basic security controls that are common to most software

packages. Watchdogs are routines that perform a specific check to ensure something is actually operating as designed and not stuck in a loop or fault condition. An example of a watchdog method is the time to live (TTL) component of the network protocol. As each hop happens, the TTL counter is reduced by one; if it gets to zero, it tells the system that something has gone wrong and to discard the packet. Another example is a heartbeat component, which is a signal that repeats on a regular basis to signal a process is still functioning. Sequence numbers are another example; a sequence number is a key value that increments in a pattern to tell the system whether things are okay. One can build sequence watchdogs that ensure entire sequences are actually happening, or they trigger an alert, indicating abnormal conditions.

Another common security control is a mechanism such as file integrity monitoring (FIM). If a program is going to load data, such as a configuration file, then having a data value that indicates the file has not been tampered with can be important. If the hash value or checksum of the file is stored in another location, then upon loading the veracity of that file can be checked prior to relying upon its contents. Legitimate updates to the file are done in a manner that also updates the checksum or hash value. One of the common mantras of secure coding is to validate all inputs and doing this for critical files is no different.

Applying Security via the Build Environment

Creating software in a modern development environment is a multistep process. Once the source code is created, it must still be compiled, linked, tested, packaged (including signing), and distributed. There is typically a tool or set of tools for each of these tasks. Building software involves partially applying these tools with the correct options set to create the correct outputs. Options on elements such as compilers are important, because the options can determine what tests and error checks are performed during the compiling process.

Organizations employing a secure development lifecycle (SDL) process will have clearly defined processes and procedures to ensure the correct tools are used and used with the correct settings. Using these built-in protections can go a long way toward ensuring that the code being produced does not

have issues that should have been caught during development.



EXAM TIP Compilers can have flag options, such as Microsoft's /GS compiler switch, which enables stack overflow protection in the form of a cookie to be checked at the end of the function, prior to the use of the return address. Using these options can enhance code security by eliminating common stack overflow conditions.

Determining the correct set of tools and settings is not a simple task. Language dependencies and legacy issues make these choices difficult, and yet these are essential steps if one is to fully employ the capabilities of these tools. Microsoft's SDL guidelines have required settings for compilers, linkers, and code analysis tools. Enabling these options will result in more work earlier in the process but will reduce the potential for errors later in the development process, where remediation is more time-consuming and expensive.

In addition to the actual tools used for building, there is an opportunity to define safe libraries. Approved libraries of cryptographic and other difficult tasks can make function call errors a lesser possibility. Create a library of safe function calls for common problem functions such as buffer overflows, cross-site scripting (XSS), and injection attacks. Examples of these libraries are the Open Web Application Security Project (OWASP) Enterprise Security API project and the Microsoft Anti-Cross Site Scripting Library for .NET.

Integrated Development Environment

Automated tools can be built into the integrated development environment (IDE), making it easy for the developer to do both forms of static and dynamic checking automatically. Integrated development environments have come a long way in their quest to improve workflow and developer productivity. The current version of Microsoft's Visual Studio integrates from requirements to data design to coding and testing, all on a single team-based platform that offers integrated task management, workflow, code analysis, and bug tracking.

A wide array of IDEs exists for different platforms and languages, with varying capabilities. Using automation such as a modern IDE is an essential part of an SDL, because it eliminates a whole range of simple errors and allows tracking of significant metrics. Although using an advanced IDE means a learning curve for the development team, this curve is short compared to the time that is saved with the team using the tool. Each daily build and the number of issues prevented early due to more efficient work results in saved time that would be lost to rework and repair after issues are found, either later in testing or in the field.

Anti-tampering Techniques

Anti-tampering techniques are employed to make it more difficult for an attacker to modify code and pass it off as genuine. There are a couple of principal methods used. Code signing makes it difficult if not impossible to alter code without the recipient being able to tell it has been altered. Obfuscation is a set of techniques employed to make it difficult to understand how to actually alter the code effectively.

An important factor in ensuring that software is genuine and has not been altered is a method of testing the software integrity. With software being updated across the Web, how can one be sure that the code received is genuine and has not been tampered with? The answer comes from the application of digital signatures to the code, a process known as *code signing*.

Code Signing

Code signing involves applying a digital signature to code, which provides a mechanism where the end user can verify the code integrity. In addition to verifying the integrity of the code, digital signatures provide evidence as to the source of the software. Code signing rests upon the established public key infrastructure. To use code signing, a developer will need a key pair. For this key to be recognized by the end user, it needs to be signed by a recognized certificate authority.

Automatic update services, such as Microsoft's Windows Update service, use code signing technologies to ensure that updates are applied only if they are proper in content and source. This technology is built into the update application, requiring no specific interaction from the end user to ensure

authenticity or integrity of the updates.



EXAM TIP Code signing provides a means of authenticating the source and integrity of code. It cannot ensure that code is free of defects or bugs.

Steps to Code Signing

Here are the steps to implement code signing:

1. The code author uses a one-way hash of the code to produce a digest.
2. The digest is encrypted with the signer's private key.
3. The code and the signed digest are transmitted to end users.
4. The end user produces a digest of the code using the same hash function as the code author.
5. The end user decrypts the signed digest with the signer's public key.
6. If the two digests match, the code is authenticated and integrity is assured.

Code signing should be used for all software distribution and is essential when the code is distributed via the Web. End users should not update or install software without some means of verifying the proof of origin and the integrity of the code being installed. Code signing will not guarantee that the code is defect free; it demonstrates only that the code has not been altered since it was signed, and it identifies the source of the code.

Configuration Management: Source Code and Versioning

Development of computer code is not a simple “write it and be done” task. Modern applications take significant time to build all the pieces and assemble a completely functioning product. The individual pieces all go through a series of separate builds or versions. Some programming shops do daily builds slowly, building a stable code base from stable parts. Managing the versions and changes associated with all these individual pieces is referred to as *version control*. Sometimes also referred to as *revision control*, its objective is to uniquely mark and manage each individually different release. This is typically done with numbers or combinations of numbers and letters, with the numbers to the left of the decimal point indicating major releases, and the numbers on the right indicating the level of change relative to the major release.

As projects grow in size and complexity, a version control system, capable of tracking all the pieces and enabling complete management, is needed. Suppose you need to go back two minor versions on a config file—which one is it, how do you integrate it into the build stream, and how do you manage the variants? These are all questions asked by the management team and handled by the version control system. The version control system can also manage access to source files, locking sections of code so that only one developer at a time can check out and modify pieces of code. This prevents two different developers from overwriting each other’s work in a seamless fashion. This can also be done by allowing multiple edits and then performing a version merge of the changes, although this can create issues if collisions are not properly managed by the development team.

Configuration management and version control operations are highly detailed, with lots of record keeping. Management of this level of detail is best done with an automated system that removes human error from the operational loop. The level of detail across the breadth of a development team makes automation the only way in which this can be done in an efficient and effective manner. A wide range of software options is available to a development team to manage this information. Once a specific product is chosen, it can be integrated into the SDL process to make its use a nearly transparent operation from the development team’s perspective.

Code Obfuscation

For code to operate in many instances, the executable code is exposed to the

external world. When you run code on a machine, the executable code can be copied and loaded into a series of debugging programs that will help an analyst determine what the code is doing. This can reveal a whole host of secrets, including elements such as encryption keys, algorithms, and more. Once an attacker has these details, in many cases, they can change bytes of the executable code to change the function of the program, bypassing protection measures, etc., to prevent this type of attack. Programmers can use *obfuscators*, programs that take the operational code and rearrange it to make the decompiling process more difficult to understand. There are many techniques and methods that can be employed, and this is currently an arms race between those trying to protect the details of how their code functions and those trying to break the code apart and understand it. Sensitive code routines and data that have to be on the client side should take these measures to protect code when the risk equation warrants this level of protection. While obfuscation is not perfect, it makes the bar higher for the attacker, and if different methods are used on subsequent releases, then each upgrade forces the attackers to begin from ground zero.

Defensive Coding Techniques

Secure code is more than just code that is free of vulnerabilities and defects. Developing code that will withstand attacks requires additional items, such as defensive coding practices. Adding a series of controls designed to enable the software to operate properly even when conditions change or attacks occur is part of writing secure code. This section will examine the principles behind defensive coding practices.

Declarative vs. Programmatic Security

Security can be instantiated in two different ways in code: in the container itself or in the content of the container. *Declarative programming* is when programming specifies the “what,” but not the “how,” with respect to the tasks to be accomplished. An example is SQL, where the “what” is described and the SQL engine manages the “how.” Thus, declarative security refers to defining security relations with respect to the container. Using a container-based approach to instantiating security creates a solution that is more flexible, with security rules that are configured as part of the deployment and

not the code itself. Security is managed by the operational personnel, not the development team.

Imperative programming, also called *programmatic security*, is the opposite case, where the security implementation is embedded into the code itself. This can enable a much greater granularity in the approach to security. This type of fine-grained security, under programmatic control, can be used to enforce complex business rules that would not be possible under an all-or-nothing container-based approach. This is an advantage for specific conditions, but it tends to make code less portable or reusable because of the specific business logic that is built into the program.

The choice of declarative or imperative security functions, or even a mix of both, is a design-level decision. Once the system is designed with a particular methodology, then the SDL can build suitable protections based on the design. This is one of the elements that requires an early design decision, as many other elements are dependent upon it.

Bootstrapping

Bootstrapping refers to the self-sustaining startup process that occurs when a computer starts or a program is initiated. When a computer system is started, an orchestrated set of activities is begun that includes power-on self-test (POST) routines, boot loaders, and operating system initialization activities. Securing a startup sequence is a challenge—malicious software is known to interrupt the bootstrapping process and insert its own hooks into the operating system.

When coding an application that relies upon system elements, such as environment variables like PATH, care must be taken to ensure that values are not being changed outside the control of the application. Using configuration files to manage startup elements and keeping them under application control can help in securing the startup and operational aspects of the application.

Cryptographic Agility

Cryptography is a complex issue, and one that changes over time as weaknesses in algorithms are discovered. When an algorithm is known to have failed, as in the case of the Data Encryption Standard (DES), MD5,

RC2, and a host of others, there needs to be a mechanism to efficiently replace it in software. History has shown that the cryptographic algorithms we depend upon today will be deprecated in the future. Cryptography can be used to protect the confidentiality and integrity of data at rest, in transit (communication), or even in some cases when being acted upon. This is achieved through careful selection of proper algorithms and proper implementation.

Cryptographic agility is the ability to manage the specifics of cryptographic function that are embodied in code without recompiling, typically through a configuration file. Most often, this is as simple as switching from an insecure to a more secure algorithm. The challenge is in doing this without replacing the code itself.

Producing cryptographically agile code is not as simple as it seems. The objective is to create software that can be reconfigured on the fly via configuration files. There are a couple of ways of doing this, and they involve using library calls for cryptographic functions. The library calls are then abstracted in a manner by which assignments are managed via a configuration file. This enables the ability to change algorithms via a configuration file change and a program restart.

Cryptographic agility can also assist in the international problem of approved cryptography. In some cases, certain cryptographic algorithms are not permitted to be exported to or used in a particular country. Rather than creating different source-code versions for each country, agility can allow the code to be managed via configurations.

Cryptographic agility functionality is a design-level decision. Once the decision is made with respect to whether cryptographic agility is included or not, then the SDL can build suitable protections based on the design. This is one of the elements that requires an early design decision, as many other elements are dependent upon it.



EXAM TIP When communications between elements involve sessions—unique communication channels tied to transactions or users—it is important to secure the session to prevent failures that can cascade into unauthorized

activity. Session management requires sufficient security provisions to guard against attacks such as brute-force, man-in-the-middle, hijacking, replay, and prediction attacks.

Handling Configuration Parameters

Configuration parameters can change the behavior of an application. Securing configuration parameters is an important issue when configuration can change programmatic behaviors. Managing the security of configuration parameters can be critical. To determine the criticality of configuration parameters, one needs to analyze what application functionality is subject to alteration. The risk can be virtually none for parameters of no significance to extremely high if critical functions such as cryptographic functions can be changed or disabled.

Securing critical data such as configuration files is not a subject to be taken lightly. As in all risk-based security issues, the level of protection should be commensurate with the risk of exposure. When designing configuration setups, it is important to recognize the level of protection needed. The simplest levels include having the file in a directory protected by the access control list (ACL); the extreme end would include encrypting the sensitive data that is stored in the configuration file.

Configuration data can also be passed to an application by a calling application. This can occur in a variety of ways—for example, as part of a URL string or as a direct memory injection—based on information provided by the target application. Testing should explore the use of uniform resource locators (URLs), cookies, temp files, and other settings to validate the correct handling of configuration data.

Interface Coding

Application programming interfaces (APIs) define how software components are connected to and interacted with. Modern software development is done in a modular fashion, using APIs to connect the functionality of the various modules. APIs are significant in that they represent entry points into software. The attack surface analysis and threat model should identify the APIs that could be attacked and the mitigation plans to limit the risk. Third-party APIs that are being included as part of the application should also be

examined, and errors or issues should be mitigated as part of the SDL process. Older, weak, and deprecated APIs should be identified and not allowed into the final application.

On all interface inputs into your application, it is important to have the appropriate level of authentication. It is also important to audit the external interactions for any privileged operations performed via an interface.

Memory Management

Memory management is a crucial aspect of code security. Memory is used to hold the operational code, data, variables, and working space. Memory management is a complex issue because of the dynamic nature of using memory across a single program, multiple programs, and the operating system. Allocating and managing memory are the responsibilities of both the operating systems and the application. In managed code applications, the combination of managed code and the intermediate code execution engine takes care of memory management, and type safety makes the tasking easier. Memory management is one of the principal strengths of managed code. Another advantage of managed code is the automatic lifetime control over all resources. Because the code runs in a sandbox environment, the runtime engine maintains control over all resources.

In unmanaged code situations, the responsibility for memory management is shared between the operating system and the application, with the task being even more difficult because of the issues associated with variable type mismatch. In unmanaged code, virtually all operations associated with resources and memory are the responsibility of the developer, including garbage collection, thread pooling, memory overflows, and more. As in all situations, complexity is the enemy of security.

Type-Safe Practice

Type safety is the extent to which a programming language prevents errors resulting from different data types in a program. Type safety can be enforced either statically at compile time or dynamically at runtime to prevent errors. Type safety is linked to memory safety. Type-safe code will not inadvertently access arbitrary locations of memory outside the expected memory range. Type safety defines all variables, and this typing defines the memory lengths. One of the results of this definition is that type-safe programming resolves

many memory-related issues automatically.

Locality

Locality is a principle that given a memory reference by a program, subsequent memory accesses are often predictable and are in close proximity to previous references. Buffer overflows are a significant issue associated with memory management and malicious code. There are various memory attacks that take advantage of the locality principle. There are also defenses against memory corruption based on locality attacks. *Address space layout randomization* (ASLR) is a specific memory management technique developed by Microsoft to defend against locality attacks.

Primary Mitigations

There are a set of primary mitigations that have been established over time as proven best practices. As a Certified Secure Software Lifecycle Professional (CSSLP), you should have these standard tools in your toolbox. An understanding of each, along with where and how it can be applied, is essential knowledge for all members of the development team. These will usually be employed through the use of the threat report. The standard best practice-based primary mitigations are as follows:

- Lock down your environment.
- Establish and maintain control over all of your inputs.
- Establish and maintain control over all of your outputs.
- Assume that external components can be subverted and your code can be read by anyone.
- Use libraries and frameworks that make it easier to avoid introducing weaknesses.
- Use industry-accepted security features instead of inventing your own.
- Integrate security into the entire software development lifecycle.
- Use a broad mix of methods to comprehensively find and prevent weaknesses.

Defensive coding is not a black art; it is merely applying the materials

detailed in the threat report. Attack surface reduction, an understanding of common coding vulnerabilities, and standard mitigations are the foundational elements of defensive coding. Additional items in the defensive coding toolkit include code analysis, code review, versioning, cryptographic agility, memory management, exception handling, interface coding, and managed code.



EXAM TIP This was covered in [Chapter 8](#), but it bears repeating here—to maintain the security of sensitive data, a common practice is *tokenization*. Tokenization is the replacement of sensitive data with data that has no external connection to the sensitive data. In the case of a credit card transaction, for example, the credit card number and expiration date are considered sensitive and are not to be stored, so restaurants typically print only the last few digits, with XXXXs for the rest, creating a token for the data while not disclosing the data.

Secure Integration of Components

A modern enterprise is never going to be singular in its architectural form. Information technology (IT) systems have grown over time through an accretive process where the new “system” is designed to meet requirements and then joins the other systems in the enterprise. Cross-integration between architectures allows data reuse and significantly increases the overall utility of the enterprise architecture as a whole. As new services and opportunities are presented to the IT enterprise, the need to fully integrate, as opposed to rebuilding existing data services, is both a cost- and risk-reducing proposition. Going forward, enterprise accretion will continue, with the addition of new capability and the retirement of no longer used or needed capabilities. This activity will include many initiatives including software acquisition, adoption of open source solutions, and custom code generation. All of these activities must occur under the umbrella of a secure process to ensure risks are appropriately accounted for in the enterprise. Learning of an untenable risk after deploying a solution is a costly endeavor, and taking

steps to understand and manage risk early in the process is essential.

Secure Reuse of Third-Party Code or Libraries

Software is typically comprised of a series of modules working together to perform a specific business function. The individual modules are specific function-limited elements to assist in constructing and implementing the overall project. Gone are the days of writing one long program from start to finish; those monstrosities proved to be virtually impossible to efficiently maintain and modify over time. Modular development also opened the door for module reuse and the inclusion of outside code into a project. Why write a sorting algorithm every time you need something sorted? When you decompose a system into its functions, such as encryption, decryption, hashing, and others, you end up with a list of modules. Whether you code these modules yourself, or use ones you have used before, the result is the set of modules. Reusing previous modules may require minor tweaking to fit the current system, but that is still faster and more efficient than doing it fresh from scratch every time. And more secure as well, as bugs have ideally been eliminated from the modules that are being reused.

Maintaining a library of reuse modules and all of the associated information for each is important if one is going to rely upon these elements. The open-source movement has resulted in an explosion of available code bases that can be reused, albeit one with its own set of risks and troubles. Software composition analysis (SCA) is the set of methodologies employed to manage the information associated with third-party libraries and code segments used in a project. There are SCA tools that can manage automated scans of a code base, including all open-source components, their license compliance data, and any known security vulnerabilities, and they even extend this activity into related artifacts such as containers, registries, imported libraries, and other related elements of the project. By providing visibility into the third-party elements, including open-source elements, the SCA tools offer information to manage external code vulnerabilities and risk.

System-of-Systems Integration

Software integration includes connecting different subsystems into a whole functioning element. This system-of-systems point of view also provides a

means of specifying security provisions between these subsystems. Anytime data travels from one system to another, it crosses a trust boundary. Every time data crosses a trust boundary, it must be revalidated as correct to prevent improper or hostile inputs to a system. Trust contracts are just that—descriptions of the requirements for trust when crossing the boundary between elements. These are important so that the supplier of data across the boundary does not inadvertently send something that would be rejected. Just specifying that something is an integer value is not enough. Across what range of integers is legal? 8 bit, 16 bit, 32 bit? Positive and negative numbers? The number zero? The specifics can matter significantly, and they need to be specified. This also provides the necessary information for effective security testing and analysis of the boundaries and potential input locations.

Chapter Review

This chapter opened with an examination of security controls and how they play a role in managing risks associated with enterprise software deployment. An analysis of applying security via the build environment including using the security features of the IDE was covered. Anti-tampering techniques such as code signing and source code management were covered, as well as the use of obfuscation to resist attackers modifying code once deployed.

An examination of defensive coding techniques begins with an analysis of the differences between declarative and programmatic (imperative) security. An examination of bootstrapping, cryptographic agility, and secure handling of configuration parameters followed suit. The security implications of the interface coding associated with APIs was presented. Memory management and the related issues of type-safe practices and locality were presented. The chapter closed with an examination of the primary mitigations that are used in defensive coding, followed by secure integration of components including secure reuse of third-party code or libraries and system-of-systems integration.

Quick Tips

- Declarative security refers to defining security relations with respect to the container.

- Programmatic security is where the security implementation is embedded into the code itself.
- Cryptographic agility is the ability to manage the specifics of cryptographic function that are embodied in code without recompiling, typically through a configuration file.
- Securing configuration parameters is an important issue when configuration can change programmatic behaviors.
- Memory management is a crucial aspect of code security.
- In managed code applications, the combination of managed code and the intermediate code execution engine takes care of memory management, and type safety makes the tasking easier.
- In unmanaged code situations, the responsibility for memory management is shared between the operating system and the application, with the task being even more difficult because of the issues associated with variable type mismatch.
- Type-safe code will not inadvertently access arbitrary locations of memory outside the expected memory range.
- Locality is a principle that, given a memory reference by a program, subsequent memory accesses are often predictable and are in close proximity to previous references.
- APIs are significant in that they represent entry points into software.
- A set of primary mitigations has been established over time as proven best practices.
- Software is designed as a series of functions with interfaces between them, and these functions can be implemented using third-party code such as common libraries.

Questions

To further help you prepare for the CSSLP exam, answer the following questions. The correct answers are provided in the following section.

1. Code signing cannot provide which of the following to the end user?
 - A. Proof of integrity of the code
 - B. Authentication of the source

- C. Assurance that the code is free of vulnerabilities
 - D. Assurance that the code is authentic
- 2. To ensure that the end user is receiving valid code, which of the following methodologies is employed?
 - A. Code signing
 - B. Version control
 - C. Configuration management
 - D. Revision control
- 3. Compiler options should be what?
 - A. Not relied upon, as they can provide errant information
 - B. Limited to key checks for efficiency
 - C. Left to developer discretion to enable efficient production
 - D. Predefined as part of the SDL
- 4. What is managing the versions and changes associated with all the individual pieces of a software project referred to as?
 - A. Version control
 - B. Secure development process
 - C. Configuration management
 - D. Source code repository
- 5. To deal with international distribution issues associated with cryptography, a clean method is via what?
 - A. Versioning
 - B. Use of international cryptography for all versions
 - C. Use of approved cryptographic libraries only
 - D. Use of cryptographic agility
- 6. What does the level of security applied to configuration files depends upon?
 - A. The location of the actual configuration file
 - B. The number of data elements in the configuration file (scaling)
 - C. Whether configuration files are locked down in a read-only

directory

- D. The risk introduced by bad configurations
7. What should you employ to manage replacement of deprecated cryptographic functions?
- A. Cryptographic agility
 - B. Only custom cryptography so you can control when it gets deprecated
 - C. Only approved cryptographic library functions
 - D. Only current cryptographic functions
8. All of the following are concerns of memory management except what?
- A. Garbage collection
 - B. Buffer overflows
 - C. Memory allocation
 - D. Exception management
9. What do type-safe practices enable?
- A. Memory management safety
 - B. Smaller programs
 - C. Avoidance of exceptions in managed code
 - D. Easier-to-maintain code
10. Elements of defensive coding include all of the following except what?
- A. Custom cryptographic functions to avoid algorithm disclosure
 - B. Exception handling to avoid program termination
 - C. Interface coding efforts to avoid API-facing attacks
 - D. Cryptographic agility to make cryptographic functions stronger

Answers

1. C. Code signing can provide no information as to the degree to which code is free of errors or defects.
2. A. Code signing can provide information as to source validity and

integrity.

3. D. Compilers can provide a wide array of defensive efforts if properly employed and should be a defined part of the software development lifecycle.
4. A. The management of the versions and changes of all the components of a project is referred to as version control.
5. D. Cryptographic agility can also assist in the international problem of approved cryptography. In some cases, certain cryptographic algorithms are not permitted to be exported to or used in a particular country. Rather than creating different source-code versions for each country, agility can allow the code to be managed via configurations.
6. D. Security responses should always be commensurate with risk.
7. A. Cryptographic agility enables the replacement of deprecated cryptographic functions without recompiling.
8. D. Exception management is not directly related to memory management.
9. A. Type-safe languages have greatly reduced memory management issues due to the removal of one of the memory management challenges.
10. A. Custom cryptographic functions are always a bad idea and frequently lead to failure.

PART V

Secure Software Testing

- [**Chapter 11**](#) Security Test Cases
- [**Chapter 12**](#) Security Testing Strategy and Plan
- [**Chapter 13**](#) Software Testing and Acceptance

Security Test Cases

In this chapter you will

- Learn about developing security test cases
 - Learn about attack surface validation
 - Explore penetration tests as part of testing
 - Explore fuzzing
 - Examine scanning
 - Learn about simulation
 - Explore the role of failures in testing
 - Examine types of cryptographic validation
 - Explore the use of regression testing
 - Explore integration testing
 - Examine the use of continuous testing
-

Designers design to a given specification, developers code to the design, but testing is where the fidelity of the actual code is examined with respect to the goal of performing what is required by the objectives, and nothing else. This chapter will examine some of the elements associated with building test cases to explore various scenarios as part of a comprehensive test program.

Security Test Cases

A security test case is a document that describes an input, action, or event that is expected to produce a predictable response. The fundamental aim of all test cases is to find out if a specified feature in a system or software product is working properly. A test case should contain particulars such as test case identifier, test case name, objective, test conditions/setup, input data

requirements, steps, and expected results. The process of developing test cases can help find problems in the requirements or design of an application, since it requires completely thinking through the operation of the application. For this reason, it's a useful habit to prepare test cases as early in the development process as possible.

Functionally, testing involves examining a system or application under controlled conditions and then evaluating the results. For example, the tester asks, "If the user is in interface A of the application while using hardware B and does C, then D should happen." From a practical standpoint, the controlled conditions ought to include both normal and abnormal conditions. In that respect, testing should intentionally try to make things go wrong in the product to determine what undesirable events might occur when they shouldn't, or which desirable events don't happen when they should.

There are numerous forms of testing, all with a slightly different purpose and focus. But generally, these fall into two categories: white box and black box. Testing that tries to exercise as much of the code as possible within some set of resource constraints is called *white-box* testing. Techniques that do not consider the code's structure when test cases are selected are called *black-box* techniques. What we are going to do is examine the most common approaches within those categories, keeping in mind that even the explicit methodologies we are going to discuss are implemented in different ways depending on the integrity level required for the software product being tested.

Attack Surface Evaluation

The attack surface for a system is technically the set of points on the boundary of a system, a system element, or an environment where an attacker can try to enter and cause an effect on, or extract data from, that system, system element, or environment. In most software systems, this means the place where the data or information enters the system, either legitimately or in an unauthorized fashion. During the design phase, an estimate of the risks and the mitigation efforts associated with the risks is performed. The various entry points and the risks associated with them should be examined. Based on the results of this design, the system is developed, and during development, the actual system design goals may or may not have been met. Testing the code for obvious failures at each step along the way provides significant

information as to which design elements were not met.

It is important to document the actual attack surface throughout the development process. Testing the elements and updating the attack surface model provide the development team with feedback, ensuring that the design attack surface objectives are being met through the development process. Testing elements such as the level of code accessible by untrusted users, the quantity of elevated privilege code, and the implementation of mitigation plans detailed in the threat model is essential in ensuring that the security objectives are being met through the development process.

Microsoft has developed and released a tool called the *attack surface analyzer*, which is designed to measure the security impact of an application on a Windows environment. Acting as a sophisticated scanner, the tool can detect the changes that occur to the underlying Windows OS when an application is installed. Designed to specifically look for and alert on issues that have been shown to cause security weaknesses, the attack surface analyzer enables a development team or an end user to do the following:

- View changes in the Windows attack surface resulting from the installation of the application
- Assess the aggregate attack surface change associated with the application in the enterprise environment
- Evaluate the risk to the platform where the application is proposed to exist
- Provide incident response teams detailed information associated with a Windows platform

One of the advantages of the attack surface analyzer is that it operates independently of the application that is under test. The attack surface analyzer scans the Windows OS environment and provides actionable information on the security implications of an application when installed on a Windows platform. For this reason, it is an ideal scanner for final security testing as part of the secure development lifecycle (SDL) for applications targeted to Windows environments.

Penetration Testing

Penetration testing, sometimes called *pen testing*, is an active form of examining the system for weaknesses and vulnerabilities. While scanning activities are passive in nature, penetration testing is more active.

Vulnerability scanners operate in a sweep, looking for vulnerabilities using limited intelligence; penetration testing harnesses the power of human intellect to make a more targeted examination. Penetration testers attack a system using information gathered from it and expert knowledge in how weaknesses can exist in systems. Penetration testing is designed to mimic the attacker's ethos and methodology, with the objective of finding issues before an adversary does. It is a highly structured and systematic method of exploring a system and finding and attacking weaknesses.

Penetration testing is a valuable part of the SDL process. It can dissect a program and determine if the planned mitigations are effective. Pen testing can discover vulnerabilities that were not thought of or mitigated by the development team. It can be done with a white-, black-, or gray-box testing mode.

Penetration Testing

Penetration testing is a structured test methodology. The following are the basic steps employed in the process:

1. Reconnaissance (discovery and enumeration)
2. Attack and exploitation
3. Removal of evidence
4. Reporting

The penetration testing process begins with specific objectives being set out for the tester to explore. For software under development, these could be input validation vulnerabilities, configuration vulnerabilities, and vulnerabilities introduced to the host platform during deployment. Based on the objectives, a test plan is created and executed to verify that the software is free of known vulnerabilities. As the testers probe the software, they take notes of the errors and responses, using this information to shape subsequent tests.

Penetration testing is a slow and methodical process, with each step and the results being validated. The records of the tests should demonstrate a reproducible situation where the potential vulnerabilities are disclosed. This information can give the development team a clear picture of what was found so that the true root causes can be identified and fixed.

Common Methods

Several common methods are employed when testing to make different determinations concerning software fitness. *Fuzzing* is a technique used to find specific errors associated with input validation. *Scanning* is another method used to look for specific conditions that result in undesired states or outcomes. *Simulations* are a means of testing how the software performs under typical operating conditions. *Failure mode testing* checks that known failure modes are understood and caught. *Cryptographic testing* is important because of the highly technical nature of cryptography. *Regression testing* is used to ensure that errors that are found are tested against different versions of the code base that are in production. *Integration testing* is done to manage interfaces with external data sources, and continuous testing is performed.

Fuzzing

Fuzz testing is a brute-force method of addressing input validation issues and vulnerabilities. Fuzzing a program includes applying large numbers of inputs to determine which ones cause faults and which ones might be vulnerable to exploitation. Fuzz testing can be applied to anywhere data is exchanged to verify that input validation is being performed properly. Network protocols can be fuzzed, file protocols can be fuzzed, and web protocols can be fuzzed. The vast majority of browser errors are found via fuzzing.

Fuzz testing works well in white-, black-, or gray-box testing, as it can be independent of the specifics of the application under test. Fuzz testing works by sending a multitude of input signals and seeing how the program handles them. Specifically, malformed inputs can be used to vary parser operation and to check for memory leaks, buffer overflows, and a wide range of input validation issues. Since input validation errors are one of the top issues in software vulnerabilities, fuzzing is the best method of testing against these issues, such as cross-site scripting and injection vulnerabilities.

There are several ways to classify fuzz testing. One set of categories is smart and dumb, indicating the type of logic used in creating the input values. Smart testing uses knowledge of what could go wrong and creates malformed inputs with this knowledge. Dumb testing just uses random inputs. Another set of terms used to describe fuzzers is generation-based and mutation-based.



EXAM TIP Fuzz testing is a staple of SDL-based testing, finding a wide range of errors with a single test method.

Generation-based fuzz testing uses the specifications of input streams to determine the data streams that are to be used in testing. Mutation-based fuzzers take known good traffic and mutate it in specific ways to create new input streams for testing. Each of these has its advantages, and the typical fuzzing environment involves both used together.

Scanning

Scanning is automated enumeration of specific characteristics of an application or network. These characteristics can be of many different forms, from operating characteristics to weaknesses or vulnerabilities. Network scans can be performed for the sole purpose of learning what network devices are available and responsive. Systems can be scanned to determine the specific operating system (OS) in place, a process known as *OS fingerprinting*. Vulnerability scanners can scan applications to determine if specific vulnerabilities are present.

Scanning can be used in software development to characterize an application on a target platform. It can provide the development team with a wealth of information as to how a system will behave when deployed into production. There are numerous security standards, including the Payment Card Industry Data Security Standard (PCI DSS), that have provisions requiring the use of scanners to identify weaknesses and vulnerabilities in enterprise platforms. The development team should take note that enterprises will be scanning the application as installed in the enterprise. Gaining an understanding of the footprint and security implications of an application

before shipping will help the team to identify potential issues before they are discovered by customers.

Scanners have been developed to search for a variety of specific conditions. There are scanners that can search code bases for patterns that are indicative of elements of the OWASP Top 10 and the SANS Top 25 lists. There are scanners tuned to produce reports for PCI and Sarbanes-Oxley (SOX) compliance. A common mitigation for several regulatory compliance programs is a specific set of scans against a specified set of vulnerabilities.

Vulnerabilities are special forms of errors, in that they can be exploited by an adversary to achieve an unauthorized result. As in all other types of defects, vulnerabilities can range in severity, and this is measured by the potential impact on the overall system. Scanning for known vulnerabilities is important as attackers will do this very thing on code bases. Vulnerabilities are frequently found during activities such as penetration testing and fuzz testing. The nature of these testing environments and the types of results make vulnerability discovery their target of opportunity. By definition, these types of errors are more potentially damaging, and they will score higher on bug bar criteria than many other error types.

Simulations

Simulation testing involves testing the application in an environment that mirrors the associated production environment. Examining issues such as configuration issues and how they affect the program outcome is important. Data issues that can result in programmatic instability can also be investigated in the simulated environment.

Setting up an application and startup can be time-consuming and expensive. When developing a new application, considering the challenges associated with the instantiation of the system can be important with respect to customer acceptance. Simple applications may have simple setups, but complex applications can have significant setup issues. Simulation testing can go a long way toward discovering issues associated with the instantiation of an application and its operation in the production environment.

Simulation testing can provide that last testing line of defense to ensure the system is properly functioning prior to deployment. This is an opportunity to verify that the interface with the operating system is correct and that roles are properly configured to support access and authorization. It also checks

that firewall rules (or other enforcement points) between tiers/environments are properly documented, configured, and tested to ensure that attack surface/exposure is managed. Other benefits of simulation testing include validating that the system itself can stand up to the rigors of production performance—for example, using load testing to “beat up” the application to ensure availability is sustainable and that the controls don’t “break” when the load reaches a particular threshold.

Failure Modes

Not all errors in code result in failure. Not all vulnerabilities are exploitable. During the testing cycle, it is important to identify errors and defects, even those that do not cause a failure. Although a specific error, say one in dead code that is never executed, may not cause a failure in the current version, this same error may become active in a later version and result in a failure. Leaving an error such as this alone or leaving it for future regression testing is a practice that can cause errors to get into production code.

Fault testing is a specific test that uses faults to ensure they generate errors and the errors are properly handled. Another name for fault testing is *break test*. Break testing is where one uses inputs that are specifically designed to trigger failures. Then the handling of these failures is measured to ensure the software can still function. Stress testing is the use of heavy loads on software to ensure that it still functions reliably, both with use and misuse cases.

Although most testing is for failure, it is equally important to test for conditions that result in incorrect values, even if they do not result in failure. Incorrect values have resulted in the loss of more than one spacecraft in flight; even though the failure did not cause the program to fail, it did result in system failure. A common failure condition is load testing, where the software is tested for capacity issues. Understanding how the software functions under heavy load conditions can reveal memory issues and other scale-related issues. These elements can cause failure in the field, and thus extensive testing for these types of known software issues is best conducted early in the development process where issues can be addressed prior to release.

Cryptographic Validation

Having secure cryptography is easy: use approved algorithms and implement them correctly and securely. The former is relatively easy—pick the algorithm from a list. The latter is significantly more difficult. Protecting the keys and the seed values and ensuring proper operational conditions are met have proven to be challenging in many cases. Other cryptographic issues include proper random number generation and key transmission.

Cryptographic errors come from several common causes. One typical mistake is choosing to develop your own cryptographic algorithm. Developing a secure cryptographic algorithm is far from an easy task, and even when done by experts, weaknesses can occur that make them unusable. Cryptographic algorithms become trusted after years of scrutiny and attacks, and any new algorithms would take years to join the trusted set. If you instead decide to rest on secrecy, be warned that secret or proprietary algorithms have never provided the desired level of protection. One of the axioms of cryptography is that security through obscurity has never worked in the long run.

Deciding to use a trusted algorithm is a proper start, but there still are several major errors that can occur. The first is an error in instantiating the algorithm. An easy way to avoid this type of error is to use a library function that has already been properly tested. Sources of these library functions abound and provide an economical solution to this functionality's needs. Given an algorithm and a proper instantiation, the next item needed is the random number to generate a random key.

The generation of a real random number is not a trivial task. Computers are machines that are renowned for reproducing the same output when given the same input, so generating a string of pure, nonreproducible random numbers is a challenge. There are functions for producing random numbers built into the libraries of most programming languages, but these are pseudo-random number generators, and although the distribution of output numbers appears random, it generates a reproducible sequence. Given the same input, a second run of the function will produce the same sequence of “random” numbers. Determining the seed and random sequence and using this knowledge to “break” a cryptographic function has been used more than once to bypass the security. This method was used to subvert an early version of Netscape’s Secure Sockets Layer (SSL) implementation. An error in the

Debian instantiation of OpenSSL resulted in poor seed generation, which then resulted in a small set of random values.



EXAM TIP Cryptographically random numbers are essential in cryptosystems and are best produced through cryptographic libraries.

The second issue is one of entropy. Entropy is a measure of the randomness of a bit sequence, and true random numbers have high entropy. High entropy sequences can also be an indication that the value is a random number, making the storing of a random number in memory discoverable to an attacker. This means that random numbers must be handled in special and specific ways when in memory and stored to prevent their discovery and recovery by an attacker.

Using a number that is cryptographically random and suitable for an encryption function resolves the random seed problem, and again, the use of trusted library functions designed and tested for generating such numbers is the proper methodology. Trusted cryptographic libraries typically include a cryptographic random number generator.

Poor key management has failed many a cryptographic implementation. A famous exploit where cryptographic keys were obtained from an executable and used to break a cryptographic scheme involved hackers using this technique to break DVD encryption and develop the DeCSS program. Tools have been developed that can search code for “random” keys and extract them from the code or running process. The bottom line is simple: do not hard-code secret keys in your code. They can, and will, be discovered. Keys should be generated and then passed by reference, minimizing the travel of copies across a network or application. Storing them in memory in a noncontiguous fashion is also important to prevent external detection.

FIPS 140-2

FIPS 140-2 is a prescribed standard, part of the Federal Information Processing Standards series that relates to the implementation of cryptographic functions. FIPS 140-2 deals with issues such as selecting

approved algorithms, such as AES, RSA, and DSA. FIPS 140-2 also deals with the environment where the cryptographic functions are used, as well as the means of implementation.



EXAM TIP FIPS 140-2 specifies requirements, specifications, and testing of cryptographic systems for the U.S. federal government.

Regression Testing

As changes to software code bases occur, they must be tested against functional and nonfunctional requirements. This is the normal testing that occurs as part of any change process. Regression testing is the testing of the changes when applied to older versions of a code base. This is common in large software projects that have multiple versions distributed across a customer base. The challenge is not in the direct effects of a change, but in interactive changes that occur because of other code differences between the two versions of a program. Regression testing can be expensive and time-consuming and is one of the major challenges for a software vendor that is supporting multiple versions of a product.

Software is a product that continually changes and improves over time. Multiple versions of software can have different and recurring vulnerabilities. Anytime that software is changed, whether by configuration, patching, or new modules, the software needs to be tested to ensure that the changes have not had an adverse impact on other aspects of the software. Regression testing is a minor element early in a product's lifecycle, but as a product gets older and has advanced through multiple versions, including multiple customizations, etc., the variance between versions can make regression testing a slow, painful process.

Regression testing is one of the most time-consuming issues associated with patches for software. Patches may not take long to create—in fact, in some cases, the party discovering the issue may provide guidance on how to patch. But before this solution can be trusted across multiple versions of the software, regression testing needs to occur. When software is “fixed,” several things can happen. First, the fix may cause a fault in some other part of the

software. Second, the fix may undo some other mitigation at the point of the fix. Third, the fix may repair a special case, for example, entering a letter instead of a number, but miss the general case of entering any non-numeric value. The list of potential issues can go on, but the point is that when a change is made, the stability of the software must be checked.

Regression testing is not as simple as completely retesting everything—this would be too costly and inefficient. Depending upon the scope and nature of the change, an appropriate regression test plan needs to be crafted. Simple changes to a unit may require a level of testing be applied only to the unit, making regression testing fairly simple. In other cases, regression testing can have a far-reaching impact across multiple modules and use cases. A key aspect of the patching process is determining the correct level, breadth, and scope of regression testing that is required to cover the patch.

Specialized reports, such as delta analysis and historical trending reports, can assist in regression testing efforts. These reports are canned types and are present in a variety of application security test tools. When leveraging regular scan and reporting cycles, remediation meetings use these reports to enable the security tester to analyze and work with teams to fix the vulnerabilities associated with each release—release 1 versus release 2, or even over the application’s release lifetime (compare release 1 to 2 to 3 and so on).

Integration Testing

Even if each unit tests properly per the requirements and specifications, a system is built up of many units that work together to achieve a business objective. There are emergent properties that occur in systems, and integration (or systems-level) testing should be designed to verify that the correct form and level of the emergent properties exist in the system. A system can be more than just the sum of the parts, and if part of the “more” involves security checks, these need to be verified.

Systems or integration testing is needed to ensure that the overall system is compliant with the system-level requirements. It is possible for one module to be correct and another module to also be correct but for the two modules to be incompatible, causing errors when connected. System tests need to ensure that the integration of components occurs as designed and that data transfers between components are secure and proper.

Continuous Testing

Continuous testing is the use of automated testing as part of the software delivery process. This is done to collect data on the business risk issues associated with a software release candidate in a rapid form, allowing the development team fast access to the results. Continuous testing is embedded within the development process, not tacked on at the end, and its feedback is an integral part of the development process. Continuous testing requires the team to continuously review the test suite to eliminate redundancy and optimize risk reporting.

Chapter Review

In this chapter, we examined the technologies employed in testing security aspects of software. The chapter opened with an examination of security test cases. This was followed by an exploration of attack surfaces and how they play a role in securing software. The next topic was penetration testing. This was followed by an examination of the common methods employed in security testing including fuzzing, scanning, simulations, failure modes, cryptographic validations, regression testing, integration testing, and continuous testing.

Quick Tips

- A security test case is a document that describes an input, action, or event that is expected to produce a predictable response.
- Testing potential entry points identified in the attack surface model is critical.
- Penetration testing is where testers apply the tools, techniques, and mindset of an attacker to a system.
- Fuzzing is brute-force testing of input validation.
- Scanning is an automated enumeration of specific elements in a test.
- Simulation testing is examining how a system would perform in production.
- Failure mode testing examines whether failures are properly handled.

- Cryptographic validation is the specific testing of cryptographic elements including random number generation, algorithms, and secrets protection.
- Regression testing is testing performed on previous versions to ensure fixes to the current software don't break other versions.
- Integration testing is testing performed to ensure error-free integrations of components in a system.
- Continuous testing is testing performed as part of the delivery process to ensure release candidates meet testing objectives with respect to system risk.

Questions

To further help you prepare for the CSSLP exam, answer the following questions. The correct answers are provided in the following section.

1. Testing different versions of an application to verify patches don't break something is referred to as what?
 - A. Penetration testing
 - B. Simulation testing
 - C. Fuzz testing
 - D. Regression testing
2. What is testing an application in an environment that mirrors the production environment called?
 - A. Simulation testing
 - B. Fuzz testing
 - C. Scanning
 - D. Penetration testing
3. Verification of cryptographic function includes all of the following except what?
 - A. Use of secret encryption methods
 - B. Key distribution
 - C. Cryptographic algorithm

- D. Proper random number generation
- 4. An automated enumeration of specific characteristics of an application or network is referred to as what?
 - A. Penetration testing
 - B. Scanning
 - C. Fuzz testing
 - D. Simulation testing
- 5. To examine a system for input validation errors, what is the most comprehensive test?
 - A. Scanning
 - B. Penetration testing
 - C. Regression testing
 - D. Fuzz testing
- 6. Fuzz testing data can be characterized by what?
 - A. Mutation or generation
 - B. Input validation, file, or network parsing
 - C. Size and character set
 - D. Type of fault being tested for
- 7. What steps does penetration testing include?
 - A. Reconnaissance, testing, reporting
 - B. Reconnaissance, exploitation, recovery
 - C. Attacking, testing, recovery
 - D. Reconnaissance, attacking, removal of evidence, reporting
- 8. FIPS 140-2 is a federal standard associated with what?
 - A. Software quality assurance
 - B. Security testing
 - C. Cryptographic implementation in software
 - D. Software security standards
- 9. What is one of the challenges associated with regression testing?

- A. Determining an appropriate test portfolio
 - B. Determining when to employ it
 - C. The skill level needed to perform it
 - D. Designing an operational platform for testing
- 10.** Regression testing is employed across what?
- A. Different modules based on fuzz testing results
 - B. Different versions after patching has been employed
 - C. Cryptographic modules to test stability
 - D. The production environment

Answers

- 1.** **D.** Regression testing is used to ensure patches don't break different versions of an application.
- 2.** **A.** Simulation testing involves mimicking the production environment.
- 3.** **A.** Secret encryption methods are not part of cryptographic validation.
- 4.** **B.** Scanning can be used to automate the enumeration of system elements.
- 5.** **D.** Fuzz testing is used to test for input validation errors, memory leaks, and buffer overflows.
- 6.** **A.** Fuzz testing datasets are built by either mutation or generation methods.
- 7.** **D.** The steps for penetration testing are reconnaissance, attacking, removal of evidence, and reporting.
- 8.** **C.** FIPS 140-2 is a cryptographic standard associated with the U.S. government.
- 9.** **A.** Determining the scope and type of tests required based on a patch is one of the challenges of regression testing.
- 10.** **B.** Regression testing is performed after patching and occurs across different versions of an application.

CHAPTER 12

Security Testing Strategy and Plan

In this chapter you will

- Learn about security testing strategy and planning
 - Explore functional security testing
 - Explore nonfunctional security testing
 - Learn about different testing techniques
 - Understand the role of the testing environment
 - Explore testing standards
 - Explore the role of crowd sourcing
-
-

Security testing is an essential part of software creation. It is important from a quality perspective as well as a security perspective. Testing can occur as a result of a myriad of activities, but to be fully effective the testing regime must be integrated as part of the software build process. Several specific elements need to be tested for, including both functional and nonfunctional requirements. This chapter explores these concepts as well as strategies and standards to assist in full-coverage testing.

Develop a Security Testing Strategy and a Plan

Testing software during its development is an integral part of the development process. Developing a test plan, which is a document detailing a systematic approach to testing a system such as a machine or software, is the first step. The plan begins with the test strategy in outline form that describes the overall testing approach. From this plan, test cases are created. A test case is designed to answer the question, “What am I going to test, and what will

‘correct’ look like?” A document enumerating these cases includes information such as unique test identifiers, links to requirement references from design specifications, notes on any preconditions, references to any test harnesses and scripts required, and any additional notes.

Security testing should always be guided by a plan. The plan spells out the scope, approach, resources, and schedule of the testing activity. The plan estimates the number of tests and test data sets and their duration and defines the test completion criteria. The plan also includes provisions for identifying risks and allocating resources.



NOTE Testing occurs for many reasons. Simple functional testing of a unit-level code fragment is common to ensure that the unit of code properly performs its function and handles errors appropriately. Security testing occurs to ensure that the software is free of defects or vulnerabilities that increase risk. Qualification testing is also called *acceptance testing* and is a suite of tests designed to ensure that the completed product is ready for use. Each of these types of tests has overlapping elements, and it is important to view them as all one big set of requirements, no one specifically more important than the other and all playing a role in ensuring the final product is correct per the specifications.

The design and execution of the qualification tests themselves are normally dictated in the contract. That usually includes consideration of the following:

- Required features to be tested
- Requisite load limits
- Number and types of stress tests
- All necessary risk mitigation and security tests
- Requisite performance levels
- Interfaces to be tested
- Test cases to address each of the following questions

Elements of a Testing Plan

A testing plan should answer the following practical questions:

- Who's responsible for generating the test designs/cases and procedures?
- Who's responsible for executing the tests?
- Who's responsible for building/maintaining the test bed?
- Who's responsible for configuration management?
- What are the criteria for stopping the test effort?
- What are the criteria for restarting testing?
- When will source code be placed under change control?
- Which test designs/cases will be placed under configuration management?
- What level will anomaly reports be written for?

Each test case must stipulate the actual input values and expected results. The general goal of the testing activity is to exercise the component's logic in a way that will expose any latent defects that might produce unanticipated or undesirable outcomes. The overall aim of the testing process is to deploy the smallest number of cases possible that will still achieve sufficient understanding of the quality and security of the product. In that respect, each explicit testing procedure contributes to that outcome.

Functional Security Testing

Functional software testing is performed to assess the level of functionality associated with the software as expected by the end user. Functional testing is used to determine compliance with requirements in the areas of reliability, logic, performance, and scalability. Reliability measures that the software functions as expected by the customer at all times. It is not just a measure of availability, but functionally complete availability. Resiliency is a measure of how strongly the software can perform when it is under attack by an adversary.

Steps for Functional Testing

Functional testing involves the following steps in this order:

1. Identifying the functions (requirements) that the software is expected to perform
2. Creating input test data based on the function's specifications
3. Determining expected output test results based on the function's specifications
4. Executing the test cases corresponding to functional requirements
5. Comparing actual and expected outputs to determine functional compliance

Unit Testing

Unit testing is conducted by developers as they develop the code. This is the first level of testing and is essential to ensure that logic elements are correct and that the software under development meets the published requirements. Unit testing is essential to the overall stability of the project, as each unit must stand on its own before being connected together. At a minimum, unit testing will ensure functional logic, understandable code, and a reasonable level of vulnerability control and mitigation.



EXAM TIP One of the principal advantages of unit testing is that it is done by the development team and catches errors early, before they leave the development phase.

Nonfunctional Security Testing

Part of the set of requirements for the software under development should be the service levels of agreement that can be expected from the software.

Testing software for requirements such as reliability, performance, and scalability can be as important as the functional requirements associated with the specific functionality of the software. These nonfunctional requirements are frequently associated with ease of use and other system factors that when not present can significantly reduce the usefulness of the software.

Nonfunctional requirements such as these still need to be specified in measurable terms for the development team to be able to address them. Typically, these are expressed in terms of a service level agreement (SLA). The typical objective in performance testing is not to find specific bugs; rather, the goal is to determine bottlenecks and performance factors for the systems under test. These tests are frequently referred to as *load testing* and *stress testing*. Load testing involves running the system under a controlled speed environment. Stress testing takes the system past this operating point to see how it responds to overload conditions.



EXAM TIP Recoverability is the ability of an application to restore itself to expected levels of functionality after the security protection is breached or bypassed.

Testing Techniques

Testing includes white-box testing, where the test team has access to the design and coding elements; black-box testing, where the team does not have access; and gray-box testing, where information is greater than black-box testing but short of white-box testing. This nomenclature does not describe the actual tests being performed but rather indicates the level of information present to the tester before the test.

White-Box Testing

White-box testing is performed on a system with full knowledge of the working components, including the source code and its operation. This is commonly done early in the development cycle. The advantage of white-box testing is that the attacker has knowledge of how the system works and can

spend their time compromising it. The unit testing of a section of code by the development team is an example of white-box testing. White-box testing, by design, provides the attacker with complete documentation, including source code and configuration parameters. This information can then be used to devise potential methods of attacking the software. Thus, white-box testing can focus on the structural basis of the software and the operational deployment considerations with respect to its use or misuse.



EXAM TIP When testers have access to full knowledge of a system, including the source code, it is referred to as *white-box* testing.

Black-Box Testing

Black-box testing is where the attacker has no knowledge of the inner workings of the software under test. This is common in more advanced system-level tests, such as penetration testing. The lack of knowledge of the specific implementation is not as important as one may think at times, because the attacker still has the same knowledge that an end user would possess, so they know what inputs are requested. Using their knowledge of how things work and what patterns of vulnerabilities are likely to exist, an attacker is not as blind in black-box testing as you might think. Black-box testing focuses on the behavioral characteristics of the application.



EXAM TIP When testers have access to no knowledge of how a system works, including no knowledge of source code, it is referred to as *black-box* testing.

Gray-Box Testing

Gray-box testing is aptly named, as an attacker has more knowledge of the inner workings than with black-box testing but less than total access to source

code. Gray-box testing is relatively rare outside of internal testing.

Testing Environment

Software applications operate within a specific environment, which also needs to be tested. Trust boundaries, described earlier in the book, are devices used to demarcate the points where data moves from one module set to another. Testing the data movement across trust boundaries from end to end of the application is important. When the complete application, from end to end, is more than a single piece of code, interoperability issues may arise and need to be tested for. When security credentials, permissions, and access tokens are involved, operations across trust boundaries and between modules become areas of concern. Verifying that all dependencies across the breadth of the software are covered, both logically and from a functional security credential point of view, is important.

Comparison of Common Testing Types

White-Box Testing	Black-Box Testing
Full knowledge, including source code.	Zero knowledge.
Assesses software structure and design.	Assesses software behavior.
Low false positives.	High false positives.
Logic flaws are detected.	Logic flaws are typically not visible.

Environment

The actual testing environment can make a huge difference in both integrating the testing process into the software development process and collecting the necessary data to document the successful completion of testing. Having an established environment for testing allows for the interoperability of artifacts and metrics between systems to improve the usability of the test results.

It is important to test data movement across trust boundaries from end to end of the application. When the complete application, from end to end, is more than a single piece of code, interoperability issues may arise and need to be tested before use. It's not enough to test and resolve issues that reside within each module of a system; a key component of testing is to check for errors that occur in the connections between the modules. The U.S. space program has lost more than one spacecraft because of a mismatch between units on interconnected systems. Each section worked as expected, but one module sent a number in foot-pounds to a system expecting newton-meters. Both numbers were in range of normal operation for the individual subsystem, but the mismatch, which is an approximately 40 percent difference, results in overall system failure.

A *test harness* is a means of documenting the software, tools, samples of data input and output, and configurations used to complete a set of tests. The individual steps of the tests can be encoded in a series of test scripts. Test scripts are important for several reasons. They replicate user actions, and by automating the series of steps (also known as *actions*) to follow, including inputs, they remove the errors that could occur with manual testing. They can also be automated to collect outputs and compare the returned values to expected results, improving the speed and accuracy of test interpretation.

The tests are sometimes grouped into collections referred to as *test suites*. It is common to have test suites for specific functionality cases, such as security, user inputs, boundary condition checking, databases, etc. By grouping them into suites, it makes management easier and promotes reuse as opposed to continual redevelopment of the same types of materials.

Standards

Quality is defined as fitness for use according to certain requirements. This can be different from security, yet there is tremendous overlap in the practical implementation and methodologies employed. In this regard, lessons can be learned from international quality assurance standards, because although they may be more expansive in goals than just security, they can make sense there as well.

ISO/IEC 25010:2011

The International Standard ISO/IEC 25010:2011, “Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models,” provides guidance for establishing quality in software products. With respect to testing, this standard focuses on a quality model built around functionality, reliability, and usability. Additional issues of efficiency, maintainability, and portability are included in the quality model of the standard. With respect to security and testing, it is important to remember the differences between quality and security. Quality is defined as fitness for use, or conformance to requirements. Security is less cleanly defined but can be defined by requirements. One issue addressed by the standard is the human side of quality, where requirements can shift over time or be less clear than needed for proper addressing by the development team. These are common issues in all projects, and the standard works to ensure a common understanding of the goals and objectives of the projects as described by requirements. This information is equally applicable to security concerns and requirements.

SSE-CMM

The Systems Security Engineering Capability Maturity Model (SSE-CMM) is also known as ISO/IEC 21827:2008 and is an international standard for the secure engineering of systems. The SSE-CMM addresses security engineering activities that span the entire trusted product or secure system lifecycle, including concept definition, requirements analysis, design, development, integration, installation, operations, maintenance, and decommissioning. The SSE-CMM is designed to be employed as a tool to evaluate security engineering practices and assist in the definition of improvements to them. The SSE-CMM is organized into processes and corresponding maturity levels. There are 11 processes that define what needs to be accomplished by security engineering. The maturity level is a standard CMM metric representing how well each process achieves a set of goals. As a model, the SSE-CMM has become a de facto standard for evaluating security engineering capabilities in an organization.

OSSTMM

The Open Source Security Testing Methodology Manual (OSSTMM) is a

peer-reviewed system describing security testing. The OSSTMM provides a scientific methodology for assessing operational security built upon analytical metrics. As shown in [Table 12-1](#), it includes five security testing sections: data networks, telecommunications, wireless, physical, and human. The purpose of the OSSTMM is to create a system that can accurately characterize the security of an operational system in a consistent and reliable fashion.

OSSTMM Section	Test/Audit Area
Data networks	Information security controls
Telecommunications	Telecommunication networks
Wireless	Mobile devices Wireless networks and devices
Physical	Access controls Building and physical perimeter controls
Human	Social engineering controls User security awareness training End-user security controls

Table 12-1 OSSTM Sections and Test/Audit Areas

The OSSTMM provides a scientific methodology that can be used in the testing of security. The Institute for Security and Open Methodologies (ISECOM), the developer of OSSTMM, has developed a range of training classes built around the methodology. The OSSTMM can also be used to assist in auditing, as it highlights what is important to verify regarding functional operational security.

Crowd Sourcing

There are other sources of testing that occur for software, including those from outside of the development process. Customers may test software, as might third parties, but two important groups are hackers and security researchers. Hackers represent a group of attackers using methods such as

fuzzing to find holes in software that can then be exploited. Security researchers do much the same thing. The difference lies in what they do with the test results. Attackers never tell the development team; they use the information to either attack the system themselves or sell it to others who will. Security researchers, on the other hand, will tell the company about their findings and ideally do so under the rules of responsible disclosure. These rules are in essence a period of time where the discovery is kept secret, giving the firm time to create and issue a patch. Forward-thinking companies can establish bug bounty programs, where they have formal channels with the security research community for the expressed purpose of performing these external tests. In exchange for the labor involved, the firms pay bounties, or rewards, to the researchers. This ecosystem has become popular and has significant advantages for all parties involved.

Chapter Review

This chapter examined the security testing strategy and testing plan. It discussed testing functional and nonfunctional requirements, as well as common testing techniques. Several common techniques such as white-box, black-box, and gray-box testing were compared. This was followed by examining the testing environment and a review of relevant standards. The chapter closed with a look at the use of bug bounty programs to expand testing to include the general security research community in a safe and responsible way.

Quick Tips

- A test plan is a document detailing a systematic approach to testing a system.
- Functional software testing is performed to assess the level of functionality associated with the software as expected by the end user.
- Nonfunctional testing of software includes elements such as reliability, performance, and scalability.
- White-box testing is performed on a system with full knowledge of the working components, including the source code and its operation.
- Black-box testing is where the attacker has no knowledge of the inner

workings of the software under test.

- It is important to test data movement across trust boundaries from end to end of the application.
- International Standard ISO/IEC 25010 provides guidance for establishing quality in software products.
- The SSE-CMM is designed to be employed as a tool to evaluate security engineering practices and assist in the definition of improvements to them.
- A bug bounty program is a formal arrangement between software firms and researchers where researchers are paid to find vulnerabilities rather than have the errors publicly exposed.

Questions

To further help you prepare for the CSSLP exam, answer the following questions. The correct answers are provided in the following section.

1. What is testing without knowledge of the inner workings of a system called?
 - A. Pen testing
 - B. White-box testing
 - C. Black-box testing
 - D. Vulnerability scanning
2. What is OSSTMM used for?
 - A. Assessing operational security using analytical metrics
 - B. Security engineering
 - C. Quality assurance for software
 - D. Evaluating security engineering practices
3. Functional testing includes all of the following except what?
 - A. System testing
 - B. Attack surface area testing
 - C. Unit testing
 - D. Performance testing

4. Functional testing is used to determine which of the following characteristics?

 - A. Reliability, bugs, performance, and scalability
 - B. Resiliency, logic, security, and testability
 - C. Resiliency, bugs, requirements, and scalability
 - D. Reliability, logic, performance, and scalability
5. When testing is done with complete knowledge of the source code, it is called what?

 - A. Unit testing
 - B. Functional testing
 - C. White-box testing
 - D. Code walkthrough
6. What type of testing is used to assess software behavior, albeit with significant false-positive results because of no system knowledge?

 - A. OSSTMM testing
 - B. Environmental testing
 - C. Trust boundary testing
 - D. Black-box testing
7. Environmental testing can be used to do what?

 - A. Test data movement across trust boundaries from end to end of the application
 - B. Ensure the code will run in the cloud
 - C. Ensure code compiles completely
 - D. Verify mutual authentication functions in the application
8. Functional testing includes what steps?

 - A. Requirements, test data creation, expected output results, execute test cases, comparison of actual and expected outputs
 - B. Create test data, perform functional test, score output
 - C. Requirements, create test data, perform functional test
 - D. Requirements, perform functional test, score output

9. What is an international standard for establishing quality in software products?

 - A. ISO 9000
 - B. ISO 27001
 - C. ISO 21827
 - D. ISO 25010
10. Which of the following is not necessarily spelled out in a test plan?

 - A. Scope
 - B. Schedule
 - C. Results
 - D. Resources

Answers

1. C. When testers do not have access to any knowledge of how a system works, including any knowledge of source code, it is referred to as black-box testing.
2. A. The OSSTMM is a scientific methodology for assessing operational security built upon analytical metrics.
3. B. Attack surface area calculations are part of the SDL process, not the actual function of the application.
4. D. Functional testing is used to determine compliance with requirements in the areas of reliability, logic, performance, and scalability.
5. C. Testing with full knowledge of source code is called white-box testing.
6. D. Black-box testing is characterized by no knowledge of the system and can examine system behaviors, although it can have a higher false-positive rate due to lack of specific knowledge.
7. A. When the complete application, from end to end, is more than a single piece of code, interoperability issues may arise and need to be tested before use.

- 8.** **A.** The steps are 1) the identification of the functions (requirements) that the software is expected to perform; 2) the creation of input test data based on the function's specifications; 3) the determination of expected output test results based on the function's specifications; 4) the execution of the test cases corresponding to functional requirements; and 5) the comparison of actual and expected outputs to determine functional compliance.
- 9.** **D.** The international standard ISO/IEC 25010:2011 provides guidance for establishing quality in software products.
- 10.** **C.** The results of testing are not known at the time of completing the plan.

CHAPTER 13

Software Testing and Acceptance

In this chapter you will

- Learn about verification and validation testing including software, documentation, and supporting elements
 - Learn to identify undocumented functionality
 - Analyze security implications of test results
 - Explore classifying and tracking security-related errors
 - Explore bug tracking
 - Examine risk scoring
 - Learn about the importance of securing test data
 - Learn about generating test data
 - Explore the reuse of production data
-

Software acceptance is the portion of the secure lifecycle development process where software is determined to meet the requirements specified earlier in the development process. Testing criteria is used to help determine if the software is acceptable for use. The purpose of the acceptance phase of the lifecycle is to determine whether a purchased product or service has met the delivery criteria itemized in the contract. Thus, acceptance is the point in the process where a check gets written. As a result, all aspects of the acceptance process ought to be rigorous. That rigor is embodied by the suite of technical and managerial assessments that the customer organization will employ to ensure that each individual acceptance criterion is duly satisfied.

The evidence to support that judgment is gathered through a series of tests and structured audits that are, as a rule, planned at the beginning of the project and specified in the contract. As a group, those audits and tests seek to determine whether the product meets the functional requirements

stipulated in the contract. Then, based on the results of that assessment, responsibility for the product is transitioned from the supplier organization to the customer. The general term for this step is *delivery*, and the processes that ensure its successful execution are called software *qualification* or *acceptance*. Both of these processes involve testing.

Perform Verification and Validation Testing

Verification and validation (V&V) testing is essential to ensure that software is correct in function. These functions are separate but work together to determine if the system satisfies the requirements. Verification is checking to see if requirements are met. Validation means checking whether the requirements are correct and complete. The V&V processes should support all of the primary process activities of the organization as well as their management. V&V activities also apply to all stages of the software lifecycle. Those activities are established and documented in the software validation and verification plan (SVVP). That plan includes the specification of the information and facilities necessary to manage and perform V&V activities and the means for coordinating all relevant validation and verification activities with other related activities of the organization and its projects.



EXAM TIP Validation means that the software meets the specified user requirements. Verification describes proper software construction. Barry Boehm clarifies the difference in this simple fashion:

Validation: Are we building the right product?

Verification: Are we building the product right?

Planners should assess the overall V&V effort to ensure that the software V&V plan remains effective and continues to actively monitor and evaluate all V&V outputs. This should be based on the adoption of a defined set of metrics. At a minimum, the V&V processes should support configuration

baseline change assessment, management and technical reviews, the interfaces between development and all organizational and supporting processes, and all V&V documentation and reporting. Documentation and reporting includes all V&V task reports, activity summary reports, anomaly reports, and final V&V reports for project closeouts. To do this properly, the software V&V plan should specify the administrative requirements for the following:

- Anomaly resolution and reporting
- Exception/deviation policy
- Baseline and configuration control procedures
- Standards practices and conventions adopted for guidance
- Form of the relevant documentation, including plans, procedures, cases, and results

Management V&V

There are two generic forms of V&V: management and technical. V&V activities for management fittingly examine management plans, schedules, requirements, and methods for the purpose of assessing their suitability to the project. That examination supports decisions about corrective actions, the allocation of resources, and project scoping. It is carried out for the purpose of supporting the management personnel who have direct responsibility for a system. Management reviews and other forms of V&V are meant to discover and report variations from plans and/or defined procedures. They might also recommend corrective action as required. Since management V&V is done to support the management process, it is likely to involve the following management roles:

- Decision-maker
- Review leader
- Review recorder
- Management staff
- Technical staff
- Customer (or user) representative

Management reviews normally consider such things as the statement of project objectives, the status of the software product itself, the status of the project management plan with respect to milestones, any identified anomalies or threats, standard operating procedures, resource allocations, project activity status reports, and other pertinent regulations. Management reviews are scheduled as part of initial project planning and are usually tied to milestones and terminal phases. This does not exclude ad hoc management reviews, which can be scheduled and held for the purposes of risk or threat analysis, software quality management, or operational/functional management, or at the request of the customer.

Technical V&V

Technical V&V evaluates the software product itself, including the requirements and design documentation and the code, test and user documentation and manuals and release notes, and the build and installation procedures. Technical reviews support decisions about whether the software product conforms to its specifications; adheres to regulations, standards, and plans; and has been correctly implemented or changed.

Technical reviews are carried out for the purpose of supporting the customer's and supplier's technical and management personnel who have direct responsibility for the system. They are meant to discover and report defects and anomalies in the software under construction and/or changes. The reviews primarily focus on the software product and its artifacts. Technical reviews are typically done by technical staff and technical management personnel. They potentially involve the following management roles:

- Decision-maker
- Review leader
- Review recorder
- Technical staff and technical managers
- Customer (or user) technical staff

Technical reviews normally consider the statements of objectives for the technical review, the software product itself, the project management plan, anomalies, defects, and security risks. Any relevant prior review reports and pertinent regulations also have to be considered.

Technical reviews should be scheduled as part of initial project planning. These can also be held to evaluate impacts of anomalies or defects. This does not exclude ad hoc technical reviews, which can be scheduled and held for the purposes of supporting functional or project management, system engineering, or software assurance.

It is common practice to perform V&V on the software; this is an obvious answer. But equally important is the performing of V&V on the supporting material including the installation and setup instructions, error messages, user guides, and release notes. All of these elements are important for the successful operation of the software in practice, and anything that is needed to make the system perform correctly in production should be subject to V&V.

Independent Testing

Independent testing can be carried out by third parties to ensure confidence in the delivered product. By involving a disinterested third party in the evaluation process, the customer and supplier can both ensure maximum trust in product integrity. The testing process is similar to the testing activities described earlier. The difference is that the third party carries out those tests. If a third party is involved in this part of the process, it is often termed *independent validation and verification* (IV&V). The essential requirement of independent testing lies in the word *independence*.

The testing manager has to ensure that the testing agent has all necessary latitude to conduct tests and audits in a manner that they deem proper to achieve the assurance goals written into the contract. In general, this means the testing agent maintains a reporting line that is not through the management of the product being evaluated. It also means that the testing agent should report to a person at a sufficient level in the organization to enforce findings from the testing process. The idea in IV&V is to ensure that the people whose product is undergoing evaluation have no influence over the findings of the testers.

Notwithstanding classic testing, audits are perhaps the most popular mechanism for independent acceptance evaluations. Audits provide third-party certification of conformance to regulations and/or standards. Items that may be audited include the following for the product:

- Plans

- Contracts
- Complaints
- Procedures
- Reports and other documentation
- Source code
- Deliverables

At the acceptance stage, audits are typically utilized to ensure confidence that the delivered product is correct, complete, and in compliance with all legal requirements. The process itself is normally conducted by a single person, who is termed the *lead auditor*. The lead auditor is responsible for the audit, including administrative tasks. Audits are normally required by the customer organization to verify compliance with requirements; by the supplier organization to verify compliance with plans, regulations, and guidelines; or by a third party to verify compliance with standards or regulations. The auditor is always a third-party agency, and the initiator is usually *not* the producer.

Audits are initiated by planning activities. Plans and empirical methods have to be approved by all the parties involved in the audit, particularly the audit initiator. All the parties involved in the audit participate in an opening meeting to get all the ground rules set. This meeting might sound like a bureaucratic exercise, but it is important since audits themselves are intrusive, and it is important to ensure that they are conducted as efficiently as possible. The auditors then carry out the examination and collect the evidence. Once all the evidence has been collected, it is analyzed, and a report is prepared. Normally, this report is reviewed by the audited party prior to release to head off any misinformation or misinterpretation. After the initial meeting with the audited party, a closing meeting is held with all parties in attendance, and a report is generated. That report includes the following:

- Preliminary conclusions
- Problems experienced
- Recommendations for remediation

Once the report is accepted, the auditors typically are also responsible for following up on the resolution of any problems identified. In that process, the

auditor examines all the target items to provide assurance that the rework has been properly performed for each item. On acceptance of the final problem resolutions, the auditor submits a final report that itemizes the following:

- The purpose and scope
- The audited organization
- The software product(s) audited
- Any applicable regulations and standards
- The audit evaluation criteria
- An observation list classifying each anomaly detected as major and minor
- The timing of audit follow-up activities

Software Qualification Testing

Qualification or acceptance testing is the formal analysis that is done to determine whether a system or software product satisfies its acceptance criteria. Thus, in practical terms, the customer does qualification testing to determine whether to accept the product. The qualification testing process ensures that the customer's requirements have been met and that all components are correctly integrated into a purchased product.



EXAM TIP The formal analysis that is performed to determine whether a system or software product satisfies its acceptance criteria is called *qualification or acceptance testing*.

Software qualification testing provides evidence that the product is compliant with the requisite levels of design, performance, and assurance that are stipulated in the contract. Thus, the software qualification phase should be designed to prove that the system meets or exceeds the acquirer's requirements. Qualification audit and testing procedures look for meaningful defects in the software's design and execution that might cause the software to fail or that might be exploited in actual use. As a consequence, the scope of

the software qualification audit and testing elements of this phase is tailored to specifically assess whether the design and development of the software are correct.

Given those purposes, the audits and tests that are part of the qualification phase have to be designed so that they not only evaluate compliance with the stipulations of the initial requirements document, but also evaluate compliance with all pertinent contract, standard, and legal requirements. The tests involve a detailed assessment of the important elements of the code under both normal and abnormal conditions. The audits assure that all requisite documentation has been done correctly and that the product satisfies all performance and functional requirements stipulated in the contract.

Qualification Testing Hierarchy

Software testing itself begins at the component level in the development stage of the lifecycle and proceeds up through the hierarchy of testing levels to the fully integrated system that is assured at the acceptance phase. Targets of this top-level phase include the software architecture, components, interfaces, and data. Testing of the delivered product is normally an iterative process because software is built that way. Activities at the acceptance level include the following:

- Software design traceability analysis (e.g., trace for correctness)
- Software design evaluation
- Software design interface analysis
- Test plan generation (by each level)
- Test design generation (by each level)

Assurance concerns at the qualification stage are usually supported by analyzing how well the code adheres to design specifications and coding standards. This assessment is normally supported by such activities as source code and interface traceability analysis and by evaluations of the documentation that is associated with each unit tested. The actual testing is supported by targeted test cases and test procedures that are exclusively generated for the particular object that is undergoing analysis.

Identify Undocumented Functionality

Software is designed to achieve a specific set of functionalities. These functionalities are specified in the requirements. Should software do additional things, this set of additional functionalities may cause security issues. Even more important are additional functionalities that are not known, because these may also cause issues, but the source would truly be unknown. Software is defined as secure when it does what it is supposed to do and only what it is supposed to do. The first part of this description is the set of defined requirements. The second part is the prevention of undocumented functionalities. If software has entry points that can be used by an attacker to achieve undesired functionalities, then it will eventually become hacked. Understanding all entry points, documenting them, and applying the correct safeguards to prevent unauthorized use of them is a key element in producing secure code.

Analyze Security Implications of Test Results

Bugs found during software development are scored based on impact. Impact to the final outcome in the form of risk is essential, because while it is impossible to fix all errors or bugs, it is imperative to fix the ones that will cause significant impacts. The primary driver of whether a bug must be fixed or not is impact, although this is not an excuse to allow other bugs to not be fixed, especially if corrective actions can be taken early at virtually no cost. Another type of bug that must be resolved is one that breaks the build criteria. To prevent costly repair issues later in the build process, many software firms use a regular build process, where modules are checked out, modified, and then must be cleanly re-merged into the next build. This prevents errors from becoming entrenched from older code in a build.

All known defects should be tracked in a bug tracking process. During the course of development, numerous bugs are recorded in the bug tracking system. As part of the bug clearing or corrective action process, a prioritization step determines which bugs get fixed and when. Not all bugs are exploitable, and among those that are exploitable, some have a greater impact on the system. In an ideal world, all bugs would be resolved at every

stage of the development process. In the real world, however, some errors are too hard (or expensive) to fix, and the risk associated with them does not support the level of effort required to fix them in the current development cycle. If a bug required a major redesign, then the cost could be high. If this bug is critical to the success or failure of the system, then resolving it becomes necessary. If it is inconsequential, then resolution may be postponed until the next major update and redesign opportunity.

To make the ultimate determination of whether a bug must be fixed or not falls to a metric in which a bug is scored against a risk standard. If it is determined as a system requirement that no serious or critical bugs are left unmitigated or fixed, then this level is known as the *bug bar*. Errors that score at this level or higher must be fixed; lower errors can be tracked.

Classify and Track Security Errors

Software defects, or bugs, can be characterized in different ways. One method is by the source or effect of the defect. Defects can be broken into five categories:

- **Flaws** Errors in design
- **Bugs** Errors in coding
- **Behavioral anomalies** Issues in how the application operates
- **Errors and faults** Outcome-based issues from other sources
- **Vulnerabilities** Items that can be manipulated to make the system operate improperly

Whether the source of error is coding or design, the errors need to be managed. Management includes identification, characterization, and tracking of how the error is handled. In the following sections we will examine the specifics of bug handling and management.

Bug Tracking

Bug tracking is a basic part of software development. As code is developed, bugs are discovered. Bugs are elements of code that have issues that result in undesired behaviors. Sometimes, the behavior results in something that can

be exploited, and this makes it a potential security bug. Bugs need to be fixed, and hence, they are tracked to determine their status. Some bugs may be obscure, impossible to exploit, and expensive to fix; thus, the best economic decision may be to leave them until the next major rewrite, saving cost now on something that is not a problem. Tracking all the bugs and keeping a log so that things can be fixed at appropriate times are part of managing code development.

Security bug tracking is similar to regular bug tracking. Just because something is deemed a security bug does not mean that it will always be fixed right away. Just as other bugs have levels of severity and exploitability, so do security bugs. A security bug that is next to impossible to exploit and has a mitigating factor covering it may not get immediate attention, especially if it would necessitate a redesign. Under these circumstances, the security bug could be left until the next update of the code base.

There are many ways to score security bugs; a common method is based on the risk principle of impact (damage) times the probability of occurrence. The DREAD (damage, reproducibility, exploitability, affected users, discoverability) model addresses this in a simple form:

$$\text{Risk} = \text{Impact} \times \text{Probability}$$

- Impact = **DREAD**
 - **Damage** Damage needs to be assessed in terms of confidentiality, integrity, and availability.
 - **Affected users** How large is the user base affected?
- Probability = **DREAD**
 - **Reproducibility** How difficult is it to reproduce? Is it scriptable?
 - **Exploitability** How difficult is it to use the vulnerability to affect the attack?
 - **Discoverability** How difficult is it to find?

Measuring each of the DREAD items on a 1 to 10 scale, with 1 being the least damage or least likely to occur and 10 being the most damaging or likely to occur, provides a final measure that can be used to compare the risk associated with different bugs.



EXAM TIP The acronym DREAD refers to a manner of classifying bugs: Damage potential, Reproducibility, Exploitability, Affected user base, and Discoverability.

One of the problems with scoring bugs has to do with point of view. A developer may see a particular bug as hard to exploit, whereas a tester viewing the same bug from a different context may score it as easy to exploit. Damage potential is also a highly context-sensitive issue. This makes detailed scoring of bugs subjective and unreliable. A simple triage method based on a defined set of severities—critical, important, moderate, and low—will facilitate a better response rate on clearing the important issues. A simple structure such as this is easy to implement, difficult for team members to game or bypass, and provides a means to address the more important issues first.

Defects

A defect database can be built to contain the information about defects as they occur. Issues such as where the defect occurred, in what part of the code, in what build, who developed it, who discovered it, how it was discovered, if it is exploitable, etc., can be logged. Then, additional disposition data can be tracked against these elements, providing information for security reviews.

Tracking all defects, even those that have been closed, provides a wealth of information to developers. What has gone wrong in the past, where, and how? The defect database is a tremendous place to learn what not to do and, in some cases, what not to repeat. This database provides testers with ammunition to go out hunting for defects.

Errors

Errors are examples of things gone wrong. They can be of varying levels of severity and impact. Some errors are not a significant issue at the present time, because they do not carry immediate operational risk. But like all other issues, they should be documented and put into the database. This allows them to be included in quality assurance (QA) counts and can help provide an

honest assessment of code quality over time. Errors can be found through a wide variety of testing efforts, from automated tests to unit tests to code walk-throughs. The important issue with errors is collecting the information associated with them and monitoring the metrics.

If testing is a data collection effort aimed at improving the SDL process, then error data collection should not be an effort aimed at punitive results. The collection should enable feedback mechanisms to provide information to the development team so that, over time, fewer errors are made, as the previously discovered and now-understood problems are not repeated. Monitoring error levels as part of a long-term security performance metric provides meaningful, actionable information to improve the efforts of the development team.

Bug Bar

The concept of a bug bar is an operational measure for what constitutes a minimum level of quality in the code. The bug bar needs to be defined at the beginning of the project as a fixed security requirement. Doing this establishes an understanding of the appropriate level of risk with security issues and establishes a level of understanding as to what must be remediated before release. During the testing phase, it is important to hold true to this objective and not let the bar slip because of production pressures.

A detailed bug bar will list the types of errors that cannot go forward into production. For instance, bugs labeled as critical or important may not be allowed into production. These could include bugs that permit access violations, elevation of privilege, denial of service, or information disclosure. The specifics of what constitutes each level of bug criticality need to be defined by the security team in advance of the project so that the testing effort will have concrete guidance to work from when determining level of criticality and associated go/no-go status for remediation.

Detailed requirements for testing may include references to the bug bar when performing tests. For instance, fuzzing involves numerous iterations, so how many is enough? Microsoft has published guidelines that indicate fuzzing should be repeated until there are 100,000 to 250,000 clean samples, depending upon the type of interface, since the last bug bar issue. These types of criteria ensure that testing is thorough and does not get stopped prematurely by a few low-hanging-fruit type of errors.

Risk Scoring

Understanding the potential severity of an error is key to its management. Some errors may have no material effect on the system and as such could be ignored as they will never amount to any risk to the system. Others could be catastrophic, e.g., allowing an attacker to elevate privileges to an administrative level in an undetected fashion. And there is a lot of room between these two levels. Characterizing the particular level of risk between these extremes has been a longstanding problem in the software industry. The MITRE Corporation, under a DHS contract, produced a scoring methodology to go along with its vulnerability enumeration scheme (CVE). The Common Vulnerability Scoring System (CVSS) has evolved into the de facto standard for assessing a “risk score” for a vulnerability. The first version of the standard defined the risk on a 1–10 scale, with 10 being the highest. The factors that were scored included the attack complexity, the privileges required for the attack to be successful, and any required user interactions—in essence things intrinsic to the specific vulnerability without considering external mitigations. This raised concerns from industry members who complained that a vulnerability could be scored a 10, and seemingly be important to fix, but if the system environment precluded the actual attack, was it really a 10?

The next version of CVSS included a more robust scoring mechanism. In addition to the base metrics mentioned previously, a temporal metric group and environmental metric group were added to allow situational customization of the CVSS score, making it more meaningful for enterprises to use. [Figure 13-1](#) illustrates the components of these additional groups of metrics.

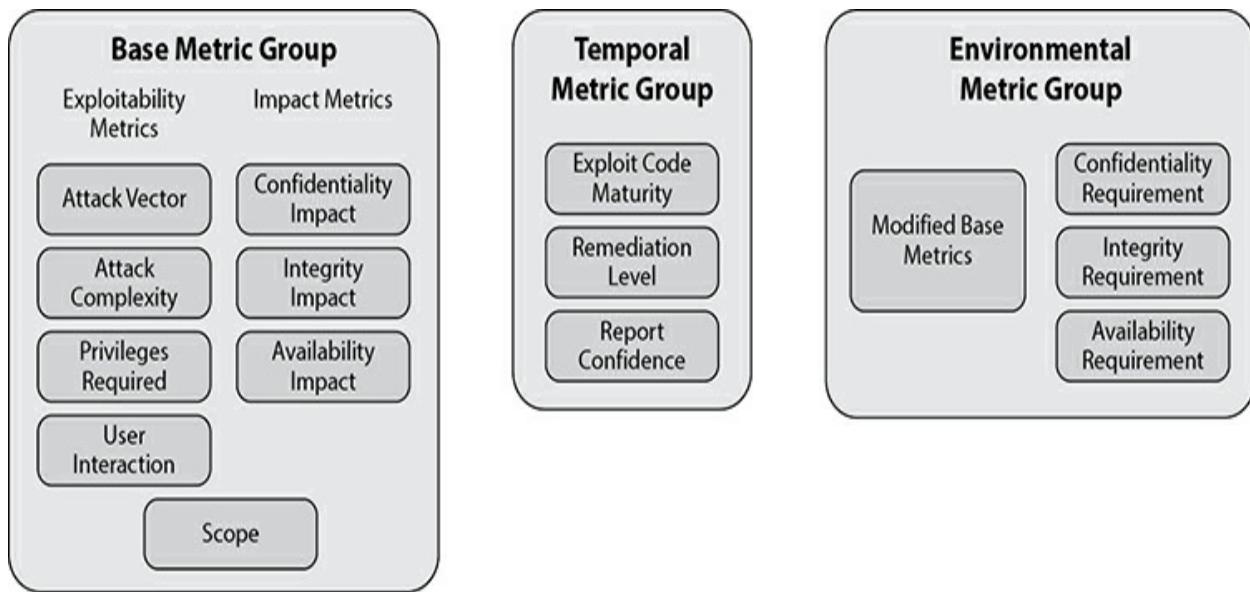


Figure 13-1 CVSS metric groups

The base metric group attempts to mirror the original intent of the CVSS score—an examination of the actual vulnerability and associated risk, without considering the system where the vulnerability resides. These elements are considered to be static over time and across a wide range of environments. As with DREAD and other risk measurements, it has two main elements: exploitability, or how easy it is to do, and impact, or the effect of the vulnerability with respect to risk.

The second set of metrics, the temporal metric group, represents how the risk can change over time. The third set of metrics, environmental metric group, represents characteristics of the vulnerability that may change from system to system based on defenses and system design. The final score is a combination of the base metric as modified by time and environmental components.

Secure Test Data

Testing is a multifaceted process that should occur throughout the development process. Beginning with requirements, use and misuse cases are created and used to assist in the development of the proper testing cases to ensure requirements coverage. As software is developed, testing can occur at various levels—from the unit level where code is first created to the final

complete system, and at multiple stages in between. To ensure appropriate and complete testing coverage, it is important for the testing group to work with the rest of the development team, creating and monitoring tests for each level of integration to ensure that the correct properties are examined at the correct intervals of the secure development process.

Generate Test Data

For testing to be representative of actual conditions, test data should be representative of the actual data being processed by a system. It should also be representative of data inputs that are designed to cause failures. These sets are called *use cases* and *misuse cases*, respectively. Developing data that possesses referential integrity, possesses appropriate statistical qualities, and is representative of both use and misuse cases can be challenging. The following sections explore several methods used to achieve these objectives.

Reuse of Production Data

Testing can require specific useful data to perform certain types of tests. Whether for error conditions or verification of correct referential integrity testing, test data must be created to mimic actual production data and specific process conditions. One manner of developing usable data, especially in complex environments with multiple referential integrity constraints, is to use production data that has been anonymized.

This is a difficult task as the process of truly anonymizing data can be more complex than just changing a few account numbers and names. Managing test data and anonymizing efforts are not trivial tasks and can require planning and process execution on the part of the testing team. Proper cleansing of production data for use in testing requires attention to identifying key fields that require protection, anonymizing the key facts that connect the data to real identities, tokenizing some keys to maintain integrity yet break specific relationships, and examining the effect of data aggregation calculations. Mitigating all the potential relationships while still maintaining meaningful data is more than changing a few names and addresses.

Chapter Review

This chapter opened with an exploration of verification and validation. Software qualification testing and planning followed. An examination of the importance of undocumented functionality and the security implications of test results were the next sections. An exploration of classifying and tracking of security errors was next. In that section, we covered bug tracking, the differences between defects and errors, the use of bug bars, and risk scoring for bugs. The chapter concluded with an examination of generating test data.

Quick Tips

- Validation answers the question, “Are we building the right product?”
- Verification answers the question, “Are we building the product right?”
- Software qualification or acceptance testing is the formal analysis that is done to determine whether a system or software product satisfies its acceptance criteria.
- The software qualification test plan is used to guide software testing efforts.
- Software should be tested at all levels, including the unit or function, subsystems, systems, and the complete product.
- Undocumented and unhandled functionalities can lead to vulnerabilities.
- Errors or bugs should be scored as to impact.
- Errors or bugs should be tracked and managed according to the level of impact associated with them.
- A bug bar is an established risk level above which a bug must be mitigated prior to release of the product.
- Risk scoring is done in several ways, but all include elements of risk impact and chance of risk occurring.
- Test data is needed to conduct realistic use and misuse tests.
- Generating good test data from existing production data is a necessary yet complex and challenging task.

Questions

To further help you prepare for the CSSLP exam, answer the following questions. The correct answers are provided in the following section.

1. What is a list of the types of errors that are not allowed to go forward as part of the SDL process called?

 - A. Bug bar
 - B. Attack surface validation
 - C. Security requirements
 - D. SDL security gate
2. An operational measure of what constitutes the minimum level of quality with respect to security in code is a description of what?

 - A. ISO 9216 process element
 - B. OSSTMM report
 - C. Bug bar
 - D. SDL process requirement
3. What is the objective of tracking bugs?

 - A. Determine the source of bugs
 - B. Ensure that they get addressed by the development team
 - C. Track where bugs are originating in software
 - D. Score developers' coding ability
4. What does a test case describe?

 - A. An output that produces a predictable input
 - B. An input that produces a predictable output
 - C. An algorithm
 - D. A result
5. Qualification testing is always guided by what?

 - A. Prior results
 - B. The customer
 - C. A plan
 - D. A beta test

6. What can changes to code often do?

 - A. Cause other parts of the system to fail
 - B. Identify latent defects
 - C. Lead to product success
 - D. Happen accidentally
7. What is the formal analysis that is performed to determine whether a system or software product satisfies its acceptance criteria called?

 - A. Requirements testing
 - B. Final QA testing
 - C. Release testing
 - D. Qualification testing
8. Software testing provides evidence that the software complies with what?

 - A. The customer's view of what they want
 - B. Legal regulations
 - C. The contract that specifies requirements
 - D. The configuration management plan
9. What is the essential requirement for IV&V?

 - A. Testing
 - B. Reviews
 - C. Audits
 - D. Independence
10. What is the essential element for scoring the severity of bugs/vulnerabilities?

 - A. Use cases
 - B. Difficulty to fix
 - C. Cost to remediate
 - D. Impact

Answers

1. **A.** The concept of a bug bar is an operational measure for what constitutes a minimum level of quality in the code.
2. **C.** A bug bar is a predetermined level of security defect that must be fixed prior to release. Errors of less significance can be either fixed or deferred. Errors that exceed the bug bar threshold must be fixed prior to software release.
3. **B.** Bugs are not always fixed at the time of discovery. Documenting and tracking them are ways to ensure they get put into work cycles for correction at a later point in time.
4. **B.** Test cases align inputs to outputs in a predictable fashion.
5. **C.** Qualification testing is established by a plan.
6. **A.** Changes can cause unintended consequences in other parts of the system.
7. **D.** The formal analysis that is performed to determine whether a system or software product satisfies its acceptance criteria is called qualification or acceptance testing.
8. **C.** The contract defines every aspect of the deliverable.
9. **D.** IV&V has to have independence from development.
10. **D.** Bugs and vulnerabilities both can result in risk to the system, and this risk is the primary concern and the essential element in scoring severity.

PART VI

Secure Software Lifecycle Management

- **Chapter 14** Secure Configuration and Version Control
- **Chapter 15** Software Risk Management

Secure Configuration and Version Control

In this chapter you will

- Learn about secure configuration and version control
 - Learn to define strategy and create a roadmap
 - Manage security within a software development methodology
 - Identify security standards and frameworks
 - Define and develop security documentation
 - Develop security metrics
 - Learn about decommissioning software
 - Report security status
-

Configuration control and version control are key aspects of a secure software development program. Managing security begins with the design, continues through the development process, persists through deployment and use, and culminates at software decommission. Various methods based on best practices and standards are employed to assist in managing the security associated with software across its lifecycle.

Secure Configuration and Version Control

Configuration control is a key aspect of secure software development. *Configuration* refers to the specific elements, such as configuration files, that control how software operates. Configurations are how multifunctional software, which is in essence programmed by external elements, enables features and functionalities at runtime. Configuration management includes

issues related to the configurations of hardware, software, documentation, interfaces, and patching processes.

The development of computer code is not a simple “write it and be done” task. Modern applications take significant time to build all the pieces and assemble a complete functioning product. The individual pieces all go through a series of separate builds or versions. Some programming shops do daily builds, slowly building a stable code base from stable parts. Managing the versions and changes associated with all these individual pieces is referred to as *version control*. Sometimes referred to as *revision control*, the objective is to uniquely mark and manage each individually different release. This is typically done with numbers or combinations of numbers and letters, with numbers to the left of the decimal point indicating major releases, and numbers on the right indicating the level of change relative to the major release.

As projects grow in size and complexity, a version control system, capable of tracking all the pieces, is needed. Suppose you need to go back two minor versions on a config file—which version is it, how do you integrate it into the build stream, and how do you manage the variants? These are all questions asked by the management team that are handled by the version control system. The version control system can also manage access to source files, locking sections of code so that only one developer at a time can check out and modify pieces of code. This prevents two different developers from overwriting each other’s work. This can also be done by allowing multiple edits and then performing a version merge of the changes, although this can create issues if collisions are not properly managed by the development team.

Configuration management and version control operations are highly detailed, with lots of recordkeeping. Management of this level of detail is best done with an automated system that removes human error from the operational loop. The level of detail across the breadth of a development team makes automation the only way in which this can be done in an efficient and effective manner. A wide range of software options are available to a development team to manage this information. Once a specific product is chosen, it can be integrated into the secure development lifecycle (SDL) process to make its use a nearly transparent operation from the development team’s perspective.

Define Strategy and Roadmap

Configuration management involves two separate roles. The customer role is responsible for the maintenance of the product after release. The supplier role is responsible for managing the configuration of the product prior to release, and any associated subcontractor role is involved in the process if the product is developed through a supply chain. The supplier is the entity that maintains the product throughout the development lifecycle, so it is the supplier organization that writes the configuration management plan. However, to maintain continuity with the customer, the configuration plan is written in conjunction with managers from that organization.

Joint creation of the plan ensures that all configuration management responsibilities are fully understood and properly defined and maintained throughout both organizations. From an implementation standpoint, the producer appoints specific overseers who have been given explicit responsibility for ensuring that the requirements of the plan are carried out throughout the development process. Those individuals are usually called *configuration managers*. Besides controlling the configuration during development, the supplier must be able to assure the security and quality of the product. This is achieved by conducting audits, which are carried out independently of the activity of the development team.

A person representing customer issues should be assigned to the configuration management process during development. This individual should have sufficient authority to be able to resolve any configuration control issues that might arise between the producer and the customer. The customer representative is responsible for approving change proposals and concluding agreements about the shape of the configuration as it evolves. The customer representative is also responsible for ensuring the transition between the configuration under development and the management of the eventual product that will be handed to the customer organization. The supplier organization is responsible, through the configuration manager, for ensuring the full understanding and participation of any subcontractor organizations in the maintenance of proper configuration control. All aspects of the producer's agreement with the customer generally apply to the subcontractor and are included in the configuration management plan.

Configuration management incorporates two large processes: configuration control and verification control. These are implemented

through three interdependent management entities. In actual practice, the activities of these three entities must fit the individual needs of each project. These activities are change process management, which is made up of change authorization, verification control, and release processing; baseline control, which is composed of change accounting and library management; and configuration verification, which includes status accounting to verify compliance with specifications.

Manage Security Within a Software Development Methodology

SDLs contain a set of common components that enable the operationalization of security design principles. The first of these components is a current team awareness and education program. Knowledge of security practices and methods, as well as current policies and procedures for the team are prerequisites for being a team member in an SDL environment. The next component is the use of security gates as a point to check compliance with security requirements and objectives. These gates offer a chance to ensure that the security elements of the process are indeed being used and are functioning to achieve desired outcomes. Security gates act as checkpoints in the process to ensure that the process tasks associated with security are completed prior to advancing a project. Three sets of tools—bug tracking, threat modeling, and fuzzing—are used to perform security-specific tasks as part of the development process. The final element is a security review, where the results of the SDL process are reviewed to ensure that all of the required activities have been performed and completed to an appropriate level.

Security is not tied to any particular development methodology. Whether your process is waterfall based or agile or any combination, the process can include security-enhancing elements.

Security in Adaptive Methodologies

Agile methods are not a single development methodology, but a whole group of related methods. Designed to increase innovation and efficiency of small programming teams, agile methods rely on quick turns involving small

increases in functionality. The use of repetitive, small development cycles can enable different developer behaviors that can result in more efficient development. There are many different methods and variations, but some of the major forms of agile development are the following:

- Scrum
- Extreme programming (XP)

XP is built around the people side of the process, while scrum is centered on the process perspective.

Additional agile methods include methods such as Lean and Kanban software development, Crystal methodologies (aka lightweight methodologies), dynamic systems development method (DSDM), and feature-driven development (FDD). The key to securing agile methods is the same as any other methodology. By incorporating security steps into the methodology, even agile can be a secure method. Conversely, any methodology that ignores security issues will not produce secure code.

Adding the necessary security steps into the work process flow, while some will claim slows down the process, will over time save development time. The earlier in the process that an error is caught or prevented, the easier it is to correct. Adding security to agile methods can have some of the greatest payoffs because of the early nature of the intervention when a mistake happens.

Security in Predictive Methodologies

The waterfall model is a development model based on simple manufacturing design. The work process begins and progresses through a series of steps, with each step being completed before progressing to the next step. This is a linear, sequential process, without any backing up and repeating of earlier stages. Depicted in [Figure 14-1](#), this is a simple model where the stages of requirements precede design and design precedes coding, etc. Should a new requirement “be discovered” after the requirement phase is ended, it can be added, but the work does not go back to that stage. This makes the model very nonadaptive and difficult to use unless there is a method to make certain that each phase is truly completed before advancing the work. This can add to development time and cost. For these and other reasons, the waterfall model,

although conceptually simple, is considered by most experts as nonworkable in practice.

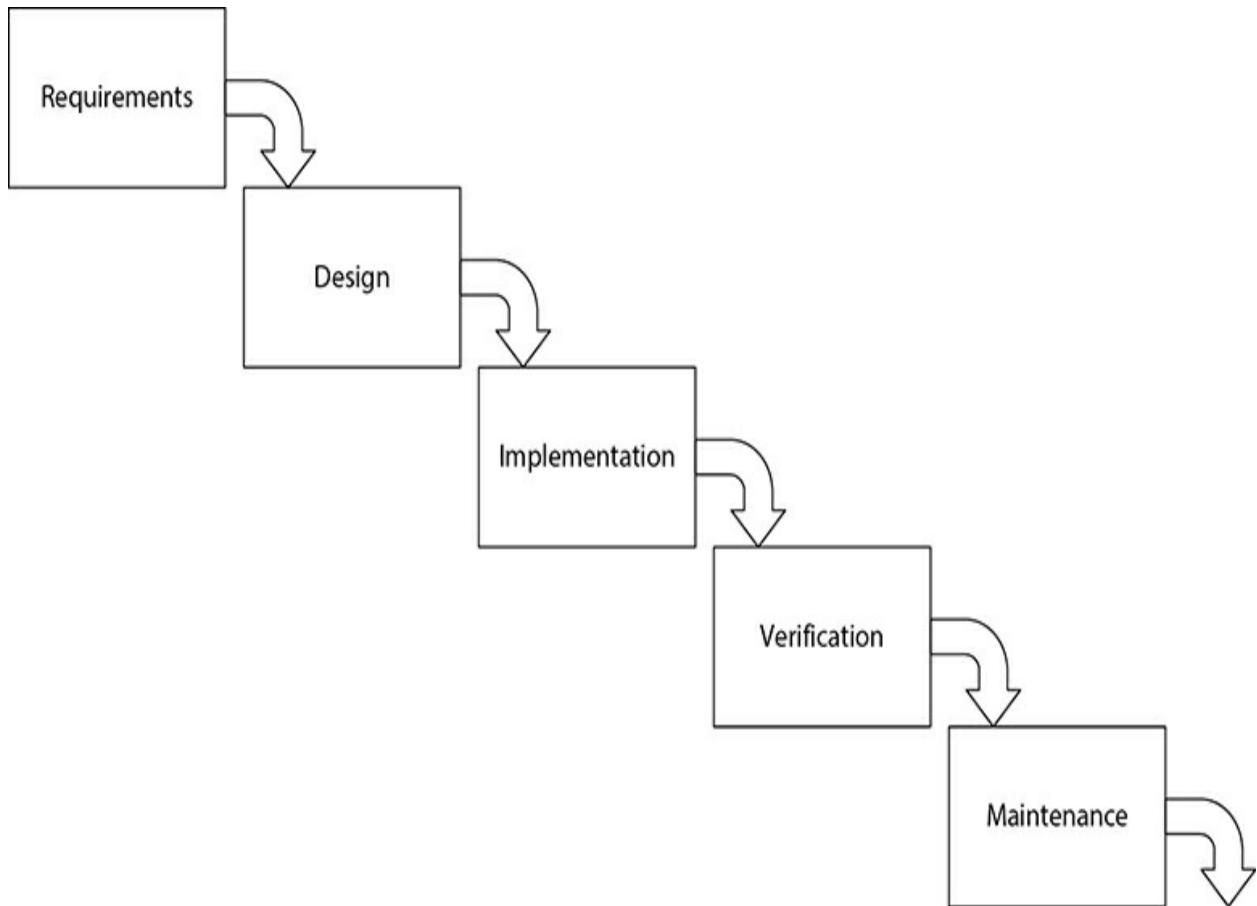


Figure 14-1 Waterfall development model

The waterfall methodology is particularly poorly suited for complex processes and systems where many of the requirements and design elements will be unclear until later stages of development. It is useful for small, bite-sized pieces and in this manner is incorporated within other models such as the spiral and agile methods.

Adding security to a waterfall process works the same as adding the steps to any process. The key to success is in identifying the security issues as early as possible in the process and fixing the issues early, when the impact to the process is minimal. Early intervention is important in waterfall processes, because going backwards in that process methodology is a challenge.

Identify Security Standards and Frameworks

Numerous security standards and frameworks have been mentioned throughout this book, including process-related standards. Each firm needs to have its own set of standards and frameworks that work within their industry and expertise level. There are numerous good sources including the following:

- BSA | The Software Alliance (BSA.org)
- OWASP (OWASP.org)
- SAFECode (Safecode.org)

The objective of using standards and frameworks is to lead to lower numbers and severities of vulnerabilities in released software, as well as develop processes that allow for correcting defects in a timely and safe fashion.

Key practices that need to have structure and guidance from standards and frameworks include the following:

- Define criteria for security checks and gates.
- Design software development process to meet security requirements and mitigate security risks:
 - Configure the compilation and build processes to improve executable security.
 - Protect all code from unauthorized access and tampering by safeguarding all build environments.
 - Provide a mechanism for verifying software integrity by digitally signing the code throughout the software lifecycle.
 - Review, analyze, and/or test all code (including third-party software) to identify vulnerabilities and verify compliance with security requirements throughout the development process.
 - Configure the software to have secure settings by default.
- Develop processes for the release and maintenance of software in a secure fashion:

- Archive and protect each software release.
- Identify, analyze, and remediate vulnerabilities on a continuous basis.

For all of these steps, the key element is processes. Having well-designed processes that provide for the desired activities is vastly superior to depending upon the good graces of your developer team. Having and following defined security-enabled processes is the only proven method of managing the level of vulnerabilities in software.

Define and Develop Security Documentation

Having a set of solid process documents that completely describe the process the firm uses to produce secure software is relatively worthless unless the team performing the tasks has access to and understands the steps. Just as security is not enabled by obscurity, developing secure software requires a level of security transparency. Many teams, including the requirement gatherers, designers, architects, coders, testers, and management, all must work from the same playbook if they are to work together. Having a level of open transparency, where this information is shared and used by all team members, is important.

Numerous security-related documents need to be developed and passed between the different teams and their members. Threat modeling, described in several places in this book, is not for the threat modelers. The threat model that is developed is used as an open communication tool to all, illustrating areas where threats can interact with the software and highlighting where mitigations and other measures need to be taken. The key part of that description is that it is a communication tool. The same goes with bug reports. Without open sharing of previous security issues, a firm is bound to repeat them. Having the information completely describing previous vulnerabilities, how they got there, and what was done to fix them is valuable early in the development cycle.

A well-documented set of security requirements, including coding standards, is valuable but only to the extent that the requirements are known and followed by everyone. Again, an open and sharing culture of this type of

material is essential for success. Test plans, including what is specifically being tested for, is important not just to testers but to the developers and designers as well.

Develop Security Metrics

Part of any management process is measuring objectives. When looking at managing “security” as part of a development process, it is important to have a means of scoring progress. Simple counting of defects corrected is common, but metrics such as this don’t scale well. This leads to the idea of defects per thousand lines of code, which is a normalized metric. Other potentially important items worth measuring include things such as the following:

- Number of repeated errors
- Number of common errors (think top ten list)
- Percentage of items above a certain criticality level
- Remediation time (average?)
- Measure of complexity associated with errors

These metrics attempt to measure what specifically is going wrong and what the costs are associated with the errors. Most of these measures have little spot utility, but the trends over time help improve the process. Using concepts from quality management, the management of defects is a well-known and understood process element. Borrowing metrics from that discipline can add value to the process improvement efforts associated with secure coding expectations.

Decommission Software

The purpose of the software disposal process is to safely terminate the existence of a system or a software entity. Disposal is an important adjunct to security because of the concept of magnetic remanence. In simple terms, old systems retain their information even if they have been put on the shelf. So, it is necessary to dispose of all software and hardware products in a way that ensures that all the information that is contained therein has been secured.

Also, the decision to perform a disposal process will, by definition, cease the active support of the product by the organization. So, at the same time the information is being secured, disposal will guide the safe deactivation, disassembly, and removal of all elements of the affected product. Disposal then transitions the functions performed by the retired system to a final condition of deactivation and leaves the remaining elements of the environment in an acceptable state of operation. The disposal process will typically delete or store the system and software elements and related product deliverables in a sound manner, in accordance with any legal agreements, organizational constraints, and stakeholder requirements. Where required, the disposal process also maintains a record of the disposals that may be audited.

Common sense dictates that the affected software or system can be retired only at the request of the owner. Just as in every other aspect of IT work, a formally documented disposal plan is drawn up to carry out the disposal process. That plan will detail the approach that will be adopted by the operation and maintenance organization to drop support of the system, software product, or service that has been designated for retirement.

In both cases of disposal and retirement, the affected users have to be kept fully and completely aware of the execution of retirement plans. Best practice stipulates that every notification should include a description of the replacement or upgrade of the retiring product with a date when the new product will be made available. The notification should also include a statement of why the software product is no longer being supported and a description of other support options that might be available once support has been dropped. Since a system is usually retired in favor of another system, migration requirements likely will be involved also in the disposal process. These requirements often include conducting parallel operations and training activities for the new product.

When it is time to switch to the new system, the users have to be formally notified that the change has taken place. Also, for the sake of preserving future integrity, all of the artifacts of the old system have to be securely archived. Finally, to maintain a sense of organizational history, all the data and associated documentation that were part of the old system have to be made readily available to any interested users. The procedures for accessing this information are usually spelled out in the plan that provides the guidance for the specific retirement activity.

A disposal strategy has to be provided and disposal constraints have to be

defined to ensure the successful implementation of the disposal process. Once the plan is drawn up and approved, the system's software elements are deleted or stored, and the subsequent environment is left in an agreed-upon state. Any records that provide knowledge of disposal actions and any analysis of long-term impacts are archived and kept available.

Then the software disposal plan is executed. The key aim of this plan is to ensure an efficient transition into retirement. Therefore, users have to be given sufficient timely notification of the plans and activities for the retirement of the affected product. These notifications ought to include such things as a description of any replacement or upgrade to the product, with its date of availability, as well as a statement of why the product will no longer be supported. Alternatively, a description of other support options, once support has been dropped, can also be provided.

It is good practice to conduct parallel operations of the retiring product and any replacement product to ensure a smooth transition to the new system. During this period, user training ought to be provided as specified in the contract. Then, when the scheduled retirement point is reached, notifications are sent to all concerned stakeholders. Also, all associated development documentation, logs, and code are placed in archives, when appropriate. In conjunction with this, data used by or associated with the retired software product needs to be made accessible in accordance with any contract requirements for audits.

End-of-Life Policies

Like all formal IT processes, disposal of software and data should be conducted according to a plan. The plan defines schedules, actions, and resources that terminate the delivery of software services; transform the system into, or retain it in, a socially and physically acceptable state; and take account of the health, safety, security, and privacy applicable to disposal actions and to the long-term condition of resulting physical material and information. Disposal constraints are defined as the basis for carrying out the planned disposal activities. Therefore, a disposal strategy is defined and documented as a first step in the process. This plan stipulates the steps that will be taken by the operations and maintenance organizations to remove active support.

The key element in this strategy is ensuring a smooth transition from the

retiring system, so any planning activities have to include input from the users. The software disposal plan defines for those users when and in what manner active support will be withdrawn and the timeframe for doing that, as well as how the product and its associated elements, including documentation, will be archived. Responsibilities for any residual support issues are defined by the strategy, as well as how the organization will transition to a new product replacement, if that is the eventual goal of the organization. Finally, the method of archiving and ensuring accessibility to relevant records is defined and publicized to all affected stakeholders.

It is important to ensure any specific credentials that were deployed to support the retired software are retired in addition to the software. This requires some advanced planning as credentials tend to live in different lifecycles than code, yet at least one set of credentials must exist until the software has been retired. Configuration items may require adjustment in the enterprise, including things such as holes in firewalls, existing connections between systems, and logging mechanisms.



EXAM TIP End-of-life (EOL) policies should include sunsetting criteria, a notice of all the hardware and software that are being discontinued or replaced, and the duration of support for technical issues from the date of sale and how long that would be valid after the notice of end of life has been published. This allows customers to align their operational timelines appropriately with the developer's product timeline.

Data Disposition

When decommissioning software, there is still the data that needs to be considered. Sometimes the data needs to be retained, as it still has business relevance. Other times the data should be destroyed, because if it no longer has business value, retaining it has a cost and also may entail risks. Data retention decisions do not change just because the software that was using it is decommissioned. Secure data retention, retrieval, and destruction were covered in detail in [Chapter 7](#).

Report Security Status

Management works through the measurement of key activity and action as a result of analyzing results. This means that a set of key process metrics needs to be established to enable management to correctly assess conditions.

Management without metrics is shooting in the dark at unknown problems, and the results frequently cause more troubles than they solve. All projects run on the common “on time, on budget” mantra, but actual management requires more detailed metrics and reports than just these two metrics.

Performance reports that contain information on what the enterprise has determined to be key metrics is a common method of keeping management in the loop when it comes to operations. Understanding the current state of most metrics requires an understanding of historical precedents, and a common reporting method is through the use of run charts that show how the metrics are scoring over time. More modern implementations include this via a dashboard, allowing quick perusal by management, and if properly set up, then clicking a status will drill down to the run chart. To choose appropriate metrics, refer back to the “Develop Security Metrics” section for advice on what provides information on the process and what can be misleading.

Chapter Review

In this chapter, you became acquainted with the management of secure configuration and version control. The chapter explored the definition of strategy for the organization and the roadmap that results from this definition. The objective is to effectively manage the inclusion of security in the software development process. This inclusion of security is independent of the development methodology. The chapter covered the addition of security in the development process in adaptive development methodologies, such as the agile processes used in many enterprises. The chapter also covered the inclusion of security in predictive methodologies such as the older waterfall methodology.

The next topic covered was the need to identify security standards and frameworks that are appropriate for the enterprise that the development is targeting. An examination of common framework sources was presented. Security documentation is important and begins with the definition and documentation of the development process and the security implications. It is

also critical to have appropriate security documentation for the end user of the software under development. Ensuring that the software is employed properly in a secured environment is essential to ensure design objectives carry through to actual implementation.

Developing secure software is a process-driven endeavor. There are a wide range of security metrics that can be employed to understand the state of the process. These include measures such as defects per line of code, criticality level of defects, average remediation time, and measures of complexity. These issues directly impact the topic of reporting status of the process including management reports, dashboards, and managerial feedback.

When software reaches its end of life and is no longer needed, the issues of software decommissioning and end-of-life policies becomes important. Looking at things such as credential removal, configuration removal, license cancellation, and system archiving are included in the end-of-life section. Data disposition including retention, destruction, and data dependencies are also important as part of the software decommissioning process.

Quick Tips

- Configuration management and version control operations are highly detailed, with lots of recordkeeping.
- The management of the various elements of code, files, and settings requires a configuration management/versioning control system to do this efficiently and effectively.
- Configuration management incorporates two large processes: configuration control and verification control.
- Three sets of tools—bug tracking, threat modeling, and fuzzing—are used to perform security-specific tasks as part of the development process.
- Security can be embedded in any development process, be it waterfall based, agile, or any combination—the process can include security enhancing elements.
- Each firm needs to have their own set of standards and frameworks that work within their industry and expertise level.
- The objective of the use of standards and frameworks is to lead to

lower numbers and severities of vulnerabilities in released software, as well as to develop processes that allow for correction of defects in a timely and safe fashion.

- There are numerous security-related documents that need to be developed and passed between the different teams and their members including security requirements, coding standards, threat models, and bug reports.
- There are numerous security metrics that can be employed to understand the state of the development process and software including measures such as defects per line of code, criticality level of defects, average remediation time, and measures of complexity.
- The purpose of the software disposal process is to safely terminate the existence of a system or a software entity.
- The software disposal plan defines schedules, actions, and resources that terminate the delivery of software services; transform the system into, or retain it in, a socially and physically acceptable state; and take account of the health, safety, security, and privacy applicable to disposal actions and to the long-term condition of resulting physical material and information.
- When decommissioning software, data disposition is an issue that requires data be evaluated for retention or destruction.
- Management of the development process includes the use of reports and dashboards so that managerial feedback is targeted to the correct elements of the process.

Questions

To further help you prepare for the CSSLP exam, answer the following questions. The correct answers are provided in the following section.

1. Which of the following is not part of the secure development process?
 - A. Code under development
 - B. Developer team training
 - C. Security gates
 - D. Final security review

2. Security can be added to the development processes in which of the following? (Choose all that apply.)

 - A. Waterfall model-based process
 - B. Agile-based processes
 - C. Scrum-based processes
 - D. Spiral development processes
3. Which of the following is a poor security metric?

 - A. Errors per thousand lines of code
 - B. Number of errors
 - C. Average time to remediation
 - D. Percentage of issues above a specific criticality level
4. Why is it important to decommission old systems that are not being used and will not be used again?

 - A. Operational costs
 - B. Clear server space
 - C. Systems retain information
 - D. Free up system administrators
5. What is the primary purpose of security documentation?

 - A. Communication of security information
 - B. Reduce liability
 - C. Tell users what not to do
 - D. Attest to the security level of the software
6. A software disposal plan defines schedules, actions, and resources of which of the following?

 - A. Terminate the delivery of software services
 - B. Transform the system into, or retain it in, a socially and physically acceptable state
 - C. Take account of the health, safety, security, and privacy applicable to disposal actions
 - D. All of the above

7. What is the objective of using standards and frameworks in designing software development processes?

 - A. All of the below
 - B. Lead to lower numbers of defects
 - C. Reduce severity of vulnerabilities
 - D. Provide for timely and safe correction of defects
8. Reports for managing security status typically include the following except what?

 - A. Daily reports of errors found and corrected as part of the process
 - B. Percentage of issues identified as critical
 - C. Time to remediate issues
 - D. Tracking specific errors, such as repeat errors and high visibility errors
9. Managing the versions and changes associated with all the individual pieces of a software project is referred to as what?

 - A. Version control
 - B. Secure development process
 - C. Configuration management
 - D. Source code repository
10. Version control is best performed by what?

 - A. A manual, structured process with checks
 - B. An automated, transparent process
 - C. Each developer taking responsibility
 - D. The project manager or lead developer to ensure consistency

Answers

1. A. The code being developed is not part of the process; it is what the process produces.
2. **A, B, C, and D.** Security checks can be applied in any development process.

3. **B.** Raw error counts do not scale with project size, complexity, or other issues.
4. **C.** Old systems still can retain information that could at a later date be released, causing risk. The other answers can be ignored in many enterprises.
5. **A.** Security documentation is used to communicate to all parties relevant security issues, including risks, configuration issues, vulnerabilities that require mitigations, etc.
6. **D.** The software disposal plan defines schedules, actions, and resources that terminate the delivery of software services; transform the system into, or retain it in, a socially and physically acceptable state; and take account of the health, safety, security, and privacy applicable to disposal actions and to the long-term condition of resulting physical material and information.
7. **A.** The objective of the use of standards and frameworks is to lead to lower numbers and severities of vulnerabilities in released software, as well as develop processes that allow for correction of defects in a timely and safe fashion.
8. **A.** Daily reports of errors tend to be noisy and not connected to key process performance issues. All of the other answers are more specific to elements of key importance to customers and management.
9. **A.** The managing of the versions and changes of all the components of a project is referred to as version control.
10. **B.** Configuration management and version control operations are highly detailed operations, with lots of recordkeeping. Management of this level of detail is best done with an automated system that removes human error from the operational loop.

Software Risk Management

In this chapter you will

- Explore integrated risk management
 - Learn about promoting security culture in software development
 - Learn about implementing continuous improvement in software development
-
-

Risk management is an important aspect of driving a system to get the desired results. There are lots of elements of risk management, and organizations create risk management frameworks to help them manage and coordinate the sources of risk and resultant activities. Although everything has risk, through the use of proper processes and procedures, supported by corporate culture and backed by a continuous improvement program, risks can be managed to tolerable levels.

Incorporate Integrated Risk Management

Organizations manage risk all the time. When it comes to managing risk across an enterprise, from numerous sources of different levels and frequency, some form of risk management organization is needed. Integrated risk management (IRM) is the use of a set of practices and processes supported by enabling technologies to do just that. Add in a risk-aware culture and one can further improve decision-making and performance. IRM is the framework that enables an integrated view of risk across the organization. There are many different types and instantiations of IRM, and it is important to connect the risk management efforts of the software development process into the specific IRM employed in the enterprise. This has several advantages, foremost being the creation of management reports in

a form that management is used to seeing and acting upon. Second, most IRM solutions are fairly comprehensive, which tends to pull more information from a process than an ad hoc risk management reporting system.

Regulations and Compliance

Management has a responsibility for ensuring compliance with a wide range of requirements that are associated with the organization's business objectives and the actions they take to achieve them. These requirements have many sources—some are contractual, some are based on policy or strategic initiatives. Others may be process based, defined by the organization or industry. There are also external sources of requirements in the form of regulations or laws.

Compliance is the term typically used when referring to the activities associated with these outside requirements. *Conformance* is the term typically used when referring to the activities associated with internal requirements (organizational policies and standards).

Compliance and conformance efforts are frequently a key issue with respect to governance, risk management, and compliance (GRC) efforts. Activities related to compliance are usually given priority over conformance. There are a variety of reasons for the prioritization, but the principal reason is related to the penalties associated with noncompliance. While management actions that run counter to conformance may have internal costs in the form of dissonance, failure to comply with external regulations or legal requirements frequently carries a financial penalty.

Legal

Governance includes the act of managing legal-driven risk elements. Two specific legal issues that have significant risk to an enterprise are intellectual property and data breach events. Intellectual property is a valuable asset of the firm, and one that requires appropriate protection. In some cases, this protection can be obtained through legal action and the courts. But in other cases, the legal mechanism has no party to act against. When intellectual property is stolen by unknown criminal elements, using the Internet and international borders to avoid prosecution, it is still lost. Intellectual property

needs prevention controls in addition to the legal remedies available after loss.

When losses involve personally identifiable information (PII), additional legal issues become involved. Many states have data breach laws, with disclosure and response provisions. Two strategies can be employed with respect to PII. First are the actions taken to protect the data prior to potential loss. Encryption is the primary method employed to protect data confidentiality and also serves to meet the specifications of many data breach laws and requirements, including the Payment Card Industry Data Security Specification (PCI DSS). One of the economic drivers is the cost of complying with data breach laws notification provisions.

Second, when senior executives weigh the options for dealing with risk, legal issues and consequences play a role in determining the appropriate balance in actions. Legal consequences, whether from compliance failure or loss, are part of the overall risk equation and should be included in the decision process.

Standards and Guidelines

Standards are established norms used to define a specific set of rules governing some form of behavior. Standards exist for a wide range of elements, from business processes to outcomes. The sources of standards are many, including government bodies and industry and trade organizations. The ultimate goal of standards is to define a set of rules associated with ensuring a specified level of quality. It is important for a CSSLP to have a solid working knowledge of the relevant security standards, as this is the blueprint for designing, creating, and operating a system that reflects best practices. Each enterprise will adopt processes aligned with standards and frameworks that are appropriate for their products and processes. There are a wide range of applicable standards for software development including ones from the International Organization for Standardization (ISO), Payment Card Industry (PCI), National Institute of Standards and Technology (NIST), OWASP, Software Assurance Forum for Excellence in Code (SAFECode), Software Assurance Maturity Model (SAMM), and Building Security In Maturity Model (BSIMM). Going into details for each of these standards is a book in itself, but some obvious things are worthy of mention. If one is building a web application, then the Open Web Application Security Project

is key. If credit card payments are involved, then PCI standards become essential, etc. In most cases, several different standards will be involved, each covering their appropriate aspect of the system.

Risk Management

When risks are identified, management has a series of options to deal with them. The first option is to fix the problem. Fixing the problem involves understanding the true cause of the vulnerability and correcting the issue that led to it. When available, this is the preferred option, as it solves the problem irrespective of external changes. This option is easier the earlier the problem is found in the development cycle. The second method involves removing the problem. This can be done in a couple of ways: If the problem is associated with a particular feature, then removing the feature may remove the problem. If the problem is associated with a particular standard—that is, confidentiality associated with cleartext protocols—removing the communication or protocol may not make sense and fixing the protocol is probably not possible; what remains is some form of compensating control. Adding encryption to the communication channel and removing the cleartext disclosure issue is a form of removing the problem. In both of these cases, there is an opportunity for some form of residual risk, and for that, management needs to make a similar decision. Can we live with the residual risk? If the answer is no, then you need to repeat the corrective process, removing additional levels of risk from the residual risk.



NOTE Residual risk is the remaining risk after options to avoid, remove, or mitigate risk have been employed. Residual risk by definition is risk that is accepted.

Other options exist for dealing with the risk. Transferring the risk to another party is an option. This can be in the form of a warning to a user or transferring the risk to the user or to a third party. In the case of financial fraud associated with online credit cards, the merchant is typically protected, and the bank card issuer ends up covering the fraud loss. This cost is

ultimately borne by the customer, but only after aggregating and distributing the cost across all customers in the form of increased interest rates. It is also possible in some cases to purchase insurance to cover risk, in essence transferring the financial impact to a third party.

The last option is to do nothing. This is, in essence, accepting the risk and the consequences associated with the impact should the risk materialize. This is a perfectly viable option, but it is best only when it is selected on purpose with an understanding of the ramifications. Ignoring risk also leads to this result, but it does so without the comprehensive understanding of the potential costs.



EXAM TIP There are four options associated with how risk is handled: mitigate, accept, transfer, and avoid.

Not all risks are created equal. Some bugs pose a higher risk to the software and need fixing more than lesser risk-related bugs. Use of a system such as DREAD in the gating process of bug tracking informs us of the risk level ($\text{Risk} = \text{Impact} \times \text{Probability}$) associated with bugs that pose the greatest risk and therefore should be fixed first. The mitigation process itself defines how these bugs can be addressed.

When bugs are discovered, there is a natural tendency to want to just fix the problem. But as in most things in life, the devil is in the details, and most bugs are not easily fixed with a simple change of the code. Why? Well, these errors are caught much earlier and rooted out before they become a “security bug.” When a security bug is determined to be present, four standard mitigation techniques are available to the development team:

- Do nothing.
- Warn the user.
- Remove the problem.
- Fix the problem.

The first, do nothing, is a tacit acknowledgment that not all bugs can be

removed if the software is to ship. This is where the bug bar comes into play; if the bug poses no significant threat (i.e., is below the critical level of the bug bar), then it can be noted and fixed in a later release. Warning the user is in effect a cop-out. This pushes the problem on to the user and forces them to place a compensating control to protect the software. This may be all that is available until a patch can be deployed for serious bugs or until the next release for minor bugs. This does indicate a communication with the user base, which has been shown to be good business, because most users hate surprises, as in when they find out about bugs that they feel the software manufacturer should have warned them of in advance.

Removing the problem is a business decision, because it typically involves disabling or removing a feature from a software release. This mitigation has been used a lot by software firms when the feature was determined not to be so important as to risk an entire release of the product. If a fix will take too long or involve too many resources, then this may be the best business recourse. The final mitigation, fixing the problem, is the most desired method and should be done whenever it is feasible given time and resource constraints. If possible, all security bugs should be fixed as soon as possible, because the cascading effect of multiple bugs cannot be ignored.

Terminology

A threat is any circumstance or event with the potential to cause harm to an asset. For example, a malicious hacker might choose to hack your system by using readily available hacking tools. A vulnerability is any characteristic of an asset that can be exploited by a threat to cause harm. Your system has a security vulnerability, for example, if you have not installed patches to fix a cross-site scripting (XSS) error on your website.

Residual risk is the risk that remains after a control is utilized and reduces the specific risk associated with a vulnerability. This is the level of risk that must be borne by the entity.

Controls are defined as the measures taken to detect, prevent, or mitigate the risks associated with the threats a system faces. Controls are also sometimes referred to as countermeasures or safeguards. They can be grouped in classes associated with several types of actions: administrative, technical, or physical. For each of these classes of controls there are four types of controls: preventative, detective, corrective, and compensating. So a

control can be described by the type of action (class) and how it affects the risk element.

Impact is the loss resulting when a threat exploits a vulnerability. A malicious hacker (the threat) uses an XSS tool to hack your unpatched website (the vulnerability), stealing credit card information that is used fraudulently. The credit card company pursues legal recourse against your company to recover the losses from the credit card fraud (the impact). Sometimes an asset will experience less than a total loss, or total risk, and the term *exposure factor* is used to describe a measure of the magnitude of loss of an asset. Exposure factor is commonly used in calculating the single loss expectancy metric.

Technical Risk vs. Business Risk

Software development is, by nature, a technological endeavor. A myriad of technologies are involved in the development of software, and with this array of technologies comes risks. Some of the risks will be associated with the technology employed as part of the development process. Another set of risks is associated with the aspects of the software functionality.

It is not possible to identify all sources of risk in a business. In software engineering, risk is often simplistically divided into two areas: business risk and, a major subset, technology risk. Business risk is associated with the operation of the business as a business.

Promote Security Culture in Software Development

Software development is a team sport, driven by a process that results in the desired outputs. But teams are made of people, and people have a human side that is not driven just by rules and regulations, but by desire. Aligning the team member's desires and motivations with the enterprise goals and objectives is not an easy task. One tool that will work to this regard is the creation of a work culture that guides the team members. If the work culture includes security, then security will become part of the work output of the team. There is a note of caution that needs to be mentioned. You cannot fake culture. Workers can't be told to adopt a culture; the culture has to exist from

all who are on the team, and as new people join the team, they can join the culture. If management does not walk the walk and actively participate in the culture, the result is the workers will know it, they will know the culture is not real, and the efforts the culture should reinforce will be seen as not valued.

Security Champions

Security champions are just that, people who are there to lead and show the way to others when visibility is lost. Security champions are not necessarily management; in fact, some need to be in the normal teams as members. They exist as a resource, the place someone goes if they are lost or don't know what to do—so co-workers can be easier to approach than managers. They don't have to have all the answers, but they need the experience to know where to get the answers. Leadership is key, because champions need to be more than just knowledgeable; they need to be people whom others will accept and follow.

The true role of the champion is to be the cheerleader who keeps the team, morale, and culture all aligned. There will be times when security becomes a problem—maybe for a specific instance or team member or requirement or deliverable. The champion's job is to keep the team on track with respect to what is important. And when that means security has to change or adapt, it is the champion's job to drive the correct change.

Security Education and Guidance

All team members should have appropriate training and refresher training throughout their careers. This training should be focused on the roles and responsibilities associated with each team member. As the issues and trends associated with both development and security are ever-changing, it is important for team members to stay current in their specific knowledge so that they can appropriately apply it in their work.

Security training can come in two forms: basic knowledge and advanced topics. Basic security knowledge, including how it is specifically employed and supported as part of the SDL effort, is an all-hands issue, and all team members need to have a functioning knowledge of this material. Advanced topics can range from new threats to tools, techniques, etc., and are typically

aimed at a specific type of team member (e.g., designer, developer, tester, project manager). The key element of team awareness and education is to ensure that all members are properly equipped with the correct knowledge before they begin to engage in the development process. Education or training with respect to the specific processes employed, while seemingly not needed for many skilled new hires, is essential to keep everyone on the same page with respect to process. Work processes vary from firm to firm and at times even within firms, and having everyone understand the work process is important.

Implement Continuous Improvement

Continuous improvement is the process of improving processes to improve business results. An old adage is never buy version 1.0 of anything. Always wait for the bug fixes. This should be part of all processes, and not a reactive step, but a proactive one. Examining progress as a process works, measuring results, and finding ways to improve the process are part of all world-class and best-in-class methods. When something doesn't go as planned or the results, while OK, were not as good as expected, there is a time for a retrospective look at how one got there. A key lesson learned by the U.S. military during the Vietnam War was that when conducting after-action reviews, rank could have no place at the table. If someone just returned from a sortie and said the enemy was behind hill X, it does no good to get into a fight with a higher-ranking person who claims otherwise. When conducting a review of anything, rank, egos, and anything not factually related to the problem needs to be checked at the door. Having extraneous inputs will steer the team from the goal, which is process improvement.

Lessons learned are just experiences that hold information. Deciphering those into root causes may take some digging, and it is worth it to dig, because effective continuous improvement acts upon root causes, not symptoms. There can be confounding information and issues that can mask or hide the real causes, and unless the team digs through those issues, the results of improvement activities will not bear the desired results.

Continuous improvement is the method of controlling change to the process and change that results in better outcomes. Long term, so many things change in the software development environment: technology, threats, regulations, business practices, and more. A solid continuous improvement

process helps the process to stay abreast of those changes and also to improve results in desired metrics. Continuous improvement is one of the most important elements for long-term survival as it provides a mechanism to control the process and fix deficiencies and items that are less than optimal.

Chapter Review

This chapter opened with a review of incorporating integrated risk management elements including regulations and compliance as well as legal concerns. The chapter then explored standards and guidelines. The next section covered the integration of risk management from software development into the enterprise risk management system. The next section covered terminology including threat, controls, impact, and exposure factors. The section on risk management concluded with an examination of technical versus business risk.

Examination of the role of security culture in an organization and its role in software development opened the next major section. The importance and role of security champions were covered. The importance and specifics about training and education of team members were covered. The final topic was implementing continuous improvement.

Quick Tips

- Integrated risk management is the use of a set of practices and processes supported by enabling technologies to assist management in managing risk across the enterprise.
- Management has a responsibility to ensure compliance with a wide range of requirements including regulations, contracts, and legal items that demand compliance.
- Standards are established norms used to define a specific set of rules governing some form of behavior.
- There are a wide range of applicable standards for software development including ones from the ISO, PCI, NIST, OWASP, SAFECode, SAMM, and BSIMM.
- Residual risk is the remaining risk after options to avoid, remove, or mitigate risk have been employed.

- There are four options associated with how risk is handled: mitigate, accept, transfer, and avoid.
- A threat is any circumstance or event with the potential to cause harm to an asset.
- A control is a measure taken to detect, prevent, or mitigate the risk associated with a threat.
- Security champions are leaders who act as resources, the place team members can go if they are lost or don't know what to do—they don't need to have all the answers, but they need the experience to know where to get the answers.
- All team members require training with respect to their roles and responsibilities as process-specific knowledge is required to be a contributing member in the manner that the work is performed.
- Continuous improvement is the process of improving processes to improve business results.

Questions

To further help you prepare for the CSSLP exam, answer the following questions. The correct answers are provided in the following section.

1. Which of the following is a standard/guideline for software development?
 - A. SafetyCode
 - B. OHornet
 - C. Payment Card Industry
 - D. All of the above
2. What is the type of risk that can never be mitigated?
 - A. Technology risk
 - B. Residual risk
 - C. Accepted risk
 - D. Transferred risk
3. Standard mitigation options for handling bugs include all of the following except what?

- A. Do nothing
 - B. Remove the problem
 - C. Warn the user
 - D. Change the requirements
4. Which of the following is defined as any circumstance or event with the potential to cause harm to an asset?
- A. Attacker
 - B. Risk
 - C. Vulnerability
 - D. Threat
5. What are the four options to handle risk?
- A. Mitigate, accept, transfer, and avoid
 - B. Migrate, accept, transfer, and avoid
 - C. Mitigate, accept, transfer, and refuse
 - D. Mitigate, accept, transfer, and residual
6. What is the technical term for the loss resulting from a threat exploiting a vulnerability?
- A. Impact
 - B. Single loss expectancy
 - C. Exposure factor
 - D. Total loss
7. What is the role of the security champion in the development process?
- A. A resource of security knowledge
 - B. Security task manager
 - C. Security tester
 - D. Security cop
8. Why can compliance with a regulation be critical?
- A. Failures can result in government risk compliance (GRC) actions.
 - B. Failures can result in external costs.

- C. Failures can result in internal costs such as management dissonance.
 - D. None of the above
9. What can risk sources include?
- A. Regulation/legal-driven elements
 - B. Process-driven elements
 - C. Threat-driven elements
 - D. All of the above
10. To control risk over the long term, which element is most important?
- A. A solid continuous improvement process
 - B. Better education and training of workforce
 - C. Better testing of software before release
 - D. Compliance with regulations

Answers

- 1. C. The Payment Card Industry (PCI) issues contractual regulations on security measures for systems that handle bank card data.
- 2. B. Residual risk is the risk that remains after all steps to reduce it have been taken.
- 3. D. When a security bug is determined to be present, four standard mitigation techniques are available to the development team: do nothing, warn the user, remove the problem, or fix the problem.
- 4. D. A threat is any circumstance or event with the potential to cause harm to an asset.
- 5. A. The options for handling risk are mitigate, accept, transfer, and avoid.
- 6. A. Impact is the term used to describe the loss resulting from a threat exploiting a vulnerability on an asset.
- 7. A. The security champion is a resource, knowledgeable in security, to assist the team in working security issues.
- 8. B. Failures in compliance issues mean the firm has failed an eternal rule or regulation, which typically has a penalty associated with its failure.

- 9.** **D.** Risk can enter the equation with virtually any activity, so all of the options (regulation/legal-driven elements, process-driven elements, and threat-driven elements) can be sources of risk.
- 10.** **A.** A solid continuous improvement program over time can resolve virtually all impediments to the business.

PART VII

Secure Software Deployment, Operations, Maintenance

- [**Chapter 16**](#) Secure Software Deployment
- [**Chapter 17**](#) Secure Software Operations and Maintenance

CHAPTER 16

Secure Software Deployment

In this chapter you will

- Learn how to perform operational risk analysis
 - Explore how to release software securely
 - Learn to securely store and manage security data
 - Examine ways to ensure secure installation
 - Examine how to perform post-deployment security testing
-
-

Software deployment involves several significant activities if the objective is to release secure software. An operational risk analysis is needed to understand the threat environment, and this information is used to ensure that the deployment system protects the code and the data during deployment. Ensuring that software is released securely and that the end users can have confidence in the level of risk associated with their installation is something that needs to be managed prior to deployment.

Perform Operational Risk Analysis

Because of the number of threats in cyberspace and the complexity of the product, risks are a given in any delivered system or software product. As a result, formal procedures must be in place to ensure that risks are properly accounted for in the final acceptance. Risks comprise a range of properties that must be considered to stay secure within acceptable limits. Those properties include the following:

- Implicit and explicit safety requirements
- Implicit and explicit security requirements

- Degree of software complexity
- Performance factors
- Reliability factors

Mission-critical software, requiring a high degree of integrity, also requires a large and rigorous number of tasks to ensure these factors. These tasks are normally defined early in the development process and are enshrined in the contract. Making that assurance explicit is important to the process because, in the normal project, there will be many more risks involved than there will be resources to address them. It is critical to be able to document in the contract a process for identifying and prioritizing risks.

The evaluation process that underlies risk acceptance should always answer two questions: what is the level of certainty of the risk, which is typically expressed as likelihood, and what is the anticipated impact, which is normally an estimate of the loss, harm, failure, or danger if the event does happen? Traditionally, these are the two factors that determine which risks are acceptable and which ones aren't. These two questions should be approached logically. Practically speaking, the first consideration is likelihood, since a highly unlikely event might not be worthy of further consideration. However, the estimate of the consequences is the activity that truly shapes the form of the response. That is because there is never enough money to secure against every conceivable risk, so the potential harm that each risk represents always has to be balanced against the likelihood of its occurrence and prioritized for response.

Therefore, the basic goal of the risk assessment process is to maximize resource utilization by identifying those risks with the greatest probability of occurrence *and* that will cause the most harm. These options are then arrayed in descending order of priority and addressed based on the available resources. Since all of the decisions about the tangible form of the processes for software assurance will depend on getting these priorities absolutely correct, it should be easy to see why a rigorous and accurate risk assessment process is so critical to the overall goal of effective risk acceptance. However, software is an intangible and highly dynamic product, so without a well-defined process, it is difficult to assign priorities with absolute confidence.

It is easy to have confidence in priorities that are set in the physical world —a civil engineering problem, for instance. However, confidence is bound to diminish if the estimate is based on something as complex and hard to

visualize as software. As a result, risk acceptance assessments for software have to be built around concrete evidence. That tangible proof is usually established through tests, reviews, and analysis of threats, as well as any other form of relevant technical or managerial assessment.

Because the threats to software are extremely diverse, the data collection process has to be systematic and well-coordinated. As a result, risk acceptance assessments should always embody a commonly accepted and repeatable methodology for data collection that produces reliable and concrete evidence that can be independently verified as correct. The gathering, compilation, analysis, and verification of all data that is obtained from tests, reviews, and audits can be a time-consuming and resource-intensive process. To ensure the effectiveness and accuracy of any given risk analysis, the practical scope of the inquiry has to be precisely defined and should be limited to a particular identified threat.

Therefore, it is perfectly acceptable to approach the understanding of operational risk in a highly compartmentalized fashion, as long as the organization understands that the results of any particular risk assessment characterize only part of the problem. In fact, the need for a detailed, accurate picture of all conceivable threats almost always implies a series of specifically focused, highly integrated risk assessments that take place over a defined period, rather than a single monolithic effort. The assessments typically target the various known threats to the electronic, communication, and human-interaction integrity of the product. The insight gained from each focused assessment is then aggregated into a single comprehensive understanding of the total impact of a given threat, which serves as the basis for judging how it will be mitigated or accepted.



EXAM TIP Risk assessment requires detailed knowledge of the risks and consequences associated with the software under consideration. This information is contained in a properly executed threat model, which is created as part of the development process.

Nevertheless, targeted risk assessments only drive acceptance decisions about specific, known vulnerabilities at a precise point in time. That explicit

understanding is called a *threat picture*, or situational assessment. However, because threats are constantly appearing on the horizon, that picture has to be continuously updated. Consequently, risk acceptance assessments are always an ongoing process. The assessment should maintain continuous knowledge of these three critical factors:

- The interrelationships among all of the system assets
- The specific threats to each asset
- The precise business and technological risks associated with each vulnerability

These factors are separate considerations, in the sense that conditions can change independently for any one of them. However, they are also highly interdependent in the sense that changes to one factor will most likely alter the situation for the other two. Moreover, the same careful investigative process has to be followed to track every risk after it has been identified. That is because of the potential for latent threats to become active. A latent threat might not have immediate consequences because the conditions that would make it harmful are not present yet. As a consequence, latent threats are normally disregarded in the real-world process of risk acceptance. This is understandable, since resources ought to be concentrated on the threats that are known to cause significant harm. However, a latent threat can become an active one if those conditions change. Therefore, all latent threats that could exploit a known vulnerability have to be tracked.

Deployment Environment

Software will be deployed in the environment as best suits its maintainability, data access, and access to needed services. There are a wide range of potential deployment environments, from cloud to virtual machines, containers, bare-metal servers, on-prem, off-prem...the list can go on. Ultimately, at the finest level of detail, the functional requirements that relate to system deployment will be detailed for use in specific systems.

An example is the use of a database and web server. Corporate standards, dictated by personnel and infrastructure services, will drive many of the selections. Enterprises will dictate the deployment environments in concert with their other concerns, as they have to maintain the new system alongside

others in their data centers. Although there are many different database servers and web servers in the marketplace, most enterprises have already selected an enterprise standard, sometimes by type of data or usage. Choice of underlying OS and other factors also play a role in the selection. Understanding and conforming to all the requisite infrastructure requirements are necessary to allow seamless interconnectivity between different systems.

Personnel Training

Training is a key element in properly operating software. While many think that this step “can just be learned on the job,” there are crucial elements associated with setting up and configuring the software that can have significant impacts on the security of the system. There are typically three groups of people with different training needs. The administrators of the system, be it in a system administrator role, a database administrator role, or an application administrator role, play a key role in establishing the correct configuration of the system and users. These people will require specific training so that they can set up and initialize the system in the proper form for it to function properly and securely.

The next set of users are the power users of the system. These are the people who normal users will turn to with questions on how to do X, or “can I do Y?” These specific users need more than simple “which menu option to click” knowledge. They need to understand how the system is being implemented and why, as well as thorough training on the use of the system. This way they can provide knowledgeable and correct answers to users’ queries. Because most modern software is highly configurable, it is important that this group has an understanding of the actual configuration being employed and any implications as a result of the choices made during the configuration.

The last set is the standard users. This is the largest group and also an important constituency. The relative success or failure of a new software installation rests on these people, because they will be the ones using it. The new software either is a new business process or is replacing a previous set of software/tools, and in either case winning over this group in the form of acceptance is important for the efficient operation of the system. This user base requires specific training on the tasks they will be performing on the software. They don’t need all the details on how it is configured and what it

does, but they do need to know how to do their business responsibilities using the new system.

Safety Criticality

In some enterprises, software applications can be involved in operations that are designated as safety critical. Safety-critical applications operate in a more restrictive environment than nonsafety critical ones. Elements such as risk management, change control, and operations get special focus because of the safety component of risk. If a software system is being developed to operate in a safety-critical environment, then elements such as logging, configuration control, and other reliability elements are important. The software product needs to integrate into the working environment of the safety-critical environment, and this includes increased monitoring and reporting on any issue that might affect reliability.

System Integration

Large systems are composed of a highly complex set of integrated components. These components can be integrated into a system-of-systems arrangement to perform some particular dedicated function, such as theater control for the military or high-performance computing in a research lab. In all cases, the aim of system-of-systems integration is to produce synergy from the linkage of all the components. Obviously, the linkage of a distributed set of concurrent processing systems into a single synergistic system-of-systems arrangement requires a high degree of coordinated engineering and management effort.

System-of-systems arrays are not specifically supply chain issues, in that they do not involve classic movement up and down a supply chain ladder. However, the component systems of that array are integrated from smaller components, and the system of systems itself involves a complex interoperability concern. So it is in the integration itself that the rules for proper supply chain risk management become relevant. And in that respect, the issues of assurance of integrity in the upward integration and assurance of concurrent processing effectiveness are the same for systems of systems as they are for products that evolve from a simple supply chain.

Since ensuring concurrent interoperability is at the heart of system-of-

systems assurance, the primary responsibility for achieving the necessary synergy falls within the domain of system engineering. That domain is primarily concerned with methods for identifying and characterizing, conceptualizing, and analyzing system-of-systems applications. That analysis is supported by abstract modeling techniques such as continuous system modeling, agent-based modeling, and Unified Modeling Language (UML). The practical outcome of that conceptual modeling is then guided by the standard principles for proper design of any system, such as abstraction, modularity, and information hiding. These principles apply to system-of-systems integration control from the component level all the way up to architectural and data-based design.

Release Software Securely

When software has finished the development process, it still needs to be delivered to the customer for use. Delivery or deployment of software is technically still part of the development process and needs to be done in a fashion that preserves all the security and integrity that has been built into the system. There are many ways to release and deploy software, and it can be viewed as the delivery of code and instructions to the customer for them to instantiate. This means that the information must meet the needs of the customer with respect to the processes they employ to deploy software. One of the more robust and comprehensive deployment methods is the DevOps method.

DevOps is a form of software maintenance where changes are introduced virtually directly into production. On its face, this seems crazy and fraught with risk, but the processes involved in DevOps include highly refined and automated processes to reduce risk. Risk can come from large software changes that create significant changes in a program, as opposed to smaller, more focused changes. Rather than wait until a significant number of changes are collected, DevOps works on the principle of applying changes as they are created. DevOps was developed in response to the slow change that quarterly, semiannual, or annual update cycles force on the marketplace. With these larger changes, the level of testing required to ensure against different element interactions became one of the major time drivers. Many small changes were faster to implement, faster to test, and faster to correct if issues arose. One of the major safeguards that is used in DevOps is a high level of

automation to implement changes and back out changes if they fail.

DevOps has become the principal method of advancing large technological platforms with multitudes of redundant servers. In Gmail, Google has deployed tens of thousands of Gmail servers worldwide to manage the vast e-mail network. As changes are rolled out to this network, they are done so incrementally. Only portions of the system are changed at a given time. Other automated tasks manage the actual change process, including the implementation of the change, post-change operational testing, and automated rollback (including subsequent halting of the change process across other servers) when issues arise. This enables developers to incrementally fix and evolve code. DevOps is not skipping testing—testing still occurs before moving code to production. The magic behind DevOps is the reduction of lengthy test cycles that accrue from the large-scale changes in major releases, where multitudes of potential interactions must be tested.

Secure DevOps is a process where the development team is responsible for all aspects of code, including security all the way to production. Gone are the hand-offs to another team that may or may not have the knowledge needed to handle code-related issues if errors arise. Through the use of high degrees of automation to manage the error-prone processes of deployment, post-deployment testing, and rollback, DevOps has brought the advantages and ideas of agile methods into software maintenance and deployment. Just as adding security to agile methods does not change the nature of agile methods, security can be added to DevOps methods to manage the risk associated with maintenance efforts.

Secure Continuous Integration and Continuous Delivery Pipeline

Another name for DevOps is continuous integration/continuous development (CI/CD) where organizations that have embraced the DevOps philosophy make use of automation to manage many maintenance functions. Continuous integration, delivery, and deployment are modern approaches to the building, testing, and deployment of IT systems. CI/CD involves the entire build process with small, regular code commits automatically triggering system builds and comprehensive testing. If errors are detected, the backout process reverts to the previous successful build. This makes finding and fixing issues easier and very timely.

Continuous integration is the first step, one that is marked by the many small changes and commits to the code repository. At the time of each commit, the code is tested and verified as to fitness and function. Repeated testing prevents errors from creeping into code bases in large commits, where finding the sources and causes can be more complex.

Continuous delivery extends the continuous integration concept into the move to production phase. Extending testing to the complete system and testing the production changes, prior to release, continues the theme that small changes are easier to manage. The many small changes are then moved to production using automation, making the mechanics of the production move easier. You can still have the change management process, but with the mechanics automated, both deployment and rollback in case of an issue, the overall process becomes faster and easier to manage. Again, the concept of small changes, with checking after each one, eliminates many complex errors from being hard to trace. The fully automated move to production or rollback of the system removes risk from the mechanics of performing changes.

Continuous deployment extends the continuous delivery concept to the process of deploying changes to customers. This means the customer base has access to the latest versions of the code, in a fully automated fashion. There are no specific human interventions; code is released, delivered, and deployed via automated scripts. Should a failure occur, scripts handle the failure and return the system to the previous known good version. This method makes the change process more efficient and safer and removes issues around large releases on release day.

These automation-driven continuous principles can be applied to whole-system deployments from development to test and reference environments, prior to deploying an identical production environment. The objective of CI/CD is a deployment pipeline that minimizes the need for manual processes, enabling fully tested production deployments whenever needed. Scripts are used to deploy updates to production without human intervention rather than use a manual process.

Automated inclusion of testing ensures all checks are done and proven and built into the actual process. Security gates are coded instead of manually reviewing/testing, with the eventual desired outcome of infrastructure and security as code, not as documentation to be reviewed or approved. All of the data that shows proper testing and test results can be coded into a release gate process and automated, with the evidence of compliance being a data stream

as part of the CI/CD process. This leads to many advantages, as changes can be invoked in much smaller increments, making detection of issues easier to observe and correct.

Secure Software Tool Chain

Secure software is built as part of a process. This process requires people who are trained in doing things in a “secure” way, and to do that, they need to use tools that support the process. This means that there needs to be standardization of the tools being used as well as the methods of using them. This is not just an issue of personal choice. Code under development needs to be stored in a repository, it needs to have standard methods of proper error and syntax checking, and it needs to have proper compiler options set. Reports of issues need to be captured and managed, including the bug log and bug bar. All tools used need to support the appropriate functionality as they apply to the tool usage.

Build Artifact Verification

In many respects, the issues that apply to software authenticity and integrity assurance are the same as those for publishing and dissemination. As the components of a product move through the supply chain, they have to be authenticated at the interface between the different supply chain elements and their integrity assured. This can be done using checksums, hashes, and digital signatures. Build artifact verification involves determining that the authentication and integrity of build artifacts are true and correct. Although integrity checking is probably a continuous process throughout the production of any given artifact, the authentication normally takes place at the interface between two levels or organizations in the supply chain process.

Authentication follows the same standard principles for nonrepudiation of origin as any other access control activity. At the physical level, the endpoints have to authenticate to each other, and man-in-the-middle possibilities have to be eliminated. Endpoint authentication is a classic issue that has been addressed by numerous conventional software protocols, such as Kerberos. At the electronic level, the authenticity of the package is confirmed by any number of digital signing organizations (certificate authorities), such as a public key infrastructure (PKI) certificate authority, or

specialized checksums, hashes, and other mathematical tools for obtaining irrefutable proof of authenticity.

Determining the level of integrity of a component is always a problem since there are so many ways that integrity can be threatened. Generally, all the common secure software assurance methods apply to integrity checking, such as targeted bench checks, static and dynamic tests, audits, and reviews. However, there is a special case with counterfeiting. Counterfeits passed up a supply chain from unscrupulous suppliers can threaten the integrity of the entire system, and since those components are meant to emulate the legitimate part, they are hard to spot by conventional means. Anti-counterfeiting methods include trusted foundry vetting and other forms of direct supplier control. However, all of these approaches are in their infancy.

Securely Store and Manage Security Data

The code base is a valuable intellectual property resource, but it is also one where economic value has been added over time in the form of captured labor. It is imperative that the code base is secured from loss, unauthorized changes, or any other peril. And as versions occur, each version requires the same level of appropriate protections. Maintaining secure stores is typically done through repository software, which can manage versioning as well as checksums, hashing, and digital signatures as required.



EXAM TIP It is important to keep development, testing, and production environments separated. Having separate credentials by environment is one of the safeguards used to enforce this separation.

Credentials

All access to the code base should be via approved credentials. Maintaining the credentials is no different than other IT credentials and permissions. What does add a twist is that it is bad practice to allow a single set of credentials access to more than one of the following environments: development, testing, production. The rationale is simple: if someone has access to both

development and production, they could conceivably mix the environments when acting and make changes to production when they thought they were logged into the dev environment. The solution is managed via separate logins for environments where work is needed.

Secrets

There are a wide range of secrets that can be associated with software development and deployment. The range of secrets can be significant, from internal items such as a key for encryption to configuration elements and credentials used by the program. Secrets can also extend to the data flowing through the system. Sensitive data, such as account information, or just the data itself can require protections.

What is important for the software development team is knowing what data requires protection. Further details, such as level of protection and who is authorized to access it and who isn't, are critical. And even if the system is designed with all of these concerns in mind, upon deployment they need to be tested yet again, because what was designed to be secure may or may not be instantiated as the designers envisioned, and the protections may not be as successful as the designer believed.

Keys/Certificates

Encryption keys are one of the critical elements in security. An extensive discussion of how keys are to be safeguarded was presented in [Chapter 7](#). What is important about deployment is that the design and implementation planned and executed per [Chapter 7](#) still holds true to those ideals once deployed. Deployment involves the actual configuration of the system with the environment, and these settings can affect the security aspects of the system. It is important for testers to know about the keys/certificates that should be protected by the system and about checks upon deployment to ensure the protection schemes are working as planned.

Configurations

As already mentioned several times in the book, software configuration can play a significant role in the security posture of the software. Ensuring that the software configurations are secured per the design specification is a key

aspect of having the software operate as designed in production. As mentioned with the checking of keys/certificate security, it is best to test a production deployment, with knowledge of what configuration elements should be secured, before considering that the deployment methodology is true to the security aspects of the design.

Ensure Secure Installation

Software can be designed to be secure and even tested to show that correct installation preserves the desired security configuration, but this does not guarantee that the customer will get all the steps right or that their environment can support the secure configuration. In most cases, it is sufficient to have the documentation explain the security aspects and to let the customer's sysadmins get it right. On systems with high exposure, higher risk, or sensitivity, having some routines that verify the correct installation can be useful. The overall objective is not to have secure software in the box, but rather in operation at a customer site. Taking the installation aspect into account when planning and testing security is an important step.

Bootstrapping

Bootstrapping is a term that has been part of computing since at least 1953. In its earliest sense, the term referred to a set of instructions set to automatically initiate the execution or loading of a program into computer memory. Since those instructions, in effect, launch the intended program, they essentially “pull” the program up by its “bootstraps,” which is a 19th-century metaphor for actions that take place without external help. Over time, this metaphor has been extended to apply to any type of function that ensures self-sustaining operation. Thus, the application of the bootstrapping principle to the deployment of any product or process implies a set of operations that will both properly launch the function and ensure its continuing correctness.

With regard to the technical aspects of the deployment of a software product, bootstrapping tends to entail any one-shot process that ensures the correctness of the initial configuration. That includes setting the proper defaults and execution parameters, as well as ensuring the accuracy and correctness of the security features in the operational product. Examples of this would be the configuration of the reference monitor settings in the

operating system to ensure the desired level of access control and the definition of privacy, security, and public key infrastructure management settings to ensure effective protection of information.



EXAM TIP The term *bootstrapping* is also known as booting. For a PC, typical boot processes are the power on self-test (POST) followed by the initial program load (IPL). Integrity of these processes must be mandatory, or all subsequent processes on the machine cannot be trusted.

In the case of the overall use of the product, bootstrapping involves the definition and deployment of policies, rules, and practices to ensure that the operational-use process is effective and will always remain effective. That could include the implementation of internal review and inspection processes to gather meaningful data about product performance. In conjunction with that activity, it would also include steps to provide course correction information to ensure management responsiveness, as well as error correction and self-correcting features that would operate within the product itself to ensure its integrity. In the most practical sense, implementing error detection and correction into overall product will provide a much more secure and effective product.

The bootstrapping process is also responsible for ensuring that elements such as key generation, connection to the access control system, and other security management aspects are done correctly and functional. It is also valuable to have checks that periodically verify that these same steps are in place.

Least Privilege

Least privilege is one of the foundations of security and risk management. Limiting privilege to just what is needed prevents risk from accidents. Having software operate in administrator level of permissions solves one problem—it removes all the security provisions from the host OS and systemic environment. This is bad, as these protection mechanisms are important as part of an overall defense strategy, and being closer to the bare

metal, they offer better and more widespread levels of protection. With respect to user level permissions, all activity should always be done at the lowest level of permissions required to get the task done. If elevated permissions are needed for some specific tasks, the return to normal level of permission should be as soon as possible after the restricted task is performed. Managing permissions needs to be a mantra of every developer, because they need to understand the risk associated with elevated permissions, and as developers, they are in the unique position to properly manage the level used in the code.

Environment Hardening

Software operates in an environment, be it on a single machine, on the operating system, on a shared server, on a virtual platform, in a container, or in the cloud; there are many different environments where a software program can work. While different in many characteristics, each of these environments has means by which the environment itself can be hardened against attacks and failures.

It is important as part of the development process to understand the different environments and how the software will interact with each one. Not all software will run optimally in all environments, and understanding the use of the software and the appropriate environments is important. Defining what is required from the environment is one of the critical steps in creating a good deployment strategy. If unique accounts with unique permissions are needed as part of the software, then defining these in specific detail is important. Just adding a unique account name or hole in a firewall is no guarantee that the accounts will survive future audits without being disabled or removed. Define what is needed and document it so that at a later point in time the system owner will understand the requirements for a stable, hardened environment.

Secure Activation

Once software is installed on a platform, it requires configuration and activation of a myriad of environmentally supplied elements to facilitate proper and secure operation. Credentials need to be connected to the enterprise system for the management of credentials. If additional credentials are needed for installation and others for operation, then the details on

deploying and cleaning up this process are important. If the system will be operating in an environment with endpoint protections, then connection to that system is important as well as specifying and establishing the correct permissions. If the OS is using whitelisting (allow-listing) as a protection, then, where needed, the software needs to be enrolled in the whitelist (allow list) program.

Configurations of the environment—be it cloud, container, VM, or just host OS—are yet another important consideration. Defining what is required in terms of security is important, because different environments may have different results. Configuring networking parameters, such as holes in firewalls, to enable communication channels is essential, as these need to be instantiated and will need to persist. If the software is going to use some form of activation for licensing reasons, then understanding the intended environment and subsequent limitations is important. A program that has to be able to call out to a license server to get an activation code can have significant limitations in some secure environments with limited or blocked communications to the Internet.

Security Policy Implementation

Security policies are used to communicate the expectations of performance to workers concerning their responsibilities with respect to cybersecurity. Applications and programs can have security policies as well, such as one of not retaining customer data. The time to realize that data is not retained is before you need to get it back from the system. Clearly communicating the security policies enforced by the software before the end user installs and instantiates a working copy is important.

Secrets Injection

Some secrets used by software are not created at the beginning, but are created, injected, and used as part of the software operations. For instance, if the software is going to use SSH for certain activities, then having a secure method of receiving and using a set of SSH keys is essential. It's the same thing for OAuth tokens and other security tokens. These devices use secrecy to establish a trusted communication channel and also exchange other secrets. Understanding the sensitive nature of these elements during development and

continuing to protect them through deployment and operations is essential.

Perform Post-Deployment Security Testing

As mentioned previously, it is not enough to just design secure software. It must be developed true to design specification and deployed and operated in a manner that persists the security expectations from the requirements and design phase. Deploying something securely is still not the complete battle, because during operations, things can change in the environment that affect the security profile of the software.

When high security is required with respect to aspects of software, there needs to be a method of ensuring that the important aspects are deployed with security intact and functioning. Developing a series of post-deployment security checks is important to ensure that the system is functioning as desired. A prime example of this is the testing of patches before application. When a user uses Microsoft update services and gets patches downloaded and installed in an automatic fashion, how can one trust that it is working as desired? A look under the hood shows that there are several checks performed to ensure what is being downloaded is correct, that it hasn't changed, and that upon use, the new executables are correct. These post-deployment checks are done cryptographically, and require no user involvement, but yet serve to make certain that the process worked properly. Should all the checks not return correct answers, the action is rolled back to the previous state, and the update is marked as unsuccessful.

Having methods to automatically verify that the system is functioning securely in post-deployment operations is important for software in environments with significant risk exposures. Also logging key information elements that can be used by compliance auditors helps the system owner deal with the myriad of compliance issues automatically with a secure data stream of activity proof that can be generated to the specifications designated by the automation.

Chapter Review

In this chapter, we explored how to perform operational risk analysis as part of secure software deployment. This involved examining the deployment

environment, the necessary personnel training, relationship to safety criticality systems, and system integration issues. The next topic was exploring how to release software securely. In this section, we explored securing the continuous integration and continuous delivery pipeline. Next was an examination of secure software tool chains and building artifact verifications.

The next major section of the chapter examined how to securely store and manage security data in software deployments. Credentials, secrets, keys/certificates, and configurations were among the elements used as examples. Then came an examination of the ways to ensure secure installation of software during deployments. This section began with a look at bootstrapping followed by the concepts of least privilege and environment hardening. The topics of security activation, security policy implementation, and secrets injections followed.

The chapter closed with an examination of how to perform post-deployment security testing.

Quick Tips

- Formal procedures must be in place to ensure that risks are properly accounted for in the final acceptance of software.
- The evaluation process that underlies risk acceptance should always answer two questions: what is the level of certainty of the risk, which is typically expressed as likelihood, and what is the anticipated impact, which is normally an estimate of the loss, harm, failure, or danger if the event does happen?
- Training is a key element in properly operating software.
- The aim of system-of-systems integration is to produce synergy from the linkage of all the components.
- Delivery or deployment of software is technically still part of the development process and needs to be done in a fashion that preserves all the security and integrity that has been built into the system.
- Continuous integration/continuous development involves the entire build process with small, regular code commits automatically triggering system builds and comprehensive testing. If errors are detected, the back-out process reverts to the previous successful build.

- Build artifact verification involves determining that the authentication and integrity of build artifacts are true and correct.
- It is imperative that the code base is secured from loss, unauthorized changes, or any other peril.
- Taking the installation aspect into account when planning and testing security is an important step to ensuring a secure installation.
- Bootstrapping is a term describing the starting of a process in which the process builds itself into the correct configuration.
- It is important as part of the development process to understand the different environments the software may be deployed to and how the software will interact with each one.
- Secure activation includes elements to ensure integration with enterprise processes for credentials, whitelisting, device configuration, network configuration, and licensing elements are correct before operations commence.

Questions

To further help you prepare for the CSSLP exam, answer the following questions. The correct answers are provided in the following section.

1. Examples of secrets that need to be protected during the deployment and operation of software include the following except what?
 - A. Keys/certificates
 - B. Configuration elements such as connection paths
 - C. Source code
 - D. Credentials for connecting to other systems
2. What does build artifact verification ensure?
 - A. The integrity of a module
 - B. The authority and integrity of modules
 - C. Code is the correct size
 - D. All of the above
3. Risk analysis involves examining what properties that must be considered to stay secure within acceptable limits?

- A. Implicit security requirements
 - B. Explicit security requirements
 - C. Degree of software complexity
 - D. All of the above
4. Risk acceptance assessments should always embody a methodology for data collection that is what?
- A. Simple to use
 - B. Repeatable
 - C. Defined by the threat under consideration
 - D. Fast
5. Which of the following represents what could be typical secrets ingested by an application in operation?
- A. Security token data
 - B. SSH keys
 - C. Certificates
 - D. All of the above
6. Continuous integration/continuous development (CI/CD) is another name for?
- A. A method of deploying software developed by Microsoft
 - B. DevOps
 - C. A structured means of managing requirements in deployment for government customers
 - D. A manual system for validation and verification
7. The objective of CI/CD is a deployment pipeline that does what?
- A. Ensures all security gates have been met
 - B. Verifies build artifact verifications
 - C. Minimizes manual processes
 - D. Provides risk-free software deployment
8. One of the best ways to keep development, testing, and production environments separated is to employ what?

- A. Separate credentials
 - B. Separate networks
 - C. Separate servers
 - D. Policies mandating separation
9. In the process of bootstrapping, a system needs to perform tasks except what?
- A. Generate keys
 - B. Authenticate accounts
 - C. Connect to external resources
 - D. Restart the server to guarantee a clean start
10. Targeted risk assessments only drive acceptance decisions about what?
- A. Unknown vulnerabilities at a precise point in time
 - B. Specific, known vulnerabilities at a precise point in time
 - C. All vulnerabilities at a precise point in time
 - D. Common vulnerabilities at a precise point in time

Answers

1. C. When deploying, the source code is no longer available—an executable of the object code is deployed, not the source code.
2. A. Build artifact verification involves determining that the authentication and integrity of build artifacts are true and correct.
3. D. Risks comprise a range of properties that must be considered in order to stay secure within acceptable limits. Those properties include
 - Implicit and explicit safety requirements
 - Implicit and explicit security requirements
 - Degree of software complexity
 - Performance factors
 - Reliability factors
4. B. Risk acceptance assessments should always embody a commonly accepted and repeatable methodology for data collection that produces

reliable and concrete evidence that can be independently verified as correct.

5. **D.** All of the above are typical security secrets that can be ingested during program operation.
6. **B.** Another name for DevOps is continuous integration/continuous development (CI/CD).
7. **C.** The objective of CI/CD is a deployment pipeline that minimizes the need for manual processes, enabling fully tested production deployments whenever needed.
8. **A.** It is important to keep development, testing, and production environments separated. Having separate credentials by environment is one of the safeguards used to enforce this separation.
9. **D.** Although POST was used as an example in the text, software that starts via a bootstrap process cannot restart the server, or it would knock other services offline.
10. **B.** Targeted risk assessments only drive acceptance decisions about specific, known vulnerabilities at a precise point in time.

Secure Software Operations and Maintenance

In this chapter you will

- Learn about obtaining security approval to operate
 - Explore information security continuous monitoring
 - Explore support incident response
 - Learn about performing patch management
 - Learn about performing vulnerability management
 - Explore runtime protection methods
 - Learn about supporting continuity of operations
 - Understand how to integrate service level objectives and service level agreements
-
-

In this day and age, it is probably more appropriate to say that software is “integrated” rather than “built.” That is, the major suppliers of software build most of their products by combining parts from a range of different sources into a complete deliverable. In actual terms, this practice means that the majority of software, and particularly major applications, is purchased rather than built by a single supplier. The implications of this approach ought to be obvious. The components from multiple sources are integrated into the enterprise environment where systemic elements manage things such as operations, patching, upgrades, vulnerability management, runtime protections, high availability, and integration into the standard flow of operations.

Obtain Security Approval to Operate

Software in today's highly connected environment can pose a wide range of risks to an enterprise, both directly within the software or remotely via a connected system. Recognizing that software is not an island of risk, most enterprises impose a management review of the risk associated with the new software including sign-offs by appropriate personnel. This can happen at multiple points in the process, such as before beginning a project, before key steps in the project, and almost always before going live with the project. The term *approval to operate* comes from numerous government processes, but whether the term is specifically used or not, it is important for the appropriate executive to review all configuration risks and system risks and approve the system to be connected and operate prior to going live on the production system.

Perform Information Security Continuous Monitoring

Information security continuous monitoring (ISCM) is the process of maintaining an ongoing awareness of information security, vulnerabilities, and threats to an organization's enterprise. The term *continuous* does not mean instantaneous but rather means at a rate sufficient to support risk management decisions. There are many components to having an effective information security continuous monitoring system, including sensors to collect data and platforms to analyze the collected data in concert with threat intelligence of current conditions. These elements need to be tied into the organization's intrusion detection/incident response system.

Collect and Analyze Security Observable Data

Security involves understanding what is currently happening within the enterprise. The challenge is in differentiating what is normal business from other activities. Some actions, such as scanning, are fairly easy to detect, while other more stealthy behaviors are at times nearly impossible to separate from normal traffic. For all of this work, there is a set of data that is collected; this is data that is indicative of what is currently happening in the system. This data can come from logs, event triggers, and various telemetry and trace data. The challenge for the security practitioner is to collect and organize the

data in such a manner that facilitates the analysis of the data and the observation of key elements that lead to clues when unauthorized activity is taking place.

All software can contribute to the data collection effort through the use of log files and logging specific events with the system's event logging apparatus. The challenge is to log enough information that unauthorized activity can be detected without logging so much that the security person can't find the relevant data. Understanding how to tie into these security systems in the enterprise in a meaningful way can be critical in protecting an application and the data associated with it.

Threat Intel

Threat intelligence is the term used to describe the processes and procedures used to assess the current risk environment in an enterprise. Some of the information associated with a threat intelligence assessment comes from internal sources and some comes from external sources. Together this information paints a picture of the current actual risk environment. Information such as this can be used to tune an application and the security procedures employed with it to assist in managing the enterprise risk level.

Intrusion Detection/Response

Intrusion detection is the process of identifying a set of network communications as improper or unauthorized. Detection is a critical first step, because the enterprise cannot respond to what it cannot detect or see. Once improper activity is detected, it is classified based on its severity. Some common, low-level threats, such as perimeter scans of the outside of the firewalls, are largely ignored. A more advanced situation is when an employee begins performing IT-related tasks in an area not related to the employees' job and outside the employee's normal working hours. This type of suspicious activity would move the security group to form a response plan per their processes.

Secure Configuration

Software is typically configured via a series of configuration files and parameters. Things such as service accounts, connections to data sources,

databases, etc., are set up when software is installed in the enterprise. Because of the nature of these elements and their effect on system security, protecting the configuration parameters of a software installation is critical. What configuration elements are needed, how they interact with the system, and the level of risk if they are abused are important pieces of information to convey as part of the software delivery process. It is then incumbent upon the system administrators installing the software to heed these recommendations and apply the requisite level of protection to the configuration elements.

Continuous monitoring of these elements should be part of the plan, in case of future changes that are unauthorized. Connecting the secure configuration elements for an application to the enterprise mechanisms for handling of this type of information is part of the initial installation process.

Regulation Changes

In some systems, regulatory issues force specific security behaviors. If the application being deployed accesses any resources that are regulation bound and protected, then integrating the application into the enterprise regulatory framework is critical. In organizations where this is an issue, additional monitoring steps will be included to watch for material changes, either to the regulatory rules or to the application, to ensure timely notification of issues and give time for returning to compliance.

Support Incident Response

An incident is any event that disrupts normal operating conditions. Incidents can range from user errors to power disruptions to malicious activity. The role of incident management is to maintain the incident response capability of the organization over time. The general incident response process encompasses a set of logical monitoring, analysis, and response activities. The incident response management function integrates these activities into a substantive and appropriate response to each adverse event as it happens. Incident response management ensures that any potentially harmful occurrence is first identified and then reported and the response fully managed.

An incident response is initiated by the detection of an event that the organization deems harmful. The incident response process is then set in

motion through a formal incident reporting process. Incident reporting ensures that every potentially harmful event gets an organizationally sanctioned response. In that respect then, effective incident reporting relies on the presence of a well-established monitoring function that provides the most timely incident identification possible. In a supply chain, the goal of incident identification is to distinguish the presence of a potential violation, such as a software defect, or even a breakdown in functioning, such as a service breakdown in a supplier. The monitoring techniques that are used to ensure timeliness can range from reviews and inspections of supply chain components all the way through the use of automated intrusion detection systems (IDSs) or intrusion prevention systems (IPSSs) that function similar to malware and virus-checking scans.

The incident response management process applies to both foreseen and unforeseen events. The only difference in the execution of the process is in whether the actual steps to mitigate the incident were planned in advance. For example, most supply chains have specific procedures in place to respond to common types of breakdowns. The actual substantive response is deployed based on the type of incident. For instance, the detection of a piece of malicious code in a component will merit a different response than if the component was unavailable due to a breakdown in the supply chain. Because there are only so many ways that a supply chain disruption can occur, it is possible to anticipate critical points in the process and have an effective response waiting in the wings. In that respect, the presence of a predefined work-around will ensure a timely and generally appropriate response to any number of common occurrences.

However, since malicious code is, by definition, unanticipated, the response will depend on the form of the attack, and that cannot be specifically foreseen. Proper practice requires the actual reporting of any new occurrence be submitted to a single central coordinating entity for action. Central coordination is necessary because new attacks do not usually present themselves in a single neat package. Instead, a series of suspicious occurrences take place that represent the signature of an impending incident. Thus, a single organizational role that is responsible for analysis has to be established to ensure that the incident is effectively analyzed. Once the nature of the incident is understood, the coordinator ensures that the right people are notified, who can then make a decision about how to respond.

Root-Cause Analysis

Incident response is initiated when a potentially harmful event occurs. The incident response is then set in motion through a formal incident reporting process. Incident reporting ensures that every possibly damaging event gets an organizationally sanctioned response. Consequently, effective incident reporting is founded on a monitoring function. The monitoring must gather objective data that decision-makers can understand and act on. In addition, the monitoring has to provide the most timely incident analysis possible. The goal of effective incident identification is to be able to distinguish a potential vulnerability in the code, an attempt to exploit the software in some fashion, or even the commission of an unintentional user error.

The aim of the incident monitoring process is to identify a potentially harmful event in as timely a fashion as possible. The monitoring techniques that are used to ensure such timeliness can range from tests, reviews, and audits of system logs all the way up to automated incident management systems, dynamic testing tools, or code scanners.

Incident Triage

For the purpose of effective management control, incident reports have to reach the right decision-makers in as timely a manner as possible. Once the occurrence of an incident can be confirmed and its nature understood, an incident report is filed with the manager who is responsible for coordinating the response. That individual is normally called an *incident manager*. The incident report will document both the type and assessed impact of the event. The incidents that are reported should not just be limited to major events. Incidents that might be reported will include everything from routine defects that are found in the code, such as incorrect or missing parameters, all the way up through the discovery of intentional objects embedded in a piece of software, such as trapdoors and Trojan horses. If the incident has been foreseen, the response will typically follow the agreed-upon procedure for its resolution. That procedure is normally specified in an incident response plan that will be utilized by a formally designated incident response team.

Whatever the nature of the incident, the incident reporting process has to lead directly to a reliable and appropriate response. Executed properly, that response should be handled by a formally designated and fittingly capable incident response team. Incident response teams work like the fire

department. They are specialists who are called upon as soon as the event happens (or as soon as reasonably possible after the event happens), and they follow a process that is drilled into them to ensure the best possible response. In most instances, the incident response team should be given specialized training and equipment designed to ensure the best possible solution to the problem. The fact that the incident response team has been given that training and equipment to achieve that purpose also justifies why a designated set of responders should deal with the event rather than the local people who might have reported it in the first place.

Software can be designed to assist in incident response efforts through the proactive logging of information. How can the team decide what to add in for monitoring? Threat modeling can provide valuable information that can be used to proactively monitor the system at the points that the information is most easily exposed for monitoring.

Forensics

Forensics is the application of scientifically valid, i.e., repeatable, tests to determine specific details about what happened in the process, by whom, and, if possible, how it happened. Forensics is really an examination of specific artifacts that are retained by the system as a result of specific system activity. The most common artifact is a log entry. Log entries can be stored securely and remotely and can provide detailed information on specific user actions. The downside to recording logs of everything is the data glut that can result. Logging is just the first step. Searching the logs and compiling the specific information you need on a timeline make up the second part of building a forensics case. The more information you have to search through, the more difficult this task becomes.

Other artifacts can be remnants of specific elements that the operating system uses including registry entries, helper files, file stubs, etc. With the advent of solid-state disks and their natural cleaning processes, the length of storage for most of these artifacts is now limited in time. The real problem with these artifacts is that they are indeed transitory, and the lack of one of them remaining is not evidence that the event did not take place.

If software is performing critical functions that may require later exploration in a forensics manner, the only guaranteed method of having the information needed is via logging. Software needs to be configurable as to

the level and detail of logging to support these inquiries.

Perform Patch Management

Changes to software to fix vulnerabilities or bugs occur as the result of the application of patches from the vendor. There are numerous methods of delivery and packaging of patches from a vendor. Patches can be labeled as patches, hotfixes, or quick-fix engineering (QFE). The issue isn't in the naming, but in what the patch changes in the software. Because patches are issued as a form of repair, the questions that need to be understood before blindly applying them in production are, "What does the patch repair?" and "Is it necessary to do so in production?"

Patch release schedules vary, from immediate to regularly scheduled events per a previously determined calendar date—for example, Patch Tuesday. Frequently, patches are packaged together upon release, making the operational implementation easier for system administrators. Periodically, large groups of patches are bundled together into larger delivery vehicles called service packs. Service packs primarily exist to simplify new installations, bringing them up-to-date with current release levels with less effort than applying the myriad of individual patches.

One of the challenges associated with the patching of software is in the regression testing of the patch against all the different configurations of software to ensure that a fix for one problem does not create other problems. Although most software users rely upon the vendor to perform regression tests to ensure that the patch they receive provides the value needed, software vendors are handicapped in their ability to completely perform this duty. Only the end users can completely model the software in the enterprise as deployed, making the final level of regression testing one that should be done prior to introducing the patch into the production environment.



EXAM TIP Patch management is a crucial element of a secure production environment. Integrating the patching process in a structured way within the change management process is important to ensure stability and

completeness.

Perform Vulnerability Management

Vulnerability management has two parts to it. The first part involves the actual identification and repair of flaws in a supply chain component, such as a code module. In this case, the process is leveraged by the inspections, tests, and audits that are part of the overall software assurance process. These can take place at any point in the process, and they are usually called out in the contract for the work. From a supply chain perspective, inspections, tests, and audits should always be done at the point where the artifact transitions from one organization or level to the next within the supply chain hierarchy.

The second part of vulnerability management involves the patching of vulnerabilities once they have been identified. As we said in an earlier section, this patching normally occurs after the product has transitioned from the supplier to the customer. Patching is necessary because software products are incredibly complex and so flaws are bound to be present in any deliverable. A well-organized and disciplined patching process can help prevent exploits against known vulnerabilities. However, the key lies in the term *disciplined*. Patches have to be issued in a timely fashion, and they have to be applied when received. If this is not done, it is possible to have incidents like the SQL Slammer virus, where the patch had been issued but many users had not gotten around to applying it.

All patching and repair solutions have to be planned and tracked. A well-defined level of control is necessary to ensure timely and effective management of the resolution process. Because supply chains function at multiple levels, the ability to identify the exact point in the supply chain where a fix has to be applied is not as simple as it seems. All of the appropriate places up and down the supply chain ladder have to be targeted, and all of the components within those places altered or repaired. Therefore, all vulnerability management activity within a supply chain has to be placed under some form of coordinated, usually single, supervision.



EXAM TIP Vulnerability management involves scanning for vulnerabilities, tracking vulnerabilities, and triaging the vulnerabilities with respect to criticality. This will enable efficient patch management that is also risk responsive.

Runtime Protection

Setting up the various available runtime protection schemes requires coordination with the enterprise when setting up the software. Some protections, such as web application firewalls (WAFs), will require coordination with the network team. Others such as address space layout randomization (ASLR) require coordination with the system administrators for the OS. More complex entities like runtime application self-protection (RASP) schemes will require both system administrator and application administrator assistance in the setup and monitoring of the system.

Support Continuity of Operations

Continuity of operations is a name for the state of operations that a business enters upon incidents impacting the business at a specified level or greater. A system enters the realm of continuity of operations when a failure, minor or major, impacts operations and causes the business to enter a phase of operations at a reduced capacity, performing only required work. Depending upon the outage and its impacts, the proper level of operating the enterprise is commenced, typically in an autopilot type switch, to allow the business to concentrate on what resources they have on critical business processes.

Organizations have plans detailing the various states of reduced capacity that they have planned for and can operate under. When a new application joins the enterprise, it needs to be built into the continuity of operations plan. It might be essential, semi-essential, or one that is just not run during this period of reduced operations. A continuity of operations plan is created, providing operational details of what will be run, including any dependencies for the various conditions of impaired operations. The concept is to keep only the essential running until the enterprise has fixed whatever problem caused the reduced operations. The pathway back to full operations is via a disaster recovery mode.

Backup, Archiving, Retention

Backups are one of the cornerstones of a security program. The process of backing up software as well as the data can be a significant issue if there is ever a need to restore a system to an earlier operational point. Maintaining archives of earlier releases and the associated datasets is an important consideration as production environments are upgraded to newer versions. It is also important to remember the security implications of storing archive sets. Employing encryption to protect the sensitivity of these archives is a basic, but often overlooked, step. In the event it is legally necessary to produce data from an earlier version of the system, a restore of not just the data but the software version as well may be necessary.

A retention cycle is defined as a complete set of backups needed to restore data. This can be a full backup set or a full backup plus incremental sets. A cycle is stored for a retention period as a group, because the entire set will be needed to restore. Managing the retention cycles of data and programs with the retention periods determined as appropriate is one of the many considerations that operations personnel need to keep in mind when creating archives and storing them for future use. Whether this process is done via the change management process or part of the system lifecycle process is less important than ensuring that it is actually performed in the enterprise.

Disaster Recovery

Disaster recovery (DR) is the set of actions a firm uses to recover from a disaster. There are three operational states: normal operations, impaired operations, and disaster recovery operations. Normal operations are just that, the normal operations of a firm. Impaired operations are those that occur during some event or outage. This is the world of continuity of operations. Only essential things are running, and even those may be in a reduced capacity. Disaster recovery mode is the pathway from the impaired operations back to normal. This mode is highly dependent upon the level of restriction during continuity of operations, both of the application and its dependent elements. For instance, it might be easy to bring the application back up, but if the orders from it go into a database that is not back to fully operational status, it might have to wait.

To manage the transition of business operations back from limited operations, the continuity of operations state to normal operations requires a

plan similar to that for initiating and operating under the state of continuity of operations. A list of all services and applications, with their dependencies, and a sequence chart for what to start first, second, etc., is needed to prepare for transitions. With the addition of each new application in the enterprise, this sequence plan will need to be updated. As opposed to the initiation of a continuity of operations plan, which, when invoked, is a sudden shift, a DR recovery plan is more scheduled, with testing along the way to ensure things are working. While one can go into a continuity of operations plan very quickly, it is a more time-consuming, deliberate effort to resume normal business operations. This is the period of disaster recovery, and for some situations, it may take days or weeks depending upon the external factors involved.

Resiliency

Resiliency is a measure of how a system can react to a disturbance and then return to a correct operational state. Resiliency needs are connected to the criticality of an application, and when a new application is being set up, it is important to address the resiliency needs of the application and its dependencies. If an application is critical, it might be important to have operational redundancy of the application, any dependent elements (think databases, supporting servers), and network connectivity. There are a wide range of options that can be employed in making an application more survivable, and many of these are in the configuring elements associated with implementing the application within the enterprise. An example would be erasure code, a means of data protection where data is chunked in redundant chunks so that a loss of data or communications does not stop the processing. Resiliency systems of this level of complexity are part of the enterprise fabric, not just a feature of a system, and thus require architecting of the application into the resiliency framework of the enterprise.

Integrate Service Level Objectives and Service Level Agreements

Service level agreements (SLAs) essentially set the requisite level of performance of a given contractual service between a customer and supplier.

An SLA is comprised of a series of objectives called Service Level Objectives (SLOs). Once entered into, the SLA becomes a legally binding document. Typically, a good SLA will satisfy two simple rules. First, it will describe the entire set of product or service level objectives in sufficient detail that their requirement will be unambiguous. Second, the SLA will provide a clear means of determining whether a specified function or service has been provided at the agreed-upon level of performance.

The contract, which is the central document in enforcing an SLA, substantiates both of these conditions. It is the mechanism used to express the required levels of performance. As a result, those behaviors have to be described as externally observable behaviors. Along with the behaviors to be observed, the contract should specify a process and an accompanying set of objective criteria that will be used to judge whether those behaviors have been correctly implemented in the deliverable. In addition, the contract would normally address such legal issues as usage, ownership, warranty, and licensing rights associated with the product or service. In that respect, the contract must legally define all of the requirements for performance for all parties.

SLAs can be used internally in an organization as well. In these situations, details regarding maintenance, performance, availability, and qualified personnel will be spelled out in the agreement so that all sides of the enterprise understand the capabilities and limitations associated with the operation of a system and can then set appropriate expectations.

Normally, because technology, and software in particular, is complex, the presence and execution of desired levels of performance from a provider can be evaluated based only on a proxy. Consequently, objective, behavioral criteria that are specifically associated with the desired actions that they represent must be placed in the contract. These criteria provide the mechanism for evaluating an essentially invisible and typically complex set of activities based on observable outcomes.

Alterations to an SLA may take place in the course of events. However, if alterations are required, these changes must be accomplished within the legal structure. It should be noted that the responsibility for ongoing maintenance of changes to the contract agreement rests with the acquiring organization. Consequently, if there is any alteration to the contract, the new form of the agreement is kept by the acquirer within its contract change control system.

Chapter Review

In this chapter, you became acquainted with addressing topics related to secure software deployment, operations, and maintenance. The chapter opened with the topic of obtaining security approval to operate the new software. This included signing off on the risk acceptance by an official at an appropriate level. The next topic was performing information security continuous monitoring, which includes collecting and analyzing security observable data (e.g., logs, events, telemetry, and trace data) and threat intelligence sources. Integration with enterprise intrusion detection/response was covered. Secure configuration and necessary regulation-driven changes, where integration of the application into the enterprise operational framework is required, were presented as an additional topic.

The next section presented supporting incident response, including root-cause analysis, incident triage, and forensics. Additional sections covered performing vulnerability and patch management as well as enabling various runtime protection methods. The chapter then talked about connecting to existing operations frameworks to support continuity of operations, including backups, archiving and retention, and disaster recovery operations. The topics of resiliency examining issues of operational redundancy, erasure code, and survivability were covered. The chapter closed with an examination of how to integrate service level objectives and service level agreements.

Quick Tips

- It is important for the appropriate executive to review all configuration risks and system risks and approve the system to be connected prior to going live on the production system.
- Information security continuous monitoring is the process of maintaining an ongoing awareness of information security, vulnerabilities, and threats to an organization's enterprise.
- A challenge for security operations is to collect and organize security data in such a manner that facilitates the analysis of the data and the observation of key elements that lead to clues when unauthorized activity is taking place.
- Threat intelligence information can come from sources both internal

and external to the enterprise.

- What configuration elements are needed, how they interact with the system, and the level of risk if they are abused is important information to convey as part of the software delivery process.
- In some systems, regulatory issues force specific security behaviors.
- The aim of the incident monitoring process is to identify a potentially harmful event in as timely a fashion as possible.
- Software needs to be configurable as to the level and detail of logging to support relevant forensic inquiries.
- Patch management is a crucial element of a secure production environment. Integrating the patching process in a structured way within the change management process is important to ensure stability and completeness.
- Vulnerability management involves scanning for vulnerabilities, tracking vulnerabilities, and triaging the vulnerabilities with respect to criticality. This will enable efficient patch management that is also risk responsive.
- Software that may be critical to operations needs to be properly integrated into the enterprise continuity of operations plan.
- Software needs to manage data backups, archiving, and retention in a manner that can be configured to work with the enterprise operations.
- Software needs to have defined details regarding maintenance, performance, availability, and qualified personnel spelled out so that all aspects of the business understand the capabilities and limitations associated with the system and they can then set appropriate expectations for service levels.

Questions

To further help you prepare for the CSSLP exam, answer the following questions. The correct answers are provided in the following section.

1. Before software is turned on in production, which of the following steps is needed?
 - A. Obtain approval to operate

- B. Notify incident response
 - C. Perform vulnerability management activities
 - D. Perform patching
- 2. Patch management is built on what?
 - A. Risk reduction
 - B. Change management process
 - C. Regression testing
 - D. Customer expectations
- 3. Valid runtime protection mechanisms may include the following except what?
 - A. Runtime application self-protection (RASP)
 - B. Web application firewall (WAF)
 - C. Address space layout randomization (ASLR)
 - D. Code reviews
- 4. Which of the following is a measure of how a system can react to a disturbance and then return to a correct operational state?
 - A. Continuity of operations
 - B. Vulnerability management
 - C. Disaster recovery
 - D. Resiliency
- 5. Which of the following is not a supporting element of incident response?
 - A. Incident triage
 - B. Patching
 - C. Root-cause analysis
 - D. Forensics
- 6. Software vulnerability management has a post-delivery focus that is called what?
 - A. Patching
 - B. Certification

- C. Authentication
 - D. Validation and verification
7. SLOs and SLAs should address all of the following except what?
- A. Maintenance
 - B. Availability
 - C. Qualified personnel
 - D. Archiving
8. Which of the following is the best source of security observable data?
- A. Log files
 - B. Incident response reports
 - C. Help-desk requests
 - D. Forensics
9. When do disaster recovery operations occur?
- A. Before continuity of operations occurs
 - B. Immediately before disaster occurs
 - C. When disaster occurs
 - D. After continuity of operations phase
10. What is the name of a complete set of backup recovery data that is required to restore a system?
- A. Retention cycle
 - B. Full backup
 - C. Full backup plus incremental sets of changes
 - D. Complete backup set

Answers

1. A. Approval or authority to operate is the action where an appropriate level of management approves the use of a system in production.
2. B. Patch management operations should use the change management process, especially for production systems.

- 3.** **D.** Code reviews are part of the development process and are not a runtime protection strategy.
- 4.** **D.** Resiliency is a measure of how a system can react to a disturbance and then return to a correct operational state.
- 5.** **B.** Patching is not supportive of incident response. It might occur as a reaction to IR actions, but is not supportive.
- 6.** **A.** Patching normally occurs after the product has transitioned from the supplier to the customer.
- 7.** **D.** Archiving deals with data, not with SLAs or SLOs.
- 8.** **A.** Log files are the primary source of security information.
- 9.** **D.** Disaster recovery is the phase that moves from continuity of operations back to normal operations.
- 10.** **A.** A retention cycle is defined as a complete set of backups needed to restore data.

PART VIII

Secure Software Supply Chain

- **Chapter 18** Software Supply Chain Risk Management
- **Chapter 19** Supplier Security Requirements

Software Supply Chain Risk Management

In this chapter you will

- Learn to implement software supply chain risk management
 - Analyze the security of third-party software
 - Verify the pedigree and provenance of software elements
-
-

Software has a supply chain like any other product, and security issues can arise in the supply chain and manifest as risk in the software being produced. Applying risk management principles to the entire supply chain is a good business practice, and even more critical when software included in final products has third-party components. This chapter explores the concepts behind supply chain risk management, including how to analyze software for third-party sources of risk.

Implement Software Supply Chain Risk Management

The overall purpose of supplier risk assessment is to identify and maintain an appropriate set of risk controls within the overall supply chain. Given the threats implicit in the supply chain process, supplier risk assessments are a particularly critical part of the effort to maintain effective supplier control. Supplier risk assessments identify specific threats to the organization's supply chain and then evaluate how likely those threats are to occur, as well as the consequences of each threat should it happen. In that respect, supplier risk assessments underwrite the development of the specific strategy that will

be used to ensure trust up and down the overall supply chain.

According to the General Accounting Office (GAO), threats to supply chains fall into five generic categories. Each category has distinctly different implications for product integrity. These categories are

- Installation of malicious logic in hardware or software
- Installation of counterfeit hardware or software
- Failure or disruption in the production or distribution of a critical product or service
- Reliance on a malicious or unqualified service provider for the performance of a technical service
- Installation of unintentional vulnerabilities in software or hardware

Supplier risk assessment lessens all of these concerns by providing assurance that the supplier employs a consistent, disciplined process that ensures measurable product integrity. Supplier risk assessments identify what could go wrong in the development process, determine which risks to address, set mitigation priorities, implement actions to address high-priority risks, and bring those risks “to within acceptable levels.” Supplier risk assessments let managers deploy the necessary proactive and reactive controls to respond to threats as they arise in the supply chain. This process also monitors the effectiveness of those controls once they have been put in place. The general aim of supplier risk assessment is to ensure up-to-date knowledge about the supply chain’s overall threat situation. Systematic supplier risk assessments explicitly guide the prioritization of the steps that the organization will take to ensure the security and integrity of its products.

Risk assessment models are used to manage future risk, whether that risk emanates from within the organization or via a supplier. While there are many risk management models, the concepts are fundamentally the same, and the best model for managing supplier risk would be one that is aligned with your internal risk management. Common to all models is a set of four elements: identification, assessment, response, and monitoring.

In the identification step, the items that may bring risk into the enterprise must be identified. In the case of software components, this might seem simple, but software is often built using software components, so the entire software bill of materials is a legitimate set of items. Once all the items are

identified, they need to be assessed with respect to the threats and the vulnerabilities associated with each asset and the likelihood of their occurrence. For each identified issue, a plan needs to be created to respond to the risk. This plan should detail what is being done by the supplier and what aspects are born by the purchaser. With respect to dealing with vulnerabilities, an important issue is detailing the lifecycle of maintenance and patching as vulnerabilities are discovered over the life of a project. The response phase is just that, determining who does what over the course of a product's lifecycle with respect to the vulnerabilities. Just stating a supplier will provide patches may or may not be sufficient. There are timelines to consider, as well as the issue of patches to subsupplier-supplied items. All plans are subject to failures in practice, so a monitor phase is put in place to ensure the agreed-upon terms are being followed and working.

Analyze Security of Third-Party Software

Third-party software is software that is not written by in-house personnel. Before someone thinks that they don't have third-party software, note that open source libraries are third party. Studies have shown that the majority of code developed today includes third-party library calls, so the issues associated with understanding the risks from third-party software apply to most firms.

The primary focus of secure lifecycle development programs is to ensure the security of code being generated. Third-party code frequently gets overlooked, and there are many options associated with this gap. The first and most obvious is to have a monitoring program in place that gets notified of updates to third-party code and incorporate those updates into your normal update process. If a vulnerability is serious enough, it may trigger your update process to issue a patch to your code, fixing the embedded code. The second option, and one that should be used in conjunction with the first option, is to subject third-party software to the same security testing regime as your own code.

Software Bill of Materials

Software is frequently composed of multiple parts, including sections of

code from libraries and other sources. A *software bill of materials (SBOM)* is a listing of the provenance and lineage of all the components in software, including libraries, third-party code, and internally generated code. Listing the versions for all included elements is important when it comes to identifying and remediating vulnerabilities. An item of increasing importance is the lineage of all the components that are in a software build. This becomes a security issue when one of the components—for example, an embedded library—has a vulnerability. This has been seen in many recent security issues such as Heartbleed and OpenVPN. Even major vendors, such as Microsoft, have had this problem, as shown in the SQL Slammer worm affecting not just SQL Server products but also third-party software developed using the SQL engine. To understand the risk associated with the components inside a software build, many have called for the production of a software bill of materials detailing the constructs and versions contained within a build.

Verify Pedigree and Provenance

In many respects, the issues that apply to software authenticity and integrity assurance are the same as those for publishing and dissemination. As the components of a product move through the supply chain, they have to be authenticated at the interface between the different supply chain elements and their integrity assured. Although integrity checking is probably a continuous process throughout the production of any given artifact, the authentication normally takes place at the interface between two levels or organizations in the supply chain process.

Authentication follows the same standard principles for nonrepudiation of origin as any other access control activity. At the physical level, the endpoints have to authenticate to each other, and man-in-the-middle possibilities have to be eliminated. Endpoint authentication is a classic issue that has been addressed by numerous conventional software protocols, such as Kerberos. At the electronic level, the authenticity of the package is confirmed by any number of digital-signing organizations (certificate authorities), such as a public key infrastructure (PKI) certificate authority, or specialized checksums, hashes, and other mathematical tools for obtaining irrefutable proof of authenticity.

Determining the level of integrity of a component is always a problem since there are so many ways that integrity can be threatened. Generally, all the common secure software assurance methods apply to integrity checking, such as targeted bench checks, static and dynamic tests, audits, and reviews. However, there is a special case with counterfeiting. Counterfeits passed up a supply chain from unscrupulous suppliers can threaten the integrity of the entire system, and since those components are meant to emulate the legitimate part, they are hard to spot by conventional means. Anti-counterfeiting methods include trusted foundry vetting and other forms of direct supplier control. However, all of these approaches are in their infancy.



EXAM TIP The fact that software is procured via a supply chain does not eliminate the need to understand and verify the necessary testing regimens employed as part of the development process. These aspects are best handled via process validation and product testing.

Secure Transfer

Secure transfer of code including updates is a critical function. Unlike a physical object, which would be difficult to intercept in transit and modify without being detected, software is a digital item, a series of 1s and 0s that are subject to interception and manipulation. And if intercepted and modified, without precautions and mitigations in place, the changes would be nearly impossible to detect. This is not a new or particularly challenging problem, because known solutions in the form of code signing and cryptographically secured communication channels have existed for decades and are a common practice. It is important to ensure that these methods are actually being used and that the veracity of third-party code can be attested to before being integrated into the production process.

System Sharing/Interconnections

System sharing and interconnections are becoming more common in today's connected world. System interconnections are direct connections to external

third-party systems, and these direct connections can become a risk conduit into the internal processes. The most prominent example is cloud services. Someone jokingly once defined *cloud computing* as using someone else's computer, but in fact that is really not far from the truth. When you have external cloud services as part of your build process or internal systems to enable sharing across geographically separated offices, there is a third-party supplier risk. Examining your systems for these interconnections is not hard, but the time to identify them is before they are contracted for, ensuring that the requisite security requirements are factored in from the beginning. While this may not seem important, in 2021, a data center fire in France caused a massive outage across the Web for many days and, for some unlucky firms without a geographically separate backup system, resulted in a total loss of data. This actually led to data losses at some firms that were catastrophic and unrecoverable.

Code Repository Security

Code repositories are the locations where source code is stored while being built and for current and past versions. Just as the need for securing backups was presented in dramatic fashion in the previous section, the code repository needs even greater protections. The code repository represents the current (and past) versions of the work, and it should have only authorized code in it. This means the repository needs to have protections against unauthorized code changes, even from inside the company. If an attacker can make an undetected change to a module in the code repository, it will likely make it into production and remain undetected. For these reasons, the code repository should be given the same level of scrutiny as a company's bank accounts. Just as you would never allow anyone to make banking changes, code changes need the same level of scrutiny. This seems like a minor issue, but many companies have had issues with unauthorized changes, sometimes unintentional, resulting in significant code losses. And regardless of intentional or not, if there is a loss, that is risk being realized into real loss. Protecting the code repository with a series of overlapping protective elements at the process level, thus not being able to be bypassed, is important.

Build Environment Security

A modern build environment will typically include continuous integration, delivery, and deployment. These are the result of multiple small, regular code commits to a repository that can then automatically trigger builds and run comprehensive testing. Sometimes this is done on a scheduled basis, sometimes as a result of changes to the code base. These changes can come from internal sources or third-party sources. It is important to consider third-party code changes in the overall design of the build environment. The security of the system comes from many small incremental changes, all tracked and all capable of being backed out of the system if testing failures occur. For this system to work securely, the security of the pipeline is paramount. If the security of any of the components is compromised—such as the pipeline, its connections, and the underlying architectures—the whole system and the data it transmits become suspect. Securing the build process is critical, and that process extends from all suppliers through you to your customers.

[Chapter 16](#) presents a detailed analysis of the secure build process, and nothing from a third-party should do anything to compromise that effort. With respect to third-party elements, it is important to be cognizant of all third-party libraries being used and the methods behind their updating. Older, out-of-date versions of libraries code may contain vulnerabilities that will introduce risk into the system being developed. It is important to understand the details of vulnerabilities in all third-party code.

Cryptographically Hashed, Digitally Signed Components

Code signing is a powerful anti-tampering methodology. Code signing should be a requirement whenever possible for third-party elements, with the code signing providing a means of detecting changes from the approved source material. Instituting code-signing checks into the process of using third-party elements is a process step; the techniques can be unobtrusive to the user of the code, but if built into the process, everyone can have faith in the veracity of the included code elements.

Right to Audit

As with all process-related elements, the process is key to success, but it still

requires a monitoring step to ensure that the process is functioning correctly. With respect to third-party software elements, where it makes sense to impose process controls on suppliers, it also makes sense to have the right to audit those processes. This provides a means to determine whether the controls are functioning as designed and whether they are effective.

Chapter Review

In this chapter, you were acquainted with the role of supply chain risk management and the activities of its many facets. Risk management with respect to supply chain issues is straightforward—it's the same set of problems that occur with internal processes; they just occur in different processes in a different company. They can still be managed. Using third-party software creates a need to analyze that software for vulnerabilities and risks, as well as a need to document the pedigree and provenance of the components.

Quick Tips

- The overall purpose of supplier risk assessment is to identify and maintain an appropriate set of risk controls within the overall supply chain.
- Supplier risk assessment lessens all of these concerns by providing assurance that the supplier employs a consistent, disciplined process that ensures measurable product integrity.
- Risk assessment models are used to manage future risk, whether that risk emanates from within the organization or via a supplier.
- The four steps that are covered in supplier risk assessment are identification, assessment, response, and monitoring.
- The majority of code developed today includes third-party library calls.
- Establishing a means of identifying the provenance and pedigree of third-party code is essential to ensuring that supply chain risk is managed.
- Software code repositories require protection against integrity violations and unauthorized updates.

- The build environment requires security to ensure that the software is properly constructed without interference.
- Code signing provides the technological means to ensure integrity of third-party code.

Questions

To further help you prepare for the CSSLP exam, answer the following questions. The correct answers are provided in the following section.

1. Software pedigree is an examination of what?
 - A. Requirements for use
 - B. Reliability of the code
 - C. The source of the code
 - D. The level of risk in third-party code
2. Which of the following will most likely include third-party code?
 - A. Current code base written by internal developers in Java
 - B. Previous release of current code base
 - C. Code base associated with a product that was obtained when your firm merged with another firm
 - D. All of the above
3. Which of the following is not a phase in supplier risk assessment?
 - A. Software provenance
 - B. Identification
 - C. Assessment
 - D. Monitoring
4. Supplier risk assessments are designed to determine what?
 - A. What could go wrong in the development process
 - B. Which risks need to be addressed
 - C. Determine mitigation priorities
 - D. All of the above
5. Risk assessment models are used to manage future risk that is

-
- A. Generated by internal actions
 - B. Generated from the actions of third parties
 - C. Coming from suppliers' suppliers
 - D. All of the above
6. Code signing can be used for what?
- A. Protecting code in transit
 - B. Identifying who alters code
 - C. Managing third-party code risks
 - D. All of the above
7. Which of the following requires specific infrastructure security to protect intellectual property?
- A. Third-party sources of source code
 - B. Code repositories
 - C. Build environment processes, procedures, and elements
 - D. B and C
8. Software authenticity requires controls to ensure what?
- A. Hashing and checksums
 - B. Good coding practice
 - C. Nonrepudiation of origin
 - D. Denial of service
9. Supplier security levels are ensured through the use of what?
- A. Standards
 - B. Contracts
 - C. Supplier processes
 - D. Audits
10. What is a software bill of materials?
- A. A listing of the pedigree and provenance of all code elements included in the project

- B. A listing of who wrote the code by module
- C. A copy of source and object code bases
- D. All of the above

Answers

1. C. Pedigree is a list of where the software came from, including which software is from subsuppliers.
2. D. Almost all Java code involves third-party libraries, so all of the options are true.
3. A. Software provenance is not an element in risk modeling. The phases are identification, assessment, response, and monitoring.
4. D. Supplier risk assessments identify what could go wrong in the development process, determine which risks to address, set mitigation priorities, implement actions to address high-priority risks, and bring those risks “to within acceptable levels.”
5. D. Risk assessment models are used to manage future risk, whether that risk emanates from within the organization or via a supplier.
6. A. Code signing cannot identify who changes code, nor is it the answer for third-party risk.
7. D. Software code repositories and the entire build environment require security protections to ensure that unauthorized changes do not occur within the code.
8. C. Software can be authenticated only if its origin can be confirmed.
9. C. While auditing is a tool that can be used to check whether supplier processes are working as designed and described, it is the process of the supplier that ensures the security level, so option C is a better answer.
10. A. A software bill of materials lists all the components, where they came from, and their version.

CHAPTER 19

Supplier Security Requirements

In this chapter you will

- Learn about ensuring supplier security requirements in the acquisition process
 - Understand security information needed to support contractual requirements
-
-

Supply chains have come under a lot of scrutiny in 2021 due to several issues. First, the pandemic interrupted many supply chain elements, extended timelines, and caused overall disruptions across virtually all sectors and industries. The second major issue was a series of security incidents that involved supply chains. The attacks from supply chain perspectives illustrated the importance of understanding the source and level of risk associated with materials and their supply chain. Supplier risk assessments support supplier sourcing decisions. Supplier sourcing decisions involve all of the analysis that is needed to understand every aspect of the supplier's operation while making any sourcing decision. That analysis can embrace decisions about everything from code reuse to intellectual property protection and regulatory compliance. All of these factors, and many others, can potentially impact the integrity of the product as it moves from development to long-term sustainment. The processes and practices for each of these areas will be discussed in this chapter.

Ensure Supplier Security Requirements in the Acquisition Process

Supply chains provide critical items to firms as part of the production process

such as hardware, software, an item that is built into a finished product, or a supporting item that assists in the production efforts. In essence, supply chains supply the stuff that makes businesses work. But along with the good comes the bad, and that is risk. All activities involve risk, but when you involve outside parties in your efforts, their risks are not ones that you can easily manage. A key step toward understanding supplier risks and managing them is the definition of the risks.

Using the same model for risks that are incurred within the firm, you can specify the types and quantities of outside risk one is willing to absorb. Defining your security requirements in a measurable manner and using this data as part of the supplier-vetting process is important with respect to downstream risk. If you do not ensure that supplier risk is managed as part of the acquisition process, then later in the cycle if those risks become actualized, your hands may be tied with respect to your ability to address them. This makes ensuring that your suppliers understand and agree to your security requirements a foundational element of the acquisition process and subsequent contracts.

Just as you are setting expectations for your suppliers, you can expect them to set their expectations for their suppliers. The best method to ensure there are no gaps is to specify that your supplier requirements are a minimal set that must be passed down to all subcontractors and suppliers to your own supplier and that these requirements are subject to the same level of scrutiny. This is important as subcontractor risk control is built around all parties in the supply chain understanding their precise contractual requirements with respect to risk.

Supplier Sourcing

Supplier sourcing considerations represent one of the most important issues in the supply chain risk management process. Obviously, each node in the supply chain represents a source, and every one of those sources has to be vetted and shown to be trustworthy. So, supplier sourcing generally amounts to nothing more than doing all the analysis that is needed to understand every aspect of the supplier's operation while making any sourcing decision. It is important that the decision be thought through prior to making the decision to contract out the work.

Given the variability among subcontractors, the acquirer should explicitly

understand which elements of the proposed work the subcontractors will be allowed to perform. For the sake of ensuring a consistent outsourcing process, it is also just as important to be able to state which activities the subcontractor will *not* be allowed to perform. That understanding is necessary so that the acquirer and the prime contractor can make specific determinations about how to apportion the work and how subcontractors will then be managed and supported throughout the process. The assignment of work elements to subcontractors is usually explicitly stipulated in the contract, and all additional outsourcing decisions are guided by the criteria that are embedded in the contractual language.

In addition to outsourcing, a strategic security concern should be addressed—the determination of the degree of foreign influence and control that might be exercised over the product. Influence and control by organizations or nation-states that are not necessarily friendly to the United States will directly impact the trustworthiness of that product. Therefore, the degree of foreign influence, control, or ownership of a given contracting organization has to be determined and ensured before additional steps can be taken to manage risk.

The nature of software development results in final products that have a wide range of sources of included components. From included libraries to subroutines to modules, the sources of individual elements of software can come from a wide range of subsuppliers, both foreign and domestic. Elements of programs can be outsourced to third parties for development, with integration being handled by other parties. [Figure 19-1](#) illustrates the wide array of supplier relationships common in software development today.

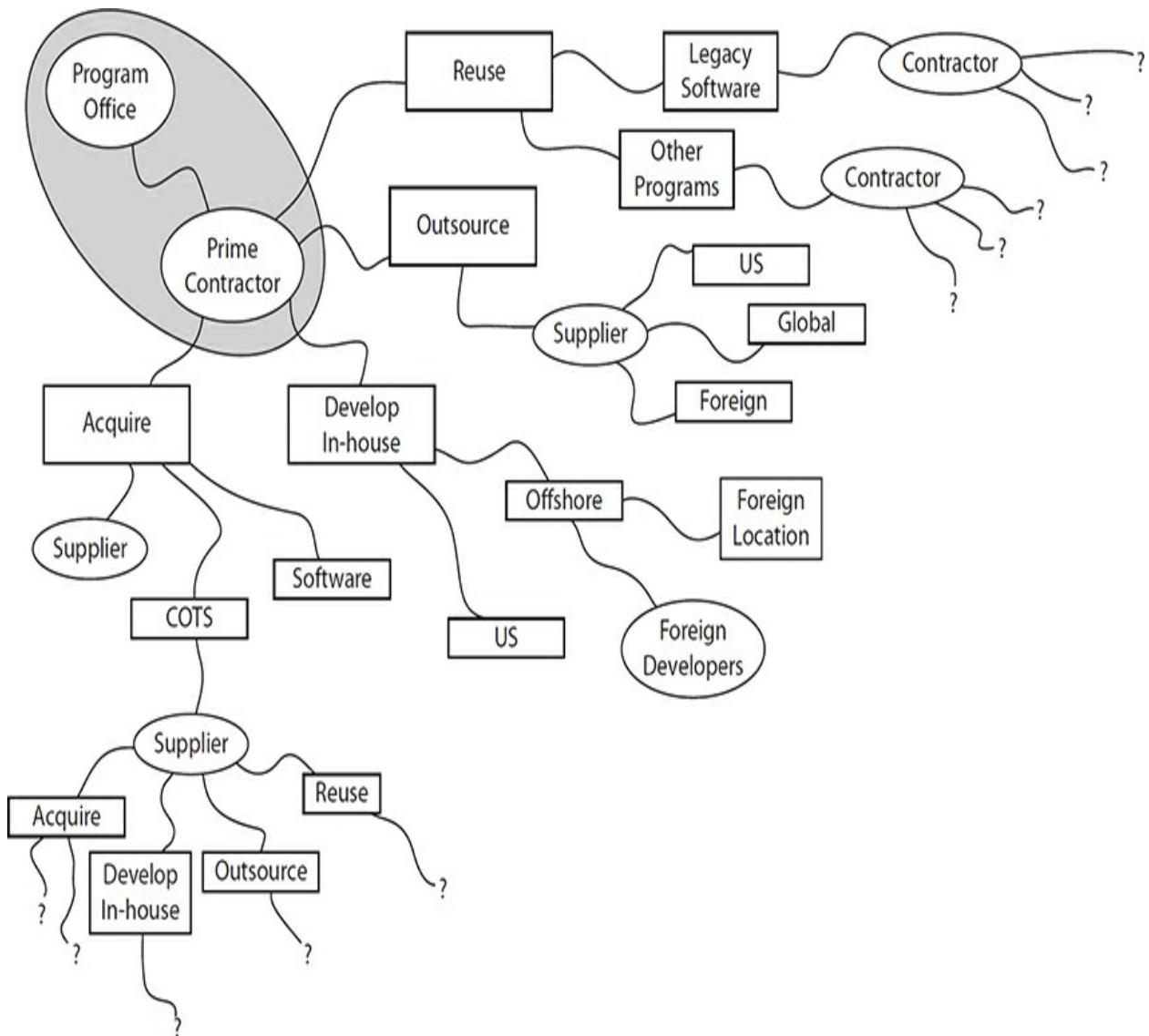


Figure 19-1 Layered security

Source: Walker, Ellen. "Software Development Security: A Risk Management Perspective." *The DoD Software Tech. Secure Software Engineering* 8, no. 2 (2005): 15–18.

This article was originally printed in the *DoD Software Tech News*, Vol. 8, No. 2. Requests for copies of the referenced newsletter may be submitted to the following address: Philip King, Editor, Data & Analysis Center for Software P.O. Box 1400 Rome, NY 13442-1400 Phone: 800-214-7921 Fax: 315-334-4964 E-mail: [news-editor@dacs.dtc.mil](mailto:news-editor@dacs.dtic.mil). An archive of past newsletters is available at www.SoftwareTechNews.com

A number of commonly accepted industry models can be utilized to source suppliers up and down the supply chain. One of the most common approaches is to rank bids based on a risk versus return score. Higher scores are given to projects that meet or exceed expectations for criteria such as technical soundness, contribution to the business goal, and overall resource constraints. Trade-offs from a security standpoint include the following:

- **Strategic improvements versus maintenance of current operations** A supplier's efforts to modernize its operation can actually increase the risk of a security breach. Therefore, there has to be an assessment of the value of changing a supplier's operation, no matter how beneficial the change might seem based on other criteria, such as cost or market share. In general, changing a successful operation's configuration is always a security risk unless the improvement can be justified.
- **High versus low risk** If the only goal is to minimize risk, then the general performance of the suppliers might be constrained. High-risk, high-return approaches can enhance the value of a product, provided the risks are capably and carefully managed. Most organizations, however, can handle only a limited number of such projects. As a result, management must always balance the amount of risk they can afford against the ability to manage it.
- **Impact of one supplier on another** Every new supplier is likely to affect, or be affected by, the suppliers in the current supply chain. Management must recognize the context in which the new supplier will operate and make decisions accordingly—particularly when it comes to the complexity of the current supplier base and the interface with a new supplier. Adding a supplier at higher levels in a supply chain can often translate to downstream ripples that can adversely affect the safety and security of the overall structure.
- **Opportunity costs** Management must always consider the impact on long-range investment. For instance, will a large current investment in suppliers and their operating costs preclude or delay better future opportunities in the supply chain? Will large current capital expenditures create even larger risks in the future?

The outcome of these comparisons is a set of concrete points of reference

that will allow managers to perform the rational trade-offs necessary to reach an optimal sourcing decision. Nevertheless, a range of requisite controls have to be put in place to ensure that any end product of the supply chain risk management process will meet all of the necessary criteria for integrity and trustworthiness. One of the most important of these is contractual integrity control.

Contractual Integrity Controls

Contracts are legal agreements to perform a specified action or deliver a specified product. Contracts are legal and binding, and they essentially control a given development process and all of its actions up and down the supply chain. As a consequence, conditions for stipulating and then ensuring contractual integrity have to be built into the supply chain management process. Those conditions include such things as the initial specification of the criteria by which maintenance of contract integrity will be judged and the specification of the standard evaluation criteria that will be utilized to ensure product integrity.

Controls can be developed from those criteria to ensure that the appropriate product assurance security testing and audits are carried out as the contract specifies. Specifically, the controls to oversee the supplier's delivery of product assurance have to be established, and the monitoring processes have to be installed to ensure that those controls remain effective. Because monitoring will occasionally lead to the issuance of nonconcurrence, there also have to be controls in the contract to ensure an explicit issue resolution process. Moreover, there is also the attendant requirement that corrective actions are implemented and monitored to resolve these issues.

On the administrative side, contractual controls can be put in place to ensure that the terms of the contract are maintained under strict configuration management. If outsourcing is involved, there can be a number of different contracts in a large project, particularly if that project involves a supply chain. All of these contracts have to be maintained in alignment to ensure control over product integrity. As a result, the provisions of all relevant project contracts have to be carefully coordinated and controlled up and down the supply chain through a well-defined and formal configuration management process.

Vendor Technical Integrity Controls for Third-Party Suppliers

Vendor integrity controls ensure the correctness of the technical process of the subcontractors within a particular supply chain. Their overall aim is to ensure that all contractual requirements are passed to subcontractors in a timely and suitable fashion and that all products of the subcontractors within the supply chain are properly validated. To achieve this purpose, those controls administer the requisite verification, validation, or test agents, as well as communicate with other parties as specified in the contract and project plans.

Vendor technical controls ensure the fundamental integrity of all software artifacts moving within the supply chain. These technical controls ensure the integrity of all product baselines throughout the development and sustainment processes, including all product baselines and repositories. The specific aim of technical controls is to ensure the completeness, consistency, and correctness of a formal set of components that comprise a given product. The objective is to ensure that the organization knows the state of those components at all times. As we have frequently pointed out, software is an abstract entity, so the only way it can be reliably managed is through the consistent and reliable execution of a formal and well-defined set of technical control activities.

The controls themselves are references to an identified set of technical items that are normally maintained in a formal baseline. In practical terms, that baseline identifies, defines, and assures the integrity of a given software product or service. All modifications and releases of the software or product are controlled and made available through that baseline, and any changes to the baseline are formally recorded and reported to management. Consequently, the modification and release of any technical item has to be properly instituted and then inspected for correctness. The records and reports on the status of each of these items is then maintained as part of the permanent product baseline.

The controls frequently comprise test- and audit-based behaviors. Their aim is to coordinate contract security testing activities between the various interfaces. In particular, the technical security controls ensure joint security assurance in accordance with contractual specifications. Security assurance tests should ensure that sufficient verification and validation have been done to support a conclusion that requirements are properly met at each level in the supply chain. In addition, those controls should generate reports and make

them available to all parties as specified in the contract.

A range of technical controls might be deployed to guide the technical phases of the project at each level in the supply chain. The overall aim of these approaches is to ensure that all of the participants in the supply chain follow proper practices in producing the technical work. Examples of what might be considered a control include assurance of proper resource management, proper use of data structures, proper reuse of testing practices, and general software engineering best practices. The latter might include such specifications as proper commenting, variable naming conventions, and code structure. These techniques are almost always stipulated in the internal standards and guidelines of the customer organization and passed through to each of the third-party organizations in the supply chain.

Ensuring the correctness of technical processes normally involves three types of general assurance activities, all of which usually occur during the coding and integration process: unit testing, integration testing, and qualification testing. Unit testing is handled as part of the coding process, and it normally involves customer-supplier joint security testing throughout the construction phase. Integration testing is part of joint security testing during the software integration process as the product moves up the supply chain. Qualification testing takes place once major deliverables pass through acceptance checkpoints. All three forms are typically based on product requirements and are supported by Institute of Electrical and Electronics Engineers (IEEE) standards that dictate the proper way to plan and execute each testing activity.



EXAM TIP The fact that software is procured via a supply chain does not eliminate the need to understand and verify the necessary testing regimens employed as part of the development process. These aspects are best handled via process validation and product testing.

Supplier Transitioning

The components of a supply chain transition from supplier to supplier. In the transition process, small components move up the integration ladder from

developer to integrator as the product moves from a low level of preparation to its final form. That transitioning raises a wide range of concerns that have to be addressed in a transitioning plan. First and foremost is the issue of assuring component integrity at the interface between organizations or levels.

The only place where this assurance can be addressed is in a detailed transitioning plan, which is normally captured as part of the contract. Without a detailed and comprehensive plan, errors and malicious objects likely will transition through the cracks in an ad hoc inspection process into major product vulnerabilities at the point of delivery.

Another point of concern that has to be addressed in detail in the contract is the issue of intellectual property rights. Because small parts are integrated into large systems through the supply chain process, it is possible for the intellectual property of smaller suppliers to get lost in the products of larger integrators further up the ladder. Worse, it is also possible for components that would represent a copyright violation, or even outright piracy, to transition into a product from lower levels in the supply chain.

The management controls that are embedded in a formal transition plan have to account for every one of the things that could potentially go wrong when moving from one entity or level to another. Thus, transition plans include how to label and baseline individual components and how to maintain them in escrow in a project archive in order to protect ownership rights. Plans also include the testing, review, and audit methods that will be employed to ensure that the components move from one entity or level to another transition as built and maintain contractual requirements for integrity. Finally, transition plans describe the process by which certification of correctness and authenticity will be underwritten and granted for each individual component in the overall product baseline.

Audit of Security Policy Compliance

In its general form, supplier risk assessment is an information-gathering function that focuses on understanding the consequences of all feasible risks. The risk assessment process identifies and evaluates each identified threat, determines that threat's potential impact, and itemizes the controls that will be needed to respond properly should the threat occur. In that respect, risk assessments should always answer two distinct but highly related questions. The first question is "What is the certainty of a given risk for any given

supplier?” The answer to that question is typically expressed as likelihood of occurrence. The second is “What is the anticipated impact should that risk occur?” The answer to that question is normally expressed as an estimate of the loss, harm, failure, or danger, usually in economic or human safety terms. Ideally, both of these questions can be answered in terms that decision-makers can understand and take action on.



EXAM TIP A supplier risk assessment is used to identify specific threats to the organization’s supply chain, evaluate how likely those threats are to occur, and determine the potential consequences of each threat should it materialize.

The ability to make decisions about the requisite mitigations implies the need for an explicit management control framework to guide the decision-making process. Authoritative models that dictate how to impose the necessary management control already exist. These models dictate a practical basis for identifying and characterizing threats that might impact a given supply chain. Once those threats have been identified, these models also suggest standard mitigations that can be tailored to address each priority threat. The requirement for the systematic execution of the mitigations in any given risk management process is what provides the justification for the adoption of a well-defined set of real-world risk assessment practices to inform and guide the process. And in that respect, it is also important for the organization to be able to improve its standard risk assessment process as the business model changes.

A standardized and organization-wide process provides a stable baseline to assess the specific risks that each potential supplier represents. A common assessment process also provides the basis that will allow all of the development project’s stakeholders to stay on the same page with each other when it comes to actually performing the project work. The aim of the assessment framework is to factor the identified risks and their mitigations into a “who, what, when” structure of necessary practices, as well as identify the required interrelationships.

The Certified Secure Software Lifecycle Professional (CSSLP) body of

knowledge is centered around the processes and procedures necessary to manage security risk in software. If part of your software build process is using outsourced software pieces, should their development not have the same level of scrutiny, or better, than your own? This means that understanding the secure software development practices and risk relationships from their use is understood and acknowledged as part of the risk equation for your own development efforts. This makes the software development process of your suppliers an important auditable element of verifying supplier security.

Vulnerability/Incident Notification, Response, Coordination, and Reporting

When one is using software modules from suppliers and these same modules are used by others, there is a significant chance that vulnerabilities will be discovered by outsiders on another instance and your firm will have no idea this happened. Having a connection to the vulnerability management process on suppliers for specific products can be important. The key item to remember is you are investing not just in the use of some code but are forming a partnership that runs during the time you are using the software. You become dependent upon the supplier's ability to manage their vulnerability process, because when their product has a failure, so may yours. You may not have the insight or ability to fix the code you have purchased. This means you are dependent upon the suppliers' incident response system for their products. All of the steps, from vulnerability discovery/incident notification to response, coordination, and reporting, matter and can have significant impacts to downstream users of the software products. These are seemingly nonimportant issues until things go wrong. It is imperative that these issues be resolved in the contract before the errors show up.

Maintenance and Support Structure

Maintenance and support of software are important items that must be considered during the development process. This includes materials from third-party suppliers. When you are using a supplier as a source for what ultimately becomes your software code base, a key element is how the supplier licenses the material and what exactly you are procuring. There are

numerous GitHub locations with software routines that might be of interest to your team in the crafting of the project. So, why spend time and effort inventing/creating your own solution when one is readily available? But if you use some other group's code, what is the licensing model? Is it a commercial license or community-based one? What are the rights and costs of adoption? These are all significant questions and can have significant bearing over the lifetime of a software project that the software becomes part of at build time. What are the maintenance expectations? Will the software be maintained? Will future versions have the same licensing plan? All these questions are important, because code that is adopted into your project can become a salvation in terms of development time but can also bring long-term issues surrounding licensing and maintenance.

Security Track Record

The security track record of a company is a legitimate concern when evaluating the level of risk that having it as a supplier entails. This is not a simple “if you have had an incident, you are out as a supplier” proposition. Incidents will occur to everyone; the really important part is evaluating how well the company handled the incident. This is similar to all audits, and the key question is: “Are there any repeat audit findings?”

Support Contractual Requirements

Contracts with suppliers will have a wide range of contractual requirements associated with the supplier and what it is delivering for the software project. These requirements should include details on a range of issues including intellectual property (IP) ownership, code escrow, liability, warranty, end-user license agreement (EULA), and service level agreements requirements. Not all contracts will have all of these elements, but it is important to consider each of these in relationship to the product/service being procured as the contract terms will determine how the product/service can be used and who has risk responsibilities.

Service level agreements (SLAs) were covered in detail in [Chapter 17](#). In a similar vein, the contract with the supplier will essentially set the rules that govern responsibilities for both parties on issues such as intellectual property ownership, code escrow, liability, warranty, and EULA language elements.

None of these will be standard boilerplate, and the real control is in the hands of the supplier, so unless specifically called out in the contract, all rights will revert to the supplier. Intellectual property ownership is an issue that extends back in many cases to a supplier's suppliers. If you are procuring the right to use a code base from a supplier, the supplier should attest that it has the right to convey the license to use and the terms of the license. In many cases, the supplier will retain IP ownership and convey only a license for use—understanding the conditions of use in the conveyed license is important. Code escrow is the question of who will maintain a copy of the code. This seems trivial until the supplier goes out of business and you no longer have access to the source code. Code escrow provisions can resolve this issue.

Software components are typically purchased for a specific purpose, which raises the question of liability and warranty. If something goes wrong, is there a shared responsibility for risk or impacts from issues associated with the item or service that has been procured? Or does the user assume all risk? In most cases, all risk is passed to the user, as the supplier has limited ability to manage warranty once something is embedded inside another system. What about the issues of updates and bug fixes? To what level is the supplier responsible for maintaining the service or item being procured? Do you have the right to patch third-party code?

This list of issues seems endless, but it is important to consider all risks that may come over time and have a plan agreed to in the contract for handling them. The next several sections look at some specific areas in more detail with regard to supporting security topics in contracts.

Intellectual Property

Products moving within a supply chain can always fall prey to intellectual property theft. That is mainly because most of those products are abstract entities that can be easily stolen. Software and all of its artifacts fall into the realm of intangible intellectual property. As a result of that intangibility, software is difficult to ensure against intellectual property theft. The problems of intangibility are exacerbated if the software item is part of a supply chain, since most commercial systems are integrated up from a large collection of small, easy-to-steal software items. Any one of the items in a supply chain could be at risk of intellectual property theft. Nevertheless, a number of concrete steps might be taken to address the problem of

intellectual property violations within a supply chain.

Theft of intellectual property can involve everything from plagiarism to outright pilfering of other people's products or ideas. The legal issue revolves around the concept of "tangibility." Tangible property has substance and obvious worth. It can be seen and counted and so it is harder to steal. It is possible to assign a clear value in a court of law if the theft takes place. However, intellectual property is abstract, so its value is strictly in the eye of the beholder.

Software is difficult to visualize as a separate component and hard to value. Therefore, its legal protection is complicated. For instance, it would be hard to attach a great deal of monetary value to a software program if the courts were to consider only the actual resources that went into making the product. There might be enormous creativity and inspiration involved in its design, but the actual value of the time spent will not represent its real worth, nor will the material investment, which usually amounts to nothing more than the price of the disk and packaging. So, the problem that the legal system faces is assigning penalties if the item is stolen or misappropriated. Given these roadblocks, the question is "What can be done to protect intellectual property rights within a complex environment of systems?"



NOTE Software can contain intellectual property in many forms: processes, algorithms, or coding; thus, the intellectual property can represent business value and requires appropriate protections.

The best hope for preventing the theft of intellectual property is to perform rigorous audits and inspections of items as they move up the integration ladder and then enforce penalties for infringements. So the most important step in addressing identified and documented thefts of intellectual property involves nothing more than increasing awareness in the minds of suppliers and developers that intangible products have tangible value. More importantly, there has to be a mechanism to guarantee and enforce the right of possession up and down the supply chain.

Copyrights

Copyrights are a form of intellectual property protection that extends to the specific works, giving the “creator” of the work the ability to determine who can use the work and under what conditions. This legal exclusivity is time limited and is designed to provide legal protection against the copying of work. Source code can be registered to protect the copyright.

Two specific federal regulations address intellectual property rights enforcement directly. These are the Computer Software Rental Amendments Act of 1990, which stipulates that rental, lease, or lending of copyrighted software without the authorization of the copyright owner is expressly forbidden. Likewise, the general Copyright Law (Title 17 of the U.S. Code) has been interpreted in such a way that the creation or distribution of copies of software without authorization is illegal. That is the case even if there is no express copyright given to the owner. The only exception to this law is the allowance for the user to make backup copies for archival purposes. Under this law, the unauthorized duplication of copyrighted material, including software, can lead to fines of up to \$100,000 and jail terms of up to five years per incident.

Patents

Patents are a form of intellectual property protection that is extended to novel inventions. To patent something, it must be new (or novel) and have a purpose. A patent is not a secret—in fact, the details are provided in the patent, but in return the inventor is given legal monopoly over the idea for a set period of time. Because of the technical nature of defining the patent, the use cases it covers, and other elements, this is an area that is guided by experienced patent attorneys.

Trademarks

Trademarks are a form of protection for intellectual property associated with protecting an image, a phrase, a name, or some other distinct branding of an item. Trademarks act as a symbol that in the eyes of a customer represent a product or other item, providing a means of identification. Although it would be difficult to apply for a trademark for code, program names and elements used in marketing materials can be trademarked to provide for product identification by customers.

Trade Secrets

Trade secrets are also protected by law, preventing others from using a secret formula (such as a recipe) or other method. Trade secret law requires that the firm keep the secret, but there are issues of innocent discovery, where the “secret” is independently discovered by another party. Trade secrets can be difficult to employ in software for the very reason that code can be reversed, revealing the secret. Algorithms are seldom secret, and when instantiated in code, they can be discovered.

Licensing

Software is seldom sold to customers; rather, customers receive a license to use the software under specific conditions. There are numerous legal reasons for this business model, but in the end, they all support protecting the intellectual property that goes into the creation of the product. Licensing can be viewed as a partnership between the intellectual property rights owner (licensor) and each authorized user (licensee), usually in exchange for an agreed payment (fee or royalty).

Code Escrow

With all of the rights to software being kept by the originator, there exist times where the party purchasing a license wants protections against business failure on the part of the developer. If a company adopts a piece of code into a critical business process and the firm that develops the code goes out of business, the customer has a supply chain problem. Finding a new supplier might be impossible, and to protect itself in these circumstances, a firm may request code escrow. Code escrow is an agreement that a copy of the source code be given to a neutral third party—an escrow agent—and that access to this source code becomes possible under specific circumstances, such as the business failure of the supplier or the termination of a product line.

Legal Compliance

All software items moving within a supply chain have to comply with existing laws and regulations. Compliance simply describes a condition or state in which the component elements of the system satisfy a wide range of legal and regulatory requirements. Those requirements can come from a variety of sources, including government, industry, and contracts.

Compliance is an important consideration in the supply chain risk management universe, since the failure to comply with a legal requirement or regulation can bring significant legal penalties, not just financially but also in terms of loss of reputation and even criminal consequences.

Compliance situations usually have extensive reporting requirements associated with them. As a result, the issue with compliance is not so much to ensure that controls are in place to meet the organization's legal and regulatory obligations. Instead, the challenge is to develop objective, auditable evidence that those requirements have been satisfied and then report that evidence to the appropriate legal and regulatory entities. Those entities are varied. Each of them might have overlapping and sometimes conflicting compliance requirements. As a result, it is normally considered correct practice to establish a single, unified compliance process to coordinate the diverse number of compliance obligations that the organization will face.

Chapter Review

In this chapter, you were acquainted with the necessity of specifying security requirements to supply chains as part of a risk management strategy. And just providing security expectations is not sufficient; one must audit compliance with them. There needs to be a sharing of vulnerability information as well as associated incidents with items that are procured. Specifying the maintenance and supporting requirements is important over the life of the products. The security track record of a supplier is also important as part of the data used in reviewing a firm as a suitable supplier.

The supplier contract is the end-all for all aspects of the issues between a supplier, the goods they supply, and the buyer. This contract needs to specify a whole host of critical requirements, from security requirements to process requirements to intellectual property items and legal compliance issues. The bottom line is simple; if it isn't in the contract, the risk will most likely have to be absorbed.

Quick Tips

- If you do not ensure that supplier risk is managed as part of the acquisition process, then later in the cycle if those risks become actualized, your hands may be tied with respect to your ability to

address them.

- The overall purpose of supplier risk assessment is to identify and maintain an appropriate set of risk controls within the overall supply chain.
- Understanding a supplier's secure software development practices and risk relationships from their use is an important part of the risk equation for your own development efforts.
- Third-party modules can have vulnerabilities that require mitigations.
- Suppliers bring risk into the enterprise, so understanding their security track record is important.
- Supplier contractual agreements specify the terms by which issues can be considered to be significant and how they are to be resolved; the details are very important.
- Software can contain intellectual property in many forms: processes, algorithms, or coding; thus, the intellectual property can represent business value and hence requires appropriate protections.

Questions

To further help you prepare for the CSSLP exam, answer the following questions. The correct answers are provided in the following section.

1. Why is intellectual property protection a problem for software?
 - A. Software is complex.
 - B. Software is property that can't be described.
 - C. Software is property that can't be valued.
 - D. Software is intangible property.
2. Code-level tests in a supply chain are always assured by what?
 - A. A contract-based plan
 - B. Dynamic black-box tests
 - C. Random audits
 - D. Separation of duties
3. What is the security track record of a company?

- A. An item to consider with respect to risk transference
 - B. Only an issue for government contracts
 - C. Not an issue if you have used them successfully in the past
 - D. One of many items on a checklist for certain contracts
4. What is an important issue in the lifecycle of supplied material with respect to risk?
- A. Cost of the item
 - B. Security track record of the supplier
 - C. Security fixes as part of maintenance
 - D. All of the above
5. Which of the following is typically not a supporting element of the supplier contract and terms to define the risk relationships of a deal?
- A. Definition of security operational expectations
 - B. Specific risk transference agreements
 - C. Security audit conditions associated with supplier performance
 - D. Cost of the item under contract on a per-use basis
6. Supplier transitioning concerns focus on what?
- A. Code level
 - B. Interface level
 - C. Design level
 - D. Acceptance level
7. Software authenticity requires controls to ensure what?
- A. Hashing and checksums
 - B. Good coding practice
 - C. Nonrepudiation of origin
 - D. Denial of service
8. Vendor integrity control is built around ensuring that all subcontractors know what?
- A. Who is boss

- B. All of the other participants up and down the supply chain
 - C. Their own precise contractual requirements
 - D. Mutual authentication procedures
9. In terms of risk, what should all outsourcing decisions be based on?
- A. Price and performance
 - B. The known capability of the supplier
 - C. Where the supplier is located
 - D. The timing of the delivery process
10. It should be possible to state in advance what actions a subcontractor will _____.
- A. Be paid for and which are optional
 - B. Be forced to test and review
 - C. Do and alternatively not do
 - D. Underwrite through audits

Answers

- 1. D. Software is intangible property, so it must be labeled and deemed to have tangible value.
- 2. A. All code-level tests have to be defined and enforced in a contract.
- 3. A. The security track record of a supplier is one of many factors to consider when creating a business relationship.
- 4. C. Having a maintenance agreement that includes patching and fixes to vulnerabilities is important from a risk perspective.
- 5. D. Pricing information is separate from the terms and conditions describing the business relationship.
- 6. B. Because it controls passing from one entity to another, transitioning almost always involves an interface concern.
- 7. C. Software can be authenticated only if its origin can be confirmed.
- 8. C. Subcontractor control is built around all parties in the supply chain

understanding their precise contractual requirements.

- 9. **B.** Supplier capability is one of the primary factors that govern risk.
- 10. **C.** It is almost more important to define in advance what actions a supplier is not permitted to perform.

PART IX

Appendix and Glossary

- [Appendix About the Online Content](#)
- [Glossary](#)

About the Online Content

This book comes complete with TotalTester Online customizable practice exam software with 250 practice exam questions.

System Requirements

The current and previous major versions of the following desktop browsers are recommended and supported: Chrome, Microsoft Edge, Firefox, and Safari. These browsers update frequently, and sometimes an update may cause compatibility issues with the TotalTester Online or other content hosted on the Training Hub. If you run into a problem using one of these browsers, please try using another until the problem is resolved.

Your Total Seminars Training Hub Account

To get access to the online content you will need to create an account on the Total Seminars Training Hub. Registration is free, and you will be able to track all your online content using your account. You may also opt in if you wish to receive marketing information from McGraw Hill or Total Seminars, but this is not required for you to gain access to the online content.

Privacy Notice

McGraw Hill values your privacy. Please be sure to read the Privacy Notice available during registration to see how the information you have provided will be used. You may view our Corporate Customer Privacy Policy by

visiting the McGraw Hill Privacy Center. Visit the mheducation.com site and click **Privacy** at the bottom of the page.

Single User License Terms and Conditions

Online access to the digital content included with this book is governed by the McGraw Hill License Agreement outlined next. By using this digital content you agree to the terms of that license.

Access To register and activate your Total Seminars Training Hub account, simply follow these easy steps.

1. Go to this URL: hub.totalsem.com/mheclaim
 2. To register and create a new Training Hub account, enter your e-mail address, name, and password on the **Register** tab. No further personal information (such as credit card number) is required to create an account.
- If you already have a Total Seminars Training Hub account, enter your e-mail address and password on the **Log in** tab.
3. Enter your Product Key: **2kxp-xw0p-vwzv**
 4. Click to accept the user license terms.
 5. For new users, click the **Register and Claim** button to create your account. For existing users, click the **Log in and Claim** button.

You will be taken to the Training Hub and have access to the content for this book.

Duration of License Access to your online content through the Total Seminars Training Hub will expire one year from the date the publisher declares the book out of print.

Your purchase of this McGraw Hill product, including its access code, through a retail store is subject to the refund policy of that store.

The Content is a copyrighted work of McGraw Hill, and McGraw Hill reserves all rights in and to the Content. The Work is © 2022 by McGraw Hill.

Restrictions on Transfer The user is receiving only a limited right to use the Content for the user's own internal and personal use, dependent on purchase and continued ownership of this book. The user may not reproduce, forward, modify, create derivative works based upon, transmit, distribute, disseminate, sell, publish, or sublicense the Content or in any way commingle the Content with other third-party content without McGraw Hill's consent.

Limited Warranty The McGraw Hill Content is provided on an "as is" basis. Neither McGraw Hill nor its licensors make any guarantees or warranties of any kind, either express or implied, including, but not limited to, implied warranties of merchantability or fitness for a particular purpose or use as to any McGraw Hill Content or the information therein or any warranties as to the accuracy, completeness, correctness, or results to be obtained from, accessing or using the McGraw Hill Content, or any material referenced in such Content or any information entered into licensee's product by users or other persons and/or any material available on or that can be accessed through the licensee's product (including via any hyperlink or otherwise) or as to non-infringement of third-party rights. Any warranties of any kind, whether express or implied, are disclaimed. Any material or data obtained through use of the McGraw Hill Content is at your own discretion and risk and user understands that it will be solely responsible for any resulting damage to its computer system or loss of data.

Neither McGraw Hill nor its licensors shall be liable to any subscriber or to any user or anyone else for any inaccuracy, delay, interruption in service, error or omission, regardless of cause, or for any damage resulting therefrom.

In no event will McGraw Hill or its licensors be liable for any indirect, special or consequential damages, including but not limited to, lost time, lost money, lost profits or good will, whether in contract, tort, strict liability or otherwise, and whether or not such damages are foreseen or unforeseen with respect to any use of the McGraw Hill Content.

TotalTester Online

TotalTester Online provides you with a simulation of the CSSLP exam. Exams can be taken in Practice Mode or Exam Mode. Practice Mode provides an assistance window with hints, references to the book, explanations of the correct and incorrect answers, and the option to check

your answer as you take the test. Exam Mode provides a simulation of the actual exam. The number of questions, the types of questions, and the time allowed are intended to be an accurate representation of the exam environment. The option to customize your quiz allows you to create custom exams from selected domains or chapters, and you can further customize the number of questions and time allowed.

To take a test, follow the instructions provided in the previous section to register and activate your Total Seminars Training Hub account. When you register, you will be taken to the Total Seminars Training Hub. From the Training Hub Home page, select CSSLP from the Study drop-down menu at the top of the page to drill down to the TotalTester for your book. You can also scroll to it from the list of Your Topics on the Home page, and then click the TotalTester link to launch the TotalTester. Once you've launched your TotalTester, you can select the option to customize your quiz and begin testing yourself in Practice Mode or Exam Mode. All exams provide an overall grade and a grade broken down by domain.

Technical Support

For questions regarding the TotalTester or operation of the Training Hub, visit www.totalsem.com or e-mail support@totalsem.com.

For questions regarding book content, visit www.mheducation.com/customerservice.

GLOSSARY

***-property** Pronounced “star property,” this aspect of the Bell-LaPadula security model is commonly referred to as the “no-write-down” rule because it doesn’t allow a user to write to a file with a lower security classification, thus preserving confidentiality.

3DES Triple DES encryption—three rounds of DES encryption used to improve security.

802.11 A family of standards describing network protocols for wireless devices.

802.1X An IEEE standard for performing authentication over networks.

abuse case A use case built around a work process designed to abuse a normal work process.

acceptable use policy (AUP) A policy that communicates to users what specific uses of computer resources are permitted.

acceptance testing The formal analysis that is done to determine whether a system or software product satisfies its acceptance criteria.

access A subject’s ability to perform specific operations on an object, such as a file. Typical access levels include read, write, execute, and delete.

access control Mechanisms or methods used to determine what access permissions subjects (such as users) have for specific objects (such as files).

access control list (ACL) A list associated with an object (such as a file) that identifies what level of access each subject (such as a user) has—what they can do to the object (such as read, write, or execute).

Active Directory The directory service portion of the Windows operating system that stores information about network-based entities (such as applications, files, printers, and people) and provides a structured, consistent

way to name, describe, locate, access, and manage these resources.

ActiveX A Microsoft technology that facilitates rich Internet applications and, therefore, extends and enhances the functionality of Microsoft Internet Explorer. Like Java, ActiveX enables the development of interactive content. When an ActiveX-aware browser encounters a webpage that includes an unsupported feature, it can automatically install the appropriate application so the feature can be used.

Address Resolution Protocol (ARP) A protocol in the TCP/IP suite specification used to map an IP address to a Media Access Control (MAC) address.

adware Advertising-supported software that automatically plays, displays, or downloads advertisements after the software is installed or while the application is being used.

algorithm A step-by-step procedure—typically an established computation for solving a problem within a set number of steps.

alpha testing This is a form of end-to-end testing done prior to product delivery to determine operational and functional issues.

annualized loss expectancy (ALE) How much an event is expected to cost the business per year, given the dollar cost of the loss and how often it is likely to occur. $ALE = \text{single loss expectancy} \times \text{annualized rate of occurrence}$.

annualized rate of occurrence (ARO) The frequency with which an event is expected to occur on an annualized basis.

anomaly Something that does not fit into an expected pattern.

application A program or group of programs designed to provide specific user functions, such as a word processor or web server.

ARP See Address Resolution Protocol (ARP).

asset Resources and information an organization needs to conduct its business.

asymmetric encryption Also called public key cryptography, this is a

system for encrypting data that uses two mathematically derived keys to encrypt and decrypt a message—a public key, available to everyone, and a private key, available only to the owner of the key.

attack An action taken against an asset to exploit a vulnerability in a system.

Attack Surface Analyzer A product from Microsoft designed to enumerate the elements of a system that are subject to attack.

attack surface evaluation An examination of the elements of a system that are subject to attack and mitigations that can be applied.

attack surface measurement A measurement of the relative number of attack points in the system throughout the development process.

attack surface minimization The processes used to minimize the number of attackable elements in a system.

attack tree A graphical method of examining the required elements to successfully prosecute an attack.

audit trail A set of records or events, generally organized chronologically, that record what activity has occurred on a system. These records (often computer files) are often used in an attempt to re-create what took place when a security incident occurred, and they can also be used to detect possible intruders.

auditing Actions or processes used to verify the assigned privileges and rights of a user, or any capabilities used to create and maintain a record showing who accessed a particular system and what actions they performed.

authentication The process by which a subject's (such as a user's) identity is verified.

authentication, authorization, and accounting (AAA) Three common functions performed upon system login. Authentication and authorization almost always occur, with accounting being somewhat less common.

Authentication Header (AH) A portion of the IPsec security protocol that provides authentication services and replay-detection ability. AH can be used either by itself or with Encapsulating Security Payload (ESP). Refer to RFC

2402.

availability Part of the “CIA” of security. Availability applies to hardware, software, and data, specifically meaning that each of these should be present and accessible when the subject (the user) wants to access or use them.

backdoor A hidden method used to gain access to a computer system, network, or application. Often used by software developers to ensure unrestricted access to the systems they create. Synonymous with trapdoor.

backup Refers to copying and storing data in a secondary location, separate from the original, to preserve the data in the event that the original is lost, corrupted, or destroyed.

baseline A system or software as it is built and functioning at a specific point in time. Serves as a foundation for comparison or measurement, providing the necessary visibility to control change.

baseline management The process of managing change in a system with relationship to the baseline configuration.

Bell-LaPadula security model A computer security model built around the property of confidentiality and characterized by no-read-up and no-write-down rules.

beta testing A form of end-to-end testing performed prior to releasing a production version of a system.

Biba security model An information security model built around the property of integrity and characterized by no-write-up and no-read-down rules.

biometrics Used to verify an individual’s identity to the system or network using something unique about the individual for the verification process. Examples include fingerprints, retinal scans, hand and facial geometry, and voice analysis.

BIOS The part of the operating system that links specific hardware devices to the operating system software.

black box A form of testing where the testers have zero knowledge of the

inner workings of a system.

bootstrapping A self-sustaining process that continues through its course without external stimuli.

botnet A term for a collection of software robots, or bots, that run autonomously and automatically, and commonly invisibly, in the background. The term is most often associated with malicious software, but it can also refer to the network of computers using distributed computing software.

buffer overflow A specific type of software coding error that enables user input to overflow the allocated storage area and corrupt a running program.

bug bar The defining of thresholds for bugs that determines which ones must be fixed prior to release to production.

business continuity planning (BCP) The plans a business develops to continue critical operations in the event of a major disruption.

cache The temporary storage of information before use, typically used to speed up systems. In an Internet context, refers to the storage of commonly accessed webpages, graphics files, and other content locally on a user's PC or a web server. The cache helps to minimize download time and preserve bandwidth for frequently accessed websites, and it helps reduce the load on a web server.

canonical form The simplest form of an expression, one that all variants are resolved to prior to evaluation.

capability maturity model (CMM) A structured methodology that helps organizations improve the maturity of their software processes by providing an evolutionary path from ad hoc processes to disciplined software management processes. Developed at Carnegie Mellon University's Software Engineering Institute.

centralized management A type of privilege management that brings the authority and responsibility for managing and maintaining rights and privileges into a single group, location, or area.

certificate A cryptographically signed object that contains an identity and a

public key associated with this identity. The certificate can be used to establish identity, analogous to a notarized written document.

certificate revocation list (CRL) A digitally signed object that lists all of the current but revoked certificates issued by a given certification authority. This allows users to verify whether a certificate is currently valid even if it has not expired. CRL is analogous to a list of stolen charge card numbers that allows stores to reject bad credit cards.

certification authority (CA) An entity responsible for issuing and revoking certificates. CAs are typically not associated with the company requiring the certificate, although they exist for internal company use as well (such as Microsoft). This term is also applied to server software that provides these services. The term *certificate authority* is used interchangeably with *certification authority*.

chain of custody Rules for documenting, handling, and safeguarding evidence to ensure no unanticipated changes are made to the evidence.

Challenge Handshake Authentication Protocol (CHAP) Used to provide authentication across point-to-point links using the Point-to-Point Protocol (PPP).

change control board (CCB) A body that oversees the change management process and enables management to oversee and coordinate projects.

change management A standard methodology for performing and recording changes during software development and operation.

CIA of security Refers to confidentiality, integrity, and availability, the basic functions of any security system.

client-server A model in which a client machine is employed for users, with servers providing resources for computing.

cloud computing The automatic provisioning of computational resources on demand is referred to as cloud computing.

CLR Microsoft's Common Language Runtime—an interpreter for .NET languages on a system.

code escrow An agreement for a third party to hold the source code in the event that the originator fails as a business concern, protecting the customer's investment in a program.

code signing The application of digital signature technology to software to determine integrity and authenticity.

cognitive computing The application of machine learning (ML) and artificial intelligence (AI) to solve computing problems that are resistant to other solutions.

command injection An attack against an input validation failure designed to force a malicious command to be processed on the system.

commercial off the shelf (COTS) A software system designed for commercial use.

common vulnerability enumeration (CVE) An enumeration of common vulnerability patterns in software.

common weakness enumeration (CWE) An enumeration of common weakness patterns in software that lead to vulnerabilities.

compensating controls Compensating controls are the security controls used when a direct control cannot be applied to a requirement.

complete mediation The process of ensuring a system consistently applies the required checks on every applicable occurrence.

compliance requirements Requirements that satisfy specific issues that originate from a compliance item.

confidentiality Part of the CIA of security. Refers to the security principle that states that information should not be disclosed to unauthorized individuals.

configuration auditing The process of verifying that configuration items are built and maintained according to requirements, standards, or contractual agreements.

configuration control The process of controlling changes to items that have been baselined.

configuration identification The process of identifying which assets need to be managed and controlled.

configuration item Data and software (or other assets) that are identified and managed as part of the software change management process. Also known as computer software configuration item.

configuration management The set of processes employed to create baseline configurations in an environment and managing configurations to comply with those baselines.

configuration management database (CMDB) A database that contains the information used in the process of managing change in a system.

configuration management system (CMS) The system used in the process of managing change in a software system.

configuration status accounting Procedures for tracking and maintaining data relative to each configuration item in the baseline.

constrained data item The data element in the Clark-Wilson integrity model that is under integrity control.

continuous improvement The efforts taken to systematically reduce risk in a system.

continuous integration and continuous delivery (CI/CD) The formal term used to describe the use of DevSecOps in the deployment of software.

control A measure taken to detect, prevent, or mitigate the risk associated with a threat.

cookie Information stored on a user's computer by a web server to maintain the state of the connection to the web server. Used primarily so preferences or previously used information can be recalled on future requests to the server.

copyright A form of legal protection for intellectual property, for unique work designed to prevent copying.

countermeasure *See* control.

Counter Mode with Cipher Block Chaining Message Authentication

Code Protocol (CCMP) An enhanced data cryptographic encapsulation mechanism based upon the counter mode, with CBC-MAC from AES designed for use over wireless LANs.

cracking A term used by some to refer to malicious hacking, in which an individual attempts to gain unauthorized access to computer systems or networks. *See also* hacking.

CRC *See* cyclic redundancy check (CRC).

CRL *See* certificate revocation list (CRL).

cross-site request forgery (CSRF or XSRF) A method of attacking a system by sending malicious input to the system and relying upon the parsers and execution elements to perform the requested actions, thus instantiating the attack. XSRF exploits the trust a site has in the user's browser.

cross-site scripting (XSS) A method of attacking a system by sending script commands to the system input and relying upon the parsers and execution elements to perform the requested scripted actions, thus instantiating the attack. XSS exploits the trust a user has for the site.

cryptanalysis The process of attempting to break a cryptographic system.

cryptographic agility The ability for applications to change which cryptographic algorithms or implementations they use without having to make changes to the source code.

cryptographic validation The validation of cryptographic functions to meet specific requirements.

cryptography The art of secret writing that enables an individual to hide the contents of a message or file from all but the intended recipient.

CVE *See* common vulnerability enumeration (CVE).

CWE *See* common weakness enumeration (CWE).

cyclic redundancy check (CRC) An error detection technique that uses a series of two 8-bit block check characters to represent an entire block of data. These block check characters are incorporated into the transmission frame and then checked at the receiving end.

DAC See discretionary access control (DAC).

data classification The labeling of data elements with security, confidentiality, and integrity requirements.

data custodian The party responsible for safe custody, transport, and storage of the data and implementation of business rules with assigned data elements.

data disposition The activities taken when data is no longer needed in a system to prevent exposures of information. Includes retention, destruction, and managing dependencies.

Data Encryption Standard (DES) A private key encryption algorithm adopted by the government as a standard for the protection of sensitive but unclassified information. Commonly used in triple DES, where three rounds are applied to provide greater security.

data flow diagram (DFD) A graphical representation of how data is processed in a system. A DFD can be developed at increasing levels of detail.

data loss prevention (DLP) Technology, processes, and procedures designed to detect when unauthorized removal of data from a system occurs. DLP is typically active, preventing the loss of data, either by blocking the transfer or dropping the connection.

data owner The party responsible for data content, context, and associated business rules of specified data elements.

data protection principles This term refers to privacy principles enacted in the European Union by law.

datagram A packet of data that can be transmitted over a packet-switched system in a connectionless mode.

decision tree A data structure in which each element in the structure is attached to one or more structures directly beneath it.

declarative programming A programming methodology that describes what computations should be performed and not how to accomplish them.

decommission software The tasks one uses to deal with the issues of removing an application from the environment while protecting the other

application's security and privacy, including credential removal, configuration removal, license cancellation, and archiving.

defense in depth A security principle involving overlapping systems of different controls to form a more comprehensive defense against attacks.

DES See Data Encryption Standard (DES).

digital rights management The processes employed to control the use of digital data in a system.

digital signature A cryptography-based artifact that is a key component of a public key infrastructure (PKI) implementation. A digital signature can be used to prove identity because it is created with the private key portion of a public/private key pair. A recipient can decrypt the signature and, by doing so, receive assurance that the data must have come from the sender and that the data has not changed.

disaster recovery plan (DRP) A written plan developed to address how an organization will react to a natural or manmade disaster in order to ensure business continuity. Related to the concept of a business continuity plan (BCP).

discretionary access control (DAC) An access control mechanism in which the owner of an object (such as a file) can decide which other subjects (such as other users) may have access to the object and what access (read, write, execute) these objects can have.

distributed denial-of-service (DDoS) attack A special type of DoS attack in which the attacker elicits the generally unwilling support of other systems to launch a many-against-one attack.

diversity of defense The approach of creating dissimilar security layers so that an intruder who is able to breach one layer will be faced with an entirely different set of defenses at the next layer.

Domain Name Service (DNS) The service that translates an Internet domain name (such as www.mcgraw-hill.com) into an IP address.

DREAD An acronym used in threat modeling signifying the measurement of damage potential, reproducibility, exploitability, affected users, and

discoverability.

DRP *See* disaster recovery plan (DRP).

dynamic code analysis The analysis of software code during execution.

elliptic curve cryptography (ECC) A method of public key cryptography based on the algebraic structure of elliptic curves over finite fields.

Encapsulating Security Payload (ESP) A portion of the IPsec implementation that provides for data confidentiality with optional authentication and replay-detection services. ESP completely encapsulates user data in the datagram and can be used either by itself or in conjunction with Authentication Headers for varying degrees of IPsec services.

enterprise service bus (ESB) A software architecture model used for designing and implementing the interaction between software applications in service-oriented architecture (SOA).

escalation auditing The process of looking for an increase in privileges, such as when an ordinary user obtains administrator-level privileges.

evidence The documents, verbal statements, and material objects admissible in a court of law.

exception management The process of handling exceptions (errors) during program execution.

exposure factor A measure of the magnitude of loss of an asset. Used in the calculation of single loss expectancy (SLE).

Extensible Authentication Protocol (EAP) A universal authentication framework used in wireless networks and point-to-point connections. It is defined in RFC 3748 and has been updated by RFC 5247.

fail safe The security concept that when a system fails, it does so in a manner that ensures it enters a safe or secure state upon failure.

failure mode effects analysis (FMEA) A formal method of examining the causes and mitigation of failures in a system.

false positive Term used when a security system makes an error and

incorrectly reports the existence of a searched-for object. Examples include an intrusion detection system that misidentifies benign traffic as hostile, an antivirus program that reports the existence of a virus in software that actually is not infected, or a biometric system that allows access to a system to an unauthorized individual.

File Transfer Protocol (FTP) An application-level protocol used to transfer files over a network connection.

File Transfer Protocol Secure (FTPS) An application-level protocol used to transfer files over a network connection that uses FTP over an SSL or TLS connection.

FIPS 140-2 Federal Information Processing Standard number 140-2 is a standard for the accreditation of cryptographic modules.

firewall A network device used to segregate traffic based on rules.

FISMA Acronym for the Federal Information Systems Management Act, a law describing the implementation of information security functionality in federal data processing systems.

functional requirements A requirement for a system that defines a specific task the software is to accomplish.

functional testing The testing of software for meeting defined functional requirements.

fuzzing The process of testing input validation by sending large numbers of malformed inputs to test for exploitable vulnerabilities.

governance, risk, and compliance (GRC) A term used to describe the actions an entity takes to manage corporate efforts with respect to risk via a governance and compliance structure.

government off the shelf (GOTS) A software system built to government specifications and not for general commercial use.

Gramm-Leach-Bliley A federal law with privacy requirements associated with financial institutions.

gray box A system under test where the testers have some knowledge, but

not complete knowledge, of the inner workings of the system.

hacking The term used by the media to refer to the process of gaining unauthorized access to computer systems and networks. The term has also been used to refer to the process of delving deep into the code and protocols used in computer systems and networks. *See also* cracking.

hash Form of encryption that creates a digest of the data put into the algorithm. These algorithms are referred to as one-way algorithms because there is no feasible way to decrypt what has been encrypted.

hash value *See* message digest.

Health Information Technology for Economic and Clinical Health Act (HITECH Act) An update to HIPAA, strengthening the security and privacy provisions of PHI data.

Healthcare Insurance Portability and Accountability Act (HIPAA) A federal law with provisions for security and privacy of personal health information.

identity management The processes and systems used to perform authentication and authorization on a system.

identity provider (IdP) An authentication module that uses a user-supplied security token to verify identity for authorization purposes.

impact The result of a vulnerability being exploited by a threat, resulting in a loss.

imperative programming A programming methodology that specifies the specific sequence of commands a program should execute.

incident response The process of responding to, containing, analyzing, and recovering from a computer-related incident.

infrastructure as a service (IaaS) The automatic, on-demand provisioning of infrastructure elements operating as a service; a common element of cloud computing.

intangible asset An asset for which a monetary equivalent is difficult or impossible to determine. Examples are brand recognition and goodwill.

integer overflow An attack method that uses integer overflows to force a program to result in an error that can be exploited.

integrated development environment (IDE) A set of development tools that operate together to implement elements of the software development process.

integrated risk management (IRM) A comprehensive set of processes and procedures used to handle risk collection, analysis, and reporting in the enterprise.

integration testing A form of testing to verify that models work together to achieve requirements.

integrity Part of the CIA of security, the security principle that requires that information is not modified except by individuals authorized to do so.

integrity verification processes (IVPs) The processes involved in the Clark-Wilson model that ensure integrity in constrained data items.

Internet Key Exchange (IKE) The protocol formerly known as ISAKMP/Oakley, defined in RFC 2409. A hybrid protocol that uses part of the Oakley and part of the Secure Key Exchange Mechanism for Internet (SKEMI) protocol suites inside the Internet Security Association and Key Management Protocol (ISAKMP) framework. IKE is used to establish a shared security policy and authenticated keys for services that require keys, such as IPsec.

Internet Protocol (IP) The network-layer protocol used by the Internet for routing packets across a network.

Internet Protocol Security (IPsec) A protocol used to secure IP packets during transmission across a network. IPsec offers authentication, integrity, and confidentiality services and uses Authentication Headers (AH) and Encapsulating Security Payload (ESP) to accomplish this functionality.

intrusion detection system (IDS) A system to identify suspicious, malicious, or undesirable activity that indicates a breach in computer security.

IPsec See Internet Protocol Security (IPsec).

ITIL Information Technology Infrastructure Library (ITIL) is a set of practices for IT service management designed to align IT services with business needs.

JVM Java Virtual Machine—a sandbox environment where Java byte code is executed.

Kerberos A network authentication protocol designed by MIT for use in client-server environments.

key In cryptography, a sequence of characters or bits used by an algorithm to encrypt or decrypt a message.

keyspace The entire set of all possible keys for a specific encryption algorithm.

layered security The practice of combining multiple mitigating security controls to protect resources and data in a system.

LDAP See Lightweight Directory Access Protocol (LDAP).

least common mechanism The security concept of not sharing mechanisms used to access critical resources.

least privilege A security principle in which a user is provided with the minimum set of rights and privileges that they need to perform required functions. The goal is to limit the potential damage that any user can cause.

Level Two Tunneling Protocol (L2TP) A Cisco switching protocol that operates at the data-link layer.

licensing A business model by which only the right to use code for defined purposes is conferred, rather than the sale of the product and the transfer of almost unlimited rights.

Lightweight Directory Access Protocol (LDAP) An application protocol used to access directory services across a TCP/IP network.

Lightweight Extensible Authentication Protocol (LEAP) A Cisco-developed version of EAP that was introduced prior to 802.11i to push 802.1X and WEP adoption.

load balancers A network device that distributes computing across multiple computers.

load testing The tests used to determine system performance under expected operational loads.

MAC *See* mandatory access control (MAC).

managed code Software that has its resources managed by an external sandbox-type environment, such as CLR or JVM.

managed services The outsourcing of specific operational control of services to a third party.

mandatory access control (MAC) An access control mechanism in which the security mechanism controls access to all objects (files), and individual subjects (processes or users) cannot change that access.

man-in-the-middle attack Any attack that attempts to use a network node as the intermediary between two other nodes. Each of the endpoint nodes thinks it is talking directly to the other, but each is actually talking to the intermediary.

MD5 Message Digest 5, a hashing algorithm and a specific method of producing a message digest.

message digest The result of applying a hash function to data. Sometimes also called a hash value. *See* hash.

message queuing The use of asynchronous messages, passed through queues to communicate between modules.

misuse case *See* abuse case.

mitigate Action taken to reduce the likelihood of a threat occurring.

near-field communication (NFC) A protocol for the use of radio frequency communication over very short distances to transport data.

nonrepudiation The ability to verify that an operation has been performed by a particular person or account. This is a system property that prevents the parties to a transaction from subsequently denying involvement in the

transaction.

Oakley protocol A key exchange protocol that defines how to acquire authenticated keying material based on the Diffie-Hellman key exchange algorithm.

OAuth An open standard for authentication.

open design The security concept of not relying upon secret designs to provide security.

Open Source Security Testing Methodology Manual (OSSTMM) A peer-reviewed, open-source manual of structured security testing and analysis.

Open Vulnerability and Assessment Language (OVAL) An XML-based standard for the communication of security information between tools and services.

OpenID An open standard for authentication using cooperating third parties.

operating system (OS) The basic software that handles input, output, display, memory management, and all the other highly detailed tasks required to support the user environment and associated applications.

OVAL *See* Open Vulnerability and Assessment Language (OVAL).

P2P *See* peer-to-peer (P2P).

patch A replacement set of code designed to correct problems or vulnerabilities in existing software.

patent A form of legal protection for intellectual property provided for the invention of novel items that can be demonstrated to serve a specific, useful purpose.

Payment Card Industry Data Security Standard (PCI DSS) An industry initiative to protect credit card data in transit and storage between merchants, processors, and banks.

peer-to-peer (P2P) A network connection methodology involving direct connection from peer to peer.

penetration testing A security test in which an attempt is made to

circumvent security controls to discover vulnerabilities and weaknesses. Also called a pen test.

performance testing The testing conducted to determine operational performance test levels and the ability to meet SLAs.

permissions Authorized actions a subject can perform on an object. *See also access control.*

personal health information (PHI) Personally identifiable information containing health care information.

personally identifiable information (PII) Information that can be used to identify a single person.

phishing The use of e-mail to get a target to click a link or attachment that then spreads malware.

phreaking Used in the media to refer to the hacking of computer systems and networks associated with the phone company. *See also cracking.*

PII *See personally identifiable information (PII).*

PIN Personal identification number.

plaintext In cryptography, a piece of data that is not encrypted. It can also mean the data input into an encryption algorithm that would output ciphertext.

platform as a service (PaaS) A cloud-based computing platform offered as a service.

privacy Protecting an individual's personal information from those not authorized to see it.

protected objects Part of trusted computing, a protected object is one whose existence may be known but cannot be directly interacted with.

psychological acceptability The security principle that security-related activities need to be accepted by users or they will be circumvented as part of normal operations.

public key infrastructure (PKI) Infrastructure for binding a public key to a

known user through a trusted intermediary, typically a certificate authority.

qualification testing The formal analysis that is done to determine whether a system or software product satisfies its acceptance criteria.

qualitative risk assessment The process of subjectively determining the impact of an event that affects a project, program, or business. It involves the use of expert judgment, experience, or group consensus to complete the assessment.

quantitative risk assessment The process of objectively determining the impact of an event that affects a project, program, or business. It usually involves the use of metrics and models to complete the assessment.

radio frequency identification (RFID) RFID is a technology that allows wireless, noncontact transfer of data.

RBAC *See rule-based access control (RBAC) or role-based access control (RBAC).*

recovery The act of restoring a system to proper operating condition after a security incident.

reference monitor The mechanism that enforces access control over subjects and objects.

regression testing This is a form of testing to ensure patches do not introduce new bugs and also is effective on alternative versions of the software.

release management The business process associated with the packaging and release of software to production.

relying party The party requesting authentication services in OpenID systems.

Remote Access Service (RAS) A combination of hardware and software used to enable remote access to a network.

remote code execution The execution of code on a system by an attacker; also known as arbitrary code execution.

repudiation The act of denying that a message was either sent or received.

requirements traceability matrix (RTM) A table that correlates the requirements of a system and where they are met.

residual risk Risks remaining after an iteration of risk management.

resiliency The property of a system to return to a correct state after suffering an impact from an environmental factor.

rich Internet application A browser-based application delivered via the Web that has the functional characteristics of a desktop application.

risk The possibility of suffering a loss.

risk assessment or risk analysis The process of analyzing an environment to identify the threats, vulnerabilities, and mitigating actions to determine (either quantitatively or qualitatively) the impact and likelihood of an event affecting a project, program, or business.

risk management Overall decision-making process of identifying threats and vulnerabilities and their potential impacts, determining the costs to mitigate such events, and deciding what actions are cost effective to take to control these risks.

role-based access control (RBAC) An access control mechanism in which instead of the users being assigned specific access permissions for the objects associated with the computer system or network, a set of roles that the user may perform is assigned to each user.

rule-based access control (RBAC) An access control mechanism based on rules.

safe harbor A principle of meeting EU privacy requirements with U.S.-based actions for transnational data transfers.

safeguard *See* control.

sandboxing The principle of running an application inside a container separating it from nonmediated contact with the operating system.

Sarbanes-Oxley A federal law requiring specific security considerations

associated with public companies and their accounting data.

SBOM See software bill of material (SBOM).

scanning The process of actively interrogating a system to determine its characteristics.

secure development lifecycle (SDL) A specific set of development elements designed to build security into the software development process.

security association (SA) An instance of security policy and keying material applied to a specific data flow. Both IKE and IPsec use SAs, although these SAs are independent of one another. IPsec SAs are unidirectional and are unique in each security protocol, whereas IKE SAs are bidirectional. A set of SAs is needed for a protected data pipe, one per direction per protocol. SAs are uniquely identified by destination (IPsec endpoint) address, security protocol (AH or ESP), and security parameter index (SPI).

security baseline The end result of the process of establishing an information system's security state. It is a known good configuration resistant to attacks and information theft.

security controls A group of technical, management, or operational policies and procedures designed to implement specific security functionality. Access controls are an example of a security control.

security testing Testing the security requirements of a system.

segregation or separation of duties A basic control that prevents or detects errors and irregularities by assigning responsibilities to different individuals so that no single individual can commit fraudulent or malicious actions.

service level agreement (SLA) An agreement between parties concerning the expected or contracted uptime associated with a system.

service-oriented architecture (SOA) An architecture where resources are requested and received via remote calls.

session management The processes employed to ensure that communication sessions are secure between parties and are not subject to hijacking.

single loss expectancy (SLE) Monetary loss or impact of each occurrence of

a threat. SLE = asset value × exposure factor.

single point of failure A point of a system that has the characteristics whereby a failure here could result in failure of the entire system.

single sign-on (SSO) An authentication process by which the user can enter a single user ID and password and then move from application to application or resource to resource without having to supply further authentication information.

social engineering The art of deceiving another person so that they reveal confidential information. This is often accomplished by posing as an individual who should be entitled to have access to the information.

software as a service (SaaS) The provisioning of software as a service, commonly known as on-demand software.

software bill of material (SBOM) A list of all the software components, including versions, external libraries, functions, etc., packaged in code.

software configuration management (SCM) The processes associated with the maintenance of the configuration of software in the enterprise.

spear phishing A phishing attack against a specific target in an organization.

spiral model A development model consisting of a series of repeating steps that add value with each iteration.

spoofing Making data appear to have originated from another source so as to hide the true origin from the recipient.

SQL injection The use of malicious SQL statements to compromise a system.

stress testing The use of specific test methodologies to find performance issues before release to production.

STRIDE An acronym in threat modeling for spoofing, tampering, repudiation, information disclosure, denial of service, and elevation of privilege.

subject-object-activity matrix In access control and authorization, the

subject-object-activity matrix shows the relationships between these elements for each case.

supplier risk assessment An all-hazards assessment of the specific risks associated with a supplier or supply chain element.

symmetric encryption Encryption that needs all parties to have a copy of the key, sometimes called a shared secret. The single key is used for both encryption and decryption.

syslog The standard for logging for Linux-based computer systems.

systems testing The testing of a complete system, not just the component parts.

tangible asset An asset for which a monetary equivalent can be determined. Examples are inventory, buildings, cash, hardware, software, and so on.

threat Any circumstance or event with the potential to cause harm to an asset.

threat modeling A listing of all of the methods of attacking a system and the mitigations employed to secure the system.

trade secret A recognized legal protection for information that is protected in the course of business and relates to specific competitive material that requires secrecy to protect.

trademark A form of legal protection for intellectual property associated with the identification or branding of an item.

Transmission Control Protocol (TCP) The transport-layer protocol for use on the Internet that allows packet-level tracking of a conversation.

Transport Layer Security (TLS) A newer form of SSL being proposed as an Internet standard.

trapdoor *See* backdoor.

trusted computing base (TCB) All of the hardware and software of a system that are responsible for the security of the system.

Trusted Platform Module (TPM) A hardware chip to enable trusted

computing platform operations.

type safe The property of ensuring that type errors do not occur in programs.

unconstrained data item (UDI) A data element in the Clark-Wilson model that does not have integrity managed.

unit testing The initial testing in a system, done at the unit level of a module, where a complete function can be tested.

unmanaged code Code that runs directly on a system and is responsible for its own control of system resources.

use-case A diagram of the process steps associated with a specific business function, detailing the specific requirements.

User Datagram Protocol (UDP) A connectionless protocol for the transport of data across the Internet.

user experience (UX) The human interface experience for a software system.

validation A check as to whether the software is meeting requirements. As Boehm describes it: Are we building the right product?

verification A check as to whether the software is being properly constructed. As Boehm describes it: Are we building the product right?

vulnerability A weakness in an asset that can be exploited by a threat to cause harm.

waterfall model A development model consisting of a linear series of steps without any retracing of steps.

weakest link The point in the system that is most susceptible to attack.

web application firewall A firewall that operates at the application level, specifically designed to protect web applications by examining requests at the application stack level.

white box A testing environment where the tester has complete knowledge of the inner workings of the system under test.

X.509 The standard format for digital certificates.

INDEX

A

- ABAC (attribute-based access control), 40
- abuse cases, 93–95
- acceptance of software. *See* software testing and acceptance
- accepting risk, 275–276
- access control
 - authentication. *See* authentication
 - authorization, 19
 - models, 38–41
- access control lists (ACLs)
 - attack points, 110
 - description, 38
- access control matrix model, 40
- access tokens, 11–12
- accountability, 19–20
- ACLs (access control lists)
 - attack points, 110
 - description, 38
- acoustic attacks, 128
- acquisition process
 - software components, 96–98
 - supplier security, 327–332
- actions in requirements, 56
- activation, secure, 295–296
- activities in requirements, 56
- actors in use cases, 57

adaptive methodologies, 261–262
address space layout randomization (ASLR), 128, 160, 207
administrator training, 288
advanced persistent threat (APT) attacks, 47–48
adversaries
 groups, 46–48
 types, 45–46
AI (artificial intelligence), 128
algorithms
 cryptographic, 170, 223
 digital certificates, 16
allocation of memory, 159–160
always-on computing, 121
American National Standards Institute (ANSI), 66
American Recovery and Reinvestment Act (ARRA), 73
Anderson, Ross, 23
anonymization, 89
ANSI (American National Standards Institute), 66
anti-tampering techniques, 202–204
anti-XSS libraries, 193
application programming interfaces (APIs), 135, 161, 206–207
applications
 firewalls, 142
 mobile, 127
 programming, 59
 rich Internet, 120–121
approval to operate, 301–302
APT (advanced persistent threat) attacks, 47–48
architecture. *See* software architecture
archiving data, 308
Ariane V booster, 183
arithmetic overflows, 168
ARRA (American Recovery and Reinvestment Act), 73
artifact verification, 291–292

artificial intelligence (AI), 128

ASLR (address space layout randomization), 128, 160, 207

assurance models, 44

attack surfaces

- evaluating, 110, 218–219
- measuring, 110–111
- minimizing, 111–112

attack tree models, 108–109

attribute-based access control (ABAC), 40

audits

- OSSTM, 235
- overview, 19–20
- qualification testing, 245
- security controls, 10
- security policy compliance, 333
- supply chain management, 322
- V&V testing, 244–245

authentication

- certificates, 11
- credential management, 15–18
- description, 7–9
- identity attributes, 11
- identity management, 9–11
- identity providers, 11
- identity tokens, 11–12
- implementing, 12–15
- multifactor, 9
- smart cards, 12
- software deployment, 292
- SSH keys, 12

authorization

- access control, 19
- description, 18

automated tools

IDE, 201–202
static code analysis, 182
automatic update services, 202
automation-driven continuous principles, 291
availability
 description, 6
 requirements, 7
avoiding risk, 275–276
awareness in software teams, 24

B

backtracking attacks, 192
backups
 continuity of operations, 308
 planning, 149–150
 vulnerabilities, 169
behavioral anomalies, 248
Bell-LaPadula model, 40–41
Berne Convention, 75–76
Biba integrity model, 42
bill of materials, 319
biometrics, 8
 multifactor authentication, 9
 tokenization, 13
black-box testing
 description, 218
 overview, 232–233
bootstrapping
 description, 158, 205
 installation and deployment, 294
breach notifications, 84
break tests, 222
Brewer Nash model, 43
broken cryptographic algorithms, 170

BSA standards, 263
BSIMM (Building Security In Maturity Model), 275
buffer overflows, 160, 172–173, 175
bug bars, 247, 250
bugs
 acceptance analysis, 247
 tracking, 24–25, 248–249
build environment
 IDE, 201–202
 security, 321
build vs. buy decisions, 96
Building Security In Maturity Model (BSIMM), 275
business risk, 277
byte code, 148

C

C language
 buffer overflow, 172
 coding standards, 59
 programming failures, 175
C# language, integer overflow in, 192
C++ language
 coding standards, 59
 programming failures, 175
California Consumer Privacy Act (AB 375), 87
canonicalization errors, 173–174
CAs (certificate authorities), 16–17, 138–140
CDIs (constrained data items), 42
CERT/CC (Computer Emergency Response Team Coordination Center), 172
certificate authorities (CAs), 16–17, 138–140
Certificate Revocation Lists (CRLs), 17, 139
Certificate usage field, 16, 139
certificates
 authentication, 11

credential management, 138–140
storing, 293
X.509, 16–17

changes in regulations, 303

character code sets, 173–174

character encoding, 173

Chinese Wall model, 43

Clark-Wilson security model, 42

classification of data, 77–81, 136–137

client-server architectures, 116–117

client-side exploits, 120

cloud computing

- architectures, 124–126
- supply chain risk, 320

CLR (common language runtime), 148

code analysis, 181

- code/peer review, 185–186
- objectives, 186
- questions, 195–197
- quick tips, 194
- review, 194
- static and dynamic, 181–185
- vulnerabilities, 187–194

code and coding, 157

- anti-tampering techniques, 202–204
- configuration management, 203–204
- cryptographic failures, 169–171
- declarative vs. programmatic security, 157–159, 204–205
- defensive techniques, 204–207
- downloading without integrity checks, 171
- error handling, 160
- general failures, 175–176
- interconnectivity, 167–169
- interface, 161, 206–207

learning from mistakes, 162
memory management, 159–160
obfuscation, 204
primary mitigations, 161
principles, 162–167
questions, 178–180
quick tips, 177–178
repositories, 321
reusable, 137–138, 209
review, 177
secure standards, 59
signing, 202–203, 321–322
technology solutions, 176–177
validation failures, 171–175
code escrow, 335, 337
Code Red event, 172
cognitive computing, 128
command injection attacks, 191–192
commercial off-the-shelf (COTS) software, 96
Committee on National Security Systems, 44
Common Criteria, 68
common language runtime (CLR), 148
Common Vulnerability Scoring System (CVSS), 250–251
common weakness enumeration (CWE) vulnerability categories, 187–188
community clouds, 124
compensating controls, 115
compilers
 description, 147
 flag options, 201
 switches, 148–149
complete mediation, 36, 165
compliance
 data classification, 77–81
ISO. *See* International Organization for Standardization (ISO)

logging for, 21

NIST. *See* National Institute of Standards and Technology (NIST)

policy, 333

privacy, 81–89

questions, 90–92

quick tips, 90

regulations. *See* regulations

review, 89

risk management, 273–274

suppliers, 338

component integration, 208–210

Computer Emergency Response Team Coordination Center (CERT/CC), 172

Computer Software Rental Amendments Act, 336

concurrency

- database security, 146–147
- threads, 162

confidentiality

- description, 3–4
- requirements, 7

configuration

- secure. *See* secure configuration and version control
- storing, 293

configuration management, 168–169

- overview, 32
- parameters, 159, 206
- source code and versioning, 203–204

configuration managers, 260

conformance, 274

constant connectivity, 122

constrained data items (CDIs), 42

continuity of operations, 307–309

continuous improvement, 278–279

continuous integration in software deployment, 290–291

continuous monitoring in information security, 302–303

continuous testing, 225
contractual integrity controls, 330–331
contractual requirements for suppliers, 335
contractual terms, 97
control risk, auditing, 20
control systems, 129
controls
 anti-tampering techniques, 202–204
 build environments, 201–202
 component integration, 208–210
 defensive coding techniques, 204–207
 framework, 115
 identification and prioritization, 113–115
 implementing, 200
 mitigation, 208
 questions, 211–213
 quick tips, 210–211
 review, 210
 risk, 199–200, 277
 types, 114–115
 vendor technical integrity, 331–332
cookie cutters, 87–88
coordination in vulnerabilities and incidents, 334
Copyright Law, 336–337
copyrights, 75–76, 336–337
core concepts
 accountability, 19–22
 authentication. *See* authentication
 authorization, 18–19
 availability, 6–7
 confidentiality, 3–4
 integrity, 4–6
 nonrepudiation, 22
 overview, 3

questions, 27–29
quick tips, 27
review, 26
corrective controls, 114
COTS (commercial off-the-shelf) software, 96
credentials
 hard-coded, 169
 managing, 15–18, 138–141
 SSO, 18
 storing, 292–293
 X.509, 16–17
CRLs (Certificate Revocation Lists), 17, 139
cross-platform/system integration, 121
cross-site request forgery (CSRF), 193–194
cross-site scripting (XSS), 193
crowd sourcing, 236
cryptology
 cryptographic agility, 158–159, 205–206
 encryption. *See* encryption
 failures, 169–171
crystal methodologies, 262
CSF (Cybersecurity Framework), 45
CSRF (cross-site request forgery), 193–194
culture factor in secure software development, 277–278
custodians, data, 80
customer-facing privacy policies, 82
CVSS (Common Vulnerability Scoring System), 250–251
CWE (common weakness enumeration) vulnerability categories, 187–188
CWE/SANS top 25 vulnerability categories, 187–188
Cybersecurity Framework (CSF), 45

D

DAC (discretionary access control), 38–39
damage, reproducibility, exploitability, affected users, discoverability

(DREAD) model, 109, 248–249
DAST (dynamic application security testing), 183–184
data
 anonymization, 89
 archiving, 308
 classification, 77–81, 136–137
 collecting and analyzing, 302
 custodians, 80
 design considerations, 136–137
 disposal, 150, 267
 end-of-life policies, 266–267
 lifecycle, 79
 masking, 88
 minimization, 88
 ownership, 79
 risk impact, 78–79
 storage, 292–293
 types, 136–137
Data Encryption Standard (DES), 170
data flow diagrams (DFDs), 43, 106–107
data loss prevention (DLP) technologies, 142
data remanence attacks, 128
databases
 defect, 249
 security for, 145–147
declarative security
 vs. imperative security, 157–159
 vs. programmatic security, 204–205
decommissioning software, 265–266
decomposition, system, 106
defect databases, 249
defense functions, missing, 174
defense in depth, 34, 164
defensive coding techniques, 204

bootstrapping, 205
configuration parameters, 206
declarative vs. programmatic security, 204–205
interface coding, 206–207
memory management, 207
delivery pipeline in software deployment, 290–291
deployment. *See* secure software deployment
DES (Data Encryption Standard), 170
design considerations. *See* software design considerations
destruction of data, 150
detection risk, auditing, 20
detective controls, 114
DevOps
 Agile, 151
 release management, 289–290
DevSecOps movement, 151
DFDs (data flow diagrams), 43, 106–107
digital certificates
 authentication, 11
 credential management, 138–140
 storing, 293
 X.509, 16–17
Digital Millennium Copyright Act, 76
digital signatures, 202–203
digitally signed components, 321–322
directory climbing attacks, 192–193
directory traversal attacks, 192–193
dirty reads in databases, 147
disaster recovery (DR), 308–309
Disaster Recovery/Business Continuity Planning (DR/BCP) requirements, 55
disasters, natural, 45
disciplined patching, 307
disclosure statements for privacy, 82
discretionary access control (DAC), 38–39

disposal
 data, 150, 267
 software, 265–266

distributed computing, 116–117

diversity defense, 34

DLP (data loss prevention) technologies, 142

document object model (DOM-based) XSS attacks, 193

documentation
 RTM, 95
 security, 264
 supply chain, 97–98
 V&V, 242–243

DOM-based (document object model) XSS attacks, 193

dot-dot-slash attacks, 192–193

downloading code without integrity checks, 171

DR (disaster recovery), 308–309

DR/BCP (Disaster Recovery/Business Continuity Planning) requirements, 55

DREAD (damage, reproducibility, exploitability, affected users, discoverability) model, 109, 248–249

DSDM (dynamic systems development method), 262

dynamic application security testing (DAST), 183–184

dynamic code analysis, 181–184

dynamic linking, 147

dynamic systems development method (DSDM), 262

E

EALs (Evaluation Assurance Levels), 68

economy of mechanism, 35, 165

education

 security, 278

 software deployment, 288

education for software teams, 24

Electronic Product Code (EPC) tags, 123

elite hacker group, 46

embedded systems, 123–124
enclaves, memory, 177
encoding, character, 173
encryption
 broken and risky algorithms, 170
 cryptographic agility, 158–159, 205–206
 cryptographic failures, 169–171
 database security, 145–146
 digitally signed components, 321–322
 privacy-enhancing technology, 87–88
 sensitive data, 169
 validation, 222–224
end-of-life policies, 266–267
enterprise service bus (ESB), 118–119
environment
 build, 321
 deployment, 59–60, 287–288
 hardening, 295
 IDE, 201
 testing, 233–234
EPC (Electronic Product Code) tags, 123
errors
 canonicalization, 173–174
 handling, 59, 160, 168
 tracking, 249–250
 trapping, 59
ESB (enterprise service bus), 118–119
Evaluation Assurance Levels (EALs), 68
evaluation of attack surfaces, 110–113, 218–219
exception management, 32, 160, 168
exposure factor for risk, 277
eXtensible Access Control Markup Language (XACML), 40
Extensions field for digital certificates, 16, 139
extreme programming (XP), 262

F

fail-safe design, 35, 164

failures

- cryptographic, 169–171
- single points, 37, 167
- testing for, 222
- validation, 171–175

faults, 248

feature-driven development (FDD), 262

Federal Financial Institutions Examination Council (FFIEC), 74

Federal Information Processing Standards (FIPS)

- confidentiality, integrity, and availability defined, 7
- cryptography, 223–224
- data classifications, 81
- description, 70
- publications, 71

Federal Information Security Management Act (FISMA), 70, 72

federated ID systems, 14, 18, 141

FFIEC (Federal Financial Institutions Examination Council), 74

file integrity monitoring (FIM), 200

Financial Modernization Act, 73

Financial Privacy rule in GLBA, 73

fingerprinting, OS, 221

FIPS. *See* Federal Information Processing Standards (FIPS)

firewalls, 141–142

flaws, 248

flow control, 141–142

forensics, 305–306

frameworks, security, 263–264

functional requirements

- design considerations, 136
- software, 55–59

functional testing, 231

functionality, undocumented, 247

fuzz testing, 25, 220–221
gates in security requirements, 24

G

General Data Protection Regulation (GDPR), 84–87
generation of data, 79
geofencing, 122
gets function, 172
GLBA (Gramm-Leach-Bliley Act), 73, 82
good enough security, 33, 162
governance, risk management, and compliance (GRM), 274
government off-the-shelf (GOTS) software, 96
Gramm-Leach-Bliley Act (GLBA), 73, 82
gray-box testing, 233
GRM (governance, risk management, and compliance), 274
guidance for security, 278
Guide to Building Secure Web Applications and Web Services, 189
guidelines for risk management, 274–275
Gunter, Dan, 113

H

hackers, 46
hard-coded credentials, 169
hardening environment, 295
hardware platform concerns, 127–128
hardware security modules (HSMs), 127
hash functions
 attacks on, 170
 one-way without salts, 171
Health Information Technology for Economic and Clinical Health Act
 (HITECH Act), 73
Healthcare Insurance Portability and Accountability Act (HIPAA), 73
Heartbleed, 319

hidden data, 78
high impact, 81
high-risk data, 78
highly structured threats, 46–48
HIPAA (Healthcare Insurance Portability and Accountability Act), 73
HITECH Act (Health Information Technology for Economic and Clinical Health Act), 73
HSMs (hardware security modules), 127
hunting threats, 113
hybrid clouds, 124
hyper-connectedness computing, 121

I

IaaS (Infrastructure as a Service), 117, 126
IAM (identity and access management), 10
IAST (interactive application security testing), 184
IDE (Integrated Development Environment), 201–202
identification
 controls, 113–115
 security objectives, 106
 supply chain risk, 318
 threats, 107–108
identity and access management (IAM), 10
identity attributes, 11
identity management (IDM), 9–11
identity providers (IdPs), 11, 14
identity tokens, 11–12
IDM (identity management), 9–11
IdPs (identity providers), 11, 14
IDSs (intrusion detection systems), 304
IEC (International Electrotechnical Commission), 66
impact
 improperly handled data, 81
 risk, 277

supplier sourcing, 330
threats, 109

imperative programming, 157–158, 205

improvement, continuous, 278–279

in-band management, 134

incident triage, 305

incidents

- notification, coordination, and reporting, 334
- response, 303–306, 334

independent verification and validation (IV&V), 244–245

industry security policies, 70–71

infinite loops, 58, 176

information flow models, 43–45

information security continuous monitoring (ISCM), 302–303

Information Technology Laboratory (ITL), 70

Infrastructure as a Service (IaaS), 117, 126

inherent risk, auditing, 20

initial program load (IPL), 294

injection attacks

- command, 191–192
- overview, 189
- SQL, 190–191

injection of secrets, 296

input data

- defined, 78
- validation failures, 171–174

insider information, 43

insider threats, 48

installation, secure, 293–294

Institute for Security and Open Methodologies (ISECOM), 235

integer overflow, 192

Integrated Development Environment (IDE), 201–202

integrated risk management (IRM)

- legal issues, 274

overview, 273
regulations and compliance, 273–274
standards and guidelines, 274–275

integration
components, 208–210
software deployment, 289
systems, 121
test cases, 225

integrity
code downloading, 171
description, 4–5
implementing, 5–6
requirements, 7

integrity-based models, 42

integrity verification processes (IVPs), 42

intellectual property rights
issues, 74–77
suppliers, 335–337

intelligence, threat, 112

interactive application security testing (IAST), 184

interconnections
configuration management, 168–169
exception management, 168
session management, 167
supply chain, 320

interface
coding, 161, 206–207
design considerations, 133–135

internal data, 78

International Electrotechnical Commission (IEC), 66

International Organization for Standardization (ISO), 66–67
ISO 2700X standards, 67–68
ISO 9126 standard, 68–69
ISO 12207 standard, 69

ISO 15408 standards, 68
ISO 21827 standard, 235
ISO 25010 standard, 234–235
ISO 33001 standard, 69–70
Internet of Things (IoT), 121
interpreters, 147
intrusion detection and response, 302–303
intrusion detection systems (IDSs), 304
intrusion prevention systems (IPSs), 304
IoT (Internet of Things), 121
IPL (initial program load), 294
IPSs (intrusion prevention systems), 304
IRM (integrated risk management). *See* integrated risk management (IRM)
ISCM (information security continuous monitoring), 302–303
ISECOM (Institute for Security and Open Methodologies), 235
ISO. *See* International Organization for Standardization (ISO)
Issuer field for digital certificates, 16, 139
ITL (Information Technology Laboratory), 70
IV&V (independent verification and validation), 244–245
IVPs (integrity verification processes), 42

J

Java Virtual Machine (JVM), 148
JavaScript Object Notation (JSON), 119
JTC 1 - Information Technology standards, 67
JVM (Java Virtual Machine), 148

K

Kaminsky, Dan, 164
keep-it-simple principle, 35
keys
cryptographic, 223–224
SSH, 12

storing, 293

L

- labeling, 80–81
- layered security, 34
- lead auditors, 244–245
- least common mechanism, 36, 165–166
- least privilege approach
 - coding, 163
 - overview, 33
 - software deployment, 295
- legal compliance by suppliers, 338
- legal issues
 - description, 74
 - risk management, 274
- levels
 - auditing, 19–20
 - authorization, 18
- levels of confidence, 44
- leveraging existing components, 37, 166
- libraries
 - anti-XSS, 193
 - reuse, 209
 - safe, 201
- licensing, 337
- lifecycle
 - data, 79
 - SDL, 22–26
- linking code, 147
- lint tools, 182
- load testing, 232
- local logging, 21
- locality principle for memory, 160, 207
- location-based data, 122

locks, 58, 176
logging and log files
 bugs, 248
 decisions, 21
 design considerations, 134–135
 forensics, 305–306
 overview, 19–22
 retaining, 150
 secure coding standards, 59
 syslog, 22
 vulnerabilities, 169
loops, infinite, 58, 176
lost updates in databases, 147
low impact, 81
low-risk data, 78
low-water-mark policy, 42

M

MAC (mandatory access control), 38
machine learning (ML), 128
Madrid System, 76
maintaining software, 334
managed code vs. unmanaged, 149
management V&V, 242–243
mandatory access control (MAC), 38
masking data, 88
MD-5 hash algorithms, 170
measurement of attack surfaces, 110–111
mediation, complete, 36, 165
medium impact, 81
medium-risk data, 78
memory
 enclaves, 177
 managing, 159–160, 207

message queuing, 117
metadata, 80
metrics
 OSSTMM, 235
 quality-of-use, 68–69
 risk scoring, 250–251
 security, 264–265
 security status, 267–268
Microsoft Intermediate Language (MSIL), 148
minimization
 attack surfaces, 111–112
 data, 88
missing defense functions, 174
mission-critical software, 285–286
mistakes, learning from, 162
misuse cases, 93–95
 questions, 99–101
 quick tips, 98
 requirements traceability matrix, 95
 review, 98
 software acquisition, 96–98
mitigation
 analysis, 108–109
 primary, 161, 208
 risk, 199, 275–276
 SDL, 25–26
MITRE
 risk scoring, 250
 vulnerabilities, 187–188
ML (machine learning), 128
mobile applications, 127
moderate impact, 81
modular development, 209
Morris worm, 172

MSIL (Microsoft Intermediate Language), 148
multifactor authentication, 9
multilevel security model, 41–42
mutual authentication, 18
mutual exclusion, 58, 176

N

n-tier model, 116
nation-state threats, 47–48
National Institute of Standards and Technology (NIST), 70
 cloud-based computing, 124
 CSF model, 45
 hash functions, 170
 publications, 71
 risk management framework, 72
 SP 800 series, 70
natural disasters, 45
near-field communication (NFC) protocol, 123
network firewalls, 141–142
next-generation firewalls, 141
NFC (near-field communication) protocol, 123
NIST. *See* National Institute of Standards and Technology (NIST)
no-read-up rule, 40
no-write-down rule, 41
no-write-up rule, 42
NoForn system, 42
nonfunctional requirements
 design, 136
 security testing, 231–232
nonpersistent XSS attacks, 193
nonrepeatable database reads, 147
nonrepudiation, 22
notifications
 breaches, 84

vulnerabilities and incidents, 334

O

OAuth protocol, 14–15

obfuscation, code, 204

objectives, security, 106

objects

 authorization, 18

 defined, 56

observable data, collecting and analyzing, 302

one-way hashes without salts, 171

Online Certificate Status Protocol (OCSP), 17, 139

open design, 36, 165

Open Group Library, 151

Open Security Architecture (OSA), 151

Open Source Security Testing Methodology Manual (OSSTMM), 235

Open Web Application Security Project (OWASP)

 architecture design, 151–152

 overview, 71

 software contract elements, 97–98

 standards, 263

 vulnerability categories, 188–194

 web applications, 275

OpenID protocol, 14–15

OpenVPN, 319

operating systems

 controls and services, 149

 fingerprinting, 221

operational architecture, 151

operational models of security, 44

operational requirements, 59–60

Operationally Critical Threat, Asset, and Vulnerability Evaluation (OCTAVE), 71

opportunity costs in supplier sourcing, 330

OSA (Open Security Architecture), [151](#)
OSSTMM (Open Source Security Testing Methodology Manual), [235](#)
out-of-band management, [134](#)
output data
 classification, [78](#)
 validation failures, [174](#)–[175](#)
outsider threats, [48](#)
outsourcing, [96](#)–[97](#)
overflows
 arithmetic, [168](#)
 buffer, [160](#), [172](#)–[173](#), [175](#)
 integer, [192](#)
OWASP. *See* Open Web Application Security Project (OWASP)
ownership of data, [79](#)

P

PA DSS (Payment Application Data Security Standard), [73](#)–[74](#)
PaaS (Platform as a Service), [117](#), [125](#)
parameters, configuration, [159](#), [206](#)
passwords
 authentication, [8](#), [13](#)–[14](#)
 hard-coded, [169](#)
 hashes, [171](#)
 identity management, [10](#)–[11](#)
past mistakes, learning from, [162](#)
patches
 managing, [306](#)
 in regression testing, [224](#)
patents, [75](#), [337](#)
path traversals, [192](#)–[193](#)
patterns in architecture design, [151](#)–[152](#)
Payment Application Data Security Standard (PA DSS), [73](#)–[74](#)
Payment Card Industry Data Security Standard (PCI DSS), [73](#)–[74](#)
pedigree of software, verifying, [319](#)–[320](#)

PEDs (PIN entry devices), 74
peer reviews, 185–186
peer-to-peer architectures, 117
penetration testing, 219–220
performance testing in service level agreements, 232
persistent attacks
 APT, 47–48
 XSS, 193
personal health information (PHI), 73, 83–84
personally identifiable information (PII), 78, 83, 274
personnel training for software deployment, 288
pervasive/ubiquitous computing, 121–123
PET (privacy-enhancing technology), 87–88
PHI (personal health information), 73, 83–84
PII (personally identifiable information), 78, 83, 274
PIN entry devices (PEDs), 74
PIN Transaction Security (PTS), 74
PKI (public key infrastructure), 16–17, 139
PKIX (Public Key Infrastructure X.509), 16–17, 139
plans for secure testing, 229–231
Platform as a Service (PaaS), 117, 125
policies
 auditing, 20
 bootstrapping, 294
 compliance audits, 333
 data management, 80
 design principles, 151
 end-of-life, 266–267
 IAM, 10
 industries, 70–71
 privacy, 82
 software deployment, 296
post-deployment security testing, 296–297
power attacks, 128

power on self-test (POST), 294
power users, training, 288
PP (Protection Profile) in Common Criteria, 68
“Practical Model for Conducting Cyber Threat Hunting,” 113
predictive methodologies, 262–263
Pretexting Provision in GLBA, 73
preventive controls, 114
primary mitigations, 161, 208
principles in architecture design, 151–152
prioritization, 113–115
privacy
 anonymization, 89
 California Consumer Privacy Act, 87
 data masking, 88
 data minimization, 88
 General Data Protection Regulation, 84–87
 overview, 81–82
 personally identifiable information, 83
 policies, 82
 privacy-enhancing technology, 87–88
 pseudo-anonymization, 89
 tokenization, 88–89
 privacy disclosure statements, 82
 privacy-enhancing technology (PET), 87–88
 private clouds, 124
 privilege management, 33, 146, 163
 process assessment standard, 69–70
 product quality, 68–69
 production data reuse, 252
 programmatic security, 204–205
 programming
 APIs, 161
 application, 59
 coding practices. *See* code and coding

programming language environments, 147–149
Protection of Information in Computer Systems, 33
Protection Profile (PP) in Common Criteria, 68
protocol choices in design, 135
provenance of software, verifying, 319–320
proxies, 142
pseudo-anonymization, 89
pseudo-random functions, 170, 223
psychological acceptability, 36–37, 166
PTS (PIN Transaction Security), 74
public clouds, 125
Public key field for digital certificates, 16, 139
public key infrastructure (PKI), 16–17, 139
Public Key Infrastructure X.509 (PKIX), 16–17, 139

Q

qualification testing, 230, 245–246
quality vs. security, 22–23

R

race conditions, 58, 175–176
radio frequency identification (RFID), 122–123
rainbow series of manuals, 8
rainbow tables, 171
random numbers, 170, 223
RAs (registration authorities), 16, 138
RASP (runtime application self-protection), 184–185, 307
RBAC (role-based access control), 39–40
RBAC (rule-based access control), 40
reasonably believed factor in breach notifications, 84
recoverability, 232
registration authorities (RAs), 16, 138
regression testing, 224–225

regulations
 changes, 303
 GLBA, 73
 HIPAA and HITECH, 73
 intellectual property, 74–77
 legal issues, 74
 overview, 65–66
 PCI DSS, 73–74
 risk management, 273–274
 Sarbanes-Oxley Act, 73
 security standards, 66
release management, 289–290
relying parties (RPs) in federated ID systems, 14
remediation of risk, 199
remote code execution, 121
reports
 security status, 267–268
 vulnerabilities and incidents, 334
repositories, code, 321
Representational State Transfer (REST) web services, 118–119
requestors for authorization, 18
requirements, software. *See* software security requirements
requirements traceability matrix (RTM), 95
residual risk
 auditing, 20
 description, 277
 removing, 200, 275
resiliency, system, 309
response for vulnerabilities and incidents, 303–306, 334
REST (Representational State Transfer) web services, 118–119
restoration planning, 149–150
retention of data, 150, 308
retrieval of data, 150
reuse

code, 209
plans, 137–138
production data, 252
reverse engineering, 36
reviews
 architecture and design, 150
 code, 185–186
 security, 25
revision control. *See* secure configuration and version control
RFID (radio frequency identification), 122–123
rich Internet applications (RIAs), 120–121
risk assessment
 design considerations, 135
 reusable code, 137–138
risk management
 auditing, 20
 continuous improvement, 278–279
 education and guidance, 278
 IRM, 273–277
 operational risk analysis, 285–287
 options, 275–276
 questions, 280–282
 quick tips, 279–280
 review, 279
 security culture, 277–278
 supply chain. *See* supply chain risk management
 technical vs. business, 277
 terminology, 276–277
risk management framework (RMF), 72
risk scoring, 250–251
risky cryptographic algorithms, 170
RMF (risk management framework), 72
role-based access control (RBAC), 39–40
roles, 56

root-cause analysis, 304
RPs (relying parties) in federated ID systems, 14
RTM (requirements traceability matrix), 95
rule-based access control (RBAC), 40
runtime application self-protection (RASP), 184–185, 307
runtime protection, 307

S

SaaS (Software as a Service), 117, 125
SABSA framework, 151
Safe Harbor principles, 84–85
safe libraries, 201
SAFECode (Software Assurance Forum for Excellence in Code)
publications, 70–71, 94
standards, 263, 275
Safeguards rule in GLBA, 73
salts, one-way hashes without, 171
SAMM (Software Assurance Maturity Model), 275
sandboxing, 149
Sarbanes-Oxley (SOX) Act, 10, 73
SAST (static application security testing), 182–183
SBOM (software bill of materials), 97, 319
SCA (software composition analysis), 209
SCADA (supervisory control and data acquisition) systems, 129
scanning, 221
SCAP (Security Content Automation Protocol), 72
script kiddies, 45
Scrum methodologies, 262
SDL (secure development lifecycle)
components, 23–26
overview, 22
security vs. quality, 22–23
secrets
injection, 296

storing, 293

secure coding standards, 59

secure configuration and version control, 203–204, 303

- adaptive methodologies, 261–262
- data disposition, 267
- decommissioning software, 265–266
- documentation, 264
- end-of-life policies, 266–267
- metrics, 264–265
- overview, 259–260
- predictive methodologies, 262–263
- questions, 269–272
- quick tips, 268–269
- review, 268
- software development methodology, 261–263
- standards and frameworks, 263–264
- status reports, 267–268
- strategy and roadmap, 260–261

secure design principles

- models. *See* security models
- overview, 162–167
- questions, 50–52
- quick tips, 50
- review, 49
- tenets, 33–37

secure development lifecycle (SDL)

- components, 23–26
- overview, 22
- security vs. quality, 22–23

Secure Shell (SSH) protocol, 12

secure software deployment, 285

- activation, 295–296
- artifact verification, 291–292
- bootstrapping, 294

continuous integration and delivery pipeline, 290–291
data storage, 292–293
environment, 59–60, 287–288
environment hardening, 295
installation, 293–294
least privilege approach, 295
operational risk analysis, 285–287
personnel training, 288
policy implementation, 296
post-deployment security testing, 296–297
questions, 298–300
quick tips, 297–298
release management, 289–290
review, 297
secrets injection, 296
system integration, 289
tool chain, 291
secure transfer of software, 320
security champions, 278
Security Content Automation Protocol (SCAP), 72
security features vs. secure software, 23
security information and event management (SIEM) tool, 21
security-level auditing, 19
security models, 37–38
 access control, 38–41
 adversaries, 45–48
 information flow, 43–45
 integrity-based, 42
 multilevel, 41–42
 threat landscape shifts, 48–49
security overview
 programming. *See* code and coding
 standards. *See* standards
security-sensitive data, 78

Security Target (ST) in Common Criteria, 68
Seitz, Marc, 113
sensitive data, encryption for, 169
sensitivity of data, 80
sensor networks, 123
separation of duties, 33–34, 163–164
sequencing and timing, 58, 175–176
Serial number field for digital certificates, 16, 139
service level agreements (SLAs)
 contractual terms, 97
 performance testing, 232
 purpose, 309–310
Service Level Objectives (SLOs), 309–310
service-oriented architecture (SOA), 117–119
services, web, 119–120
session management, 31–32, 159, 167
SHA-1 hash algorithms, 170
shared responsibility model, 126
side channel attacks, 128
SIEM (security information and event management) tool, 21
Signature algorithm field for digital certificates, 16, 139
signing, code, 202–203, 321–322
Simple Object Access Protocol (SOAP), 118
Simple Security Rule, 40–41
simplicity principle, 35
simulation testing, 221–222
single points of failure, 37, 167
single sign-on (SSO), 14, 18, 140–141
Slammer event, 172
SLAs (service level agreements)
 contractual terms, 97
 performance testing, 232
 purpose, 309–310
SLOs (Service Level Objectives), 309–310

smart cards, 12
SOA (service-oriented architecture), 117–119
SOAP (Simple Object Access Protocol), 118
social engineering attacks, 128
software
 acceptance. *See* software testing and acceptance
 acquisition, 96–98
 bill of materials, 319
 code analysis. *See* code analysis
 code repositories, 321
 coding. *See* code and coding
 decommissioning, 265–266
 deployment. *See* secure software deployment
 end-of-life policies, 266–267
 patents and copyrights, 75–76
 pedigree and provenance verification, 319–320
 risk. *See* risk management
 secure transfer, 320
 third-party, 318–319
software architecture
 cloud, 124–126
 cognitive computing, 128
 control identification and prioritization, 113–115
 control systems, 129
 defining, 113
 distributed computing, 116–117
 embedded systems, 123–124
 hardware platform concerns, 127–128
 mobile applications, 127
 operational, 151
 pervasive/ubiquitous computing, 121–123
 questions, 130–132
 quick tips, 129–130
 review, 129

rich Internet applications, 120–121
risk assessment, 135
service-oriented architecture, 117–119
shared responsibility model, 126
threat modeling. *See* threat modeling
Software as a Service (SaaS), 117, 125
Software Assurance Forum for Excellence in Code (SAFECode)
publications, 70–71, 94
standards, 263, 275
Software Assurance Maturity Model (SAMM), 275
software bill of materials (SBOM), 97, 319
software composition analysis (SCA), 209
software design considerations, 133
 architectural risk assessment, 135
 backups, 149–150
 credential management, 138–141
 data loss prevention, 142
 data retention, 150
 databases, 145–147
 flow control, 141–142
 interface, 133–135
 logging, 134–135
 models and data classification, 136–137
 operating systems, 149
 operational architecture, 151
 programming language environment, 147–149
 protocol choices, 135
 questions, 152–154
 quick tips, 152
 reusability, 137–138
 review, 152
 review process, 150
 secure architecture, 151–152
 session management, 31–32

tenets, 31–33
trusted computing, 143–144
virtualization, 143

software development and testing
 SDL. *See* secure development lifecycle (SDL)
 security culture, 277–278

software operations and maintenance
 approval to operate, 301–302
 continuity of operations, 307–309
 forensics, 305–306
 incident response, 303–306
 incident triage, 305
 information security continuous monitoring, 302–303
 overview, 301
 patch management, 306
 questions, 312–314
 quick tips, 311
 review, 310
 root-cause analysis, 304
 runtime protection, 307
 service level agreements, 309–310
 vulnerability management, 306–307

software security requirements, 55
 functional, 55–59
 operational, 59–60
 questions, 62–64
 quick tips, 61–62
 review, 61
 summary, 60–61

software teams for code review, 185–186

software testing and acceptance, 241
 error classification and tracking, 248–251
 qualification testing, 230, 245–246
 questions, 253–255

quick tips, 252–253
review, 252
risk, 200
test data, 251–252
test result implications, 247
 undocumented functionality, 247
 verification and validation testing, 242–246
software vulnerabilities. *See* vulnerabilities and countermeasures
something about you authentication factor, 7–9
something you have authentication factor, 7–9
something you know authentication factor, 7–9
source code
 analyzers, 182
 configuration management, 203–204
sourcing, supplier, 328–330
SOX (Sarbanes-Oxley) Act, 10, 73
SQL (Structured Query Language) injection attacks, 190–191
SQL Slammer worm, 319
SSE-CMM (Systems Security Engineering Capability Maturity Model), 235
SSH (Secure Shell) protocol, 12
SSO (single sign-on), 14, 18, 140–141
ST (Security Target) in Common Criteria, 68
standard template libraries (STLs), 137
standards, 66
 ISO. *See* International Organization for Standardization (ISO)
 NIST. *See* National Institute of Standards and Technology (NIST)
 risk management, 274–275
 security, 263–264
 testing, 234–235
star property, 41
states, data, 77–78
static application security testing (SAST), 182–183
static code analysis, 181–183
static linking, 147

status reports, 267–268
STLs (standard template libraries), 137
storage of data, 292–293
stored procedures, 145
strcpy function, 175
stress testing, 232
STRIDE method, 107–108
strncpy function, 175
structured data, 136
Structured Query Language (SQL) injection attacks, 190–191
structured threats, 47
style guides, 175
Subject field for digital certificates, 16, 139
subject-object-activity matrix, 56
subjects in authorization, 18
summary issues in databases, 147
supervisory control and data acquisition (SCADA) systems, 129
supplier security requirements, 327
 acquisition process, 97–98, 327–332
 contractual integrity controls, 330–331
 contractual requirements, 335
 intellectual property, 335–337
 legal compliance, 338
 maintenance and support structure, 334
 policy compliance audits, 333
 questions, 339–341
 quick tips, 338–339
 review, 338
 sourcing, 328–330
 track records, 334
 transitioning, 332
 vendor technical integrity controls, 331–332
 vulnerabilities, 334
supply chain risk management, 317

audits, 322
build environment, 321
code repositories, 321
code signing, 321–322
implementing, 317–318
pedigree and provenance verification, 319–320
questions, 323–325
quick tips, 322
review, 322
secure transfer, 320
system sharing/interconnections, 320
third-party software, 318–319
support, supplier, 334
syslog protocol, 22
system decomposition, 106
system integration in software deployment, 289
system logs, retaining, 150
system-of-systems integration, 209–210
system sharing in supply chain, 320
system tenets, 31–32
Systems Security Engineering Capability Maturity Model (SSE-CMM), 235

T

tags, RFID, 122–123
take-grant model, 41
tangible property, 336
Target of Evaluation (TOE) in Common Criteria, 68
TC (trusted computing), 143–144
TCB (trusted computing base), 143
TCP (Transmission Control Protocol) handshakes, 32
teams for code review, 185–186
technical risk, 277
technical V&V, 243–244
tenets

secure design, 33–37
system, 31–32

test cases
attack surface evaluation, 218–219
common methods, 220
continuous, 225
cryptographic validation, 222–224
fuzzing, 220–221
integration, 225
overview, 217–218
penetration testing, 219–220
questions, 226–228
quick tips, 226
regression, 224–225
review, 225
scanning, 221
simulation, 221–222

test data, 251–252

test harnesses, 234

testing
automating, 291
failure modes, 222
post-deployment security, 296–297
software. *See* software testing and acceptance

testing strategy, 229
crowd sourcing, 236
environment, 233–234
functional, 231
nonfunctional, 231–232
plans, 229–231
questions, 237–239
quick tips, 236–237
review, 236
standards, 234–235

techniques, 232–233

third parties

- APIs, 161
- build environment, 321
- code and libraries, 209
- code signing, 321–322
- identity management, 10
- independent testing, 244–245
- logging, 21
- risk transferring, 200, 275
- software analysis, 318–319
- vendor technical integrity controls, 331–332

thread-checking routines, 183

threads in concurrency, 162

threat intelligence, 112, 302

threat modeling

- attack surface evaluation, 110–113
- description, 25
- developing, 105–110
- threat hunting, 113
- threat intelligence, 112
- validating, 109–110

threat pictures, 287

threats in landscape shifts, 48–49

three tier model, 116

time of check/time of use (TOC/TOU) attacks, 58, 175–176

timing issues and attacks, 58, 128, 175–176

TOE (Target of Evaluation) in Common Criteria, 68

tokens and tokenization

- authentication, 8
- description, 170
- identity, 11–12
- privacy, 88–89

tool chain in software deployment, 291

top 25 vulnerability categories, 187–188
TPM (Trusted Platform Module), 127, 144
TPs (transformation processes), 42
track records of suppliers, 334
tracking bugs, 248–249
trade secrets, 76, 337
trademarks, 76, 337
training
 security, 278
 software deployment, 288
transfer of software, 320
transferring risk, 200, 275–276
transformation processes (TPs), 42
transitioning, supplier, 332
Transmission Control Protocol (TCP) handshakes, 32
triage, incident, 305
triggers for database security, 146
trust boundaries, 107–108
trusted computing (TC), 143–144
trusted computing base (TCB), 143
Trusted Platform Module (TPM), 127, 144
two-factor authentication, 9
type-safe practice, 160, 207
types of data, 136–137

U

ubiquitous computing, 121–123
UDDI (Universal Description, Discovery, and Interface), 119–120
UDP (User Datagram Protocol), 32
unconstrained data items (UDIs), 42
undocumented functionality, 247
unencrypted personal information in breach notifications, 84
Unicode support, 173–174
unit testing, 231

Universal Description, Discovery, and Interface (UDDI), 119–120
unmanaged code vs. managed, 149
unstructured data, 137
unstructured threats, 46–47
updates in databases, 147
usage, data, 78
use cases
 benefits, 56–57
 data flow diagrams, 43–44
 misuse cases. *See* misuse cases
User Datagram Protocol (UDP), 32
user definitions, 56
user experience (UX) interface, 14
user-supplied input in command injection attacks, 191–192
usernames in authentication, 8
users, training, 288
UX (user experience) interface, 14

V

V&V testing. *See* verification and validation (V&V) testing
validation
 cryptographic, 222–224
 input, 171–174
 output, 174–175
 threat model, 109–110
Validity field for digital certificates, 16, 139
VDBIR (Verizon Data Breach Investigative Report), 112
vendor technical integrity controls, 331–332
verification, artifact, 291–292
verification and validation (V&V) testing, 242
 independent, 244–245
 management, 242–243
 software qualification testing, 245–246
 technical, 243–244

Verizon Data Breach Investigative Report (VDBIR), [112](#)
Version number field for digital certificates, [16](#), [139](#)
versions
 configuration management, [203](#)–[204](#)
 controlling. *See* secure configuration and version control
views for database security, [146](#)
virtualization, [143](#)
vulnerabilities and countermeasures
 CWE/SANS top 25 vulnerability categories, [187](#)–[188](#)
 description, [248](#)
 managing, [306](#)–[307](#)
 notification, response, coordination, and reporting, [334](#)
 OWASP, [188](#)–[194](#)
 scanning, [221](#)

W

walk-throughs, code, [185](#)
warranties, [77](#)
watchdog routines, [200](#)
waterfall model, [262](#)–[263](#)
weakest links in systems, [37](#), [166](#)
web beacons, [88](#)
web services, [119](#)–[120](#)
Web Services Description Language (WSDL), [119](#)–[120](#)
white-box testing, [218](#), [232](#)
wireless communications, [121](#)–[122](#)
World Intellectual Property Organization, [76](#)
WSDL (Web Services Description Language), [119](#)–[120](#)

X

X.509 credentials, [16](#)–[17](#), [138](#)–[140](#)
XACML (eXtensible Access Control Markup Language), [40](#)
XP (extreme programming), [262](#)

XSS (cross-site scripting), [193–194](#)

Z

zero-day exploits, [46](#)

Zimmerman, Phil, [87](#)