CHAPTER

3

Cloud Computing Software Security Fundamentals

People don't ever seem to realize that doing what's right is no guarantee against misfortune.

—William McFee

Security is a principal concern when entrusting an organization's critical information to geographically dispersed cloud platforms not under the direct control of that organization. In addition to the conventional IT information system security procedures, designing security into cloud software during the software development life cycle can greatly reduce the cloud attack surface.

In the document "Security Guidance for Critical Areas of Focus in Cloud Computing,"[1] the Cloud Security Alliance emphasizes the following points relative to the secure software life cycle in their listing of 15 cloud security domains:

- Domain 6, Information Life Cycle Management — "Understand cloud provider policies and processes for data retention and destruction and how they compare with internal organizational policy. Be aware that data retention assurance may be easier for the cloud provider to demonstrate, but data destruction may be very difficult. Perform regular backup and recovery tests to assure that logical segregation and controls are effective."
- Domain 11, Application Security — "IaaS, PaaS and SaaS create differing trust boundaries for the software development lifecycle, which must be accounted for during the development, testing and production deployment of applications."
- Domain 14, Storage — "Understand cloud provider storage retirement processes. Data destruction is extremely difficult in a multi-tenant environment

and the cloud provider should be utilizing strong storage encryption that renders data unreadable when storage is recycled, disposed of, or accessed by any means outside of authorized applications."

With cloud computing providing SaaS, secure software is a critical issue. From the cloud consumer's point of view, using SaaS in the cloud reduces the need for secure software development by the customer. The requirement for secure software development is transferred to the cloud provider. However, the user might still find it necessary to develop custom code for the cloud. Whoever develops the software, this process requires a strong commitment to a formal, secure software development life cycle, including design, testing, secure deployment, patch management, and disposal. Yet, in many instances, software security is treated as an add-on to extant software and not as an important element of the development process.

These and other related issues in the secure software development life cycle for cloud computing are explored in detail in this chapter.

Cloud Information Security Objectives

Developing secure software is based on applying the secure software design principles that form the fundamental basis for software assurance. Software assurance has been given many definitions, and it is important to understand the concept. The Software Security Assurance Report[2] defines *software assurance* as "the basis for gaining justifiable confidence that software will consistently exhibit all properties required to ensure that the software, in operation, will continue to operate dependably despite the presence of sponsored (intentional) faults. In practical terms, such software must be able to resist most attacks, tolerate as many as possible of those attacks it cannot resist, and contain the damage and recover to a normal level of operation as soon as possible after any attacks it is unable to resist or tolerate."

The U.S. Department of Defense (DoD) Software Assurance Initiative[3] defines *software assurance* as "the level of confidence that software functions as intended and is free of vulnerabilities, either intentionally or unintentionally designed or inserted as part of the software."

The Data and Analysis Center for Software (DACS)[4] requires that software must exhibit the following three properties to be considered secure:

- **Dependability** — Software that executes predictably and operates correctly under a variety of conditions, including when under attack or running on a malicious host
- **Trustworthiness** — Software that contains a minimum number of vulnerabilities or no vulnerabilities or weaknesses that could sabotage the software's dependability. It must also be resistant to malicious logic.

- **Survivability (Resilience)** — Software that is resistant to or tolerant of attacks and has the ability to recover as quickly as possible with as little harm as possible

Seven complementary principles that support information assurance are confidentiality, integrity, availability, authentication, authorization, auditing, and accountability. These concepts are summarized in the following sections.

Confidentiality, Integrity, and Availability

Confidentiality, integrity, and availability are sometimes known as the *CIA triad* of information system security, and are important pillars of cloud software assurance.

Confidentiality

Confidentiality refers to the prevention of intentional or unintentional unauthorized disclosure of information. Confidentiality in cloud systems is related to the areas of intellectual property rights, covert channels, traffic analysis, encryption, and inference:

- **Intellectual property rights** — Intellectual property (IP) includes inventions, designs, and artistic, musical, and literary works. Rights to intellectual property are covered by copyright laws, which protect creations of the mind, and patents, which are granted for new inventions.
- **Covert channels** — A *covert channel* is an unauthorized and unintended communication path that enables the exchange of information. Covert channels can be accomplished through timing of messages or inappropriate use of storage mechanisms.
- **Traffic analysis** — *Traffic analysis* is a form of confidentiality breach that can be accomplished by analyzing the volume, rate, source, and destination of message traffic, even if it is encrypted. Increased message activity and high bursts of traffic can indicate a major event is occurring. Countermeasures to traffic analysis include maintaining a near-constant rate of message traffic and disguising the source and destination locations of the traffic.
- **Encryption** — *Encryption* involves scrambling messages so that they cannot be read by an unauthorized entity, even if they are intercepted. The amount of effort (*work factor*) required to decrypt the message is a function of the strength of the encryption key and the robustness and quality of the encryption algorithm.
- **Inference** — *Inference* is usually associated with database security. Inference is the ability of an entity to use and correlate information protected at one level of security to uncover information that is protected at a higher security level.

Integrity

The concept of cloud information *integrity* requires that the following three principles are met:

- Modifications are not made to data by unauthorized personnel or processes.
- Unauthorized modifications are not made to data by authorized personnel or processes.
- The data is internally and externally consistent — in other words, the internal information is consistent both among all sub-entities and with the real-world, external situation.

Availability

Availability ensures the reliable and timely access to cloud data or cloud computing resources by the appropriate personnel. Availability guarantees that the systems are functioning properly when needed. In addition, this concept guarantees that the security services of the cloud system are in working order. A denial-of-service attack is an example of a threat against availability.

The reverse of confidentiality, integrity, and availability is disclosure, alteration, and destruction (DAD).

Cloud Security Services

Additional factors that directly affect cloud software assurance include authentication, authorization, auditing, and accountability, as summarized in the following sections.

Authentication

Authentication is the testing or reconciliation of evidence of a user's identity. It establishes the user's identity and ensures that users are who they claim to be. For example, a user presents an identity (user ID) to a computer login screen and then has to provide a password. The computer system authenticates the user by verifying that the password corresponds to the individual presenting the ID.

Authorization

Authorization refers to rights and privileges granted to an individual or process that enable access to computer resources and information assets. Once a user's identity and authentication are established, authorization levels determine the extent of system rights a user can hold.

Auditing

To maintain operational assurance, organizations use two basic methods: system audits and monitoring. These methods can be employed by the cloud customer, the cloud provider, or both, depending on asset architecture and deployment.

- A *system audit* is a one-time or periodic event to evaluate security.
- *Monitoring* refers to an ongoing activity that examines either the system or the users, such as intrusion detection.

Information technology (IT) auditors are often divided into two types: internal and external. Internal auditors typically work for a given organization, whereas external auditors do not. External auditors are often certified public accountants (CPAs) or other audit professionals who are hired to perform an independent audit of an organization's financial statements. Internal auditors usually have a much broader mandate than external auditors, such as checking for compliance and standards of due care, auditing operational cost efficiencies, and recommending the appropriate controls.

IT auditors typically audit the following functions:

- System and transaction controls
- Systems development standards
- Backup controls
- Data library procedures
- Data center security
- Contingency plans

In addition, IT auditors might recommend improvements to controls, and they often participate in a system's development process to help an organization avoid costly reengineering after the system's implementation.

An *audit trail or log* is a set of records that collectively provide documentary evidence of processing, used to aid in tracing from original transactions forward to related records and reports, and/or backward from records and reports to their component source transactions. Audit trails may be limited to specific events or they may encompass all of the activities on a system.

Audit logs should record the following:

- The transaction's date and time
- Who processed the transaction

- At which terminal the transaction was processed
- Various security events relating to the transaction

In addition, an auditor should examine the audit logs for the following:

- Amendments to production jobs
- Production job reruns
- Computer operator practices
- All commands directly initiated by the user
- All identification and authentication attempts
- Files and resources accessed

Accountability

Accountability is the ability to determine the actions and behaviors of a single individual within a cloud system and to identify that particular individual. Audit trails and logs support accountability and can be used to conduct postmortem studies in order to analyze historical events and the individuals or processes associated with those events. Accountability is related to the concept of *nonrepudiation*, wherein an individual cannot successfully deny the performance of an action.

Relevant Cloud Security Design Principles

Historically, computer software was not written with security in mind; but because of the increasing frequency and sophistication of malicious attacks against information systems, modern software design methodologies include security as a primary objective. With cloud computing systems seeking to meet multiple objectives, such as cost, performance, reliability, maintainability, and security, trade-offs have to be made. A completely secure system will exhibit poor performance characteristics or might not function at all.

Technically competent hackers can usually find a way to break into a computer system, given enough time and resources. The goal is to have a system that is secure enough for everyday use while exhibiting reasonable performance and reliability characteristics.

In a 1974 paper that is still relevant today,[5] Saltzer and Schroeder of the University of Virginia addressed the protection of information stored in a computer system by focusing on hardware and software issues that are necessary to support information protection. The paper presented the following 11 security design principles:

- Least privilege
- Separation of duties

- Defense in depth
- Fail safe
- Economy of mechanism
- Complete mediation
- Open design
- Least common mechanism
- Psychological acceptability
- Weakest link
- Leveraging existing components

The fundamental characteristics of these principles are summarized in the following sections.

Least Privilege

The principle of *least privilege* maintains that an individual, process, or other type of entity should be given the minimum privileges and resources for the minimum period of time required to complete a task. This approach reduces the opportunity for unauthorized access to sensitive information.

Separation of Duties

Separation of duties requires that completion of a specified sensitive activity or access to sensitive objects is dependent on the satisfaction of a plurality of conditions. For example, an authorization would require signatures of more than one individual, or the arming of a weapons system would require two individuals with different keys. Thus, separation of duties forces collusion among entities in order to compromise the system.

Defense in Depth

Defense in depth is the application of multiple layers of protection wherein a subsequent layer will provide protection if a previous layer is breached.

The Information Assurance Technical Framework Forum (IATFF), an organization sponsored by the National Security Agency (NSA), has produced a document titled the "Information Assurance Technical Framework" (IATF) that provides excellent guidance on the concepts of defense in depth.

The IATFF encourages and supports technical interchanges on the topic of information assurance among U.S. industry, U.S. academic institutions, and U.S. government agencies. Information on the IATFF document can be found at `www.niap-ccevs.org/cc-scheme/IATF_3.1-Chapter_03-ISSEP.pdf`.

The IATF document 3.1[6] stresses the importance of the *people* involved, the *operations* required, and the *technology* needed to provide information assurance and to meet the organization's mission.

The defense-in-depth strategy as defined in IATF document 3.1 promotes application of the following information assurance principles:

- **Defense in multiple places** — Information protection mechanisms placed in a number of locations to protect against internal and external threats
- **Layered defenses** — A plurality of information protection and detection mechanisms employed so that an adversary or threat must negotiate a series of barriers to gain access to critical information
- **Security robustness** — An estimate of the robustness of information assurance elements based on the value of the information system component to be protected and the anticipated threats
- **Deploy KMI/PKI** — Use of robust key management infrastructures (KMI) and public key infrastructures (PKI)
- **Deploy intrusion detection systems** — Application of intrusion detection mechanisms to detect intrusions, evaluate information, examine results, and, if necessary, take action

Fail Safe

Fail safe means that if a cloud system fails it should fail to a state in which the security of the system and its data are not compromised. One implementation of this philosophy would be to make a system default to a state in which a user or process is denied access to the system. A complementary rule would be to ensure that when the system recovers, it should recover to a secure state and not permit unauthorized access to sensitive information. This approach is based on using permissions instead of exclusions.

In the situation where system recovery is not done automatically, the failed system should permit access only by the system administrator and not by other users, until security controls are reestablished.

Economy of Mechanism

Economy of mechanism promotes simple and comprehensible design and implementation of protection mechanisms, so that unintended access paths do not exist or can be readily identified and eliminated.

Complete Mediation

In complete meditation, every request by a subject to access an object in a computer system must undergo a valid and effective authorization procedure.

This mediation must not be suspended or become capable of being bypassed, even when the information system is being initialized, undergoing shutdown, being restarted, or is in maintenance mode. Complete mediation entails the following:

1. Identification of the entity making the access request
2. Verification that the request has not changed since its initiation
3. Application of the appropriate authorization procedures
4. Reexamination of previously authorized requests by the same entity

Open Design

There has always been an ongoing discussion about the merits and strengths of security designs that are kept secret versus designs that are open to scrutiny and evaluation by the community at large. A good example is an encryption system. Some feel that keeping the encryption algorithm secret makes it more difficult to break. The opposing philosophy believes that exposing the algorithm to review and study by experts at large while keeping the encryption key secret leads to a stronger algorithm because the experts have a higher probability of discovering weaknesses in it. In general, the latter approach has proven more effective, except in the case of organizations such as the National Security Agency (NSA), which employs some of the world's best cryptographers and mathematicians.

For most purposes, an open-access cloud system design that has been evaluated and tested by a myriad of experts provides a more secure authentication method than one that has not been widely assessed. Security of such mechanisms depends on protecting passwords or keys.

Least Common Mechanism

This principle states that a minimum number of protection mechanisms should be common to multiple users, as shared access paths can be sources of unauthorized information exchange. Shared access paths that provide unintentional data transfers are known as *covert channels*. Thus, the *least common mechanism* promotes the least possible sharing of common security mechanisms.

Psychological Acceptability

Psychological acceptability refers to the ease of use and intuitiveness of the user interface that controls and interacts with the cloud access control mechanisms. Users must be able to understand the user interface and use it without having to interpret complex instructions.

Weakest Link

As in the old saying "A chain is only as strong as its weakest link," the security of a cloud system is only as good as its weakest component. Thus, it is important to identify the weakest mechanisms in the security chain and layers of defense, and improve them so that risks to the system are mitigated to an acceptable level.

Leveraging Existing Components

In many instances, the security mechanisms of a cloud implementation might not be configured properly or used to their maximum capability. Reviewing the state and settings of the extant security mechanisms and ensuring that they are operating at their optimum design points will greatly improve the security posture of an information system.

Another approach that can be used to increase cloud system security by leveraging existing components is to partition the system into defended sub-units. Then, if a security mechanism is penetrated for one sub-unit, it will not affect the other sub-units, and damage to the computing resources will be minimized.

Secure Cloud Software Requirements

The requirements for secure cloud software are concerned with nonfunctional issues such as minimizing or eliminating vulnerabilities and ensuring that the software will perform as required, even under attack. This goal is distinct from security functionality in software, which addresses areas that derive from the information security policy, such as identification, authentication, and authorization.

Software requirements engineering is the process of determining customer software expectations and needs, and it is conducted before the software design phase. The requirements have to be unambiguous, correct, quantifiable, and detailed.

Karen Goertzel, Theodore Winograd, and their contributors in "Enhancing the Development Life Cycle to Produce Secure Software"[7] from the United States Department of Defense Data and Analysis Center for Software (DACS) state that all software shares the following three security needs:

- It must be dependable under anticipated operating conditions, and remain dependable under hostile operating conditions.
- It must be trustworthy in its own behavior, and in its inability to be compromised by an attacker through exploitation of vulnerabilities or insertion of malicious code.

- It must be resilient enough to recover quickly to full operational capability with a minimum of damage to itself, the resources and data it handles, and the external components with which it interacts.

In the following sections, cloud software considerations related to functional security and secure properties are explored in the context of software requirements engineering. Secure requirements for security-related cloud software functions generally define what the software has to accomplish to perform a task securely.

Secure Development Practices

There are many methods for developing code. Any of them can be used to develop a secure cloud application. Every development model must have both requirements and testing. In some models, the requirements may emerge over time. It is very important that security requirements are established early in the development process.

Security in a cloud application tends to be subtle and invisible. Security is prominent at only two times in the development life cycle: requirements definition and testing. At other times, deadlines, capabilities, performance, the look and feel, and dozens of other issues tend to push security to the back. This is why it is important to ensure that security requirements are prominent at the beginning of the software development life cycle.

In many respects, the tools and techniques used to design and develop clean, efficient cloud applications will support the development of secure code as well. Special attention, however, should be shown in the following areas:

- **Handling data** — Some data is more sensitive and requires special handling.
- **Code practices** — Care must be taken not to expose too much information to a would-be attacker.
- **Language options** — Consider the strengths and weakness of the language used.
- **Input validation and content injection** — Data (content) entered by a user should never have direct access to a command or a query.
- **Physical security of the system** — Physical access to the cloud servers should be restricted.

Handling Data

As the Internet continues to be a driving force in most of our everyday lives, more and more personal and sensitive information will be put on cloud servers. Requirements for handling this private information did not exist five years ago, while other data, such as passwords, has always required special handling. Following are some special cases for the handling of sensitive or critical data:

- Passwords should never be transmitted in the clear. They should always be encrypted.
- Passwords should never be viewable on the user's screen as they are entered into the computer. Even though asterisks (*) are being displayed, care must be taken to ensure that it is not just because the font is all asterisks. If that is the case, someone could steal the password by copying and pasting the password from the screen.
- If possible, passwords should always be encrypted with one-way hashes. This will ensure that no one (not even a system administrator) can extract the password from the server. The only way to break the password would be through brute-force cracking. With one-way hashing, the actual passwords are not compared to authenticate the user; rather, the hashed value is stored on the server and is compared with the hashed value sent by the user. If the passwords cannot be decrypted, users cannot be provided their passwords when they forget them. In such cases, the system administrator must enter a new password for the user, which the user can change upon re-entering the application.
- Credit card and other financial information should never be sent in the clear.
- Cloud servers should minimize the transmissions and printing of credit card information. This includes all reports that may be used for internal use, such as troubleshooting, status, and progress reports.
- Sensitive data should not be passed to the cloud server as part of the query string, as the query string may be recorded in logs and accessed by persons not authorized to see the credit card information. For example, the following query string includes a credit card number:

```
http://www.server site.com/process_card.asp?cardnumber=1234567890123456
```

Code Practices

The minimum necessary information should be included in cloud server code. Attackers will spend countless hours examining HTML and scripts for information that can be used to make their intrusions easier to accomplish.

Comments should be stripped from operational code, and names and other personal information should be avoided. HTML comment fields should not reveal exploitable information about the developers or the organization. Comments are not bad per se, but those embedded in the HTML or client script and which may contain private information can be very dangerous in the hands of an attacker.

Third-party software packages, such as Web servers and FTP servers, often provide banners that indicate the version of the software that is running. Attackers can use this information to narrow their search of exploits to apply to these targets. In most cases, these banners can be suppressed or altered.

Language Options

One of the most frequently discovered vulnerabilities in cloud server applications is a direct result of the use of C and C++. The C language is unable to detect and prevent improper memory allocation, which can result in buffer overflows.

Because the C language cannot prevent buffer overflows, it is left to the programmer to implement safe programming techniques. Good coding practices will check for boundary limits and ensure that functions are properly called. This requires a great deal of discipline from the programmer; and in practice even the most experienced developers can overlook these checks occasionally.

One of the reasons Java is so popular is because of its intrinsic security mechanisms. Malicious language constructs should not be possible in Java. The Java Virtual Machine (JVM) is responsible for stopping buffer overflows, the use of uninitialized variables, and the use of invalid opcodes.

Input Validation and Content Injection

All user input that cannot be trusted must be verified and validated. Content injection occurs when the cloud server takes input from the user and applies the content of that input into commands or SQL statements. Essentially, the user's input is injected into a command that is executed by the server. Content injection can occur when the server does not have a clear distinction and separation between the data input and the commands executed.

Physical Security of the System

Any cloud server is vulnerable to an attacker with unlimited time and physical access to the server. Additionally, physical problems could cause the server to have down time. This would be a loss of availability, which you may recall is one of the key principles of the security triad — confidentiality, integrity, and availability (CIA). The following items should be provided to ensure server availability:

- Provide an uninterruptible power supply (UPS) unit with surge protection.
- Provide fire protection to minimize the loss of personnel and equipment.
- Provide adequate cooling and ventilation.
- Provide adequate lighting and workspace for maintaining and upgrading the system.

- Restrict physical access to the server. Unauthorized persons should not get near the server. Even casual contact can lead to outages. The server space should be locked and alarmed. Any access to the space should be recorded for later evaluation should a problem occur. Inventory should be tightly controlled and monitored.
- The physical protections listed here should extend to the network cables and other devices (such as routers) that are critical to the cloud server's operation.

Approaches to Cloud Software Requirements Engineering

Cloud system software requirements engineering demands extensive interaction with the user, and the product of the process includes both nonfunctional and functional software performance characteristics. Figure 3-1 illustrates the major elements of the software requirements engineering process.

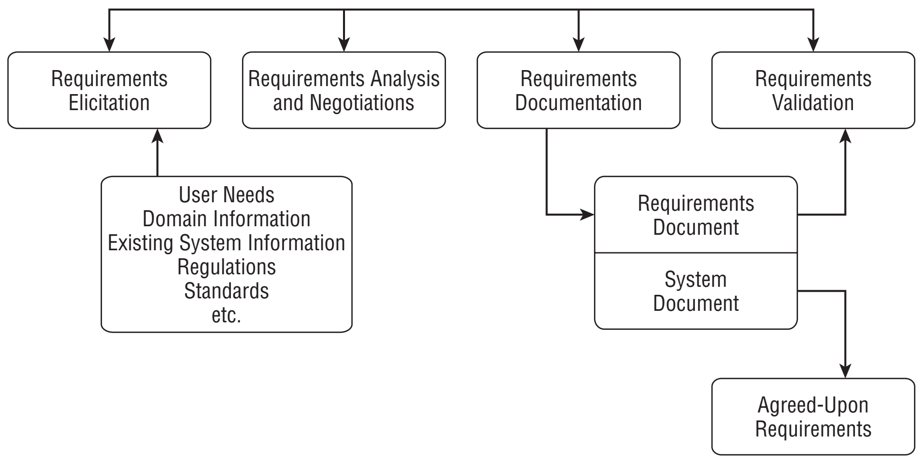

Figure 3-1: Software requirements engineering components

Source: Information Assurance Technology Analysis Center (IATC), Data and Analysis Center for Software (DACS), "State-of-the-Art Report," July 31, 2007.

Figure 3-2 illustrates additional elements that can be used to augment traditional software requirements engineering to increase cloud software security.

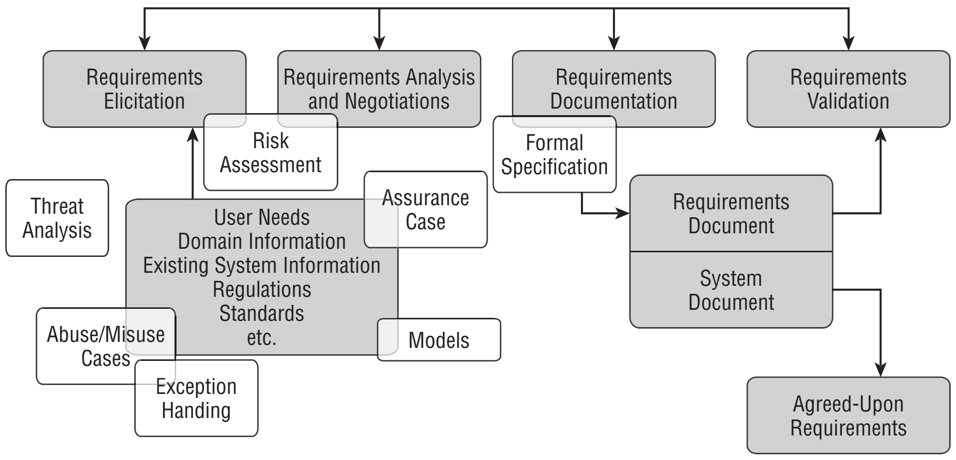

Figure 3-2: Additions to the software requirements engineering process to promote secure software

Source: Information Assurance Technology Analysis Center (IATC), Data and Analysis Center for Software (DACS), "State-of-the-Art Report," July 31, 2007.

A Resource Perspective on Cloud Software Security Requirements

Approaching software security requirements derivation from a resource perspective provides an effective method for addressing cloud software security requirements. In their April 1995 paper "SMART Requirements" (`www.win.tue.nl/~wstomv/edu/2ip30/references/smart-requirements.pdf`), Mike Mannion and Barry Keepence of Napier University, Edinburgh, U.K., take this approach by defining the following SMART basic properties that requirements should possess:

- **Specific** — The requirement should be unambiguous and direct. Mannion and Keepence define this characteristic as being clear, consistent, and simple.
- **Measurable** — The requirement should be measurable to ensure that it has been met.
- **Attainable** — The system must be able to exhibit the requirement under the specified conditions.
- **Realizable** — The requirement must be achievable under the system and project development constraints.
- **Traceable** — The requirement should be traceable both forward and backward throughout the development life cycle from conception through design, implementation, and test.

The Open Web Application Security Project (OWASP) has modified the SMART acronym (`www.owasp.org/index.php/Document_security-relevant_requirements`) to be SMART+ requirements. These requirements, taken from the OWASP website, are as follows:

- **Specific** — Requirements should be as detailed as necessary so there are no ambiguities.
- **Measurable** — It should be possible to determine whether the requirement has been met, through analysis, testing, or both.
- **Appropriate** — Requirements should be validated, thereby ensuring both that they derive from a real need or demand and that different requirements would not be more appropriate.
- **Reasonable** — While the mechanism or mechanisms for implementing a requirement need not be solidified, one should conduct some validation to determine whether meeting the requirement is physically possible, and possible given other likely project constraints.
- **Traceable** — Requirements should also be isolated to make them easy to track/validate throughout the development life cycle.

Goal-Oriented Software Security Requirements

Another complementary method for performing cloud software security requirements engineering is a *goal-oriented* paradigm in which a goal is a software objective. The types of goals that are targeted are functional goals, nonfunctional goals, security robustness, and code correctness. As Axel van Lamsweerde, Simon Brohez, Renaud De Landtsheer, and David Janssens write in "From System Goals to Intruder Anti-Goals: Attack Generation and Resolution for Security Requirements Engineering," "A goal is a prescriptive statement of intent about some system (existing or to-be) whose satisfaction in general requires the cooperation of some of the agents forming that system. Agents are active components such as humans, devices, legacy software or software-to-be components that play some role towards goal satisfaction. Goals may refer to services to be provided (functional goals) or to quality of service (nonfunctional goals)."[8]

One implementation of goal-oriented requirements engineering is the *nonfunctional requirements (NFR) framework,*[9] which provides a basis for determining if a goal has been satisfied through meeting lower-level goals.

Nonfunctional requirements include characteristics of a software system such as reliability, performance, security, accuracy, costs, reliability, and maintainability. According to Goertzel and Winograd et al., these requirements should specify the following:[10]

- Properties the software must exhibit (e.g., its behavior must be correct and predictable; it must remain resilient in the face of attacks)
- Required level of assurance or risk-avoidance of individual security functions and constraints
- Controls and rules governing the processes by which the software will be built, deployed, and operated (e.g., it must be designed to operate within a virtual machine, and its source code must not contain certain function calls)

Goertzel and Winograd et al. also provide an example of a negative nonfunctional requirement as follows: "The software must validate all input to ensure it does not exceed the size specified for that type of input."

A related goal-oriented requirements engineering approach is the MILOS[11] project methodology for goal-oriented security requirements engineering. The MILOS security model uses generic specification patterns that map to the information system's properties of confidentiality, integrity, availability, privacy, authentication, authorization, and nonrepudiation. The security patterns are transformed into goals that are used to develop a correlated "anti-model" that comprises a pattern of "anti-goals" an attacker would use to prevent meeting the specified system security goals.

NOTE **Cloud software security requirements address necessary attributes for software behavior and limitations on software functionality, whereas cloud software requirements are concerned with necessary software functionality and performance specifications.**

Monitoring Internal and External Requirements

The requirements of the information system security policy relative to software assurance should be analyzed to ensure their consistency and correctness. Two types of secure software requirements analysis should be performed:

- **Internal** — Necessary in order to ascertain that the requirements are complete, correct, and consistent with the related specification requirements. The analysis should address the following:
  - Security constraints
  - The software's nonfunctional properties
  - The software's positive functional requirements
- **External** — Necessary to determine the following:
  - The software assurance requirements address the legal regulatory and required policy issues.

- The nonfunctional security requirements represent a proper decomposition of the system security goals.
- Software assurance requirements don't conflict with system security goals.
- The software is resilient.

Also, in the context of internal and external access to information systems, the issues in Table 3-1 should be considered.

Table 3-1: Internal and External Security Requirements

INTERNAL	EXTERNAL
Maintain identity of active users	External connections must incorporate adequate controls to safeguard IT resources.
Implement internal access controls	At a minimum, all external connections must incorporate a firewall.
Use secure gateways to allow internal users to connect to external networks	If the user access originates from outside the protected network, user must be identified and authenticated at the gateway.
Hide internal Domain Name Systems (DNSs)	Use external authentication databases, such as RADIUS.
Dial-up modems should not be connected to computers that are connected to the internal network.	Employ content filtering to permit or deny services to specific external hosts.
E-mail messages flowing through the information systems should be monitored for internal policy compliance.	Accredit external connections prior to use.
	External connections should be periodically reviewed by an independent organization.

Source: National Institute of Standards and Technology, "An Introduction to Computer Security: The NIST Handbook, Special Publication 800-12," October 1995.

Cloud Security Policy Implementation and Decomposition

Cloud software security requirements are a function of policies such as system security policies, software policies, and information system policies. Cloud providers also have to satisfy regulations and directives such as FISMA, Gramm-Leach-Bliley, Sarbanes-Oxley, and HIPAA. For proper secure cloud software implementation, these issues have to be accounted for during the software development life cycle and through an effective cloud software security policy.

Implementation Issues

Important areas addressed by a software system's cloud security policy include the following:

- Access controls
- Data protection
- Confidentiality
- Integrity
- Identification and authentication
- Communication security
- Accountability

In the context of secure software, a requirement should follow from the general policy statements. An example of such a process is provided by Goertzel and Winograd et al.,[12] for the high-level policy functional requirement: "The server should store both public-access and restricted Web pages." From this high-level statement, the following activities should result as presented by Goertzel, Winograd, et al:

- Derive the detailed functional requirements, e.g., "The server should return public-access Web pages to any browser that requests those pages."
- Identify the related constraint requirements, e.g., "The server should return restricted Web pages only to browsers that are acting as proxies for users with authorized privileges sufficient to access those Web pages."
- Derive the functional security requirements, e.g., "The server must authenticate every browser that requests access to a restricted Web page."
- Identify the related negative requirements, e.g., "The server must not return a restricted Web page to any browser that it cannot authenticate."

The security requirements in a software security policy can also be specified in terms of functionality properties, such as restrictions on system states and information flows.

Goertzel and Winograd et al. list the following common sources of security requirements:

- Stakeholders' expressed security concerns
- Security implications of the functional specification
- Requirements for security functions
- Compliance and conformance mandates
- Secure development and deployment standards, guidelines, and best practices

- Attack models and environment risk analysis
- Known and likely vulnerabilities in the technologies and commercial-off-the-shelf (COTS) and open-source software (OSS) components that, due to preexisting commitments, must be used

An additional source of inputs to secure software policies is NIST FIPS Publication 200,[13] which specifies the following items:

- **System and Services Acquisition** — "Organizations must . . . (ii) employ system development life cycle processes that incorporate information security considerations; (iii) employ software usage and installation restrictions; and (iv) ensure that third-party providers employ adequate security measures to protect information, applications, and/or services outsourced from the organization."
- **System and Communications Protection** — "Organizations must . . . (ii) employ architectural designs, software development techniques, and systems engineering principles that promote effective information security within organizational information systems."
- **System and Information Integrity** — "Organizations must: (i) identify, report, and correct information and information system flaws in a timely manner; (ii) provide protection from malicious code at appropriate locations within organizational information systems."

Security policies are the foundation of a sound cloud system security implementation. Often organizations will implement technical security solutions without first creating this foundation of policies, standards, guidelines, and procedures, unintentionally creating unfocused and ineffective security controls.

According to the Data and Analysis Center for Software (DACS), "Information security policy is concerned, in large part, with defining the set of rules by which system subjects are allowed to change the states of data objects in the system. In practical terms, this means defining for every system subject whether, and if so how, it may store, transmit, create, modify, or delete a given data object (or type of data object)."[14]

The same document also lists three main objectives common to all system security policies and the mechanisms and countermeasures used to enforce those policies:

- They must allow authorized access and connections to the system while preventing unauthorized access or connections, especially by unknown or suspicious actors.
- They must enable allowable reading, modification, destruction, and deletion of data while preventing unauthorized reading (data leakage), modification (data tampering), destruction (denial of service), or deletion (denial of service).

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

Decomposing Critical Security Issues into Secure Cloud Software Requirements

An information system security policy addresses the critical issues of confidentiality, integrity, availability, identification, authentication, authorization, and auditing; and decomposes their elements into the following secure software requirements.

Confidentiality

Confidentiality in a cloud system policy is concerned with protecting data during transfers between entities. A policy defines the requirements for ensuring the confidentiality of data by preventing the unauthorized disclosure of information being sent between two end points. The policy should specify who can exchange information and what type of data can be exchanged. Related issues include intellectual property rights, access control, encryption, inference, anonymity, and covert channels. These policy statements should translate into requirements that address the following:

- Mechanisms that should be applied to enforce authorization
- What form of information is provided to the user and what the user can view
- The means of identity establishment
- What other types of confidentiality utilities should be used

Integrity

A cloud policy has to provide the requirements for ensuring the integrity of data both in transit and in storage. It should also specify means to recover from detectable errors, such as deletions, insertions, and modifications. The means to protect the integrity of information include access control policies and decisions regarding who can transmit and receive data and which information can be exchanged. Derived requirements for integrity should address the following:

- Validating the data origin
- Detecting the alteration of data
- Determining whether the data origin has changed

The policy should also provide for the integrity of data stored on media through monitoring for errors. Consideration should be given to determining the attributes and means that will be used as the basis for the monitoring and the actions that need to be taken should an integrity error occur. One type of integrity can also be described as maintaining a software system in a predefined “legitimate” state.

Availability

Cloud policy requirements for availability are concerned with denying illegitimate access to computing resources and preventing external attacks such as denial-of-service attacks. Additional issues to address include attempts by malicious entities to control, destroy, or damage computing resources and deny legitimate access to systems. While availability is being preserved, confidentiality and integrity have to be maintained. Requirements for this category should address how to ensure that computing resources are available to authorized users when needed.

Authentication and Identification

A cloud system policy should specify the means of authenticating a user when the user is requesting service on a cloud resource and presenting his or her identity. The authentication must be performed in a secure manner. Strong authentication using a public key certificate should be employed to bind a user to an identity. Exchanged information should not be alterable. This safeguard can be accomplished using a certificate-based digital signature. Some corresponding requirements include the following:

- Mechanisms for determining identity
- Binding of a resource to an identity
- Identification of communication origins
- Management of out-of-band authentication means
- Reaffirmations of identities

Authorization

After authentication, the cloud system policy must address authorization to allow access to resources, including the following areas:

- A user requesting that specified services not be applied to his or her message traffic
- Bases for negative or positive responses
- Specifying responses to requests for services in a simple and clear manner

- Including the type of service and the identity of the user in an authorization to access services
- Identification of entities that have the authority to set authorization rules between users and services
- Means for the provider of services to identify the user and associated traffic
- Means for the user to acquire information concerning the service profile kept by the service provider on the user

These policy issues should generate requirements that address the following:

- Specific mechanisms to provide for access control
- Privileges assigned to subjects during the system's life
- Management of access control subsystems

Auditing

The auditing of a cloud system has characteristics similar to auditing in the software development life cycle (SDLC) in that the auditing plan must address the following:

- Determination of the audit's scope
- Determination of the audit's objectives
- Validation of the audit plan
- Identification of necessary resources
- Conduct of the audit
- Documentation of the audit
- Validation of the audit results
- Report of final results

The Information Systems Audit and Control Association (ISACA) has developed information systems (IS) audit standards, guidelines, and a code of ethics for auditors that are directly applicable to cloud platforms. This information can be found on the ISACA website at `www.isaca.org`. The cloud system security policy should decompose the audit requirements to risk-based elements that consider the following three types of audit-related risks:

- **Inherent risk** — The susceptibility of a process to perform erroneously, assuming that no internal controls exist
- **Detection risk** — The probability that an auditor's methods will not detect a material error
- **Control risk** — The probability that extant controls will not prevent or detect errors in a timely fashion

The cloud system security policy decomposition for auditing should also consider organizational characteristics such as supervisory issues, institutional ethics, compensation policies, organizational history, and the business environment. In particular, the following elements of the cloud system organizational structure and management should be taken into account:

- Organizational roles and responsibilities
- Separation of duties
- IS management
- IS training
- Qualifications of IS staff
- Database administration
- Third party–provided services
- Managing of contracts
- Service-level agreements (SLAs)
- Quality management and assurance standards
- Change management
- Problem management
- Project management
- Performance management and indicators
- Capacity management
- Economic performance
    - Application of SOP 98-1,[15] which is an accounting statement of position that defines how information technology software development or acquisition costs are to be expended or capitalized
    - Expense management and monitoring
- Information system security management
- Business continuity management

The cloud policy decomposition for the audit component is recursive in that the audit has to address the cloud system security policy, standards, guidelines, and procedures. It should also delineate the three basic types of controls, which are preventive, detective, and corrective; and it should provide the basis for a qualitative audit risk assessment that includes the following:

- Identification of all relevant assets
- Valuation of the assets

- Identification of threats
- Identification of regulatory requirements
- Identification of organizational risk requirements
- Identification of the likelihood of threat occurrence
- Definition of organizational entities or subgroupings
- Review of previous audits
- Determination of audit budget constraints

The cloud policy should ensure that auditing can pass a test of *due care*, which is defined by the ISACA as "the level of diligence that a prudent and competent person would exercise under a given set of circumstances."[16]

In the event that it is necessary to conduct forensic investigations in cloud systems, the confidentiality and integrity of audit information must be protected at the highest level of security.

In 1996, the ISACA introduced a valuable audit planning and execution tool, the "Control Objectives for Information and Related Technology (COBIT)" document. As of this writing, COBIT is now in version 4.1. It is divided into four domains comprising 34 high-level control objectives. These 34 control objectives are further divided into 318 specific control objectives. COBIT defines a *control objective* that is a goal aimed at preventing a set of risks from occurring . The four COBIT domains are as follows:

- **Planning and organization (PO) —** Provides direction to solution delivery (AI) and service delivery (DS)
- **Acquisition and implementation (AI) —** Provides the solutions and passes them to be turned into services
- **Deliver and support (DS) —** Receives the solutions and makes them available for end users
- **Monitor and evaluate (ME) —** Monitors all processes to ensure that the direction provided is followed

NIST 33 Security Principles

In June 2001, the National Institute of Standards and Technology's Information Technology Laboratory (ITL) published NIST Special Publication 800-27, "Engineering Principles for Information Technology Security (EP-ITS)," to assist in the secure design, development, deployment, and life cycle of information systems. The document was revised (Revision A) in 2004. It presents 33 security principles that begin at the design phase of the information system or application and continue until the system's retirement and secure disposal.

Some of the 33 principles that are most applicable to cloud security policies and management are as follows:

- Principle 1 — Establish a sound security policy as the "foundation" for design.
- Principle 2 — Treat security as an integral part of the overall system design.
- Principle 3 — Clearly delineate the physical and logical security boundaries governed by associated security policies.
- Principle 6 — Assume that external systems are insecure.
- Principle 7 — Identify potential trade-offs between reducing risk and increased costs and decreases in other aspects of operational effectiveness.
- Principle 16 — Implement layered security; ensure there is no single point of vulnerability.
- Principle 20 — Isolate public access systems from mission-critical resources (e.g., data, processes, etc.).
- Principle 21 — Use boundary mechanisms to separate computing systems and network infrastructures.
- Principle 25 — Minimize the system elements to be trusted.
- Principle 26 — Implement least privilege.
- Principle 32 — Authenticate users and processes to ensure appropriate access control decisions both within and across domains.
- Principle 33 — Use unique identities to ensure accountability.

Secure Cloud Software Testing

Secure cloud software testing involves a number of activities. Each activity is based on a formal standard or methodology and adds unique value to the overall secure software testing process. An organization typically selects testing activities based on a number of factors, including secure cloud software requirements and available resources.

Analyses of test results form the basis for assessing risk to cloud information and means of remediation. Standards and methodologies such as the International Organization for Standardization (ISO) 9126 Standard for Software Engineering/Product Quality, the Systems Security Engineering Capability Maturity Model (SSE-CMM) and the Open Source Security Testing Methodology Manual (OSSTMM) provide additional guidance for secure software evaluation and mitigation. After software has been modified, regression testing provides assurance that the original software system functionality and security characteristics are not negatively affected by the respective changes.

Testing for Security Quality Assurance

Secure software testing has considerations in common with quality assurance testing. For example, the correct version of the software should always be tested. However, secure software testing must also measure the quality of the software's security properties. For example, software should be tested to ensure that it meets its functional specifications, and does nothing else. Testing that software does nothing else — that is, does not contain any unintended functionality — is a measure of security quality.

There is a lack of commonly agreed-upon definitions for software quality, but it is possible to refer to software quality by its common attributes. One well-known characterization of software quality is the International Organization for Standardization (ISO) 9126 standard. The ISO 9126 standard characterizes software quality with six main attributes and 21 subcharacteristics, as shown in Table 3-2.

Table 3-2: The ISO 9126 Software Quality Standards

ATTRIBUTES	SUBCHARACTERISTICS	DEFINITION
Functionality	Suitability	Attributes of software that bear on the presence and appropriateness of a set of functions for specified tasks
	Accurateness	Attributes of software that bear on the provision of right or agreed upon results or effects
	Interoperability	Attributes of software that bear on its ability to interact with specified systems
	Compliance	Attributes of software that make the software adhere to application-related standards or conventions or regulations in laws and similar prescriptions
	Security	Attributes of software that bear on its ability to prevent unauthorized access, whether accidental or deliberate, to programs or data
Reliability	Maturity	Attributes of software that bear on the frequency of failure by faults in the software
	Fault tolerance	Attributes of software that bear on its ability to maintain a specified level of performance in case of software faults or infringement of its specified interface

Continued

Table 3-2 *(continued)*

ATTRIBUTES	SUBCHARACTERISTICS	DEFINITION
	Recoverability	Attributes of software that bear on the capability to re-establish its level of performance and recover the data directly affected in case of a failure and on the time and effort needed for it
Usability	Understandability	Attributes of software that bear on the users' effort for recognizing the logical concept and its applicability
	Learnability	Attributes of software that bear on the users' effort for learning its application
	Operability	Attributes of software that bear on the users' effort for operation and operation control
Efficiency	Time behavior	Attributes of software that bear on response and processing times and on throughput rates in performing its function
	Resource behavior	Attributes of software that bear on the amount of resources used and the duration of such use in performing its function
Maintainability	Analyzability	Attributes of software that bear on the effort needed for diagnosis of deficiencies or causes of failures or for identification of parts to be modified
	Changeability	Attributes of software that bear on the effort needed for modification, fault removal, or environmental change
	Stability	Attributes of software that bear on the risk of unexpected effect of modifications
	Testability	Attributes of software that bear on the effort needed for validating the modified software

ATTRIBUTES	SUBCHARACTERISTICS	DEFINITION
Portability	Adaptability	Attributes of software that bear on the opportunity for its adaptation to different specified environments without applying other actions or means than those provided for this purpose for the software considered
	Installability	Attributes of software that bear on the effort needed to install the software in a specified environment
	Conformance	Attributes of software that make the software adhere to standards or conventions relating to portability
	Replaceability	Attributes of software that bear on opportunity and effort using it in the place of specified other software in the environment of that software

Conformance Testing

The National Institute of Standards and Technology (NIST) states that "conformance testing activities assess whether a software product meets the requirements of a particular specification or standard."[17] These standards are typically well regarded and widely accepted, such as those from the International Organization for Standardization (ISO), the Institute of Electrical and Electronics Engineers, Inc. (IEEE), or the American National Standards Institute (ANSI). They reflect a commonly accepted "reference system" whose standards recommendations are sufficiently defined and tested by certifiable test methods. They are used to evaluate whether the software product implements each of the specific requirements of the standard or specification.

Conformance testing methodologies applicable to cloud services have been developed for operating system interfaces, computer graphics, documented interchange formats, computer networks, and programming language processors. Most testing methodologies use test case scenarios (e.g., abstract test suites, test assertions, test cases), which themselves must be tested.

Standardization is an important component of conformance testing. It usually includes developing the functional description and language specification, creating the testing methodology, and "testing" the test case scenarios.

A major benefit of conformance testing is that it facilitates interoperability between various cloud software products by confirming that each software product meets an agreed-upon standard or specification.

One type of conformance testing, protocol-based testing, uses an application's communication protocol as a direct basis for testing the application. This method is useful for cloud-based applications. Protocol-based testing is especially important for security testing in Web-based applications, because Web protocols provide the easiest way for remote attackers to access such applications.[18]

Functional Testing

In functional testing, a cloud software application is tested at runtime to determine whether it conforms to its functional requirements. Requirements that state how the application should respond when a specific event occurs are referred to as *positive requirements*. Typically, a positive requirement is mapped to a specific software artifact meant to implement that requirement. This provides traceability from requirements to implementation and informs the tester of which code artifact to test to validate the expected functionality.

An example of a positive requirement is "the application should lock the user account after three failed login attempts." A tester can validate the expected functionality (the lockout) by attempting to log in to the application three times with the same username and incorrect passwords. This type of test can be easily automated with a functional testing tool suite, such as the open-source Canoo WebTest (`http://webtest.canoo.com`).

Functional testing also includes *negative requirements*, which specify what software should *not* do. An example of a negative requirement is "the cloud application should not allow for the stored data to be disclosed." This type of requirement is more difficult to test because the expected behavior is not implemented in a specific software artifact. Testing this requirement properly would require the tester to anticipate every anomalous input, condition, cause, and effect. Instead, the testing should be driven by risk analysis and threat modeling. This enables the negative requirement to be documented as a threat scenario, and the functionality of the countermeasure as a factor to mitigate the threat. The following steps summarize this approach from the Open Web Application Security Project (OWASP) Testing Guide (`www.owasp.org/index.php/Category:OWASP_Testing_Project`).

First, the security requirements are documented from a threats and countermeasures perspective:

- Encrypt authentication data in storage and transit to mitigate risk of information disclosure and authentication protocol attacks.

- Encrypt passwords using nonreversible encryption such as a hashing algorithm and a salt to prevent dictionary attacks. *Salt* refers to inserting random bits into algorithms used for key generation to mitigate against dictionary attacks.
- Lock out accounts after reaching a login failure threshold and enforce password complexity to mitigate risk of brute-force password attacks.
- Display generic error messages upon validation of credentials to mitigate risk of account harvesting/enumeration.
- Mutually authenticate client and server to prevent nonrepudiation and man-in-the-middle (MITM) attacks.

Artifacts produced in the threat modeling process, such as threat trees and attack libraries, can then be used to derive negative test scenarios.

A threat tree will assume a root attack (e.g., attack might be able to read other users' messages) and identify different exploits of security controls (e.g., data validation fails because of a SQL injection vulnerability) and necessary countermeasures (e.g., implement data validation and parameterized queries) that could be tested for effectiveness in mitigating such attacks.

Typically, functional testing is used to test the functionality of implemented features or after the software feature is complete. However, code coverage is limited by the number of available use cases. If a test is not created for a specific use case, then a number of execution paths in the software will remain untested. Therefore, even if the functionality is validated for all available use cases, that is not a guarantee that the software is free of defects.

Logic testing is a type of functional testing that involves ensuring that business logic is predicated on the correct assumptions. Business logic is the code that satisfies the business purpose of cloud software and typically models and automates a "real-life" or "paper" business process such as e-commerce or inventory management. Business logic is composed of both business rules and workflows:

- Business rules that express business policy (such as channels, location, logistics, prices, and products)
- Workflows based on the ordered tasks of passing documents or data from one participant (a person or a software system) to another

Business logic flaws are typically specific to the cloud application being tested, and are difficult to detect. Automated tools do a poor job of discovering logic flaws because they do not understand the context of the decisions. Therefore, discovering logic flaws is typically a manual process performed by a human tester.

When looking for business logic flaws, the tester begins by considering the rules for the business function being provided by the cloud application. Next, the tester searches for any limits or restrictions on people's behavior. Then the application can be tested to validate that it enforces those rules. A classic example of a business logic flaw is the modification of prices that was sometimes allowed by e-commerce applications on the early Web-based Internet.

Setting the price of a product on an e-commerce site as a negative number could result in funds being credited to an attacker. A countermeasure to this vulnerability is to implement positive validation of the price so that the application allows only positive numbers in a specific numerical range. Of course, the application should never accept and process any data from the client that did not require user input in the first place.

Performance Testing

In an online report (`http://vote.nist.gov/vvsg-report.htm`), NIST states that "what distinguishes performance testing from functional testing is the form of the experimental result. A functional test yields a yes or no verdict, while a performance test yields a quantity." Performance testing measures how well the cloud software system executes according to its required response times, throughput, CPU, usage, and other quantifiable features in operation. The quantity resulting from a test may subsequently be reduced to a yes or no verdict by comparison with a benchmark.

Performance testing is also commonly known by other names and/or associated with other testing activities, such as stress testing, capacity testing, load testing, volume testing, and benchmark testing. These various performance testing activities all have approximately the same goal: "measuring the cloud software product under a real or simulated load."[19]

Typically, performance testing is conducted late in the software life cycle when the software is fully developed. In order to obtain accurate measurements, the cloud software is deployed and tested in an environment that simulates the operational environment. This can be achieved by creating a cloud "staging" environment, essentially a mirror copy of the production infrastructure, and simulating typical operating conditions.

A major benefit of performance testing is that it is typically designed specifically for pushing system limits over a long period of time. This form of testing has commonly been used to uncover unique failures not discovered during conformance or interoperability tests. In addition, benchmarking is typically used to provide competitive baseline performance comparisons. For instance, these tests are used to characterize performance prior to manufacturing as well as to compare performance characteristics of other software products prior to purchase.

Performance testing procedures provide steps for determining the ability of the cloud software to function properly, particularly when near or beyond the boundaries of its specified capabilities or requirements. These boundaries are usually stated in terms of the volume of information used. The specified metrics are usually stated in terms of time to complete an operation. Ideally, performance testing is conducted by running a software element against standard datasets or scenarios, known as *reference data*.

Performance measures and requirements are quantitative, which means they consist of numbers that can be measured and confirmed by rational experiments. A performance specification consists of a set of specified numbers that can be reduced to measured numbers, often in the form of a probability distribution. The numbers measured for the software product are either less than, more than, or equal to the specified values. If less than, the software product fails; if more than or equal to, the software product passes the tests. Every performance specification is a variation of these simple ideas. Common metrics used in performance testing include the following:

- **Throughput** — The rate at which the system processes transactions, commonly measured in bytes per second
- **Processing delay** — The time it takes to process those transactions, measured in seconds
- **Load** — The rate at which transactions are submitted to a software product, measured in arriving transactions per second

Stress testing is a kind of performance testing that involves increasing the load on a software system beyond normal operating capacity and observing the results. Stress testing can be used to ensure that the cloud software remains stable and reliable, and can continue to provide a specific quality of service, although the software is often tested to the point of failure. Extreme operating conditions, such as those associated with resource exhaustion — out of memory or hardware failures that might occur in a cloud environment — are simulated.

Stress testing can also be used to test the security properties of cloud software because it can induce anomalous behavior. For example, extreme operating conditions may cause an error that is poorly handled by the cloud application, causing it to fail insecurely. In a real-world scenario, a DoS attack targeted against a cloud application could slow down the execution of the application such that it exposes a race condition, which could subsequently be exploited as a security vulnerability.[20]

NOTE The Microsoft Web Application Stress Tool (`www.microsoft.com`) is a freely available tool that simulates multiple browsers requesting pages from a website. It can be used to gather performance and stability information about a Web application. It simulates a large number of requests with a relatively small number of client machines. The goal is to create an environment that is as close to production as possible so that problems can be discovered and eliminated in a Web application prior to deployment.

Security Testing

Security testing should assess the security properties and behaviors of cloud software as it interacts with external entities (human users, environment, other software) and as its own components interact with each other. Security testing should verify that software exhibits the following properties and behaviors, as summarized from the "Software Security Assurance State-of-the-Art Report (SOAR)":[21]

- Its behavior is predictable and secure.
- It exposes no vulnerabilities or weaknesses.
- Its error and exception handling routines enable it to maintain a secure state when confronted by attack patterns or intentional faults.
- It satisfies all of its specified and implicit nonfunctional security requirements.
- It does not violate any specified security constraints.
- As much of its runtime-interpretable source code and byte code as possible has been obscured or obfuscated to deter reverse engineering.

A security test plan should be included in the overall cloud software test plan and should define all testing activities, including the following:

- Security test cases or scenarios
- Test data, including attack patterns
- Test oracle (if one is to be used), which is used to determine if a software test has passed or failed
- Test tools (white box, black box, static, and dynamic)
- Analyses to be performed to interpret, correlate, and synthesize the results from the various tests and outputs from the various tools

Software security testing techniques can be categorized as white box, gray box, or black box:

- **White box** — Testing from an internal perspective, i.e., with full knowledge of the software internals; the source code, architecture and design documents, and configuration files are available for analysis.
- **Gray box** — Analyzing the source code for the purpose of designing the test cases, but using black box testing techniques; both the source code and the executable binary are available for analysis.
- **Black box —** Testing the software from an external perspective, i.e., with no prior knowledge of the software; only the binary executable or intermediate byte code is available for analysis.

An example of a white box testing technique is the static analysis of source code that should be performed iteratively as the software is being written. Table 3-3 lists other common security testing techniques and how they are typically categorized.

Table 3-3: Common Security Testing Techniques

TESTING TECHNIQUE	CATEGORY
Source code analysis	White box
Property-based	White box
Source code fault injection	White box, gray box
Dynamic code analysis	Gray box
Binary fault injection	Gray box, black box
Fuzz testing	Black box
Binary code analysis	Black box
Byte code analysis	Black box
Black box debugging	Black box
Vulnerability scanning	Black box
Penetration testing	Black box

Fault Injection

Fault injection is a technique used to improve code coverage by testing all code paths, especially error-handling code paths that may not be exercised during functional testing. In fault injection testing, errors are injected into the cloud software to simulate unintentional user errors and intentional attacks on the software through its environment, and attacks on the environment itself.

Source Code Fault Injection

In source code fault injection, the tester decides when environment faults should be triggered. The tester then "instruments" the source code by non-intrusively inserting changes into the program that reflect the changed environment data that would result from those faults.[22] The instrumented source code is then compiled and executed, and the tester observes how the executing software's state changes when the instrumented portions of code are executed. This enables the tester to observe the secure and nonsecure state changes in the software resulting from changes in its environment.

The tester can also analyze how the cloud software's state changes as a result of a fault propagating through the source code. This type of analysis is typically referred to as *fault propagation analysis*, and involves two techniques of source code fault injection: *extended propagation analysis* and *interface propagation analysis*.

To prepare for fault propagation analysis, the tester must generate a fault tree from the software's source code. To perform an extended propagation analysis, the tester injects faults into the fault tree, then traces how each injected fault propagates through the tree. This shows the impact a particular fault will have on the overall behavior of the software.

Interface propagation analysis focuses on how faults are propagated through the interfaces between the component/module and other cloud application-level and environment-level components. In interface propagation analysis, the tester injects a fault into the data inputs between components, views how the resulting faults propagate, and observes whether any new anomalies result. This type of analysis enables the tester to determine how the failure of one component can affect the failure of a neighboring component, particularly important if a neighboring cloud component is of high consequence.

Source code fault injection is particularly useful in detecting the following:

- Incorrect use of pointers and arrays
- Use of dangerous calls
- Race conditions

Source code fault injection should be performed iteratively as the software is being written. When new threats (attack types and intrusion techniques) are discovered, the source code can be re-instrumented with faults that are representative of those new threat types.

Binary Fault Injection

Binary fault injection is a runtime analysis technique whereby an executing cloud application is monitored as faults are injected. By monitoring system call traces, a tester can identify the names of system calls, the parameters to each call, and the call's return code. This enables the tester to discover the names and types of resources being accessed by the calling software, how the resources are being used, and the success or failure of each access attempt. In binary fault analysis, faults are injected into the environment resources that surround the cloud application. Environmental faults provide the tester with a number of benefits:

- They simulate real-world attack scenarios and can be easily automated.
- They simulate environment anomalies without requiring an understanding of how those anomalies actually occur in the real world. This enables fault injection by testers who do not have prior knowledge of the environment whose faults are being simulated.
- The tester can choose when to trigger a particular environmental fault. This avoids the problem of a full environment emulation in which the environment state when the application interacts with it may not be what is expected, or may not have the expected effect on the software's behavior.

It is difficult to predict the complex inputs the cloud software will actually receive in its target environment. Therefore, fault injection scenarios should be designed to give the tester the most complete understanding possible of the security of the behaviors, states, and properties of the software system under all possible operating conditions. Once the cloud application has been deployed to production, the tester can employ penetration testing and vulnerability scanning to provide an additional measure of the application's security posture.

Binary fault injection tools include binary fault injectors and brute-force testers. These tools should support the common functionality found in the application. For example, the commercial fault injection tool Holodeck (`www.securityinnovation.com/holodeck/`) is often used to simulate faults in Microsoft operating system applications. Holodeck injects faults for common functionality found in a typical Windows environment such as the following:

- **Network** — Cable disconnected, network not installed, wrong Winsock version, Winsock task limit reached, no ports available, network is down
- **Disk** — Insufficient disk space, cyclic redundancy check (CRC) errors, too many files open, disk is write-protected, no disk is in the drive
- **Memory** — Insufficient memory, failure to allocate, locked memory

Holodeck also supports monitoring an application to watch its interactions with the environment.

Dynamic Code Analysis

Dynamic code analysis examines the code as it executes in a running cloud application, with the tester tracing the external interfaces in the source code to the corresponding interactions in the executing code, so that any vulnerabilities or anomalies that arise in the executing interfaces are simultaneously located in the source code, where they can then be fixed.

Unlike static analysis, dynamic analysis enables the tester to exercise the software in ways that expose vulnerabilities introduced by interactions with users and changes in the configuration or behavior of environment components. Because the software isn't fully linked and deployed in its actual target environment, these interactions and their associated inputs and environment conditions are essentially simulated by the testing tool.

An example of a dynamic code analysis toolset is the open-source Valgrind (`www.valgrind.org`), which is useful in detecting memory management and thread bugs.

Property-Based Testing

Property-based testing is a formal analysis technique developed at the University of California at Davis.[23] Property-based testing validates that the software's implemented functionality satisfies its specifications. It does this by examining security-relevant properties revealed by the source code, such as the absence of insecure state changes. Then these security-relevant properties in the code are compared against the software's specification to determine whether the security assumptions have been met.

Like direct code analysis, property-based testing requires the full concentration of the tester and is a detail-oriented process. Because it requires the tester to dedicate a significant amount of time to the code, it is often used only to analyze code that implements high-consequence functions.

Black Box Debugging

Black box debugging[24] is a technique to monitor behaviors external to the binary component or system while it is executing, and thereby observe the data that passes between that component/system and external entities.

Additionally, by observing how data passes across the software's boundary, the analyst can also determine how externally sourced data might be manipulated to force the software down certain execution paths, or to cause the software to fail. This can reveal errors and failures that originate not in the cloud software itself, but are forced by the external entities with which it interacts, or by an incorrectly implemented API.

Interoperability Testing

Interoperability testing evaluates whether a cloud application can exchange data (interoperate) with other components or applications. Interoperability testing activities determine the capability of applications to exchange data via a common set of exchange formats, to read and write the same file formats, and to communicate using the same protocols. A major goal of interoperability testing is to detect interoperability problems between cloud software applications before these applications are put into operation. Interoperability testing requires the majority of the application to be completed before testing can occur.

Interoperability testing typically takes one of three approaches:

- **Testing all pairs** — This is often conducted by a third-party independent group of testers who are knowledgeable about the interoperability characteristics across software products and between software vendors.
- **Testing some of the combinations** — This approach involves testing only part of the combinations and assuming the untested combinations will also interoperate.
- **Testing against a reference implementation** — This approach establishes a reference implementation, e.g., using the accepted standard, and testing all products against the reference. In a paper on metrology in information technology, researchers in the NIST Information Technology Laboratory state that a typical procedure used to conduct interoperability

testing includes "developing a representative set of test transactions in one software product for passage to another software product for processing verification."[25]

One challenge in cloud software component integration is how to build a secure composite system from components that may or may not be individually secure. In a paper by Verizon Communications and the University of Texas,[26] researchers describe a systematic approach for determining interoperability of components from a security perspective and unifying the security features, policies, and implementation mechanisms of components. This is a goal-oriented and model-driven approach to analyzing the security features of components to determine interoperability. Along with this approach, the researchers provide a guideline for integrating the components to fulfill the security goals of the composite system. Following the proposed analysis procedure could lead to discovery of some classes of security interoperability conflicts that help to determine whether or not the components should be used together.

Cloud Penetration Testing

A penetration test is a security testing methodology that gives the tester insight into the strength of the target's network security by simulating an attack from a malicious source. The process involves an active analysis of the cloud system for any potential vulnerabilities that may result from poor or improper system configuration, known and/or unknown hardware or software flaws, or operational weaknesses in process or technical countermeasures. This analysis is carried out from the position of a potential attacker, and can involve active exploitation of security vulnerabilities.

Any security issues that are found are presented to the system owner together with an assessment of their impact, and often with a proposal for mitigation or a technical solution. The intent of a penetration test is to proactively determine the feasibility of an attack or to retroactively determine the degree to which a successful exploit has affected the business. It is a component of a full security audit, which includes the following:

- **A Level I, high-level assessment** — A top-down look at the organization's policies, procedures, standards, and guidelines. A Level I assessment is not usually hands-on, in that the system's security is not actually tested.
- **A Level II, network evaluation** — More hands-on than a Level I assessment, a Level II assessment has some of the Level I activities plus more information gathering and scanning.
- **A Level III, penetration test** — A penetration test is not usually concerned with policies. It is more about taking the adversarial view of a hacker, by seeing what can be accomplished, and with what difficulty.

Several factors have converged in the cloud environment to make penetration testing a necessity. The evolution of information technology has focused on ease of use at the operational end, while exponentially increasing the complexity of the computing resources. Unfortunately, the administration and management requirements of cloud systems have increased for several reasons:

- The skill level required to execute a hacker exploit has steadily decreased.
- The size and complexity of the network environment has mushroomed.
- The number of network and cloud-based applications has increased.
- The detrimental impact of a security breach on corporate assets and goodwill is greater than ever.

Penetration testing is most commonly carried out within a "black box," that is, with no prior knowledge of the infrastructure to be tested. At its simplest level, the penetration test involves three phases:

1. **Preparation** — A formal contract is executed containing nondisclosure of the client's data and legal protection for the tester. At a minimum, it also lists the IP addresses to be tested and the time to test.
2. **Execution** — In this phase the penetration test is executed, with the tester looking for potential vulnerabilities.
3. **Delivery** — The results of the evaluation are communicated to the tester's contact in the organization, and corrective action is advised.

Whether the penetration test is a full knowledge (white box) test, a partial knowledge (gray box) test, or a zero knowledge (black box) test, after the report and results are obtained, mitigation techniques have to be applied to reduce the risk of compromise to an acceptable or tolerable level. The test should address vulnerabilities and corresponding risks to such areas as applications, remote access systems, Voice over Internet Protocol (VoIP), wireless networks, and so on.

Legal and Ethical Implications

Because an ethical hacker conducting a penetration test works for an organization to assist in evaluating its network security, this individual must adhere to a high standard of conduct. In fact, there is a Certified Ethical Hacker (CEH) certification sponsored by the International Council of E-Commerce Consultants (EC-Council) at `www.eccouncil.org` that attests to the ethical hacker's knowledge and subscription to ethical principles. The EC-Council also provides the Licensed Penetration Tester (LPT) certification, which, to quote their website, provides the following benefits:

- Standardizes the knowledge base for penetration testing professionals by incorporating best practices followed by experienced experts in the field

- Ensures that each professional licensed by EC-Council follows a strict code of ethics
- Is exposed to the best practices in the domain of penetration testing
- Is aware of all the compliance requirements required by the industry
- Trains security professionals to analyze the security posture of a network exhaustively and recommend corrective measures

When an ethical hacker or licensed penetration tester agrees to conduct penetration tests for an organization, and to probe the weaknesses of their information systems, he or she can be open to dismissal and prosecution unless contract terms are included to protect the individuals conducting the test. It is vitally important that the organization and ethical hacking team have an identical understanding of what the team is authorized to do and what happens if the team inadvertently causes some damage.

Attacking a network from the outside carries ethical and legal risk to the tester, and remedies and protections must be spelled out in detail before the test begins. For example, the Cyber Security Enhancement Act of 2002 indicates life sentences for hackers who "recklessly" endanger the lives of others, and several other U.S. statutes address cyber crime. Statute 1030, "Fraud and Related Activity in Connection with Computers," specifically states that whoever intentionally accesses a protected computer without authorization, and as a result of such conduct, recklessly causes damage or impairs medical treatment, can receive a fine or imprisonment of five to twenty years. It is vital that the tester receive specific written permission to conduct the test from the most senior executive possible. A tester should be specifically indemnified against prosecution for the task of testing.

For his or her protection, the ethical hacking tester should keep the following items in mind:

- **Protect information uncovered during the penetration test** — In the course of gaining access to an organization's networks and computing resources, the ethical hacker will find that he or she has access to sensitive information that would be valuable to the organization's competitors or enemies. Therefore, this information should be protected to the highest degree possible and not divulged to anyone, either purposely or inadvertently.
- **Conduct business in an ethical manner** — Ethical is a relative term and is a function of a number of variables, including socio-economic background, religion, upbringing, and so on. However, the ethical hacker should conduct his or her activities in a trustworthy manner that reflects the best interests of the organization that commissioned the penetration testing. Similarly, the organization should treat the ethical hacker with the same respect and ethical conduct.

- **Limitation of liability** — As discussed earlier in this section, during a penetration test, the ethical hacking team will most likely have access to sensitive files and information. The ethical hacker is trained to not cause any harm, such as modifying files, deleting information, and so on, in the course of his or her activities; but because errors do occur, the organization and the ethical hacker should have terms in the contract between them that address the situation where harm is done inadvertently. There should be a limitation to the liability of the ethical hacker if this scenario occurs. Another commonly used option by consultants is to obtain an insurance policy that will cover the consultant's activities in his or her chosen profession.
- **Remain within the scope of the assignment** — The scope of the penetration testing should be delineated beforehand and agreed upon by all parties involved. With that accomplished, the testing team should conduct the testing strictly within those bounds. For example, the penetration testing should include only the networks and computing resources specified, as well as the methods and extent of trying to break in to the information system.
- **Develop a testing plan** — As with any endeavor, the ethical hacking team should develop a test plan in advance of the testing and have it approved by the hiring organization. The plan should include the scope of the test, resources to be tested, support provided by the hiring organization, times for the testing, location of the testing, the type of testing (white box, gray box, or black box), extent of the penetration, individuals to contact in the event of problems, and deliverables.
- **Comply with relevant laws and regulations** — Business organizations are required to comply with a variety of laws and regulations, including the Health Insurance Portability and Accountability Act (HIPAA), Sarbanes-Oxley, and the Gramm-Leach-Bliley Act (GLBA). These acts are one of the reasons why companies hire ethical hackers, and demonstrate that they are acting to protect their information resources.

The Open-Source Security Testing Methodology Manual, OSS OSSTMM 2.2, (`http://isecom.securenetltd.com/osstmm.en.2.2.pdf`), also provides rules of engagement for ethical practices in a number of areas, including penetration testing. The following list summarizes some of the pertinent rules from the OSSTMM 2.2:

- Testing of very insecure systems and installations is not to be performed until appropriate remediation measures have been taken.
- The auditor is required to ensure nondisclosure of client proprietary information.

- Contracts should limit the liability of the auditor.
- The engagement contract should provide permissions for the specific types of tests to be performed.
- The scope of the testing effort should be clearly defined.
- The auditor must operate legally.
- In reporting test results, the auditor must respect the privacy of all concerned.

The Three Pre-Test Phases

Penetration testing is usually initiated with *reconnaissance*, which comprises three pre-test phases: footprinting, scanning, and enumerating. These pre-test phases are very important and can make the difference between a successful penetration test that provides a complete picture of the target's network and an unsuccessful test that does not.

The reconnaissance process seeks to gather as much information about the target network as possible, using the following seven steps during the footprinting, scanning, and enumerating activities:

1. Gather initial information.
2. Determine the network range.
3. Identify active machines.
4. Discover open ports and access points (APs).
5. Fingerprint the operating system.
6. Uncover services on ports.
7. Map the network.

Footprinting

Footprinting is obtaining information concerning the security profile of an organization. It involves gathering data to create a blueprint of the organization's networks and systems. It is an important way for an attacker to gain information about an organization without the organization's knowledge.

Footprinting employs the first two steps of reconnaissance, gathering the initial target information and determining the network range of the target. It may also require manual research, such as studying the company's Web page for useful information such as the following:

- Company contact names, phone numbers, and e-mail addresses
- Company locations and branches

- Other companies with which the target company partners or deals
- News, such as mergers or acquisitions
- Links to other company-related sites
- Company privacy policies, which may help identify the types of security mechanisms in place

Other resources that may have information about the target company include the following:

- The U.S. Securities and Exchange Commission (SEC) EDGAR database, if the company is publicly traded
- Job boards, either internal to the company or external sites
- Disgruntled employee blogs
- The target organization's website and other related websites
- Business social networking websites such as LinkedIn
- Personal/business websites such as Facebook
- Trade press

Scanning

The next four steps of gathering information — identifying active machines, discovering open ports and access points, fingerprinting the operating system, and uncovering services on ports — are considered part of the scanning phase. The goal in this step is to discover open ports and applications by performing external or internal network scanning, pinging machines, determining network ranges, and scanning the ports of individual systems. (Scanning is discussed in more detail later in this chapter.)

Enumerating

The last step in reconnaissance, mapping the network, is the result of the scanning phase and leads to the enumerating phase. As the final pre-test phase, the goal of enumeration is to paint a fairly complete picture of the target.

To enumerate a target, a tester tries to identify valid user accounts or poorly protected resource elements by using directed queries and active connections to and from the target. The type of information sought by testers during the enumeration phase can be names of users and groups, network resources and shares, and applications.

The techniques used for enumerating include the following:

- Obtaining Active Directory information and identifying vulnerable user accounts
- Discovering the NetBIOS name with NBTscan

- Using the SNMPutil command-line utility for Simple Network Management Protocol (SNMP)
- Employing Windows DNS queries
- Establishing null sessions and connections

Penetration Testing Tools and Techniques

A variety of tools and techniques, including some used by malicious hackers, can be valuable in conducting penetration tests on cloud systems. Some tools, such as Whois and Nslookup, are public software that can help gather information about the target network. Whois is usually the first stop in reconnaissance. With it, you can find information such as the domain's registrant, its administrative and technical contacts, and a listing of their domain servers. Nslookup enables you to query Internet domain name servers. It displays information that can be used to diagnose Domain Name System (DNS) infrastructure and find additional IP addresses. It can also use the MX record to reveal the IP of the mail server.

Another information source is American Registry of Internet Numbers (ARIN). ARIN enables you to search the Whois database for a network's autonomous system numbers (ASNs), network-related handles, and other related point-of-contact information. ARIN's Whois function enables you to query the IP address to find information on the target's use of subnet addressing.

The common traceroute utility is also useful. Traceroute works by exploiting a feature of the Internet Protocol called *time-to-live (TTL)*. It reveals the path that IP packets travel between two systems by sending out consecutive User Datagram Protocol (UDP) packets with ever-increasing TTLs. As each router processes an IP packet, it decrements the TTL. When the TTL reaches zero, the router sends back a "TTL exceeded" Internet Control Message Protocol (ICMP) message to the origin. Thus, routers with DNS entries reveal their names, network affiliations, and geographic locations.

A utility called Visual Trace by McAfee displays the traceroute output visually in map view, node view, or IP view. Additional useful Windows-based tools for gathering information include the following:

- **VisualRoute** — VisualRoute by VisualWare includes integrated traceroute, ping tests, and reverse DNS and Whois lookups. It also displays the actual route of connections and IP address locations on a global map.
- **SmartWhois** — Like Whois, SmartWhois by TamoSoft obtains comprehensive info about the target: IP address; hostname or domain, including country, state or province; city; name of the network provider; administrator; and technical support contact information. Unlike Whois utilities, SmartWhois can find the information about a computer located in any part of the world, intelligently querying the right database and delivering all the related records within a few seconds.

- **Sam Spade** — This freeware tool, primarily used to track down spammers, can also be used to provide information about a target. It comes with a host of useful network tools, including ping, Nslookup, Whois, IP block Whois, dig, traceroute, finger, SMTP, VRFY, Web browser, keep-alive, DNS zone transfer, SMTP relay check, and more.

Port Scanners

Port scanning is one of the most common reconnaissance techniques used by testers to discover the vulnerabilities in the services listening to well-known ports. Once you've identified the IP address of a target through footprinting, you can begin the process of port scanning: looking for holes in the system through which you — or a malicious intruder — can gain access. A typical system has 2^{16} –1 port numbers, each with its own Transmission Control Protocol (TCP) and UDP port that can be used to gain access, if unprotected.

NMap, the most popular port scanner for Linux, is also available for Windows. NMap can scan a system in a variety of stealth modes, depending upon how undetectable you want to be. NMap can determine a wealth of information about a target, including what hosts are available, what services are offered, and what OS is running.

Other port-scanning tools for Linux systems include SATAN, NSAT, VeteScan, SARA, PortScanner, Network Superscanner, CGI Port Scanner, and CGI Sonar.

Vulnerability Scanners

Nessus, a popular open-source tool, is an extremely powerful network scanner that can be configured to run a variety of scans. While a Windows graphical front end is available, the core Nessus product requires Linux to run.

Microsoft's Baseline Security Analyzer (MBSA) is a free Windows vulnerability scanner. MBSA can be used to detect security configuration errors on local computers or on computers across a network. It does have some issues with Windows Update, however, and can't always tell if a patch has been installed.

Popular commercial vulnerability scanners include Retina Network Security Scanner, which runs on Windows, and SAINT, which runs on several Unix/Linux variants, including Mac OS X.

Password Crackers

Password cracking doesn't have to involve fancy tools, but it is a fairly tedious process. If the target doesn't lock you out after a specific number of tries, you can spend an infinite amount of time trying every combination of alphanumeric characters. It's just a question of time and bandwidth before you break into the system.

The most common passwords found are password, root, administrator, admin, operator, demo, test, webmaster, backup, guest, trial, member, private, beta, [company_name], or [known_username].

Three basic types of password cracking methods can be automated with tools:

- **Dictionary** — A file of words is run against user accounts. If the password is a simple word, it can be found fairly quickly.
- **Hybrid** — A hybrid attack works like a dictionary attack, but adds simple numbers or symbols to the file of words. This attack exploits a weakness of many passwords: they are common words with numbers or symbols tacked on the end.
- **Brute force** — The most time-consuming but comprehensive way to crack a password. Every combination of character is tried until the password is broken.

Some common Web password-cracking tools are:

- **Brutus** — Brutus is a password-cracking tool that can perform both dictionary attacks and brute-force attacks whereby passwords are randomly generated from a given character. It can crack the multiple authentication types, HTTP (basic authentication, HTML form/CGI), POP3, FTP, SMB, and Telnet.
- **WebCracker** — WebCracker is a simple tool that takes text lists of usernames and passwords, and uses them as dictionaries to implement basic password guessing.
- **ObiWan** — ObiWan is a Web password-cracking tool that can work through a proxy. Using word lists, it alternates numeric or alphanumeric characters with roman characters to generate possible passwords.
- **Burp Intruder** — Burp Intruder is a Web application security tool that can be used to configure and automate attacks. It can be used to test for Web application vulnerabilities to such attacks as buffer overflow, SQL injection, path traversal, and cross-site scripting.
- **Burp Repeater** — Burp Repeater is a manual tool that can be used to attack Web applications. It operates by supporting the reissuing of HTTP requests from the same window. It also provides a graphical environment to support the manual Web application testing procedures, and complements other tools such as Burp Intruder.

Trojan Horses

A *Trojan horse* is a program that performs unknown and unwanted functions. It can take one or more of the following forms:

- An unauthorized program contained within a legitimate program
- A legitimate program that has been altered by the placement of unauthorized code within it
- Any program that appears to perform a desirable and necessary function but does something unintended

Trojan horses can be transmitted to the computer in several ways — through e-mail attachments, freeware, physical installation, ICQ/IRC chat, phony programs, or infected websites. When the user signs on and goes online, the Trojan horse is activated and the attacker gains access to the system.

Unlike a worm, a Trojan horse doesn't typically self-replicate. The exact type of attack depends on the type of Trojan horse, which can be any of the following:

- Remote access Trojan horses
- Keystroke loggers or password-sending Trojan horses
- Software detection killers
- Purely destructive or denial-of-service Trojan horses

The list of Trojan horses in the wild is expanding quickly, but a few of the earliest have remained relevant since the beginning, and many of these serve as platforms for the development of more lethal variations.

Back Orifice 2000, known as BO2K, is the grandfather of Trojan horses and has spawned a considerable number of imitators. Once installed on a target PC or server machine, BO2K gives the attacker complete control of the victim.

BO2K has stealth capabilities, will not show up on the task list, and runs completely in hidden mode. Back Orifice and its variants have been credited with the highest number of infestations of Windows systems.

Another Trojan horse that has been around for a considerable time is SubSeven, although it is becoming less and less of a problem. SubSeven is a back-door program that enables others to gain full access to Windows systems through the network.

Other common Trojans and spyware currently in the wild include Rovbin, Canary, Remacc.RCPro, Jgidol, IRC.mimic, and NetBus. The SANS Internet Storm Center (`http://isc.sans.org/`) is a good source of information on the latest malware exploits and attack activity.

Buffer Overflows

A *buffer overflow* (or overrun) occurs when a program allocates a specific block length of memory for something, but then attempts to store more data than the block was intended to hold. This overflowing data can overwrite memory areas and interfere with information crucial to the normal execution of the program. While buffer overflows may be a side effect of poorly written code, they can also be triggered intentionally to create an attack.

A buffer overflow can allow an intruder to load a remote shell or execute a command, enabling the attacker to gain unauthorized access or escalate user privileges. To generate the overflow, the attacker must create a specific data feed to induce the error, as random data will rarely produce the desired effect.

For a buffer overflow attack to work, the target system must fail to test the data or stack boundaries and must also be able to execute code that resides in the data or stack segment. Once the stack is smashed, the attacker can deploy his or her payload and take control of the attacked system.

Three common ways to test for a buffer overflow vulnerability are as follows:

- Look for strings declared as local variables in functions or methods, and verify the presence of boundary checks in the source code.
- Check for improper use of input/output or string functions.
- Feed the application large amounts of data and check for abnormal behavior.

Products like Immunix's StackGuard and ProPolice employ stack-smashing protection to detect buffer overflows on stack-allocated variables. Also, vulnerability scanners like Proventia can help protect against buffer overflow.

Buffer overflow vulnerabilities can be detected by manual auditing of the code as well as by boundary testing. Other countermeasures include updating C and C++ software compilers and C libraries to more secure versions, and disabling stack execution in the program.

SQL Injection Attack

SQL injection is an example of a class of injection exploits that occur when one scripting language is embedded inside another scripting language.

The injection targets the data residing in a database through the firewall in order to alter the SQL statements and retrieve data from the database or execute commands. It accomplishes this by modifying the parameters of a Web-based application.

Preventing SQL injection vulnerability involves enforcing better coding practices and database administration procedures. Here are some specific steps to take:

- Disable verbose error messages that give information to the attacker.
- Protect the system account `sa`. It's very common for the `sa` password to be `blank`.
- Enforce the concept of *least privilege* at the database connection.
- Secure the application by auditing the source code to restrict length of input.

Cross-Site Scripting (XSS)

Web application attacks are often successful because the attack is not noticed immediately. One such attack exploits the cross-site scripting (XSS) vulnerability. An XSS vulnerability is created by the failure of a Web-based application to validate user-supplied input before returning it to the client system.

Attackers can exploit XSS by crafting malicious URLs and tricking users into clicking on them. These links enable the attacker's client-side scripting language, such as JavaScript or VBScript, to execute on the victim's browser.

If the application accepts only expected input, then the XSS vulnerability can be significantly reduced. Many Web application vulnerabilities can be minimized by adhering to proper design specifications and coding practices, and implementing security early in the application's development life cycle.

Another piece of advice: Don't rely on client-side data for critical processes during the application development process; and use an encrypted session, such as SSL, without hidden fields.

Social Engineering

Social engineering describes the acquisition of sensitive information or inappropriate access privileges by an outsider, by manipulating people. It exploits the human side of computing, tricking people into providing valuable information or allowing access to that information.

Social engineering is the hardest form of attack to defend against because it cannot be prevented with hardware or software alone. A company may have rock-solid authentication processes, VPNs, and firewalls, but still be vulnerable to attacks that exploit the human element.

Social engineering can be divided into two types: human-based, person-to-person interaction, and computer-based interaction using software that automates the attempt to engineer information.

Common techniques used by an intruder to gain either physical access or system access are as follows:

- Asserting authority or pulling rank
- Professing to have the authority, perhaps supported with altered identification, to enter a facility or system
- Attempting to intimidate an individual into providing information
- Praising, flattering, or sympathizing
- Using positive reinforcement to coerce a subject into providing access or information for system access

Some examples of successful social engineering attacks include the following:

- E-mails to employees from a tester requesting their passwords to validate the organizational database after a network intrusion has occurred
- E-mails to employees from a tester requesting their passwords because work has to be done over the weekend on the system

- An e-mail or phone call from a tester impersonating an official who is conducting an investigation for the organization and requires passwords for the investigation
- An improper release of medical information to individuals posing as medical personnel and requesting data from patients' records
- A computer repair technician who convinces a user that the hard disk on his or her PC is damaged and irreparable, and installs a new hard disk, taking the old hard disk, extracting the information, and selling it to a competitor or foreign government

The only real defense against social engineering attacks is an information security policy that addresses such attacks and educates users about these types of attacks.

Regression Testing

As software evolves, new features are added and existing features are modified. Sometimes these new features and modifications "break" existing functionality — that is, cause accidental damage to existing software components. According to the IEEE Software Engineering Body of Knowledge (IEEE610.12-90), regression testing is the "selective retesting of a system or component to verify that modifications have not caused unintended effects." Regression testing can indicate that software which previously passed the tests no longer does. The problem code can then be identified and fixed to restore the lost functionality. However, as software evolves, a fault that was previously fixed sometimes "reemerges." This kind of reemergence of faults is common and occurs for a number of reasons, including the following:

- **Poor revision control practices** — The fix was not documented properly or the change was accidentally reversed.
- **Software brittleness** — The fix for the initial fault was too narrow in scope. As the software ages and the code base grows larger (becoming "legacy" software), new problems emerge relating to the initial fault but are more difficult to fix without negatively affecting other areas of the software.
- **Repetition of mistakes** — Problematic code is sometimes copied from one area of the software to another; or when a feature is redesigned, the same mistakes are made in the redesign that were made in the original implementation of the feature.

Increasing the code execution coverage of regression testing can help prevent the reemergence of software faults. For example, regression tests could be varied — such as by introducing new sample data or combining tests — to catch problems that were missed with the existing tests. In this way, regression testing would not only verify that previous tests still work, but also mitigate the risk of unintended side effects caused by changes.

For greater assurance, regression testing should also be more extensive for code surrounding vulnerability fixes, code that may contain the same class of vulnerability, and other high-consequence areas of the software. A regression security test plan should be developed containing misuse/abuse cases and attack scenarios (based in part on relevant attack patterns). Earlier test cases can be augmented by any new abuse/misuse cases and attack scenarios suggested by real-world attacks that have emerged since the software was last tested.

Regression testing is often performed by different stakeholders in the software life cycle. During the coding phase, programmers run unit tests to verify that individual units of source code are working properly. The unit is the smallest testable part of the software, often a function or method and its encapsulated data. Unit testing as part of a software development methodology, such as extreme programming, typically relies upon an automated unit testing framework, such as JUnit (`www.junit.org`). The automated unit testing framework integrates with an IDE and enables the developer to generate a stub for a unit test that can then be completed with sample data and additional business logic. In test-driven development, unit code should be created for every software unit. When a programmer follows this discipline, it can result in more confident coding, and enable a programmer to verify that a fault has been fixed earlier in the life cycle (reducing overall development costs).

At the system level, regression testing is a form of functional testing that the software quality assurance team performs by using an automated testing suite. Typically, if a software fault is discovered in the process of testing, it is submitted to the bug-tracking system and assigned to a programmer for remediation. Once the bug is fixed, the software needs to be run through all the regression test cases once again. Consequently, fixing a bug at the quality assurance stage is more expensive than during coding.

According to the IEEE Computer Society, testing is defined as "an activity performed for evaluating product quality, and for improving it, by identifying defects and problems. Software testing consists of the dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the expected behavior."[27]

Secure software testing involves testing for quality assurance through functional, white box and black box, environment, and defect testing. Penetration tests, fuzzing, and simulation tests complemented by conducting cloud system scans provide additional secure software test tools.

Cloud Computing and Business Continuity Planning/Disaster Recovery

Business continuity planning (BCP) and disaster recovery planning (DRP) involve the preparation, testing, and updating of the actions required to protect critical business processes from the effects of major system and network failures. From the cloud perspective, these important business processes are heavily dependent on cloud-based applications and software robustness and security. BCP comprises scoping and initiating the planning, conducting a business impact assessment (BIA), and developing the plan. DRP includes developing the DRP processes, testing the plan, and implementing the disaster recovery procedures.

Designing, developing, and implementing a quality and effective BCP and DRP is a major undertaking, involving many person-hours and, in many instances, high hardware or software costs. These efforts and costs are worthwhile and necessary, but they impact a large number of organizational resources. Cloud computing offers an attractive alternative to total, in-house BCP/DRP implementations. Before exploring cloud computing solutions to BCP/DRP, it is important to establish baseline definitions of key related terms.

Definitions

A *disaster* is a rapidly occurring or unstoppable event that can cause suffering, loss of life, or damage. In many instances, the aftermath of a disaster can impact social or natural conditions for a long period of time.

A DRP is a comprehensive statement of consistent actions to be taken before, during, and after a disruptive event that causes a significant loss of information systems resources. The number one priority of DRP is personnel safety and evacuation, followed by the recovery of data center operations and business operations and processes.

Specific areas that can be addressed by cloud providers include the following:

- Protecting an organization from a major computer services failure
- Providing extended backup operations during an interruption
- Providing the capability to implement critical processes at an alternate site
- Guaranteeing the reliability of standby systems through testing and simulations
- Returning to the primary site and normal processing within a time frame that minimizes business loss by executing rapid recovery procedures.
- Minimizing the decision-making required by personnel during a disaster

- Proving an organized way to make decisions if a disruptive event occurs
- Minimizing the risk to the organization from delays in providing service

A business continuity plan addresses the means for a business to recover from disruptions and continue support for critical business functions. It is designed to protect key business processes from natural or man-made failures or disasters and the resultant loss of capital due to the unavailability of normal business processes. A BCP includes a business impact assessment (BIA), which, in turn, contains a vulnerability assessment.

A BIA is a process used to help business units understand the impact of a disruptive event. A vulnerability assessment is similar to a risk assessment in that it contains both a quantitative (financial) section and a qualitative (operational) section. It differs in that it is smaller than a full risk assessment and is focused on providing information that is used solely for the business continuity plan or disaster recovery plan.

General Principles and Practices

Several major steps are required to produce an effective DRP and BCP. In this section, the principles and practices behind DRB and BCP are reviewed in the context of their ability to be provided through cloud services.

Disaster Recovery Planning

As mentioned in the preceding section, the primary objective of a disaster recovery plan is to provide the capability to implement critical processes at an alternate site and return to the primary site and normal processing within a time frame that minimizes loss to the organization by executing rapid recovery procedures. In many scenarios, the cloud platforms already in use by a customer are extant alternate sites. Disasters primarily affect availability, which impacts the ability of staff to access the data and systems, but it can also affect the other two tenets, confidentiality and integrity. In the recovery plan, a classification scheme such as the one shown in Table 3-4 can be used to classify the recovery time-frame needs of each business function.

The DRP should address all information processing areas of the company:

- Cloud resources being utilized
- LANs, WANs, and servers
- Telecommunications and data communication links
- Workstations and workspaces
- Applications, software, and data

- Media and records storage
- Staff duties and production processes

Table 3-4: Recovery Time Frame Requirements Classification

RATING CLASS	RECOVERY TIME FRAME REQUIREMENTS
AAA	Immediate recovery needed; no downtime allowed
AA	Full functional recovery required within four hours
A	Same-day business recovery required
B	Up to 24 hours downtime acceptable
C	24 to 72 hours downtime acceptable
D	Greater than 72 hours downtime acceptable

The means of obtaining backup services are important elements in the disaster recovery plan. The typically used alternative services are as follows:

- **Mutual aid agreements** — An arrangement with another company that might have similar computing needs. The other company may have similar hardware or software configurations or may require the same network data communications or Internet access.
- **Subscription services** — Third-party commercial services that provide alternate backup and processing facilities. An organization can move its IT processing to the alternate site in the event of a disaster.
- **Multiple centers** — Processing is spread over several operations centers, creating a distributed approach to redundancy and sharing of available resources. These multiple centers could be owned and managed by the same organization (in-house sites) or used in conjunction with a reciprocal agreement.
- **Service bureaus** — Setting up a contract with a service bureau to fully provide all alternate backup-processing services. The disadvantages of this arrangement are primarily the expense and resource contention during a large emergency.

Recovery plan maintenance techniques must be employed from the outset to ensure that the plan remains fresh and usable. It's important to build maintenance procedures into the organization by using job descriptions that centralize responsibility for updates. In addition, create audit procedures that can report regularly on the state of the plan. It is important to ensure that multiple versions of the plan don't exist because that could create confusion during an emergency.

The Foreign Corrupt Practices Act of 1977 imposes civil and criminal penalties if publicly held organizations fail to maintain adequate controls over their information systems. Organizations must take reasonable steps to ensure not only the integrity of their data, but also the system controls the organization put in place.

Disaster Recovery Plan Testing

The major reasons to test a disaster recovery plan are summarized as follows:

- To inform management of the recovery capabilities of the enterprise
- To verify the accuracy of the recovery procedures and identify deficiencies
- To prepare and train personnel to execute their emergency duties
- To verify the processing capability of the alternate backup site or cloud provider

Certain fundamental concepts apply to the testing procedure. Primarily, the testing must not disrupt normal business functions, and the test should begin with the least complex case and gradually work up to major simulations.

Management Roles

The plan should also detail the roles of senior management during and following a disaster:

- Remaining visible to employees and stakeholders
- Directing, managing, and monitoring the recovery
- Rationally amending business plans and projections
- Clearly communicating new roles and responsibilities
- Monitoring employee morale
- Providing employees and family with counseling and support
- Reestablishing accounting processes, such as payroll, benefits, and accounts payable
- Reestablishing transaction controls and approval limits

WHEN A DISASTER CAN BE DECLARED TO BE OVER

A disaster is not over until all operations have been returned to their normal location and function. A very large window of vulnerability exists when transaction processing returns from the alternate backup site to the original production site. If cloud computing resources provide a large portion of the backup for the organization, any possible vulnerabilities will be mitigated. The disaster can be officially declared over only when all areas of the enterprise are back to normal in their original home, and all data has been certified as accurate.

Business Continuity Planning

A BCP is designed to keep a business running, reduce the risk of financial loss, and enhance a company's capability to recover promptly following a disruptive event. The four principle components of a BCP are as follows:

- **Scope and plan initiation** — Creating the scope and other elements needed to define the plan's parameters
- **Business impact assessment (BIA)** — Assisting the business units in understanding the impact of a disruptive event. This phase includes the execution of a vulnerability assessment.
- **Business continuity plan development** — Using information collected in the BIA to develop the actual business continuity plan. This process includes the areas of plan implementation, plan testing, and ongoing plan maintenance.
- **Plan approval and implementation** — Obtaining the final senior management sign-off, creating enterprise wide awareness of the plan, and implementing a maintenance procedure for updating the plan as needed.

The BIA

A key element of the BCP process is conducting a BIA. The purpose of a BIA is to create a document that outlines what impact a disruptive event would have on the business. The impact might be financial (quantitative) or operational (qualitative), such as the inability to respond to customer complaints. A vulnerability assessment is often part of the BIA process. A BIA has three primary goals:

- **Criticality prioritization** — Every critical business unit process must be identified and prioritized, and the impact of a disruptive event must be evaluated.
- **Downtime estimation** — The BIA is used to help estimate the maximum tolerable downtime (MTD) that the business can withstand and still remain viable; that is, what is the longest period of time a critical process can remain interrupted before the company can never recover? The BIA process often determines that this time period is much shorter than expected.
- **Resource requirements** — The resource requirements for the critical processes are also identified at this time, with the most time-sensitive processes receiving the most resource allocation.

A BIA generally involves four steps:

1. Gathering the needed assessment materials
2. Performing the vulnerability assessment

3. Analyzing the information compiled
4. Documenting the results and presenting recommendations

The Vulnerability Assessment

The vulnerability assessment is often part of a BIA. It is similar to a risk assessment but it is smaller than a full risk assessment and is focused on providing information that is used solely for the business continuity plan or disaster recovery plan.

The function of a vulnerability assessment is to conduct a loss impact analysis. Because there are two parts to the assessment, a financial assessment and an operational assessment, it is necessary to define loss criteria both quantitatively and qualitatively.

Quantitative loss criteria can be defined as follows:

- Incurring financial losses from loss of revenue, capital expenditure, or personal liability resolution
- Incurring additional operational expenses due to the disruptive event
- Incurring financial loss resulting from the resolution of violating contract agreements
- Incurring financial loss resulting from the resolution of violating regulatory or compliance requirements

Qualitative loss criteria can consist of the following:

- The loss of competitive advantage or market share
- The loss of public confidence or credibility, or incurring public embarrassment

During the vulnerability assessment, critical support areas must be defined in order to assess the impact of a disruptive event. A critical support area is defined as a business unit or function that must be present to sustain continuity of the business processes, protect life, provide safety, or avoid public relations embarrassment.

Critical support areas could include the following:

- Telecommunications, data communications, or information technology areas
- Physical infrastructure or plant facilities, transportation services
- Accounting, payroll, transaction processing, customer service, purchasing

Typical steps in performing a vulnerability assessment are as follows:

1. List potential disruptive events (i.e., natural, technological, and man-made).
2. Estimate the likelihood of occurrence of a disruptive event.

3. Assess the potential impact of the disruptive event on the organization (i.e., human impact, property impact, and business impact).
4. Assess external and internal resources required to deal with the disruptive event.

Enterprise wide awareness of the plan is important because an organization's ability to recover from an event will most likely depend on the efforts of many individuals. Employee awareness of the plan also emphasizes the organization's commitment to its employees. Specific training may be required for certain personnel to carry out their tasks; and quality training is perceived as a benefit, which increases the interest and commitment of personnel in the BCP process.

Using the Cloud for BCP/DRP

Adopting a cloud strategy for BCP/DRP offers significant benefits without large amounts of capital and human resource investments. Effective cloud-based BCP/DRP requires planning, preparation, and selecting the cloud provider that best meets an organization's needs. A critical issue is the stability and viability of the vendor. The vendor should have the financial, technical, and organizational resources to ensure it will be around for both the short term and the long term. In addition, in order for cloud BCP/DRP to reach its full potential, standardization across a variety of architectures has to evolve.

Proper design of a cloud-based IT system that meets the requirements of a BCP and DRP should include the following:

- Secure access from remote locations
- A distributed architecture with no single point of failure
- Integral redundancy of applications and information
- Geographical dispersion

Redundancy Provided by the Cloud

Cloud-based BCP and DRP eliminate the need for expensive alternative sites and the associated hardware and software to provide redundancy. This approach also provides for low cost and widely available, dynamically scalable, and virtualized resources.

With a cloud computing paradigm, the backup infrastructure is always in place. Thus, data access and running business applications are available on cloud servers. Another option is to implement a hybrid cloud with collocation of resources and services. Cloud service providers also offer organizations the option to control the backup process thorough the use of storage area networks (SANs). Examples of elements that require backup are application data, media files, files that have changed, recent documents, the operating system, and archival files.

Secure Remote Access

In order for cloud-based BCP/DRP to be effective, the cloud applications and data must be securely accessible from all parts of the globe. One solution is for the cloud vendor to establish a global traffic management system that provides the following customer services:

- Meets service-level agreements for availability and performance
- Regulates and controls traffic among virtual machines located at multiple data centers
- Maximizes speed and performance by directing traffic to the closest and most logical cloud data center

These services have to be implemented and conducted in a secure environment to protect both the cloud consumer and cloud provider from compromises and attacks.

Integration into Normal Business Processes

Services provided by a cloud vendor at a remote location are, in almost all cases, isolated geographically from the customer's facilities. The cloud enterprise is strongly protected both physically and technically. At the consumer's site, if cloud processing and data storage are integrated into the daily routine of the business, recovery from a disruptive event at the user organization can be more rapid and involve less time and personnel. In many instances, the cloud resources will be used in normal operations and will be available during a disruptive event at the organization's location without large amounts of transfer activity.

Summary

Security of cloud-based applications and data is one of the principal concerns of cloud customers. Secure software and secure software life cycle management are intrinsic to the protection of cloud services. The information security of cloud systems depends on the classical principles of confidentiality, availability, and integrity, but applied to distributed, virtualized, and dynamic architectures. Important secure software design and application principles include least privilege, separation of duties, defense in depth, fail-safe, and open design.

Secure cloud software also depends on applying software requirements engineering, design principles, code practices, security policy implementation and decomposition, and secure software testing. Valuable testing types are penetration testing, functional testing, performance testing, and vulnerability testing.

The availability of an organization's applications and data residing on cloud servers is a prime consideration of acquiring cloud services. Business continuity planning and disaster recovery planning are important activities for any organization. Cloud computing can offer a low entry cost into providing redundant, resilient, backup capabilities for an organization, and minimize interference in business processes during and following a disruptive event.

Notes

1. Cloud Security Alliance Guidance Version 2.1, 2009, (`http://www.cloudsecurityalliance.org/guidance/csaguide.pdf`).
2. Information Assurance Technology Analysis Center (IATAC), Data and Analysis Center for Software (DACS), Software Security Assurance, State-of-the-Art Report (SOAR), July 31, 2007.
3. Komaroff, M., and Baldwin, K., DoD Software Assurance Initiative, September 13, 2005 (`https://acc.dau.mil/CommunityBrowser.aspx?id=25749`).
4. Goertzel, K., Winograd, T., et al., "Enhancing the Development Life Cycle to Produce Secure Software," Draft Version 2.0. Rome, New York: United States Department of Defense Data and Analysis Center for Software, July 2008.
5. Saltzer, J. H., and Schroeder, M. D., "The Protection of Information in Computer Systems," Fourth ACM Symposium on Operating Systems Principles, October 1974.
6. National Security Agency, "Information Assurance Technical Framework (IATF)," Release 3.1, September 2002.
7. Goertzel, K., Winograd, T., et al., "Enhancing the Development Life Cycle to Produce Secure Software.
8. van Lamsweerde A., Brohez, S., De Landtsheer, R., and Janssens, D., "From System Goals to Intruder Anti-Goals: Attack Generation and Resolution for Security Requirements Engineering," in *Proceedings of the Requirements for High Assurance Workshop*, Monterey Bay, CA, September 8, 2003, pp. 49–56.
9. Chung, L., "Representing and Using Nonfunctional Requirements," Ph.D. Thesis, Dept. of Computer Science, University of Toronto, 1993.

10. Goertzel, Winograd, T., et al., "Enhancing the Development Life Cycle to Produce Secure Software."
11. van Lamsweerde, Brohez, De Landtsheer, and Janssens, "From System Goals to Intruder Anti-Goals: Attack Generation and Resolution for Security Requirements Engineering."
12. Goertzel, Winograd, et al., "Enhancing the Development Life Cycle to Produce Secure Software."
13. NIST FIPS Publication 200, "Minimum Security Requirements for Federal Information and Information Systems," March 2006.
14. Goertzel, Winograd, et al., "Enhancing the Development Life Cycle to Produce Secure Software."
15. American Institute of Certified Public Accountants (AICPA), "Accounting for the Costs of Computer Software Developed or Obtained for Internal Use," AICPA Statement of Position (SOP) No. 98-1, March 1998, www.aicpa.org.
16. ISACA, "IS Auditing Guideline on Due Professional Care," Information Systems Audit and Control Association, March 1, 2008, www.isaca.org.
17. Tassey, G., "The Economic Impacts of Inadequate Infrastructure for Software Testing," National Institute of Standards and Technology, Technical Report, 2002.
18. Sun, X., Feng, C., Shen, Y., and Lombardi, F., *Protocol Conformance Testing Using Unique Input/Output Sequences* (Hackensack, NJ: World Scientific Publishing Co., 1997).
19. Tassey, G., "The Economic Impacts of Inadequate Infrastructure for Software Testing."
20. Du, W., and Mathur, A. P., "Testing for Software Vulnerability Using Environment Perturbation," *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2000)*, Workshop on Dependability versus Malicious Faults, New York, NY, June 25–28, 2000 (Los Alamitos, CA: IEEE Computer Society Press, 2000), pp. 603–12.
21. Information Assurance Technology Analysis Center (IATAC)/Data and Analysis Center for Software (DACS), "Software Security Assurance State-of-the-Art Report (SOAR)," July 31, 2007.
22. Goertzel, K. M., Winograd, T., et al., "Enhancing the Development Life Cycle to Produce Secure Software."
23. Fink, G., and Bishop, M., "Property-Based Testing: A New Approach to Testing for Assurance," *SIGSOFT Software Engineering Notes* 22, 4 (July 1997): 74–80.

24. Whittaker, J. A., and Thompson, H. H., "Black Box Debugging," *Queue 1*, 9 (December/January 2003–2004).

25. National Institute of Standards and Technology (NIST), 1997, "Metrology for Information Technology (IT)," www.nist.gov/itl/lab/nistirs/ir6025 .htm.

26. Oladimeji, E. A., and Chung, L., "Analyzing Security Interoperability during Component Integration," in *Proceedings of the 5th IEEE/ACIS International Conference on Computer and Information Science and 1st IEEE/ACIS International Workshop on Component-Based Software Engineering, Software Architecture and Reuse* (July 10–12, 2006). ICIS-COMSAR, IEEE Computer Society, Washington, DC, pp. 121–29.

27. Abran, A., Moore J., (executive editors), Bourque, P., Dupuis, R., and Tripp, L. (editors), "Guide to the Software Engineering Body of Knowledge," IEEE Computer Society, 2004.