

Chapter 8

Principles of Security Models, Design, and Capabilities

THE CISSP TOPICS COVERED IN THIS CHAPTER INCLUDE:

✓ Domain 3.0: Security Architecture and Engineering

- 3.1 Research, implement and manage engineering processes using secure design principles
 - 3.1.4 Secure defaults
 - 3.1.5 Fail securely
 - 3.1.7 Keep it simple and small
 - 3.1.8 Zero trust or trust but verify
 - 3.1.9 Privacy by design
 - 3.1.11 Secure access service edge
- 3.2 Understand the fundamental concepts of security models (e.g. Biba, Star Model, Bell-LaPadula)
- 3.3 Select controls based upon systems security requirements
- 3.4 Understand security capabilities of Information Systems (IS) (e.g., memory protection, Trusted Platform Module (TPM), encryption/decryption)
- 3.10 Manage the information system lifecycle
 - 3.10.1 Stakeholders needs and requirements
 - 3.10.2 Requirements analysis
 - 3.10.3 Architectural design
 - 3.10.4 Development/implementation
 - 3.10.5 Integration
 - 3.10.6 Verification and validation
 - 3.10.7 Transition/deployment
 - 3.10.8 Operations and maintenance/sustainment

■ 3.10.9 Retirement/disposal

Understanding the philosophy behind security solutions helps limit your search for the best controls for your specific security needs. In this chapter, we discuss secure system design principles, security models, the Common Criteria, and security capabilities of information systems.

Domain 3 includes a variety of topics that are discussed in other chapters, including the following:

- [Chapter 1](#), “Security Governance Through Principles and Policies”
- [Chapter 6](#), “Cryptography and Symmetric Key Algorithms”
- [Chapter 7](#), “PKI and Cryptographic Applications”
- Chapter 8, “Principles of Security Models, Design, and Capabilities”
- [Chapter 9](#), “Security Vulnerabilities, Threats, and Countermeasures”
- [Chapter 10](#), “Physical Security Requirements”
- [Chapter 14](#), “Controlling and Monitoring Access”
- [Chapter 16](#), “Managing Security Operations”
- [Chapter 20](#), “Software Development Security”
- [Chapter 21](#), “Malicious Code and Application Attacks”

Secure Design Principles

Security should be a consideration at every stage of a system's development. Programmers, developers, engineers, and so on should strive to build security into every application or system they develop, with greater levels of security provided to critical applications and those that process sensitive information. It's imperative to consider the security implications of a development project in the early stages because it's much easier to build security into a system during development than adding security

to an existing system. Developers should research, implement, and manage engineering processes using secure design principles.

Objects and Subjects

Controlling access to any resource in a secure system involves two entities. The *subject* is the active entity that requests access to a resource. A subject is commonly a user, but it can also be a process, program, computer, or organization. The *object* is the passive entity that the subject wants to access. An object is commonly a resource, such as a file or printer, but it can also be a user, process, program, computer, or organization. You want to keep a broad understanding of the terms subject and object, rather than only considering users and files. Access is the relationship between a subject and object, including reading, writing, modifying, deleting, printing, moving, backing up, and many other operations or activities. Authorization or access control is the management of the relationship between subjects and objects.

Remember that the actual entities referenced by the terms *subject* and *object* are specific to an individual access request. The entity serving as the object in one access event could serve as the subject in another. For example, process A may ask for data from process B. To satisfy process A's request, process B must ask for data from process C. In this example ([Table 8.1](#)), process B is the object of the first request and the subject of the second request.

[TABLE 8.1](#) Subjects and objects

Request	Subject	Object
First request	Process A	Process B
Second request	Process B	Process C

This also serves as an example of transitive trust. *Transitive trust* is the concept that if A trusts B and B trusts C, then A inherits the trust of C through the transitive property ([Figure 8.1](#))—which works as it would in a mathematical equation: if $a = b$ and $b = c$, then $a = c$. In the previous example, when A requests data from B

and then B requests data from C, the data that A receives is essentially from C. Transitive trust is a serious security concern because it may enable the bypassing of restrictions or limitations between A and C, especially if A and C both support interaction with B. An example would be when an organization blocks access to Facebook or YouTube to increase worker productivity. Thus, workers (A) do not have access to certain Internet sites (C). However, if workers are able to have access to a web proxy, virtual private network (VPN), or anonymization service, then this can serve as a means to bypass the local network restriction. In other words, if workers (A) are accessing VPN service (B), and the VPN service (B) can access the blocked internet service (C), then A can access C through B via a transitive trust exploitation.

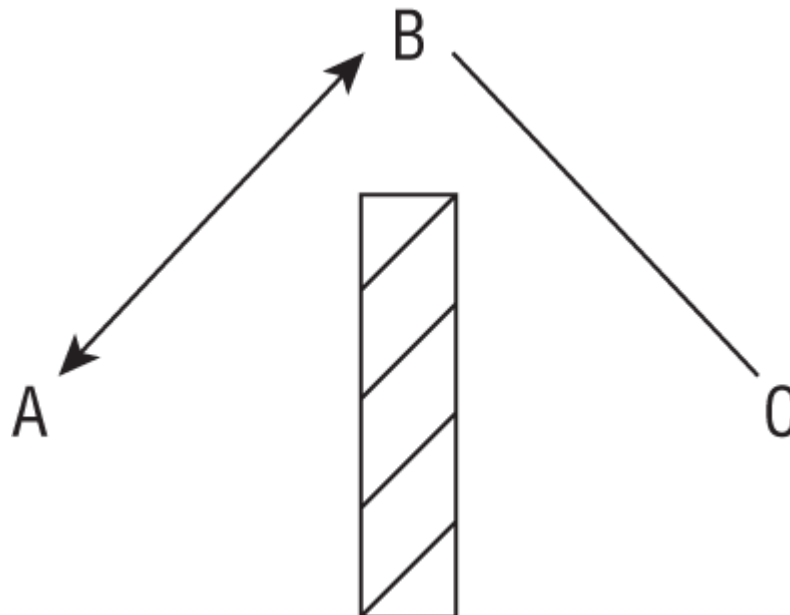


FIGURE 8.1 Transitive trust

Closed and Open Systems

Systems are designed and built according to one of two differing philosophies. A *closed system* is designed to work well with a narrow range of other systems, generally all from the same manufacturer. The standards for closed systems are often proprietary and not normally disclosed. *Open systems*, on the other hand, are designed using agreed-on industry standards. Open systems are much easier to integrate with systems from

different manufacturers that support the same standards or that use compatible application programming interfaces (APIs).

An API is a defined set of interactions allowed between computing elements, such as applications, services, networking, firmware, and hardware. An API defines the types of requests that can be made, the exact means to make the requests, the data forms of the exchange, and other related requirements (such as authentication and/or session encryption). APIs make interoperability of computing elements possible. Without APIs, computing components would be unable to interact directly, and information sharing would not be easy. APIs are what make modern computing and the Internet possible. The app on your smartphone talks to the phone's operating system via an API; the phone's operating system talks over the telco or Wi-Fi network via an API to reach the cloud service's API to submit a request and receive a response.

Closed systems are harder to integrate with unlike systems, but this “feature” could make them more secure. A closed system is often composed of proprietary hardware and software that does not incorporate industry standards or offer an open API. This lack of integration ease means that attacks that typically focus on generic system components either will not work or must be customized to be successful. In many cases, attacking a closed system is harder than launching an attack on an open system, since a unique exploit of a unique vulnerability would be required. In addition to the lack of known vulnerable components on a closed system, it is often necessary to possess more in-depth knowledge of the specific target system to launch a successful attack.

Open systems are generally far easier to integrate with other open systems. It is easy, for example, to create a local area network (LAN) with a Microsoft Windows Server machine, a Linux machine, and a Macintosh machine. Although all three computers use different operating systems and could represent up to three different hardware architectures, each supports industry standards and open APIs, which makes it easy for network (or other) communications to occur. This ease of

interoperability comes at a price, however. Because standard communications components are incorporated into each of these three open systems, there are far more predictable entry points and methods for launching attacks. In general, their openness makes them more vulnerable to attack, and their widespread availability makes it possible for attackers to find plenty of potential targets. Also, open systems are more popular and widely deployed than closed systems and thus attract more attention from attackers. An attacker who develops basic attacking skills will find more targets that are open systems than closed ones. Inarguably, there's a greater body of shared experience and knowledge on how to attack open systems than there is for closed systems. The security of an open system is therefore more dependent on the use of secure and defensive coding practices and a thoughtful defense-in-depth deployment strategy (see [Chapter 1](#)).

Open Source vs. Closed Source

It's also helpful to keep in mind the distinction between open-source and closed-source systems. An *open-source* solution is one where the source code, and other internal logic, is exposed to the public. A closed-source solution is one where the source code and other internal logic is hidden from the public. Open-source solutions often depend on public inspection and review to improve the product over time. *Closed-source* solutions are more dependent on the vendor/programmer to revise the product over time. Both open-source and closed-source solutions can be available for sale or at no charge, but the term *commercial* typically implies closed-source. However, closed-source code is sometimes revealed through either vendor compromise or through decompiling or disassembly. The former is always a breach of ethics and often the law, whereas the latter is a standard element in ethical reverse engineering or systems analysis.

It is also the case that a closed-source program can be either an open system or a closed system, and an open-source program can be either an open system or a closed system. Since these terms are so similar, it is essential to read questions carefully. Additional coverage of open-source and other software issues is included in [Chapter 20](#), “Software Development Security.”



CISSP Objective 3.1 lists 11 secure design principles. Six of them are covered in this chapter (i.e., secure defaults, fail securely, keep it simple and small, zero trust or trust but verify, privacy by design, and secure access service edge); the other five are covered in other chapters where they integrate best with broader coverage of similar topics. For threat modeling and defense in depth, see [Chapter 1](#); for least privilege and segregation of duties, see [Chapter 16](#); and for shared responsibility, see [Chapter 9](#).

Secure Defaults

You have probably heard the phrase “the tyranny of the default.” But do you know what this means? Tyranny has several definitions, but the one that applies here is “a rigorous condition imposed by some outside agency or force” (attributed to American historian Dixon Wecter). Many assume that the settings present in a software or hardware product when it is first installed are optimal. This is based on the assumption that the designers and developers of a product know the most about that product, so the settings they made are likely the best ones. However, this assumption overlooks the fact that often, the default settings of a product are selected to minimize installation problems to avoid increased load on the technical support services. For example, consider the fact that most devices have a default password, which minimizes the costs of support when installing or using the product for the first time. Unfortunately, default settings often make the discovery and exploitation of equipment trivial for attackers.

Never assume that the default settings of any product are secure. They typically are not because secure settings would likely get in the way of existing business tasks or system operations. It is always up to the system's administrator and/or company security staff to alter a product's settings to comply with the organization's security policies. Unless your organization hired

the developer, that developer did not craft the code or choose settings specifically for your organization's use of their product.

A much better assumption is that the default settings of a product are the worst possible options for your organization. Therefore, you need to review every setting to determine what it does and what you need configured to do to optimize security while supporting business operations.

Fortunately, there is some movement toward more *secure defaults*. Some products, especially security products, may now be designed with their most secure settings enabled by default. However, such a locked-down product will have fewer enabled capabilities and will likely be less user-friendly. Thus, while being more secure, secure defaults may be an obstacle for those who only want their systems to “just work.”

If you are a developer, you must create detailed explanations of each of your product's configuration options. You can't assume that customers know everything about your product, especially the configuration settings and what each option does to alter its features, operations, communications, etc. You may be required to have default settings to make the product as easy to install as possible. Still, you may be able to provide one or more configurations in either written instructional form or in a file that can be imported or applied. This will go a long way to assist customers with gaining the most advantage from your product while minimizing the security risks.



Restrictive defaults refer to a practice or policy where the default settings or options are intentionally configured to be more limiting or restrictive to enhance security, privacy, or compliance. This concept is often applied in software, systems, or services where administrators or users are provided with a preconfigured set of options and the default settings prioritize security, safety, or regulatory compliance.

Fail Securely

System failures can occur due to a wide range of causes. Once the failure event occurs, how the system or environment handles the failure is important. The most desired result is for an application to *fail securely*. The first type of failure management is programmatic error handling (aka *exception handling*). This is the process where a programmer codes in mechanisms to anticipate and defend against errors to avoid the termination of execution. Error handling includes code that will attempt to handle errors when they arise before they can cause harm or interrupt execution.

One such mechanism, supported by many languages, is a `try...catch` statement. This logical block statement is used to place code that could result in an error on the `try` branch and then code that will be executed if there is an error on the `catch` branch. This is similar to `if...then...else` statements, but it is designed to handle errors deftly.

Other mechanisms are to avoid or prevent errors, especially regarding user input. Input sanitization, input filtering, and input validation are terms used to refer to this concept. This often includes checking the input for length, filtering against a block list of unwanted input, and escaping metacharacters. See more about secure coding practices in [Chapter 9](#); [Chapter 15](#), “Security Assessment and Testing”; and [Chapter 20](#).

There are several similar terms that can be confusing and thus require a bit of focus to comprehend. These terms are fail-soft, fail-secure, fail-safe, fail-open, and fail-closed. Typically, confusion occurs when not understanding the context where these terms are used. The two primary contexts are the physical world and the digital environment. In the physical world, entities primarily prioritize the protection of people. However, there are some circumstances where assets are protected in priority over people. In the digital world, entities focus on protecting assets, but the type of protection may vary among the CIA Triad.

When a program fails securely, it is able to do so only because it was designed and programmed to. When secure failure is

integrated into a system, the designer must make a few difficult choices about what the results of a failure event will be. The first question to be resolved is whether the system can operate in a fail-soft mode. To *fail-soft* is to allow a system to continue to operate after a component fails. This is an alternative to having a failure cause a complete system failure. An example is a typical multitasking operating system that can support numerous simultaneous applications. If one application fails, the others can typically continue to operate.

If fail-soft isn't a viable option, then the designer needs to consider the type of product, its deployment scenarios, and the priorities related to failure response. In other words, when the product fails without a fail-soft design, it will fail completely. The designer/developer must decide what type of complete failure to perform and what to protect or sacrifice to achieve the planned failure result. There are numerous scenarios to consider. The initial distinction is whether the product is something that affects the physical world, such as a door-locking mechanism, or primarily a digital asset—focused product, such as a firewall. If a product can affect the physical world, then the life and safety of humans must be considered and likely prioritized. This human protection prioritization is called *fail-safe*. The idea is that when a failure occurs, the system, device, or product will revert to a state that protects the health and safety of people. For example, a fail-safe door will open easily in an emergency to allow people to escape a building. However, this implies that the protection of assets may be sacrificed in favor of personnel safety. However, in some physical world situations, a product could be designed and intended to protect assets in priority above people, such as a bank vault, medical lab, or even a data center. A fail-secure system prioritizes the physical security of assets over any other consideration. For example, a vault door may automatically close and lock when the building enters a state of emergency. This prioritization of asset protection may occur at the potential cost of harming personnel who could be trapped inside. Obviously, the prioritization of physical world products should be considered carefully. In the context of the physical world, the term *fail-open*

is a synonym for fail-safe, and fail-closed is a synonym for fail-secure.

If the product is primarily digital, then the focus of security is completely on digital assets. That means the designer must then prioritize the security aspect—namely, availability or confidentiality and integrity. If the priority is for maintaining availability, then when the product fails, the connection or communication is allowed to continue. This is known as fail-open. If the priority is for maintaining confidentiality and integrity, then when the product fails, the connection or communication is cut off. This is known as fail-secure, fail-closed, and/or fail-safe (again, in the context of a digital environment).



The Internet Engineering Task Force (IETF) recommends avoiding using the term fail-safe when discussing digital-only issues as it introduces the concept of human safety, which is not a concern in a digital context and thus causes unnecessary confusion.

However, when the context switches from the physical world to the digital world, the definition of fail-safe changes. An example could be a firewall, which, if designed to fail-open, would allow communications without filtering. In contrast, implementing a fail-secure, fail-closed, or fail-safe solution would cut off communications. The fail-open state protects availability by sacrificing confidentiality and integrity, whereas the fail-closed state sacrifices availability to preserve confidentiality and integrity. Another example of a digital environment event following a fail-secure, fail-closed, and/or fail-safe procedure is when an operating system encounters a processing or memory isolation violation, it terminates all executions then initiates a reboot. This mechanism is known as a stop error or the Blue Screen of Death (BSOD) in Windows.

A condensed summary of the context and protection priority of these terms is presented in [Table 8.2](#).

TABLE 8.2 Fail terms' definitions related to physical and digital products

Physical	State	Digital
Protect People	Fail-Open	Protect Availability
Protect People	Fail-Safe	Protect Confidentiality and Integrity
Protect Assets	Fail-Closed	Protect Confidentiality and Integrity
Protect Assets	Fail-Secure	Protect Confidentiality and Integrity

Keep It Simple and Small

Keep it simple is a shortened form of the classic statement “keep it simple, stupid” or “keep it stupid simple.” This is sometimes called the KISS principle. In security, this concept encourages avoiding overcomplicating the environment, organization, or product design. The more complex a system, the more difficult it is to secure. The more lines of code, the more challenging it is to test it thoroughly. The more parts there are, the more places there are for things to go wrong. The more features and capabilities, the larger the attack surface. Thus, keeping a system's design or software coding simple and small will directly relate to the difficulty of establishing, testing, and verifying its security.

There are many other concepts that have a similar or related emphasis, such as the following:

“Don't Repeat Yourself” (DRY) The idea is to eliminate redundancy in software by not repeating the same code in multiple places. Otherwise, duplicating code would increase the difficulty if changes are needed.

Computing Minimalism Crafting code to use the least necessary hardware and software resources possible is computing minimalism. This is also the goal of the program

evaluation and review technique (PERT), which is discussed in [Chapter 20](#).

Rule of Least Power Use the least powerful programming language that is suitable for the needed solution.

“Worse Is Better” (aka New Jersey Style) The quality of software does not necessarily increase with increased capabilities and functions; there is often a worse software state (i.e., fewer functions), which is the better (i.e., preferred, maybe more secure) option.

“You Aren’t Gonna Need It” (YAGNI) Programmers should not write capabilities and functions until they are necessary, so rather than create them when you think of them, create them only when you need them.

It is easy to get caught up in adding complexity to a system, whether that system is a software program or an organizational IT security structure. The KISS principle encourages us all to avoid the overly complex in favor of the streamlined, optimized, and reduced solution. Simpler solutions are easier to secure, easier to troubleshoot, and easier to verify.

Zero-Trust

Zero trust is a security concept where nothing and no person inside the organization is automatically trusted. There has long been an assumption that everything on the inside is trusted and everything on the outside is untrusted. This has led to a significant security focus on endpoint devices, the locations where users interact with company resources. An endpoint device could be a user's workstation, a tablet, a smartphone, an Internet of Things (IoT) device, an industrial control system (ICS), an edge computing sensor, or any public-facing servers in a screened subnet or extranet. The idea that a security perimeter exists between the safe inside and the harmful outside is problematic. There have been too many occurrences of security breaches caused by insiders as well as external attacker breaches that

gained the freedom to perform lateral movement internally once they breached the security barrier.

The concept of a security perimeter is further complicated by the use of mobile devices, the cloud, and the proliferation of endpoint devices. If a device can operate inside a private network, then be used externally with direct internet access, and then returned to the private network, there is no actual security perimeter. For most organizations, there is no longer a clearly defined line between inside and outside.

Zero trust is an alternate approach to security where nothing and no person is automatically trusted. Instead, each request for activity or access is assumed to be from an unknown and untrusted location until otherwise verified. The concept is “never trust, always verify.” Since anyone and anything could be malicious, every transaction should be verified before it is allowed to occur. The zero-trust model is based on “assume breach,” meaning that you should always assume a security breach has occurred and that whoever or whatever is making a request could be malicious. The goal is to have every access request be authenticated, authorized, and encrypted prior to the access being granted to a resource or asset. The implementation of a zero-trust architecture does involve a significant shift from historical security management concepts. This shift typically requires internal microsegmentation and strong adherence to the principle of least privilege. This approach prevents lateral movement so that if there is a breach or even a malicious insider, their ability to move about the environment is severely restricted.



Microsegmentation is dividing up an internal network into numerous subzones. Each zone is separated from the others by internal segmentation firewalls (ISFWs), subnets, or virtual local area network (VLANs). Zones could be as small as a single device, such as a high-value server or even a client or endpoint device. Any and all communications between zones are filtered, may be required to authenticate, often require session encryption, and may be subjected to allow list and block list control.

Zero trust is implemented using a wide range of security solutions, including internal segmentation firewalls (ISFWs), multifactor authentication (MFA), identity and access management (IAM), and next-generation endpoint security (see [Chapter 11](#)). A zero-trust approach to security management can only be successful if a means to validate and monitor user activities continuously is implemented. If a one-time validation mechanism is used, then the opportunity to abuse the system remains since threats, users, and connection characteristics are always subject to change. Thus, zero-trust networking can only work if real-time vetting and visibility into user activities are maintained.

A summary of zero trust is that all devices are segmented and isolated from each other to prevent any and all communications. Then, inter-device transactions must be authorized, authenticated, encrypted, monitored/analyzed, and logged.



In some situations, complete isolation may be needed instead of controlled and filtered interaction. This type of isolation is achieved using an air gap. An *air gap* is a network security measure employed to ensure that a secure system is physically isolated from other systems. Air gap implies that neither cabled nor wireless network links are available. Due to the proliferation of wireless connectivity options, a Faraday cage may be necessary to enforce air gap isolation (see [Chapter 8](#)).

To implement a zero-trust system, an organization must be capable of and willing to abandon some long-held assumptions about security. First and foremost, it must be understood that there is no such thing as a trusted source. No entity, asset, or subject—internal or external—is to be trusted by default. Instead, always assume attackers are already on the inside, on every system. From this new “no assumed trust” position, it is obvious that traditional default access controls are insufficient. Each and every subject, each and every time, needs to be authenticated, authorized, and encrypted. From there, a continuous real-time monitoring system should be established to look for violations and suspicious events. But even with zero trust integrated into the IT architecture, it is only an element of a holistic security strategy that is integrated into the entire organization's management processes.

Zero trust has been formalized in NIST SP 800-207, “Zero Trust Architecture.” Please consult this document to learn more about this revolution in security design.

Trust but Verify

The phrase “trust but verify” (a quote from a Russian proverb) was made famous by former president Ronald Reagan when discussing U.S. relations with the Soviet Union. However, our focus on this phrase is on its use in the security realm. A more

traditional security approach of trusting subjects and devices within the company's security perimeter (i.e., internal entities) automatically can be called “*trust but verify*.” This type of security approach leaves an organization vulnerable to insider attacks and grants intruders the ability to easily perform lateral movement among internal systems. Often, the trust but verify approach depends on an initial authentication process to gain access to the internal “secured” environment and then relies on generic access control methods. Due to the rapid growth and changes in the modern threatscape, the trust but verify model of security is no longer sufficient. Most security experts now recommend designing organizational security around the zero-trust model. So, in regard to the question of “zero trust or trust but verify?”, today's answer should only be zero trust.

Privacy by Design

Privacy by design (PbD) is a guideline to integrate privacy protections into products during the early design phase rather than attempting to tack it on at the end of development. It is effectively the same overall concept as “security by design” or “integrated security,” where security is to be an element of the design and architecture of a product starting at initiation and being maintained throughout the software development life cycle (SDLC).

As described in Ann Cavoukian's paper “Privacy by Design – The 7 Foundational Principles: Implementation and Mapping of Fair Information Practices,” the PbD framework is based on seven foundational principles:

- Proactive, not reactive; preventive, not remedial
- Privacy as the default
- Privacy embedded into design
- Full functionality – positive-sum, not zero-sum
- End-to-end life cycle protection
- Visibility and transparency

- Respect for user privacy

The goal of PbD is to have developers integrate privacy protections into their solutions to avoid privacy violations in the first place. The overall concept focuses on prevention rather than remedies for violations.

PbD is also the driving factor behind an initiative to have privacy protections integrated throughout an organization, not just by developers. Business operations and systems design can also integrate privacy protections into their core functions. This in turn, has led to the *Global Privacy Standard (GPS)*, which was crafted to create a single set of universal and harmonized privacy principles. GPS is to be adopted by countries to use as a guide in developing privacy legislation, used by organizations to integrate privacy protection into their operations, and used by developers to integrate privacy into the products they produce. There is some integration of a few of the principles of PbD in the EU's GDPR (see [Chapter 4](#), “Laws, Regulations, and Compliance”).

For more on PbD and GPS, please visit gpsbydesign.org, review the Cavoukian paper mentioned earlier, and read an additional paper, “Privacy by Design in Law, Policy and Practice.” Learn more about privacy in [Chapter 4](#) and about software development security in [Chapter 20](#).

Secure Access Service Edge (SASE)

Secure Access Service Edge (SASE) is a framework that combines network security functions with wide area network (WAN) capabilities, catering to the dynamic, secure access needs of modern organizations. SASE is designed to respond to the evolving IT landscape marked by trends such as cloud adoption, a mobile workforce, and an increased emphasis on network security.

SASE features a cloud-native architecture, unifying traditionally separate network and security services. This cloud-native approach allows organizations to deploy and manage their network and security services from the cloud, eliminating the

need for on-premises hardware and offering scalability and flexibility.

A core principle of SASE is identity-centric security, prioritizing the identity of users and devices over the traditional perimeter-based security model. This is particularly relevant in the current environment where users access resources from diverse locations and devices. This core principle is implemented through zero trust network access (ZTNA), which is founded on the concept that no entity, whether inside or outside the organization's network, should be trusted by default. Every user and device must undergo authentication (with MFA for users when possible) and authorization processes for access.

SASE also leverages edge computing, bringing security and networking closer to users and devices, reducing latency, and improving performance, especially for cloud-based applications. The framework emphasizes a globally distributed network, ensuring consistent security and performance for users regardless of their geographical location. SASE is often delivered as a service, allowing organizations to subscribe to the specific capabilities they require, simplifying management, reducing capital expenditures, and enabling scalability.

Continuous monitoring of user behavior and network conditions is a key aspect of SASE, enabling adaptive security measures that respond to changes in real time, contributing to an enhanced overall security posture.

Implementing SASE addresses the challenges posed by the modern IT landscape, providing a more agile and scalable approach to network and security services. It aligns well with the needs of a distributed and mobile workforce and accommodates the increasing reliance on cloud-based applications and resources.

Techniques for Ensuring CIA

To ensure the confidentiality, integrity, and availability (CIA) of data, you must ensure that all components that have access to data are secure and well behaved. Software designers use

different techniques to ensure that programs do only what is required and nothing more. Although the concepts we discuss in the following sections all relate to software programs, they are also commonly used in all areas of security. For example, physical confinement guarantees that all physical access to hardware is controlled.

Confinement

Software designers use process confinement to restrict the actions of a program. Simply put, process *confinement* allows a process to read from and write to only certain memory locations and resources. This is also known as *sandboxing*. It is the application of the principle of least privilege to processes. The goal of confinement is to prevent data leakage to unauthorized programs, users, or systems.

The operating system, or some other security component, disallows illegal read/write requests. If a process attempts to initiate an action beyond its granted authority, that action will be denied. In addition, further actions, such as logging the violation attempt, may be taken. Generally, the offending process is terminated. Confinement can be implemented in the operating system itself (such as through process isolation and memory protection), through the use of a confinement application or service (for example, Sandboxie at [sandboxie.com](https://www.sandboxie.com)), or through a virtualization or hypervisor solution (such as VMware or Oracle's VirtualBox).

Bounds

Each process that runs on a system is assigned an authority level. The authority level tells the operating system what the process can do. In simple systems, there may be only two authority levels: user and kernel. The authority level tells the operating system how to set the bounds for a process. The *bounds* of a process consist of limits set on the memory addresses and resources it can access. The bounds state the area within which a process is confined or contained. In most systems, these bounds

segment logical areas of memory for each process to use. It is the responsibility of the operating system to enforce these logical bounds and to disallow access to other processes. More secure systems may require physically bounded processes. Physical bounds require each bounded process to run in an area of memory that is physically separated from other bounded processes, not just logically bounded in the same memory space. Physically bounded memory can be very expensive, but it's also more secure than logical bounds. Bounds can be a means to enforce confinement.

Isolation

When a process is confined through enforcing access bounds, that process runs in isolation. Process isolation ensures that any behavior will affect only the memory and resources associated with the isolated process. *Isolation* is used to protect the operating environment, the kernel of the operating system, and other independent applications. Isolation is an essential component of a stable operating system. Isolation is what prevents an application from accessing the memory or resources of another application, whether for good or ill. Isolation allows for a fail-soft environment so that separate processes can operate normally or fail/crash without interfering or affecting other processes. Isolation is achieved through the enforcement of containment using bounds. Hardware and software isolation implementations are discussed throughout [Chapter 9](#).

These three concepts (confinement, bounds, and isolation) make designing secure programs and operating systems more difficult, but they also make it possible to implement more secure systems. Confinement is making sure that an active process can only access specific resources (such as memory). Bounds is the limitation of authorization assigned to a process to limit the resources the process can interact with and the types of interactions allowed. Isolation is the means by which confinement is implemented through the use of bounds. The goals of these concepts are to ensure that the predetermined scope of resource access is not violated and that any failure or

compromise of a process has minimal to no effect on any other process.

Access Controls

To ensure the security of a system, you need to allow subjects to access only authorized objects. Access controls limit the access of a subject to an object. Access rules state which objects are valid for each subject. Further, an object might be valid for one type of access and be invalid for another type of access. There are a wide range of options for access controls, such as discretionary, role-based, and mandatory. Please see [Chapter 14](#) for an in-depth discussion of access controls.

Trust and Assurance

A *trusted system* is one in which all protection mechanisms work together to process sensitive data for many types of users while maintaining a stable and secure computing environment. In other words, trust is the presence of a security mechanism, function, or capability. *Assurance* is the degree of confidence in the satisfaction of security needs. In other words, assurance is how reliable the security mechanisms are at providing security. Assurance must be continually maintained, updated, and reverified. This is true if the secured system experiences a known change (good or bad—i.e., a vendor patch or a malicious exploit) or if a significant amount of time has passed. In either case, change has occurred at some level. Change is often the antithesis of security; it often diminishes security. This is why change management, patch management, and configuration management are so important to security management.

Assurance varies from one system to another and often must be established on individual systems. However, there are grades or levels of assurance that can be placed across numerous systems of the same type, systems that support the same services, or systems that are deployed in the same geographic location. Thus, trust can be built into a system by implementing specific security

features, whereas assurance is an assessment of the reliability and usability of those security features in a real-world situation.

Understand the Fundamental Concepts of Security Models

In information security, models provide a way to formalize security policies. Such models can be abstract or intuitive, but all are intended to provide an explicit set of rules that a computer can follow to implement the fundamental security concepts, processes, and procedures of a security policy. A *security model* provides a way for designers to map abstract statements into a security policy that prescribes the algorithms and data structures necessary to build hardware and software. Thus, a security model gives software designers something against which to measure their design and implementation.

Tokens, Capabilities, and Labels

Several different methods are used to describe the necessary security attributes for an object. A security *token* is a separate object that is associated with a resource and describes its security attributes. This token can communicate security information about an object prior to requesting access to the actual object. In other implementations, various lists are used to store security information about multiple objects. A *capabilities list* maintains a row of security attributes for each controlled object. Although not as flexible as the token approach, a capabilities list generally offers quicker lookups when a subject requests access to an object. A third common type of attribute storage is called a *security label*, which is generally a permanent part of the object to which it's attached. Once a security label is set, it usually cannot be altered. This permanence provides another safeguard against tampering that neither tokens nor capabilities lists provide.

You'll explore several security models in the following sections; all of them can shed light on how security enters into computer architectures and operating system design:

- Trusted computing base
- State machine model
- Information flow model
- Noninterference model
- Take-grant model
- Access control matrix
- Bell–LaPadula model
- Biba model
- Clark–Wilson model
- Brewer and Nash model

Other Security Models

There are several more security models you can learn about if you formally study computer security, systems design, or application development. Some of those include the object-capability model, Lipner's Model, the Boebert and Kain Integrity model, the two-compartment exchange (Kärger) model, Gong's JDK Security Model, the Lee–Shockley model, the Jueneman model, and more.

Trusted Computing Base

The *trusted computing base (TCB)* design principle is the combination of hardware, software, and controls that work together to form a trusted base to enforce your security policy. The TCB is a subset of a complete information system. It should be as small as possible so that a detailed analysis can reasonably ensure that the system meets design specifications and

requirements. The TCB is the only portion of that system that can be trusted to adhere to and enforce the security policy. It is the responsibility of TCB components to ensure that a system behaves properly in all cases and that it adheres to the security policy under all circumstances.

Security Perimeter

The *security perimeter* of your system is an imaginary boundary that separates the TCB from the rest of the system ([Figure 8.2](#)). This boundary ensures that no insecure communications or interactions occur between the TCB and the remaining elements of the computer system. For the TCB to communicate with the rest of the system, it must create secure channels, also called *trusted paths*. A trusted path is a channel established with strict standards to allow necessary communication to occur without exposing the TCB to security exploitations.

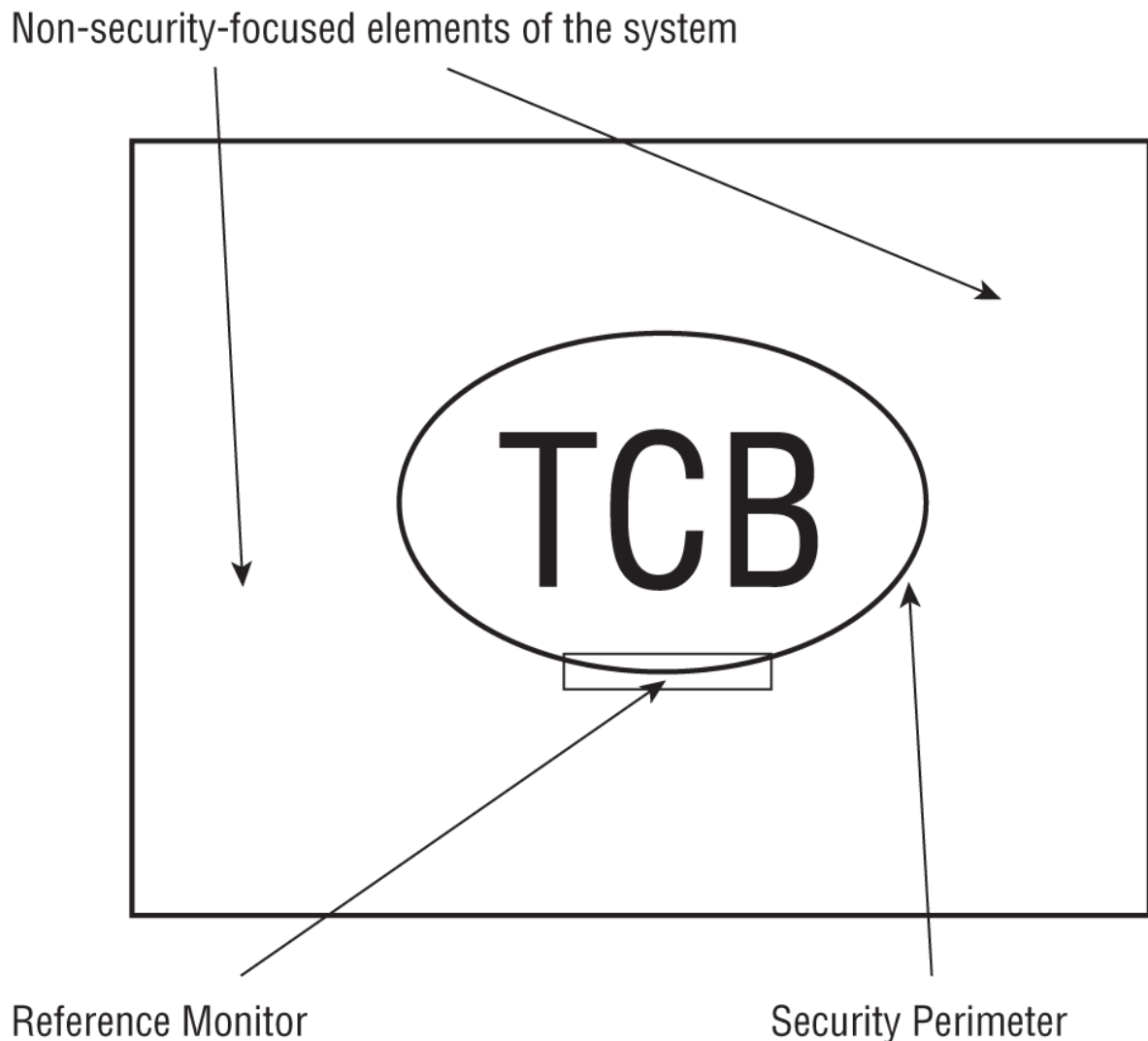


FIGURE 8.2 The TCB, security perimeter, and reference monitor

Reference Monitors and Kernels

The part of the TCB that validates access to every resource prior to granting access requests is called the *reference monitor* ([Figure 8.2](#)). The reference monitor stands between every subject and object, verifying that a requesting subject's credentials meet the object's access requirements before any requests are allowed to proceed. Effectively, the reference monitor is the access control enforcer for the TCB. The reference monitor enforces access control or authorization based on the desired security model, whether discretionary, mandatory, role-based, or some other form of access control.

The collection of components in the TCB that work together to implement reference monitor functions is called the *security kernel*. The reference monitor is a concept or theory that is put into practice via the implementation of a security kernel in software and hardware. The purpose of the security kernel is to launch appropriate components to enforce reference monitor functionality and resist all known attacks. The security kernel mediates all resource access requests, granting only those requests that match the appropriate access rules in use for a system.

State Machine Model

The *state machine model* describes a system that is always secure no matter what state it is in. It's based on the computer science definition of a *finite state machine (FSM)*. An FSM combines an external input with an internal machine state to model all kinds of complex systems, including parsers, decoders, and interpreters. Given an input and a state, an FSM transitions to another state and may create an output. Mathematically, the next state is a function of the current state and the input next state—that is, the next state = $F(\text{input}, \text{current state})$. Likewise, the output is also a function of the input and the current state output—that is, the output = $F(\text{input}, \text{current state})$.

According to the state machine model, a *state* is a snapshot of a system at a specific moment in time. If all aspects of a state meet the requirements of the security policy, that state is considered secure. A transition occurs when accepting input or producing output. A transition always results in a new state (also called a *state transition*). All state transitions must be evaluated. If each possible state transition results in another secure state, the system can be called a *secure state machine*. A secure state machine model system always boots into a secure state, maintains a secure state across all transitions, and allows subjects to access resources only in a secure manner compliant with the security policy. The secure state machine model is the basis for many other security models.

Information Flow Model

The *information flow model* focuses on controlling the flow of information. Information flow models are based on the state machine model. Information flow models don't necessarily deal with only the direction of information flow; they can also address the type of flow.

Information flow models are designed to prevent unauthorized, insecure, or restricted information flow, often between different levels of security (known as multilevel models). Information flow can be between subjects and objects at the same or different classification levels. An information flow model allows all authorized information flows, and prevents all unauthorized information flows.

Another interesting perspective on the information flow model is that it is used to establish a relationship between two versions or states of the same object when those two versions or states exist at different points in time. Thus, information flow dictates the transformation of an object from one state at one point in time to another state at another point in time. The information flow model also addresses covert channels by specifically excluding all undefined flow pathways.

Noninterference Model

The *noninterference model* is loosely based on the information flow model. However, instead of being concerned about the flow of information, the noninterference model is concerned with how the actions of a subject at a higher security level affect the system state or the actions of a subject at a lower security level. Basically, the actions of subject A (high) should not affect or interfere with the actions of subject B (low) or even be noticed by subject B. If such violations occur, subject B may be placed into an insecure state or be able to deduce or infer information about a higher level of classification. This is a type of information leakage and implicitly creates a covert channel. Thus, the noninterference model can be imposed to provide a form of protection against

damage caused by malicious programs, such as Trojan horses, backdoors, and rootkits.

Composition Theories

Some other models that fall into the information flow category build on the notion of inputs and outputs between multiple systems. These are called composition theories because they explain how outputs from one system relate to inputs to another system. There are three composition theories:

- **Cascading:** Input for one system comes from the output of another system.
- **Feedback:** One system provides input to another system, which reciprocates by reversing those roles (so that system A first provides input to system B and then system B provides input to system A).
- **Hookup:** One system sends input to another system but also sends input to external entities.

Take-Grant Model

The *take-grant model* employs a *directed graph* ([Figure 8.3](#)) to dictate how rights can be passed from one subject to another or from a subject to an object. Simply put, a subject (X) with the grant right can grant another subject (Y) or another object (Z) any right that subject (X) possesses. Likewise, a subject (X) with the take right can take a right from another subject (Y). In addition to these two primary rules, the take-grant model has a create rule and a remove rule to generate or delete rights. The key to this model is that using these rules allows you to figure out when rights in the system can change and where leakage (that is, unintentional distribution of permissions) can occur.

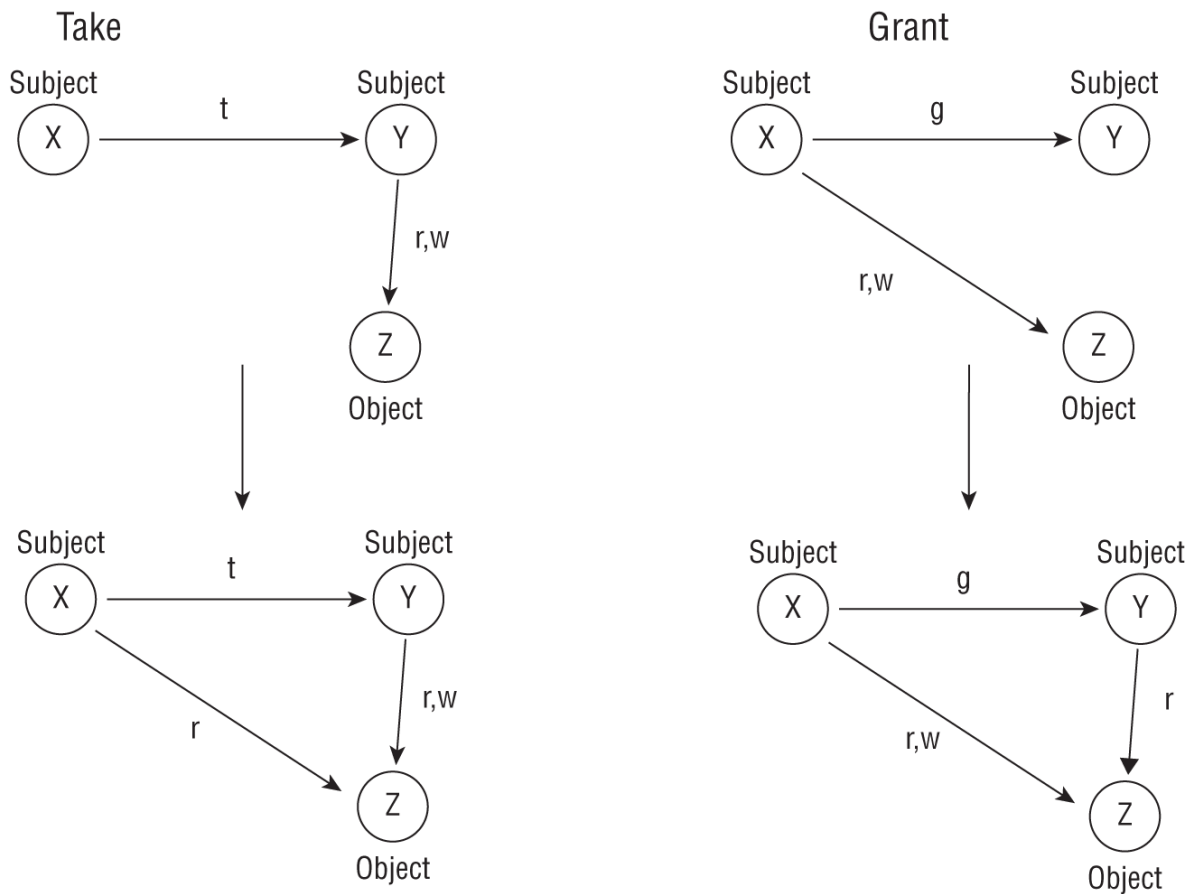


FIGURE 8.3 The take-grant model's directed graph

In essence, here are the four rules of the take-grant model:

- *Take rule:* Allows a subject to take rights over an object
- *Grant rule:* Allows a subject to grant rights to an object
- *Create rule:* Allows a subject to create new rights
- *Remove rule:* Allows a subject to remove rights it has

It is interesting to ponder that the take and grant rules are effectively a copy function. This can be recognized in modern operating systems in the process of inheritance, such as subjects inheriting a permission from a group or a file inheriting access control list (ACL) values from a parent folder. The two additional rules (create and remove), not defined by a directed graph, are also commonly present in modern operating systems. For example, to obtain permission on an object, that permission does not have to be copied from a user account that already has that permission; instead, it is simply created by an account with the

privilege capability of create or assign permissions (which can be the owner of an object or a subject with full control or administrative privileges over the object).

Access Control Matrix

An *access control matrix* is a table of subjects and objects that indicates the actions or functions that each subject can perform on each object. Each column of the matrix is an ACL pulled from objects. Once sorted, each row of the matrix is a capabilities list for each listed subject. An ACL is tied to an object; it lists the valid actions each subject can perform. A capability list is tied to the subject; it lists valid actions that can be taken on each object included in the matrix.

From an administration perspective, using only capability lists for access control is a management nightmare. A capability list method of access control can be accomplished by storing on each subject a list of rights the subject has for every object. This effectively gives each user a key ring of access and rights to objects within the security domain. To remove access to a particular object, every user (subject) that has access to it must be individually manipulated. Thus, managing access on each user account is much more difficult than managing access on each object (in other words, via ACLs). A capabilities table can be created by pivoting an access control matrix; this results in the columns being subjects and the rows being ACLs from objects.

The access control matrix shown in [Table 8.3](#) is for a discretionary access control system. A mandatory or role-based matrix can be constructed simply by replacing the subject names with classifications or roles. Access control matrices are used by systems to quickly determine whether the requested action by a subject for an object is authorized.

TABLE 8.3 An access control matrix

Subjects	Document file	Printer	Network folder share
Bob	Read	No Access	No Access
Mary	No Access	No Access	Read
Amanda	Read, Write	Print	No Access
Mark	Read, Write	Print	Read, Write
Kathryn	Read, Write	Print, Manage Print Queue	Read, Write, Execute
Colin	Read, Write, Change Permissions	Print, Manage Print Queue, Change Permissions	Read, Write, Execute, Change Permissions

Bell–LaPadula Model

The *Bell–LaPadula model* was developed for the U.S. Department of Defense (DoD) in the 1970s based on the DoD's multilevel security policies. The multilevel security policy states that a subject with any level of clearance can access resources at or below its clearance level. However, within clearance levels, access to compartmentalized objects is granted only on a need-to-know basis.

By design, the Bell–LaPadula model prevents the leaking or transfer of classified information to less secure clearance levels. This is accomplished by blocking lower-classified subjects from accessing higher-classified objects. With these restrictions, the Bell–LaPadula model is focused on maintaining confidentiality and does not address any other aspects of object security.

Lattice-Based Access Control

This general category for nondiscretionary access controls is covered in [Chapter 13](#), “Managing Identity and Authentication.” Here's a quick preview on that more detailed coverage of this subject (which drives the underpinnings for most access control security models): Subjects under *lattice-based access controls* are assigned positions in a lattice (i.e., a multilayered security structure or multileveled security domains). Subjects can access only those objects that fall into the range between the least upper bound (LUB) (the nearest security label or classification higher than their lattice position) and the greatest (i.e., highest) lower bound (GLB) (the nearest security label or classification lower than their lattice position) of the labels or classifications for their lattice position.

This model is built on a state machine concept and the information flow model. It also employs mandatory access controls and is a lattice-based access control concept. The *lattice* tiers are the *classification levels* defined by the organization's security policy.

There are three basic properties of this state machine:

- The *Simple Security Property* (i.e., the ss-Property) states that a subject may not read information at a higher sensitivity level (no read-up).
- The **-Property (star-property)* states that a subject may not write information to an object at a lower sensitivity level (no write-down).
- The *Discretionary Security Property* states that the system uses an access matrix to enforce discretionary access control.

These first two properties define the states into which the system can transition. No other transitions are allowed. All states accessible through these two rules are secure states. Thus, Bell–

LaPadula–modeled systems offer state machine model security (see [Figure 8.4](#)).

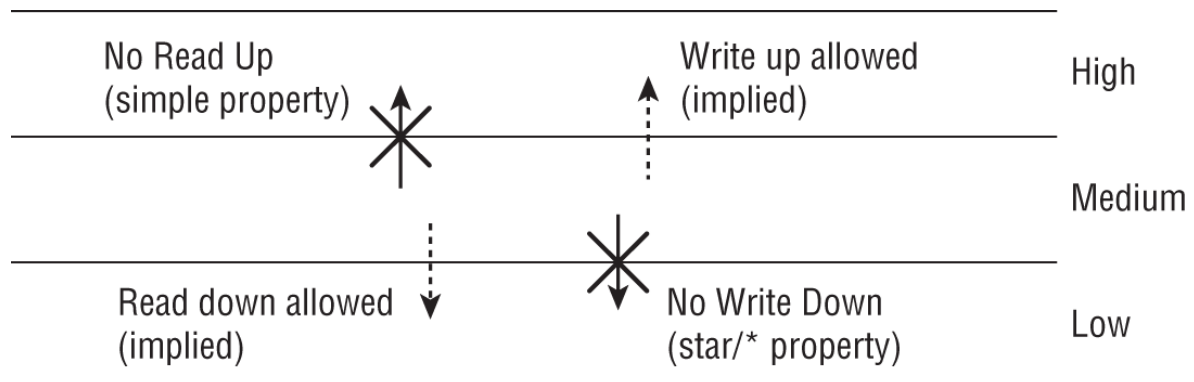


FIGURE 8.4 The Bell–LaPadula model

The Bell–LaPadula properties are in place to protect data confidentiality. A subject cannot read an object that is classified at a higher level than the subject is cleared for. Because objects at one level have data that is more sensitive or secret than data in objects at a lower level, a subject (who is not a trusted subject) cannot write data from one level to an object at a lower level. That action would be similar to pasting a top-secret memo into an unclassified document file. The third property enforces a subject's job/role-based need to know to access an object.



An exception in the Bell–LaPadula model states that a “trusted subject” is not constrained by the *-Property. A trusted subject is defined as “a subject that is guaranteed not to consummate a security-breaching information transfer even if it is possible.” This means that a trusted subject is allowed to violate the *-Property and perform a write-down, which is necessary when performing valid object declassification or reclassification.

The Bell–LaPadula model was designed in the 1970s, so it does not support many operations that are common today, such as file sharing and networking. It also assumes secure transitions between security layers and does not address covert channels (see [Chapter 9](#)).

Biba Model

The *Biba model* was designed after the Bell–LaPadula model, but it focuses on integrity. The Biba model is also built on a state machine concept, is based on information flow, and is a multilevel model. In fact, the Biba model is the inverted Bell–LaPadula model. The properties of the Biba model are as follows:

- The *Simple Integrity Property* states that a subject cannot read an object at a lower integrity level (no read-down).
- The ** (star) Integrity Property* states that a subject cannot modify an object at a higher integrity level (no write-up).
- The *Invocation Property* states that a process from below cannot request higher access (neither read nor write); only with subjects at an equal or lower level.



In both the Biba and Bell–LaPadula models, there are two properties that are inverses of each other: simple and * (star). However, they may also be labeled as axioms, principles, or rules. What you should focus on is the *simple* and *star* designations. Take note that *simple* is always about reading, and *star* is always about writing. In both cases, the rules define what cannot or should not be done. Usually, what is not prevented or blocked is allowed. Thus, even though a rule is stated as a No declaration, its opposite direction is implied as allowed.

[Figure 8.5](#) illustrates these Biba model properties.

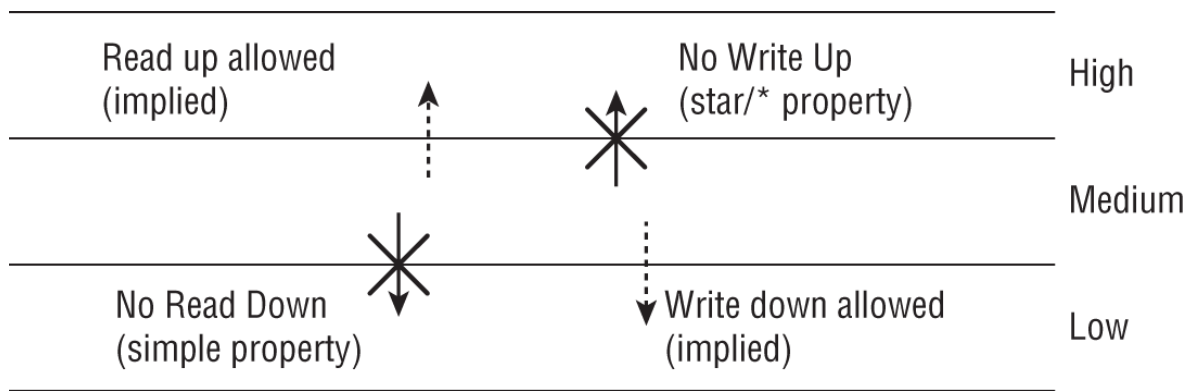
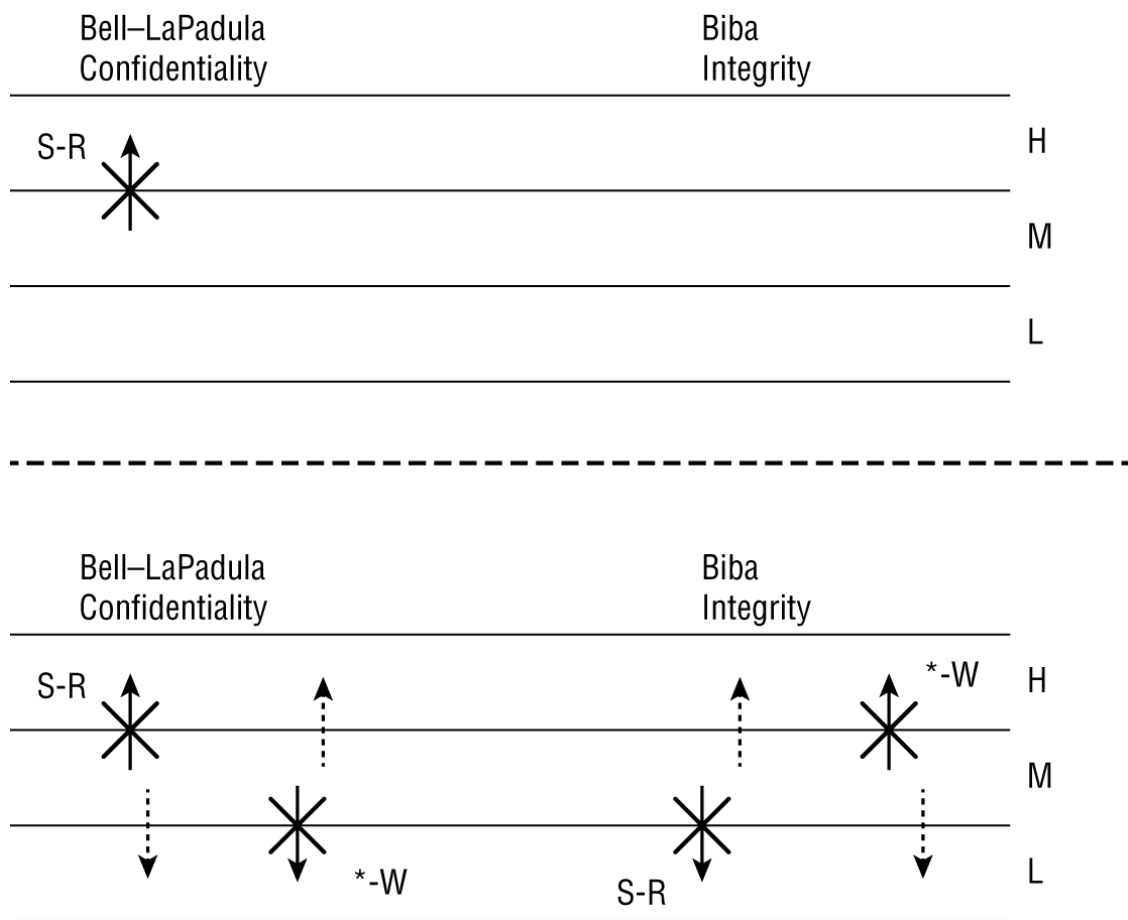


FIGURE 8.5 The Biba model

Consider the Biba properties. The second property of the Biba model is pretty straightforward. A subject cannot write to an object at a higher integrity level. That makes sense. What about the first property? Why can't a subject read an object at a lower integrity level? The answer takes a little thought. Think of integrity levels as being like the purity level of air. You would not want to pump air from the smoking section into the clean room environment. The same applies to data. When integrity is important, you do not want unvalidated data read into validated documents. The potential for data contamination is too great to permit such access.



Memorizing the properties of Bell–LaPadula and Biba can be challenging, but there is a shortcut. If you can memorize the graphical layout in the following figure above the dotted line, then you can figure out the rest. Notice that Bell–LaPadula is placed on the left and Biba is on the right, and the security benefit of each is listed below the model name. Then, only the Bell–LaPadula model's simple property is listed. That property is “No Read Up,” which is represented by an arrow pointing upward that is crossed out and labeled by an “S” for simple and an “R” for read. From there, all of the other rules are the opposing element of the pair or inverted.



Biba requires that all subjects and objects have a classification label (it is still a DoD-derived security model). Thus, data integrity protection is dependent on data classification.

Critiques of the Biba model reveal a few drawbacks:

- It addresses only integrity, not confidentiality or availability.
- It focuses on protecting objects from external threats; it assumes that internal threats are handled programmatically.
- It does not address access control management, and it doesn't provide a way to assign or change an object's or subject's classification level.
- It does not prevent covert channels.

Clark–Wilson Model

The *Clark–Wilson model* uses a multifaceted approach to enforcing data integrity. Instead of defining a formal state machine, the Clark–Wilson model defines each data item and allows modifications through only a limited or controlled intermediary program or interface.

The Clark–Wilson model does not require the use of a lattice structure; rather, it uses a three-part relationship of subject/program/object (or subject/transaction/object) known as a triple or an *access control triplet*. Subjects do not have direct access to objects. Objects can be accessed only through programs. Through the use of two principles—well-formed transactions and separation of duties—the Clark–Wilson model provides an effective means to protect integrity.

Well-formed transactions take the form of programs. A subject is able to access objects only by using a program, interface, or access portal ([Figure 8.6](#)). Each program has specific limitations on what it can and cannot do to an object (such as a database or other resource). This effectively limits the subject's capabilities. This is known as a constrained, limiting, or restrictive interface. If the programs are properly designed, then the triple relationship provides a means to protect the integrity of the object.

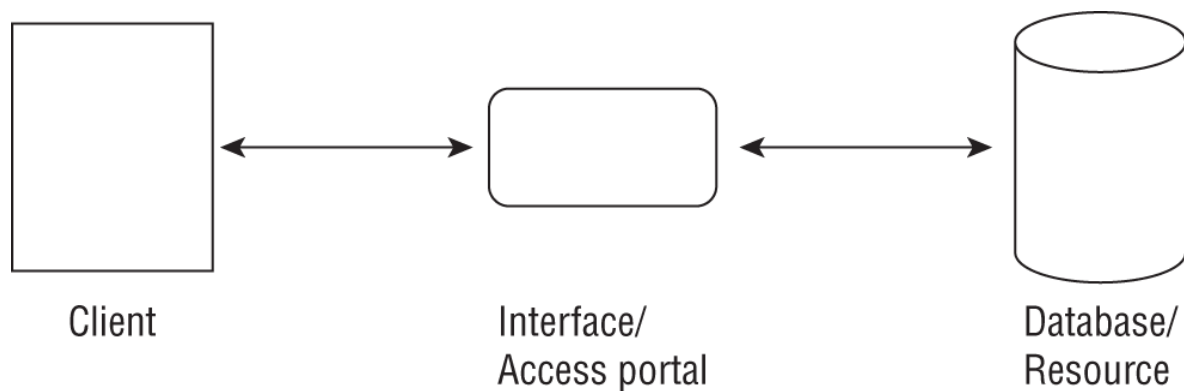


FIGURE 8.6 The Clark–Wilson model

Clark–Wilson defines the following items and procedures:

- A *constrained data item (CDI)* is any data item whose integrity is protected by the security model.
- An *unconstrained data item (UDI)* is any data item that is not controlled by the security model. Any data that is to be input and hasn't been validated, or any output, would be considered an unconstrained data item.
- An *integrity verification procedure (IVP)* is a procedure that scans data items and confirms their integrity.
- *Transformation procedures (TPs)* are the only procedures that are allowed to modify a CDI. The limited access to CDIs through TPs forms the backbone of the Clark–Wilson integrity model.

The Clark–Wilson model uses security labels to grant access to objects, but only through transformation procedures and a *restricted interface model*. A restricted interface model uses classification-based restrictions to offer only subject-specific authorized information and functions. One subject at one classification level will see one set of data and have access to one set of functions, whereas another subject at a different classification level will see a different set of data and have access to a different set of functions. The different functions made available to different levels or classes of users may be implemented by either showing all functions to all users but deactivating those that are not authorized for a specific user or by

showing only those functions granted to a specific user. Through these mechanisms, the Clark–Wilson model ensures that data is protected from unauthorized changes from any user. In effect, the Clark–Wilson model enforces separation of duties. The Clark–Wilson design makes it a common model for commercial applications.



The Clark–Wilson model was designed to protect integrity using the access control triplet. However, though the intermediary interface can be programmed to limit what can be done to an object by a subject, it can just as easily be programmed to limit or restrict what objects are shown to a subject. Thus, this concept can lend itself readily to protect confidentiality. In many situations, there is an intermediary program between a subject and an object. If the focus of that intermediary is to protect integrity, then it is an implementation of the Clark–Wilson model. If it is intended to protect confidentiality, then they are benefiting from an alternate use of the intermediary program.

Brewer and Nash Model

The *Brewer and Nash model* was created to permit access controls to change dynamically based on a user's previous activity (making it a kind of state machine model as well). This model applies to a single integrated database; it seeks to create security domains that are sensitive to the notion of conflict of interest (for example, someone who works at company C who has access to proprietary data for company A should not also be allowed access to similar data for company B if those two companies compete with each other). This model creates a class of data that defines which security domains are potentially in conflict and prevents any subject with access to one domain that belongs to a specific conflict class from accessing any other domain that belongs to the same conflict class. Metaphorically, this puts a wall around all other information in any conflict class. Thus, this model also uses

the principle of data isolation within each conflict class to keep users out of potential conflict-of-interest situations (for example, management of company datasets). Because company relationships change all the time, dynamic updates to members of and definitions for conflict classes are important.

Another way of looking at or thinking of the Brewer and Nash model is of an administrator having full control access to a wide range of data in a system based on their assigned job responsibilities and work tasks. However, at the moment an action is taken against any data item, the administrator's access to any conflicting data items is temporarily blocked. Only data items that relate to the initial data item can be accessed during the operation. Once the task is completed, the administrator's access returns to full control.

The Brewer and Nash model was sometimes known as the Chinese Wall model, but this term is deprecated. Instead, other terms of “ethical wall” and “cone of silence” have been used to describe Brewer and Nash.

Disambiguating the Word “Star” in Models

The term *star* presents a few challenges when it comes to security models. For one thing, there is no formal security model named “Star Model.” However, both the Bell–LaPadula and the Biba models have a *star property*, which is discussed in their respective sections in this chapter.

Although not a model, the Cloud Security Alliance (CSA) also has a STAR program. CSA's Security Trust Assurance and Risk (STAR) program focuses on improving cloud service provider (CSP) security through auditing, transparency, and integration of standards.

Although not related to security, there is also Galbraith's Star Model, which helps businesses organize divisions and departments to achieve business missions and goals and adjust over time for long-term viability. This model is based on five main areas of business administration that need to be managed, balanced, and harnessed toward the mission and goals of the organization. The five areas of Galbraith's Star Model are Strategy, Structure, Processes, Rewards, and People.

Understanding how “star” is used in the context of the Bell–LaPadula and Biba models, CSA's STAR program, and Galbraith's Star Model will help you distinguish what is meant when you see the word used in different contexts.

Select Controls Based on Systems Security Requirements

Those who purchase information systems for certain kinds of applications—for example, national security agencies whose sensitive information may be extremely valuable (or dangerous in the wrong hands) or central banks or securities traders that have certain data that may be worth billions of dollars—often want to understand the security strengths and weaknesses of

systems prior to acquisition. Such buyers are often willing to consider only systems that have been subjected to formal evaluation processes in advance and have received some kind of security rating.

Often, trusted third parties are used to perform security evaluations; the most important result from such testing is their “seal of approval” that the system meets all essential criteria.

Common Criteria

The *Common Criteria (CC)* defines various levels of testing and confirmation of systems' security capabilities. Nevertheless, it's wise to observe that even the highest CC ratings do not equate to a guarantee that such systems are completely secure or that they are entirely devoid of vulnerabilities or susceptibilities to exploit. The Common Criteria was designed as a dynamic subjective product evaluation model and replaced previous static systems, such as the U.S. Department of Defense's Trusted Computer System Evaluation Criteria (TCSEC) and the EU's Information Technology Security Evaluation Criteria (ITSEC).

A document titled “Arrangement on the Recognition of Common Criteria Certificates in the Field of Information Technology Security” was signed by representatives from government organizations in Canada, France, Germany, the United Kingdom, and the United States in 1998, making the document an international standard. Since then, 23 additional countries have signed the arrangement. The arrangement documentation was formally adopted as a standard and published as ISO/IEC 15408:2022 and labeled as “Information security, cybersecurity and privacy protection: Evaluation criteria for IT security.”



The latest versions of ISO/IEC standards are available at standards.iso.org/ittf/PubliclyAvailableStandards/index.html. The current (as of this writing) version of the Common Criteria is to be replaced with ISO/IEC WD 15408-1 (www.iso.org/standard/88134.html). This could occur in 2024.

The objectives of the CC guidelines are as follows:

- To add to buyers' confidence in the security of evaluated, rated IT products
- To eliminate duplicate evaluations (among other things, this means that if one country, agency, or validation organization follows the CC in rating specific systems and configurations, others elsewhere need not repeat this work)
- To keep making security evaluations more cost-effective and efficient
- To make sure evaluations of IT products adhere to high and consistent standards
- To promote evaluation and increase the availability of evaluated, rated IT products
- To evaluate the functionality (what the system does) and assurance (how much can you trust the system) of the target of evaluation (TOE)

The Common Criteria process is based on two key elements: protection profiles and security targets. *Protection profiles (PPs)* specify the security requirements and protections for a product that is to be evaluated (the TOE), which are considered the security desires, or the “I want” of a customer. *Security targets (STs)* specify the claims of security from the vendor that are built into a TOE. STs are considered the implemented security measures, or the “I will provide” from the vendor. In addition to

offering security targets, vendors may offer packages of additional security features. A package is an intermediate grouping of security requirement components that can be added to or removed from a TOE (like the option packages when purchasing a new vehicle). This system of the PP and ST allows for flexibility, subjectivity, and customization of an organization's specific security functional and assurance requirements over time.

An organization's PP is compared to various STs from the selected vendor's TOEs. The closest or best match is what the client purchases. The client initially selects a vendor based on published or marketed *evaluation assurance levels (EALs)* for currently available systems. Using Common Criteria to choose a vendor allows clients to request exactly what they need for security rather than having to use static fixed security levels. It also allows vendors more flexibility on what they design and create. A well-defined set of Common Criteria supports subjectivity and versatility, and it automatically adapts to changing technology and threat conditions. Furthermore, the EALs provide a method for comparing vendor systems that is more standardized (like the old TCSEC).

[Table 8.4](#) summarizes EALs 1 through 7. For a complete description of EALs, consult the CC standard documents.

TABLE 8.4 Common Criteria evaluation assurance levels

Level	Assurance level	Description
EAL1	Functionally tested	Applies when some confidence in correct operation is required but where threats to security are not serious. This is of value when independent assurance that due care has been exercised in protecting personal information is necessary.
EAL2	Structurally tested	Applies when delivery of design information and test results are in keeping with good commercial practices. This is of value when developers or users require low to moderate levels of independently assured security. It is especially relevant when evaluating legacy systems.
EAL3	Methodically tested and checked	Applies when security engineering begins at the design stage and is carried through without substantial subsequent alteration. This is of value when developers or users require a moderate level of independently assured security, including thorough investigation of TOE and its development.
EAL4	Methodically designed, tested and reviewed	Applies when rigorous, positive security engineering and good commercial development practices are used. This does not require substantial specialist knowledge, skills, or resources. It involves independent testing of all TOE security functions.
EAL5	Semi-formally verified designed and tested	Uses rigorous security engineering and commercial development practices, including specialist security engineering techniques. This applies when developers

Level	Assurance level	Description
		or users require a high level of independently assured security in a planned development approach, followed by rigorous development.
EAL6	Semi-formally verified design and tested	Uses the application of high assurance security engineering techniques to a rigorous development environment in order to produce a premium TOE for protecting high-value assets against significant risks. It is therefore applicable to the development of security TOEs for application in high risk situations where the value of the protected assets justifies the additional costs.
EAL7	Formally verified design and tested	Used only for highest-risk situations or where high-value assets are involved. This is limited to TOEs where tightly focused security functionality is subject to extensive formal analysis and testing.

Though the CC guidelines are flexible and accommodating enough to capture most security needs and requirements, they are by no means perfect. As with other evaluation criteria, the CC guidelines do nothing to make sure that how users act on data is also secure. The CC guidelines also do not address administrative issues outside the specific purview of security. As with other evaluation criteria, the CC guidelines do not include evaluation of security *in situ*—that is, they do not address controls related to personnel, organizational practices and procedures, or physical security. Likewise, controls over electromagnetic emissions are not addressed, nor are the criteria for rating the strength of cryptographic algorithms explicitly laid out. Nevertheless, the CC guidelines represent some of the best techniques whereby systems may be rated for security.



Additional Common Criteria documentation is available at commoncriteriaportal.org. Visit this site to get information on the current version of the CC guidelines and guidance on using the CC along with lots of other useful, relevant information.

Authorization to Operate

For many environments, it is necessary to obtain an official approval to use secured equipment for operational objectives. This is often referred to as an *authorization to operate (ATO)*. ATO is the current term for this concept as defined by the Risk Management Framework (RMF) (see [Chapter 2](#), “Personnel Security and Risk Management Concepts”), which replaces the previous term of accreditation. An ATO is an official authorization to use a specific collection of secured IT/IS systems to perform business tasks and accept the identified risk. The assessment and assignment of an ATO is performed by an *authorizing official (AO)*. An AO is an authorized entity who can evaluate an IT/IS system, its operations, and its risks, and potentially issue an ATO.



NIST maintains an excellent glossary with references at csrc.nist.gov/glossary.

A typical ATO is issued for 3 years (although assigned time frames vary and the AO can adjust the time frame even after issuing an ATO) and must be reobtained whenever one of the following conditions occurs:

- The ATO time frame has expired.
- The system experiences a significant security breach.
- The system experiences a significant security change.

The AO has the discretion to determine which breaches or security changes result in a loss of ATO. Either a modest intrusion event or the application of a substantial security patch could cause the negation of an ATO.

An AO can issue four types of authorization decisions:

Authorization to Operate This decision is issued when risk is managed to an acceptable level.

Common Control Authorization This decision is issued when a security control is inherited from another provider and when the risk associated with the common control is at an acceptable level and already has a ATO from the same AO.

Authorization to Use This decision is issued when a third-party provider (such as a cloud service) provides IT/IS servers that are deemed to have risk at an acceptable level; it is also used to allow for reciprocity in accepting another AO's ATO.

Denial of Authorization This decision is issued when risk is unacceptable.

Please see NIST SP 800-37r2 for more on the Risk Management Framework and authorization.

Understand Security Capabilities of Information Systems

The security capabilities of information systems include memory protection, virtualization, Trusted Platform Module (TPM), encryption/decryption, interfaces, and fault tolerance. It is important to carefully assess each aspect of the infrastructure to ensure that it sufficiently supports security. Without an understanding of the security capabilities of information systems, it is impossible to evaluate them or implement them properly.

Memory Protection

Memory protection is a core security component that must be designed and implemented into an operating system. It must be enforced regardless of the programs executing in the system. Otherwise, instability, violation of integrity, denial of service, and disclosure are likely results. Memory protection is used to prevent an active process from interacting with an area of memory that was not specifically assigned or allocated to it.

Memory protection is discussed throughout [Chapter 9](#) in relation to the topics of isolation, virtual memory, segmentation, memory management, and protection rings, as well as protections against buffer (i.e., memory) overflows.

Virtualization

Virtualization technology is used to host one or more operating systems within the memory of a single host computer or to run applications that are not compatible with the host OS.

Virtualization can be a tool to isolate operating systems, test suspicious software, or implement other security protections. See [Chapter 9](#) for more information about virtualization.

Trusted Platform Module (TPM)

The Trusted Platform Module (TPM) is both a specification for a cryptoprocessor chip on a mainboard and the general name for the implementation of the specification. A TPM can be used to implement a broad range of cryptography-based security protection mechanisms. A TPM chip is often used to store and process cryptographic keys for a hardware-supported or OS-implemented local storage device encryption system. A TPM is an example of a hardware security module (HSM). An HSM is a cryptoprocessor used to manage and store digital encryption keys, accelerate crypto operations, support faster digital signatures, and improve authentication. An HSM can be a chip on a motherboard, an external peripheral, a network-attached device, or an extension card (which is inserted into a device, such as a router, firewall, or rack-mounted server blade). HSMs include tamper

protection to prevent their misuse even if an attacker gains physical access.

Interfaces

A *constrained* or *restricted interface* is implemented within an application to restrict what users can do or see based on their privileges. Users with full privileges have access to all the capabilities of the application. Users with restricted privileges have limited access.

Applications constrain the interface using different methods. A common method is to hide the capability if the user doesn't have permission to use it. Commands might be available to administrators via a menu or by right-clicking an item, but if a regular user doesn't have permissions, the command does not appear. Other times, the command is shown but is dimmed or deactivated. The regular user can see it but will not be able to use it.

The purpose of a constrained interface is to limit or restrict the actions of both authorized and unauthorized users. The use of such an interface is a practical implementation of the Clark–Wilson model of security.

Fault Tolerance

Fault tolerance is the ability of a system to suffer a fault but continue to operate. Fault tolerance is achieved by adding redundant components such as additional disks within a redundant array of independent disks (RAID) (aka redundant array of inexpensive disks [RAID]) array, or additional servers within a failover clustered configuration. Fault tolerance is an essential element of security design. It is also considered part of avoiding single points of failure and the implementation of redundancy. For more details on fault tolerance, redundant servers, RAID, and failover solutions, see [Chapter 18](#), “Disaster Recovery Planning.”

Encryption/Decryption

Encryption is the process of converting plaintext to ciphertext, whereas decryption reverses that process. Symmetric and asymmetric methods of encryption and decryption can be used to support a wide range of security solutions to protect confidentiality and integrity. Please see the full coverage of cryptography in [Chapters 6](#) and [7](#).

Manage the Information System Life Cycle

Managing the information system life cycle is a comprehensive process that involves various stages, each with specific activities and considerations. Managing the information system life cycle involves a structured and organized approach to developing, deploying, and maintaining an information system.

The typical stages of the information system life cycle are:

Stakeholders' Needs and Requirements This initial phase focuses on identifying and understanding the needs, expectations, and requirements of stakeholders who will interact with the information system. It encompasses a thorough analysis of the diverse set of requirements presented by end users, managers, regulatory bodies, and other relevant parties.

Requirements Analysis This phase involves a detailed examination of these requirements, including determining both functional and nonfunctional requirements, considering constraints, and ensuring alignment with the overall goals of the organization.

Architectural Design In this phase, a blueprint for the information system is created, defining the overall structure, components, modules, data flow, and interfaces of the system.

Development/Implementation This phase is where the actual coding and development of the information system take place. Developers work on creating the software,

configuring hardware, and integrating various components to bring the system to life.

Integration This phase is the process of combining different modules or components of the system to ensure they work together seamlessly. The goal here is to ensure that the individual elements of the system function as a unified whole.

Verification and Validation This phase is focused on confirming that the developed system meets the specified requirements. Verification ensures that each component of the system is correctly implemented, whereas validation ensures that the system as a whole fulfills its intended purpose.

Transition/Deployment This phase is when the system is deployed for actual use. Transition involves the migration of the system from the development environment to the operational environment, making it available to end users.

Operations and Maintenance/Sustainment In this phase, the system is actively used in the operational environment. Operations involve day-to-day management, monitoring, and support, while maintenance ensures that the system continues to function correctly by addressing issues, applying updates, and making improvements. Sustainment refers to the ongoing focus on maintaining and supporting the operational functionality of an information system over an extended period.

Retirement/Disposal Eventually, the information system reaches the end of its life cycle. The retirement or disposal phase involves decommissioning the system in an organized manner, considering data disposal, ensuring compliance with regulations, and making decisions about the future of the system or its replacement.

Effectively managing the information system life cycle requires collaboration among different stakeholders, adherence to best

practices, and continuous improvement to meet evolving needs and technological advancements.

Summary

Secure systems are not just assembled; they are designed to support security. Systems that must be secure are judged for their ability to support and enforce the security policy. Programmers should strive to build security into every application they develop, with greater levels of security provided to critical applications and those that process sensitive information.

There are numerous issues related to the establishment and integration of security into a product, including managing subjects and objects and their relationships, using open or closed systems, managing secure defaults, designing a system to fail securely, abiding by the “keep it simple” postulate, implementing zero trust (instead of trust but verify), incorporating privacy by design, and using Secure Access Service Edge (SASE). CIA can be protected using confinement, bounds, and isolation. Access controls are used to implement security protections.

Proper security concepts, controls, and mechanisms must be integrated before and during the design and architectural period to produce a reliably secure product. A trusted system is one in which all protection mechanisms work together to process sensitive data for many types of users while maintaining a stable and secure computing environment. In other words, trust is the presence of a security mechanism or capability. Assurance is the degree of confidence in the satisfaction of security needs. In other words, assurance is how reliable the security mechanisms are at providing security.

When security systems are designed, it is often helpful to derive security mechanisms from standard security models. Some of the security models that should be recognized include the trusted computing base, state machine model, information flow model, noninterference model, take-grant model, access control matrix, Bell–LaPadula model, Biba model, Clark–Wilson model, and the Brewer and Nash model.

Several security criteria exist for evaluating computer security systems. The Common Criteria uses a subjective system to meet security needs and a standard evaluation assurance level (EAL) to evaluate reliability.

The NIST Risk Management Framework (RMF) establishes an authorization to operate (ATO) issued by an authorizing official (AO) to ensure that only systems with acceptable risk levels are used to perform IT operations.

It is important to carefully assess each aspect of the infrastructure to ensure that it sufficiently supports security. Without an understanding of the security capabilities of information systems, it is impossible to evaluate them, nor is it possible to implement them properly. The security capabilities of information systems include memory protection, virtualization, Trusted Platform Module (TPM), encryption/decryption, interfaces, and fault tolerance.

Managing the information system life cycle is a comprehensive process that involves various stages of a structured and organized approach to developing, deploying, and maintaining an information system.

Study Essentials

Be able to describe open and closed systems. Open systems are designed using industry standards and are usually easy to integrate with other open systems. Closed systems are generally proprietary hardware and/or software. Their specifications are not normally published, and they are usually harder to integrate with other systems.

Know about secure defaults. Never assume the default settings of any product are secure. It is always up to the system administrator and/or company security staff to alter a product's settings to comply with the organization's security policies.

Understand the concept of fail securely. Failure management includes programmatic error handling (aka

exception handling) and input sanitization; secure failure is integrated into the system (fail-safe versus fail-secure).

Know about the principle of “keep it simple.” “Keep it simple” is the encouragement to avoid overcomplicating the environment, organization, or product design. The more complex a system, the more difficult it is to secure.

Understand zero trust. Zero trust is a security concept where nothing inside the organization is automatically trusted. Each request for activity or access is assumed to be from an unknown and untrusted location until otherwise verified. The concept is “never trust, always verify.” The zero-trust model is based around “assume breach” and microsegmentation.

Know about privacy by design. Privacy by design (PbD) is a guideline to integrate privacy protections into products during the early design phase rather than attempting to tack them on at the end of development. The PbD framework is based on seven foundational principles.

Understand trust and assurance. A trusted system is one in which all protection mechanisms work together to process sensitive data for many types of users while maintaining a stable and secure computing environment. In other words, trust is the presence of a security mechanism or capability. Assurance is the degree of confidence in the satisfaction of security needs. In other words, assurance is how reliable the security mechanisms are at providing security.

Define a trusted computing base (TCB). A TCB is the combination of hardware, software, and controls that form a trusted base that enforces the security policy.

Know details about each of the security models. The state machine model ensures that all instances of subjects accessing objects are secure. The information flow model is designed to prevent unauthorized, insecure, or restricted information flow. The noninterference model prevents the actions of one subject from affecting the system state or actions of another subject. The take-grant model dictates how rights can be passed from one subject to another or from a subject to an

object. An access control matrix is a table of subjects and objects that indicates the actions or functions that each subject can perform on each object. Bell–LaPadula subjects have a clearance level that allows them to access only those objects with the corresponding classification levels, which protects confidentiality. Biba prevents subjects with lower security levels from writing to objects at higher security levels. Clark–Wilson is an integrity model that relies on the access control triplet (subject/program/object).

Know the controls used for evaluating computer security systems. The Common Criteria (ISO/IEC 15408) is a subjective security function evaluation tool that uses protection profiles (PPs) and security targets (STs) and assigns an evaluation assurance level (EAL). Authorization to operate (ATO) (from the RMF) is a formal approval to operate IT/IS based on an acceptable risk level based on the implementation of an agreed-on set of security and privacy controls.

Understand the security capabilities of information systems. Common security capabilities include memory protection, virtualization, Trusted Platform Module (TPM), encryption/decryption, interfaces, and fault tolerance.

Know about the information system life cycle. Managing the information system life cycle is a comprehensive process that involves various stages, each with specific activities and considerations. Managing the information system life cycle involves a structured and organized approach to developing, deploying, and maintaining an information system. Know the nine stages.

Written Lab

1. Name at least seven security models and the primary security benefit of using each.
2. Describe the primary components of TCB.
3. What are the two primary rules or principles of the Bell–LaPadula security model? Also, what are the two rules of