Security Policy Enforcement

Cynthia E. Irvine, *Naval Postgraduate School*

INTRODUCTION

Many chapters of this *Handbook* describe mechanisms that contribute to various facets of security. The arbitrary use of security mechanisms provides no prescription for the achievement of security goals. It is only in their application in the context of organizational objectives for the protection of information and computational assets that security can be assessed. This chapter is intended to discuss the policies that provide a rationale for those mechanisms and to broadly examine their enforcement mechanisms in computer systems. It is intended to focus primarily on fundamental concepts, which remain valid despite their longevity.

In a utopian world where nothing bad ever happened, information security would be unnecessary. There would be no accidents; all actions performed by users would be correct; no attackers would attempt to violate systems. Unfortunately, reality is dramatically different. Information owners are confronted with risks to their assets and, to address these risks, make statements regarding what needs to be protected and how well. These statements constitute the basis for information security policies.

Security policies for information and assets have been with us for centuries, but their application within computer systems requires examination. Sterne (1991) provides a useful guide to understanding how policy is expressed at several levels within an organization and how it is described in a technical context.

First, security policy applies to the protection of assets. Sterne points out that only tangible assets can be protected. Intangible assets may also be protected through the protection of tangible assets, but it is impossible to state and implement a policy to address intangible assets. For example, how can a bank protect its reputation? Not by putting guards around that “reputation.” Instead it protects tangible assets such that its reputation is unsullied and enhanced. In contrast, the bank that inadequately protects funds transfers and financial records may be attacked with consequent damage to its reputation. Thus at the highest level, the company board of directors may state policy very abstractly: “important information and other assets must be protected.” It will be up to management to translate that policy into more concrete terms and to establish the practices for its enforcement.

Policy is enforced through procedures and mechanisms. Prior to the information age, these procedures and mechanisms were manual: now they are automated. When computers are used to store and process information assets, technical policies are required to translate management strategies into engineering specifications. Even for a nontechnical asset, for example, a painting in a museum, a computer system may be used in conjunction with other protection, and a technical policy describing the policy to be enforced by that system will be required. Mechanisms within the system contribute to policy enforcement, but just as important, external technical and nontechnical procedures involving human interaction must be followed to ensure compliance with the enterprise policy. User account management provides an example where automated and procedural measures must be combined to achieve the desired result. At some nontechnical level, it must be determined whether a particular individual should have an account. Technical activities will ensure that the account, if granted, is created and maintained.

Security as a Negative Requirement

In security, confidence in an information technology (IT) system comes from knowing that a broad range of bad things, many heretofore unknown, will not occur. For many enterprises, a lack of security can result in lost opportunity. This is due to the fact that fear of unknown

Report Documentation Page

Form Approved
OMB No. 0704-0188

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

1. REPORT DATE **21 SEP 2005**	2. REPORT TYPE	3. DATES COVERED **00-00-2005 to 00-00-2005**

4. TITLE AND SUBTITLE
Security Policy Enforcement

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School,Center for Information Systems Security Studies and Research (CISR),Monterey,CA,93943

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR’S ACRONYM(S)

11. SPONSOR/MONITOR’S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**15**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

security failures will prevent organizations from exploring new IT-based business models, where security is predicated on the fact that unauthorized access to critical information requires a level of effort that adversaries cannot mount. When security is good, a large number of bad things do not happen. Because it is impossible to enumerate all of the possible bad things (and few would choose to turn off security measures to discover even a partial list of these undesirable events), the notion of "measuring" security is not helpful. Because of this measurement problem, management is often reluctant to invest in security. It is more appropriate for management to ask, "In what activities are we unwilling to engage because we do not trust computer systems to adequately protect our assets?" Thus, security is enabling technology but is expressed as a negative requirement: do not let security breaches occur. Through a process of risk analysis, it is possible to create simple models and determine where within the enterprise various security measures will result in the highest return on investment.

Security as a Constructive Effort

There are two approaches to achieving security policy enforcement, whether in individual systems or in networks of systems. The first is to apply various security measures to a system that is discovered to be insufficient after it has been deployed. Often such measures may be ad hoc. In the case of many commodity products, patches are used to remedy some flaw that has been revealed. Unfortunately, the cure may be worse than the original malady and can lead to additional flaws (Karger & Schell, 1974).

An alternative is to articulate the security policy and then construct a system sufficient to enforce it to some level of confidence. This approach allows system owners to better understand how various risks have been mitigated and those threats for which the system does not have sufficient protection. For example, if a system is intended for a student's database of favorite music, the assets are not of high value and lightweight mechanisms may be adequate to protect the information. A system intended to automate the processing of critical national security information will require considerably stronger security, because the threat posed by adversaries is much higher.

Security policies and the constructive techniques used to enforce them are the focus of what follows.

KEY DEFINITIONS FOR DESCRIBING TECHNICAL POLICIES

Having provided a motivation for technical security, some terminology is required.

Active Entities: Subjects

The heart of every computer is its central processing unit (CPU): a collection of registers and hardware mechanisms to execute code. Instructions are fetched and acted on with consequent changes to the system state. Although this is a rather simplistic description of modern processors, which in reality can be quite complex (Hennessy & Patterson, 1996), it is sufficient for this discussion. At any particular moment, the processor executes on behalf of a particular active entity. In a multiprocessing system, active entities are scheduled. These entities consist of the set of instructions being executed and some domain within the system address space that will be read, written, or both as the instructions are executed. These active entities are called *subjects* (Lampson, 1971). The people external to the computer, viz. users, are not subjects; subjects act within the computer on behalf of users. The notion of a subject is a term of art in computer security and should not be confused with threads. (See, for example, Tannenbaum, 2001, for a discussion of threads.) In many simple operating systems, subjects correspond to entire processes; however, some systems implement highly granular privilege policies such that a single process may support multiple subjects. Using hierarchical rings, Multics provided a highly granular privilege mechanism (Schroeder & Saltzer, 1972).

Passive Entities: Objects

The passive entities of a system are called *objects* (Lampson, 1971). A set of objects comprise the domain of each subject. These objects possess security attributes such that the subject has some form of access to each object within its domain. It is possible for many subjects to share access to the same object. If an object can be written to by multiple subjects, as might occur in a database, some form of synchronization among subjects may be required. Because subjects execute within processes, a wide range of interprocess synchronization mechanisms (Maekawa, Oldehoft, & Oldehoft, 1987) may be useful. These can ensure that specific actions on resources are viewed as atomic.

The objective of security policy definition and enforcement is to control the ways subjects can share and affect the objects.

In a system lacking any sort of security policy, all subjects would have the same access rights to all objects. They would be expected to behave properly. Early operating system designers quickly recognized this as a recipe for chaos and instituted controls (a form of system-internal security policy) to protect processes from each other and from the operating system itself. Saltzer and Schroeder (1975) provide a review of many of these mechanisms. With the exception of certain specialized single-process systems (Weissman, 2003), today there is an expectation of controlled sharing of resources, for example, processor time, memory, and devices, among processes. Security policy is a major factor in determining the resource-sharing mechanisms.

TYPES OF POLICIES

At a high level, an organization security policy may be expressed in terms of three objectives: confidentiality, integrity, and availability. These primary policies may be complemented by those for separation, least privilege, policy control, and other supporting policies.

Confidentiality Policies

Confidentiality is focused on protecting information from unauthorized disclosure and may apply to requirements for secrecy and privacy. Confidentiality policies are implemented in most organizations. Businesses may wish to protect trade secrets, marketing plans, accounting information, proposals, and so forth from disclosure to the general public. In addition, they may be required to protect information regarding their personnel. In the health care industry, protection of personal information is of particular interest. Here, the disclosure of medical records to unauthorized individuals could have serious consequences. A number of laws and regulations articulate confidentiality policy requirements within various sectors of society.

Integrity Policies

Integrity addresses the modification of information: it relates to the reliability or criticality of information. Information of high integrity might be considered to be either highly reliable, for example, from a highly trusted source, or intended for use in critical operations. For example, when a high integrity application is distributed, typical users might be permitted to inspect the source code or the binaries but would be prohibited from modifying either: those authorizations might be limited to certain engineers or programmers. A level of criticality might be associated with meteorological information prior to a space flight launch. As a consideration in a launch decision, such information must not be tampered with.

Availability Policies

Both confidentiality and integrity policies can be implemented by mechanisms that determine whether a particular subject, which is ultimately executing on behalf of some user, may either observe or modify the information being protected. The decision is simple: either access is permitted or it is denied. Confidentiality and integrity can be reduced to a set of yes or no decisions. In contrast, policies regarding the availability of resources are notoriously difficult to characterize and even more challenging to enforce. This is due to the subjective nature of availability.

Availability policies, by their very nature, are subjective and must be addressed on a per-organization basis. Consider the following example. If an availability policy were to state that all users must have sufficient resources for arbitrary tasks, then when several users decided that they wanted to model colliding galaxies and molecular interactions on a small, general purpose computer (assuming these codes would execute on such a machine), the availability policy would not be met. This is a rather extreme scenario; however, it is easy to see how the tension for resources can result in service inadequacies.

As another example, suppose a user has a processor adequate for supporting a defined set of applications and also suppose that this system is not connected to any network. As long as the application suite is unmodified and there are no stresses imposed on the system by the network, the user is likely to find the system to be adequate. Now, introduce a new suite of applications that consume much larger amounts of system memory and require considerably more processing. Suddenly, the system that was once satisfactory is now inadequate. Connection to a network can further complicate the user's perception of system availability by taxing system resources to support network communications and data transfer, in addition to exposing the system to denial of service attacks in which adversaries deliberately consume system networking resources. Techniques to address the threat of denial of service attacks may entail the application of specially crafted confidentiality and integrity policies. Although availability can be addressed on a case-by-case basis, researchers have yet to develop a generally applicable model for system availability.

Separation Policies

Separation systems enforce an internal policy that isolates processes from one another (Rushby & Randell, 1983). In general, absolute isolation is not particularly useful; therefore relaxation of absolute separation is usually required. This then leads to consideration of some combination of confidentiality and integrity policy enforcement. In the early years of this decade, hardware advances have allowed consideration of the practical construction of systems that employ a low-level separation kernel to create isolated blocks, where the term *block* is used in the mathematical sense. Through careful static configuration of the separation kernel, blocks represent equivalence classes. Within blocks, more granular mandatory policies may be enforced (Levin, Irvine, & Nguyen, 2004). If a set of common underlying processors supports the separation system, then issues such as availability and covert channels may emerge.

When absolute separation is required, but isolated Process A can perceive the presence of another presumably isolated process, B, then two problems arise. Process A may not have as many machine cycles with which to perform its task and thus may experience an availability problem. Second, through the manipulation of system-level resources, Process B may be able to signal to Process A in a manner not permitted by the overall separation policy. This creates a covert channel (Lampson, 1973; Levin & Clark, 2004).

Least Privilege

The principle of least privilege (Saltzer & Schroeder, 1975) states that no entity within a system should be accorded privileges greater than those required to carry out its tasks. For example, the task of audit administrator does not require authorizations to manage user accounts or to configure new system devices, nor does a user who merely wishes to write a letter require the power to configure the operating system. To effectively apply the principle of least privilege within the context of an implementation, a policy must be articulated. Within a monolithic entity, the principle of least privilege can be used as a metric by which to structure the system based on information-hiding paradigms (Parnas, 1972). Among processes, the principle of least privilege can be realized by the application of integrity, and sometimes confidentiality, mechanisms. Among processes, the principle of least privilege can be implemented to protect the integrity of the operating system, libraries, and other reliable components from software of unknown provenance (Saltzer & Schroeder, 1975; Schroeder & Saltzer, 1972). This permits processes to be organized into a set of hierarchically ordered subjects, with the process integrity policy enforced by the underlying operating system. The domains having the least privilege depend on those of increasing privilege. Subjects in the former can be prevented from corrupting the data and software of the latter.

Control Policies

The policies in the previous paragraphs describe the types of protection that may be required but do not address how those policies will be controlled. Control policies determine the ways in which policies can be modified, and fall into two categories: *discretionary* and *mandatory*. A discretionary control policy provides an interface whereby the access control policy can be modified at runtime by programs executing on behalf of users. A mandatory control policy may not be modified at runtime and instead is both global and persistent; a secret is secret no matter where the information is being accessed or what time of day it is. The difference between these control policies is of great importance. The fact that two kinds of control policies exist reflects fundamental differences in the way organizations want the policies themselves to be managed.

In organizations without computers, individuals are cognizant of all actions taken with respect to information being processed. These organizations may subdivide and compartmentalize their information. Certain individuals are vetted and authorized to handle specific, limited subsets of the information. Through the vetting and training process, each individual knows how to handle information of different sensitivities and is expected to exercise judgment so that sensitive information is not compromised.

When computers are used to augment and enhance productivity, a problem arises. Now, a program is executed on behalf of the individual. In general, software is written by third parties, be it commodity, open source, or custom. Although the software may properly perform its advertised functions, there is no guarantee that it does not contain additional, clandestine artifacts that attempt to modify, disregard, or circumvent the system security policy. Such malicious artifacts may take the form of Trojan horses or trapdoors, both of which are discussed further in the fifth section.

Figure 1 depicts a Trojan horse being executed by its victim, Alice. The access controls on her software do not permit Mallory to read her file; however, the access controls on Mallory's file permit Alice to write to his file. The Trojan horse, acting with Alice's permissions, is able to read her files and write their contents to Mallory's files in violation of Alice's intent to protect her information from access by Mallory. In a system with discretionary controls, the Trojan horse might also have used its control privileges to modify the access control list on Alice's information so that Mallory could read it.

If the intent of the policy is not correctly encoded into the underlying enforcement mechanisms, a Trojan horse can violate the policy. Another problem arises because in many systems, the policy can simply be modified, thus permitting the Trojan horse to carry out its malicious intent. When a system provides a general application interface that permits policy modification, the policy control mechanism is *discretionary*: users and programs can change the policy. This is useful when a requirement for dynamic policy modification exists and when the consequences of policy modification are insignificant. Thus, discretionary policies are appropriate in situations when a possible Trojan horse executing within an application will cause only minor, localized damage.

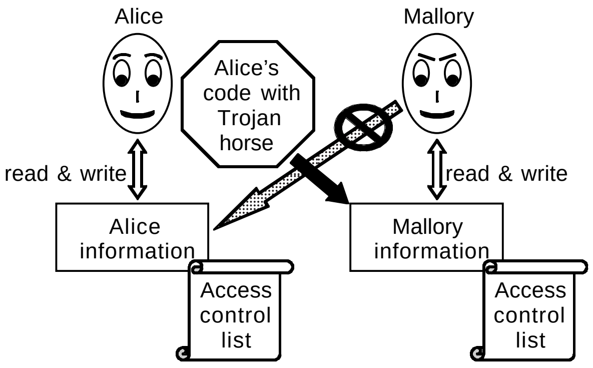

Figure 1: A Trojan horse executing in Alice's code is able to write to Mallory's information, thus circumventing Alice's intent to block Mallory's read access to her information.

There are several ways that control over discretionary policies may be exercised (DoD, 1987). The most common form is *owner control*. Here, each object has an owner, a named user of the system, and that owner can change the access rights to the object, thus granting more or less access to various other users or collections of users called *groups*. *Centralized control* places an administrator in charge of granting and revoking access rights. Few systems take this approach, because the burden on the administrator may be too high. *Hierarchical control* systems organize the information objects in a tree-like structure, as in a typical file system, and provide diminishing control over access to objects as one moves from the root to the leaves. It could be useful in military or other highly structured, top-down organizations. Finally, *laissez-faire control* permits anyone to modify the policy on anything. Of the various approaches to discretionary control, owner control is the most common and is found in most variants of UNIX and Linux. Given sufficiently elaborate underlying control structures, an operating system can be configured to support any of these control policies.

The implementation of discretionary policies offers choices regarding access rights when objects are created. For example, an object might be created where all accesses were permitted until access by certain individuals is denied. Alternatively, objects could be initialized with all access denied and subsequent access to the object would be granted on a need-to-know basis. Lunt (1989) provides a detailed discussion of various approaches to discretionary access control.

Because of their fluidity, discretionary policies offer the opportunity to revoke the rights of subjects to objects. A problem associated with revocation is that although the right to an object may be revoked, the information itself may already have been accessed and copied. Another problem with revocation is that of its timing. If the permissions on an object are changed, should the system immediately locate and terminate access by subjects no longer authorized to access the object? One must find all, viz. the transitive closure, subjects with the revoked right that have active access. This would require examining all of the current accesses of all of the subjects, which, although feasible, could have a significant effect on system performance while the access control system conducts the search. Observers have noted that because the subject once had access to the object, it could have copied the information; hence access to the information that the object contained will not be prohibited. In addition, it might be considered somewhat impolite to cause an application to suddenly crash because access permissions were modified while the application was using the subsequently revoked object (especially because the application could have copied all of the information in the first place). As pointed out by Grossman (1995), immediate revocation may be particularly difficult to define and achieve in highly distributed systems.

Other dynamic changes in access rights can occur, for example, when the user represented by the process is dynamically changed as in the UNIX *setuid* permission mechanism. Code is stored in files and owned by individual system users. To execute new code, a system mechanism replaces the current executable of the process with a new one. Usually, the user identifier associated with the process remains the same; however, when the *setuid* permission is set, the process takes on the persona associated with the new executable. It would be possible to construct systems with this functionality that could grant access rights inappropriately. Checking for such behavior is not always possible. It has been proved that it is impossible to construct a mechanism that will determine whether an arbitrary system will leak information (Harrison, Ruzzo, & Ullman, 1976). Fortunately, there are systems constructed with nondiscretionary controls on the policy mechanism for which leakage of access rights does not occur.

Some systems support the notion of access permissions associated with various collections of job functions called roles. In a hospital, these might correspond to the functions of doctor, nurse, technician, administrator, and so forth. The benefits of access controls based on roles are realized in large organizations where administrators can assign users to roles and then change those roles as user responsibilities evolve. Many systems may be organized to support role-based access controls, a notion first introduced by Ferraiolo and Kuhn (1992). A survey of a large number of role-based access control models was conducted by Sandhu, Coyne, Feinstein, and Youman (1996).

Systems enforcing nondiscretionary, or mandatory, control policies do not provide a general interface for policy modification and are applied in cases when the policy is intended to be constant in both time and space. For example, if the secret formula to a popular cola drink is secret worldwide and at all times, it will be treated as such by both the company personnel and through the design and implementation of its IT systems.

A qualitative metric for determining when mandatory control policies are required can be found in the consequences of policy violation (Brinkley & Schell, 1995). If grave harm would result and individuals would be fired or imprisoned for willfully violating the policy, then mandatory controls are probably appropriate.

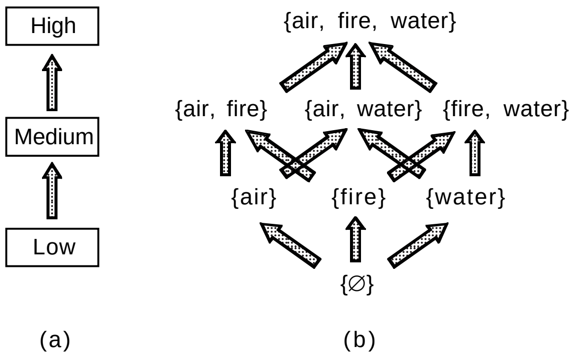

Figure 2: Lattice of access classes. A hierarchical ordering of classes is shown in (a). In (b), a set of noncomparable classes is shown. Arrows show the allowed direction of information flow.

In the context of mandatory policies, subjects and objects are allocated to equivalence classes that are partially ordered. Denning (1976) showed that a lattice provides a useful representation of the equivalence classes and their relationships with respect to the flow of information within a system enforcing a mandatory policy. Figure 2 shows both a hierarchical ordering of access classes and a set of classes created from noncomparable attributes: air, fire, and water. In the latter, any set of classes may receive information from a set of classes that is a subset of itself. The information flow policies depicted in the figure may be combined by taking their Cartesian product. The work of Denning also demonstrates that elaborate mandatory policies can be represented in a lattice through the introduction of additional equivalence classes so that a least upper bound and a greatest lower bound can be found for any pair of equivalence classes.

The Bell and LaPadula model provided a formal description of a mandatory confidentiality (secrecy) policy (Bell & LaPadula, 1973). Figure 3 illustrates the read and write accesses permitted to subjects when a mandatory confidentiality policy is enforced. Each subject may read

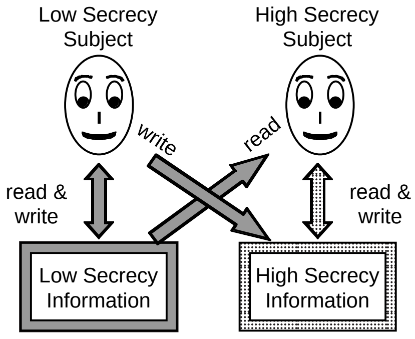

Figure 3: High confidentiality subjects have read access to low confidentiality information; at the same time confinement prevents the flow of high information to low.

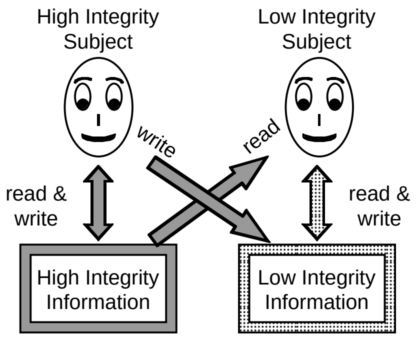

Figure 4: Confinement prevents the corruption of high integrity subjects with low integrity information; at the same time low integrity subjects benefit from high integrity information.

and write information at its own level. High secrecy subjects may read low secrecy information and low secrecy subjects may write high secrecy information, but low secrecy subjects may not read high secrecy information nor may high secrecy subjects write to low secrecy information. It is worth noting that although mandatory confidentiality policies generally permit low secrecy subjects to perform a blind write to high secrecy information, in practical implementations blind writes could result in chaos and are not generally permitted.

Figure 4 illustrates the read and write accesses permitted when a mandatory integrity policy is enforced, such as in the Biba model (Biba, 1977). Here, high integrity subjects may not be corrupted by low integrity information; at the same time low integrity information and subjects can be supplemented with high integrity information.

In the case of both confidentiality and integrity, the fact that the mandatory security levels are partially ordered makes comparison of access classes easy to implement.

It is worth noting that the Seaview model was the first to illustrate that a single set of equivalence classes could be used to enforce combined secrecy, integrity, and least privilege policies in a system with mandatory controls (Lunt et al., 1989). (Note that Seaview was a model for a database management system that allowed the enforcement of mandatory and discretionary policies. The formal name for the project was Secure Distributed Data Views.) The confidentiality policy model was similar to that formulated by Bell and LaPadula (1973). The integrity policy applied to access between processes, similar to that modeled by Biba (1977), whereas the least privilege policy, which was to be implemented using protection rings (DoD, 1994), applied to process-internal integrity (Shirley & Schell, 1981). The security levels were created by taking the Cartesian product of the partial orderings for the confidentiality and integrity policies.

The problem of characterizing security policies for the commercial sector where the integrity of information is often of equal or greater importance than its confidentiality was recognized in the early work of Lipner (1982), who described how the Biba model could be applied in commercial settings. Clark and Wilson (1987) described a more extensive commercial integrity model based on the notion of transactions, which was shown to be feasible using existing technology by Shockley (1988). To describe conflict of interest regulations that arise in the business context, Brewer and Nash (1989) developed the Chinese Wall security model. Drawing on both confidentiality and integrity, active entities are not permitted to access information once a possible conflict of interest between two data sets has been established. For example, by accessing the protected information of a particular pharmaceutical company, access to the information of similar companies would be disallowed. A sanitization policy allows certain nonsensitive information to be released to the public.

Supporting Policies

The policies discussed thus far relate to the access of subject to objects. In an operational system, supporting policies for user identification and authentication, as well as for audit, are required. The former allows a binding between the physical user external to the system and the subjects acting on the user's behalf within the system. First a user must identify himself to the system, then the user must demonstrate to the system that he is who he claims to be by presenting something that only comes from him. This might involve, for example, a password, a token, a biometric factor, or some combination of these.

Enforcement of discretionary policies generally takes the form of establishing rights to a particular object based on a name that is internally bound to each subject. For example, in many systems a user identifier is bound to the process, which also has a unique process identifier. Thus several processes may be acting on behalf of a particular user ID. The binding between the user's name and the user identifier is often found in the password file, which contains for each user a unique user name and user identifier.

For systems enforcing mandatory controls, the attribute associated with the subject will be its sensitivity level (equivalence class). This usually takes the form of a label. In a system enforcing mandatory policies, the password file might contain some maximum sensitivity level at which the user can log in. For example, a user cleared to TOP SECRET could select any one of the following levels for her current session: TOP SECRET, SECRET, or UNCLASSIFIED. Suppose a user logs on at SECRET, then, when a subject is instantiated on behalf of the user, one of its attributes will be the user's current session level, for example, SECRET.

Since the attributes bound to subjects acting on behalf of users are the basis for access control decisions, it should be clear that having a well-defined identification and authentication policy is required. For example, a system might require that each user have an individual password and be associated with a user group, that is, a set of common users such as *students* or *faculty*. Changes to groups might require additional passwords. Alternatively, a set of users might be associated with a particular activity or role for which they might authenticate. In a public library, it might be possible for anyone to access the system as a "library subscriber" and the purpose of the identification and authentication mechanism would be for accounting purposes rather than to track the individual reading habits of the population.

Audit provides a record of security-relevant events and can be used as a deterrent to such user malfeasance as white-collar crime. If bound to a rule-checking mechanism, audit may provide alerts of impending security violations and may evolve into elaborate intrusion detection systems. Policies must be established to determine what should be audited. For example, one might choose to audit all accesses to a particular file, but to no others; all activity on the system could be audited; the activities of a particular suspicious user might be recorded; the use of a particular set of system calls could be audited; and so forth. The choices regarding audit policy are extremely broad. Two important points should be noted, however. First, it is generally a good idea to audit the activities of the security administrator so that a record of security critical activities can be maintained. Second, policy makers should be aware that a voluminous audit record that is not accompanied by audit reduction tools is not likely to be particularly useful

In systems enforcing nondiscretionary policies, the management of labels and the labeling of information being transferred into and out of the system will be reflected in supporting policies. For example, there may be a requirement that all printed documents contain markings in their headers and footers indicating whether the document is "company proprietary" or "public."

System support is also required for the enforcement of administrative policies. These include user account management and security configuration, including the configuration of mandatory equivalence classes.

Object Reuse

As noted previously, objects are the information containers in systems. They are constructed using system resources, usually primary and secondary memory, but devices must also be considered. When objects are deleted, the memory from which they were constructed is returned to a pool. To prevent inadvertent access to information previously stored in now defunct objects, an object reuse policy is applied to memory resources. Such policies usually stipulate that all information must be removed from resources prior to their reuse. The system implementation determines whether the information is purged immediately after object deletion or prior to its allocation to a new object.

With this overview of policies, it is now possible to describe the various techniques and mechanisms that may be used to provide for their enforcement.

Policy Languages

Considerable work has been conducted in the area of defining languages for the expression of security policies. Only a few are presented here. An early example was KeyNote (Blaze, Feigenbaum, Ioannidis, & Keromytis, 1999). In highly networked organizations, databases accessible via Web interfaces provide a useful way to organize large quantities of information. Policies may be captured in the use of extensible markup language (XML) frameworks. Two emerging standards are SAML (Oasis, 2004) and XACML (Oasis, 2003). The former supports the exchange of security-relevant information between organizations, whereas the latter allows organizational security policies and access decisions to be expressed. A challenge for each of these standards is to create system architectures that provide a high assurance binding between security attributes and the information to be protected.

POLICY ENFORCEMENT MECHANISMS

The mechanisms used to enforce primary confidentiality and integrity policies depend on the control policies. Mechanisms for the enforcement of discretionary and nondiscretionary policies are discussed.

Mechanisms for Discretionary Policy Enforcement

Discretionary policies may be enforced in two ways: access control lists and capabilities.

Access Control Lists

Access control lists (ACLs) are lists of permissions associated with each object such as a file, directory, or device. Each ACL entry consists of the name representing an entity, such as an individual user or group, and the rights accorded to that entity. Because access control lists for a large number of similar users (for example, all of the students enrolled in a particular class) may be burdensome, it is often convenient to organize users into groups so that access rights can be granted to a number of users simultaneously. The largest group is, naturally, everyone, also known as "public" on many systems.

The types of permissions contained in an ACL may include more than merely read, write, and execute. It is possible to list the users or groups that have control access to the object, that is, who can grant or deny permission for other access rights. Furthermore, an additional level of permission can be provided through control-of-control-access, an access right that permits administration of control accesses. Access modes may be combined to create specialized access modes. For example, *append* access can be created with a combination of read and write access and restricts all writing to the end of the target object. Sometimes, it may be necessary to explicitly deny access to a particular user or group. Thus, ACLs can be enlarged to support negative access rights. Using negative access rights, it is possible to deny an individual access to an object even though he is a member of a group that possesses that access right. These several levels of access rights along with their various combinations can be used to create a highly sophisticated system.

Two implementation considerations are of particular interest for ACL-based systems. First, the initial value of each object's ACL must be determined. It is possible to provide template for an initial default ACL. This may be based on a template associated with the user or that is part of the parent directory. As noted earlier, Lunt (1989) provided an analysis of the defaults possible in systems with discretionary controls: no access (i.e., minimized access) or complete access. Where the principle of least privilege is to be observed, a limited or no access default would be appropriate.

The second implementation consideration is associated with the precedence of ACL entry interpretation. For example, suppose that there are conflicts between explicit user negative and positive permissions and those associated with one or more groups of which the user is a member. An organization may decide that negative ACLs take precedence, followed by positive user, group, and public permissions. This means that if an access right is explicitly denied, then the user will not be granted access despite the existence of positive access rights in other ACL entries.

The primary benefit of ACLs is that all permissions associated with a particular object are localized and can be easily managed. Revocation of access is achieved simply by changing the ACL. To "solve" the revocation problem described previously, implementations choose to have the revocation take effect on the next attempt to gain access to the object.

Capabilities

An alternative implementation approach is *capabilities*. In a capability-based system, the access rights to objects are bound to the subjects executing on behalf of users, rather than the objects. At the time of login, an initial set of capabilities is bound to a subject. As execution progresses, additional capabilities may be gained by subjects. Once a subject possesses a capability for a particular object, that object may be accessed with the rights associated with the capability; all the subject needs to do is present the capability.

For capability-based systems, revocation presents challenges. This is because a capability-based system distributes the access rights to each object among many subjects. If subjects are able to copy and store capabilities, the revocation problem is further exacerbated. There is no central location that can be inspected to determine which subjects have potential access to a particular object. Instead, the capability list for each subject must be inspected. If one decided to revoke access to an object, potentially every capability list in the system would require inspection to ensure that the revocation was complete.

The implementation of highly granular capability mechanisms in operating systems has been attempted in several systems; one of the most notable was the CAP system (Wilkes & Needham, 1980). Although they can be implemented, the systems are notoriously complex and their lack of a conceptually simple policy enforcement mechanism caused this approach to be abandoned.

Mechanisms for Enforcement of Nondiscretionary (Mandatory) Policies

As discussed previously, nondiscretionary policies provide no run-time interface for policy modification. Typical access control lists are unsuitable for the enforcement of mandatory policies. This is largely due to the enormous complexity of the management that would be required to ensure that all information and active entities have the proper security attributes and that those attributes cannot be modified via a run-time interface.

Two techniques are in common use for the enforcement of nondiscretionary policies. The first is physical and the second is logical. To discuss these policies, we introduce the notion of *sensitivity levels*. These are identifiers for equivalence classes of objects defined by the secrecy and integrity attributes associated with that set of objects. The choice of equivalence classes is up to the organization. For a private enterprise, the sensitivity levels might be PROPRIETARY, COMPANY CONFIDENTIAL, and PUBLIC, while a military organization might choose SECRET, CONFIDENTIAL, and UNCLASSIFIED.

To enforce policy using physical mechanisms, one must construct a separate network for each sensitivity level. All users must be authorized for the sensitivity level of the network and all information created and managed in that network must be considered to be at the network's sensitivity level. Such networks are described as being *single level*. The advantages of single-level systems or networks include the ability to identify and manage the access to information in a manner that is easy to understand. An isolated network may be maintained in a special facility and only users authorized to use that network may be granted access to the premises. The construction and maintenance of isolated networks and the facilities to house them can be costly, but when information is extremely critical the protection afforded by an isolated network may outweigh the cost.

There are serious disadvantages to physical isolation. Users must either move from room to room to access different networks, or they may have multiple systems on the desktop. The latter can lead to clutter and confusion when a user must access many networks in the course of daily activities. The user could use a KVM (keyboard, video, mouse) switch to minimize consumption of desktop space; however, multiple processors are still required and the possible advantage of seeing information at different sensitivity levels simultaneously is lost.

If nonsensitive information can be moved to networks of higher sensitivity, but without sensitivity labels associated with the information, it is impossible for users to distinguish nonsensitive information from that which is sensitive. To share the nonsensitive information with individuals having lesser authorizations, users must go back to the nonsensitive system. Alternatively, if a user wishes to transmit nonsensitive information directly from the more sensitive enclave, complex procedures are required to address the threat of unauthorized information flow resulting from the use of steganography (Kurak & McHugh, 1992) and other techniques for clandestine information hiding.

Logical isolation depends on an underlying mechanism that enforces the security policy. Because mandatory policies can always be characterized by comparisons between equivalence classes, it is possible to construct a relatively simple mechanism to determine whether a particular subject may have access to a given object. The Bell and LaPadula and Biba models permit read and write access by subjects at the same sensitivity level as the object; simultaneously, subjects are not permitted to read information of greater confidentiality, write to information of lesser confidentiality, write to objects of higher integrity, or read from objects of lesser integrity. Systems that enforce logical isolation can permit users to have a coherent view of all information at or below their sensitivity level. They also allow users to log into the system at any sensitivity level below their maximum authorization. Thus, from a single system, an authorized user is able to both access company proprietary information when logged in at "proprietary" and the Internet when logged in at the "public" sensitivity level.

CRITICALITY OF CORRECT POLICY ENFORCEMENT

Failures in security policy enforcement may result from technical flaws in information systems or the failure of people to use the system as intended. Although the latter is of considerable concern and must be addressed through appropriate information security training and awareness programs, the former is of interest here.

A large number of failures in policy enforcement result from the presence of unspecified functionality in systems. Broadly defined, unspecified functionality is the set of system flaws and unintended artifacts that permit an adversary ultimately to bypass the policy enforcement mechanism of a system. Thus, failures in design and implementation, ranging from inadequate bounds checking of interface parameters to pathological interactions between synchronizing processes, can be exploited by adversaries intent on gaining system privileges for the purpose of avoiding the constraints of the protection mechanism. Such flaws were identified by Anderson (1972) and are still found in current systems (Karger & Schell, 2002). The Common Vulnerabilities and Exposures (MITRE, 2004) Web site lists more than 3000 unique entries. A complete enumeration is not possible here; however, a few of the major categories derived from Linde (1975) and Anderson (1972) are provided in Table 1.

A more insidious form of unspecified functionality occurs when a system is subverted (Anderson, Irvine, & Schell, 2004; Myers, 1980). In this case, a member of the system's development team intentionally adds clandestine functionality that permits the adversary to bypass system security mechanisms. The term subversion is generally applied to the operating system or kernel, whereas other forms of malicious software, for example, Trojan horses, function in the context of applications. This permits several distinct characteristics of each to be identified. Trojan horses execute in the context of applications, thus they are constrained by the permissions and privileges of the user who is executing them. They can bypass the intended policy of the user but cannot bypass the policy enforcement mechanisms altogether. Trojan horses must be activated by the user. This gives the adversary less control over their execution.

In contrast, low-level subversion mechanisms execute within the operating system with full privileges and are unconstrained by policy enforcement mechanisms. They usually contain triggers for activation and deactivation, thus affording the adversary control over their execution as shown in Table 2.

It is useful to note that, generally, viruses execute in the context of applications, whereas other forms of malicious code can be placed either within applications or in the underlying system. In many cases, malicious code may be introduced into systems in the form of downloadable executables or scripts, updates, and patches. Thus, code of unknown provenance should be confined in a manner such that it does not result in pervasive damage.

Assurance

As is the case with security, assurance is a term that is often misused. For example, some state that "software assurance" will improve the "security" of systems. Both of these terms are rather meaningless without context. For software assurance, some might say that a system possesses this quality if it functions as specified and if various tests indicate that the software behaves as expected over a set of inputs, but this definition assumes that there is no malicious intent involved in the construction of the system. If, on the other hand, one assumes a malicious adversary, then assurance means correct policy enforcement in the face of a sophisticated set of attacks specifically intended to misuse system interfaces or insert artifices into the system itself.

Table 1 Examples of Errors Resulting in Security Flaws

General error category	Example
System design errors	Absence of least privilege Inappropriate mechanism for shared objects Poor choice of data types
Design errors	Error recovery results in exploitable side effects System modifications that deviate original intent of security mechanisms
Implementation errors	Buffers sizes are not checked, resulting in "buffer overflow" Failure to initialize variables Absent parameters are erroneously assumed
User interface errors	Gratuitous active execution Passwords too short Default access control lists are too permissive
Configuration errors	Insecure defaults render the system vulnerable Critical resources remain unprotected because of bad configuration choices

Table 2 Comparison of Trojan Horse and System Subversion

Trojan horse	System subversion
Requires activation and use by a victim user; the adversary cannot choose the time of activation.	No user is required. Activation and deactivation may be triggered remotely by the adversary.
Constrained by security controls imposed by the system on the victim.	Bypasses security controls.
Executes as an application.	Often executes within the operating system and, thus, has complete system privileges.

For a system to have assurance of security, the security policy must be enforced at all times and the security mechanisms must be resistant to subversion and tamper. Given a particular security policy, an organization may seek more or less assurance that the policy is correctly enforced. For example, greater assurance might be required to show that only authorized individuals have access to trade secrets, whereas less assurance might be required for the protection of the agenda for the next staff meeting. As subsequent sections reveal, assurance of correct policy enforcement in the face of a malicious adversary is not easily achieved. The pressures of product development can lead to shortcuts that reduce assurance: a problem encountered in many security systems is that of vendor claims of correct policy enforcement, when these systems are, in fact, quite vulnerable to attack. Sometimes a vendor will claim to have a "secret" technique that makes a system secure (these claims seem to be most prevalent in the area of cryptography and key management), but close inspection by knowledgeable reviewers usually reveals serious flaws (Anderson, 1972; Karger & Schell, 1974). It is generally accepted that an objective third party must provide an independent assessment of system assurance. This is similar to the ratings provided by independent consumer organizations for a wide variety of products. The current framework for third-party evaluation of system assurance is that of the Common Criteria (ISO/IEC, 2004).

Established through an international treaty, the Common Criteria support the creation of high-level requirements documents called *protection profiles* for various classes of security products (NIST, 2004). Each protection profile includes both functional and assurance requirements, where the latter achieves one of seven levels of confidence through requirements imposed on the system lifecycle processes. For specific systems, developers can create a *security target*, which in addition to all of the requirements of the protection profile, includes more detailed system-specific requirements. Through a process of analysis and testing, product team evaluators use the protection profile and security target to establish whether the product meets both the functional and assurance security requirements. A second round of testing and analysis by independent evaluators validates the team's results. Completion of the process results in an official evaluated product.

A secure system should exhibit all of the characteristics of a classic reference monitor (Anderson, 1972): resistant to tamper, always invoked, and understandable. Threat analysis reveals that there are two broad classes of threats to building a system that aspires to the objectives of the reference monitor concept. There are both developmental threats and operational threats to the system (Irvine et al., 2002). Developmental threats include the introduction of flaws into the system through mistakes in design and implementation and through deliberate system subversion. The former introduces exploitable flaws, whereas the latter introduces trapdoors. Hence, the system must be constructed using a methodology that will counter developmental threats, and it must be designed and implemented so that operational threats are mitigated.

Operational threats occur when the system is in use. Adversaries can include malicious insiders as well as external activities. The mechanisms that have been designed into the system are intended to counter operational threats; however, system security also depends on adequate user and administrator training, as well as good configuration management and system maintenance. In short, a well-constructed security system is of limited value if not used properly.

CONSIDERATIONS FOR THE CONSTRUCTION OF SECURE SYSTEMS

Two challenges confront the developer who wishes to construct a secure system. The first is the problem of policy dependencies. Suppose that the elements of the system intended to enforce mandatory policy are built using constructs exported by mechanisms enforcing discretionary policy. We must ask: what sort of assurance is possible if a mandatory policy enforcement mechanism is constructed as a layer that depends on a discretionary policy enforcement mechanism? The discretionary mechanism will export the storage resources used by the mandatory layer to create its policy enforcement mechanism. Access to these storage resources is mediated by the discretionary mechanism, which, by definition, has a runtime interface that allows its policy to be modified. Thus, in this architecture the mandatory mechanism is subject to run-time modification: global and persistent policy enforcement cannot be ensured.

The second challenge is associated with system complexity. Assurance depends on the ability for evaluators, such as those using the Common Criteria (ISO/IEC, 2004) framework, to understand the system. It must be possible to state with some level of confidence that no malicious, unspecified functionality has been added to the system. As systems become more complex, they become less understandable. Although size is a factor when considering complexity, it is not the only consideration. The interactions between subsystems can be subtle and difficult to completely describe. An adversary may attempt to subvert a system by using internal synchronization, interrupt handling, and other mechanisms to compose a trapdoor or other clandestine artifice. Understandability of the system is essential. A metric for high assurance is that all components of the protection mechanism are both necessary and sufficient to enforce the security. No additional functionality, whether malicious or merely gratuitous, is included in such a system.

The problems of composition and complexity present even greater challenges for distributed systems. Composed systems may enforce the same policy with differing levels of assurance. When this is the case, the integrity and assurance of the overall system are forced to the greatest lower bound of the composed assurance levels (Irvine & Levin, 2002). Even when systems have the same level of assurance, sometimes when they are networked together the risk of unintended disclosure can be significantly increased. The cascade problem (Horton et al., 1993) represents a good example of this composition risk.

To create a distributed system, it is necessary to understand the various security policies to be enforced and to allocate those policies to components in a manner such that dependencies among the networked components are reasonable and a coherent distributed enforcement mechanism results. Unless an extensible distributed security architecture is described, the effect of the addition of new components usually requires complete reanalysis of the distributed system.

Essential Elements for System Protection

Several key elements for the creation of an effective protection mechanism were identified by Saltzer and Schroeder (1975). These include a memory management mechanism that allows the memory resources used by applications to be distinguished from those of the underlying system. Memory management must be able to prevent applications from arbitrarily accessing system functions and databases. Unauthorized attempts to access these resources should result in a fault that can be handled by the underlying resource management mechanism. To permit applications to request services of the operating system, controlled entry points must be created. Not only must these entry points ensure the proper flow of control to the correct underlying function, but they should also validate all arguments so that misbehaving applications are unable to manipulate the called functions in unanticipated ways. The system should have at least two modes of operation: privileged and unprivileged. Hardware constructs are used by the operating system to set the mode. One or more protection bits can provide this service. The instructions for the management of the processor mode are restricted to the privileged mode. Instructions needed for primitive resource management must be privileged as well. Finally, it must be possible for the system to create an unambiguous binding between the user and processes that will execute on the user's behalf. A *trusted path* is used to both authenticate the system to the user and the user to the system. It is constructed in such a way that both entities have confidence that neither interface is being spoofed.

Constructive Security

Those building secure systems must be paranoid: someone intends to subvert the system and it may be a member of the core design team or someone else who has access to the system at some point during its lifecycle. Typical process-related and testing techniques described in software engineering are inadequate because they assume a benign development environment.

Security requirements engineering results in a description of the system to be built. As a start, it is important to understand that the system will be subject to both developmental and operational threats (Irvine et al., 2002). Operationally, the system must be demonstrated to be resistant to tamper and bypass of security mechanisms. Developmentally, one must construct the system through a process that demonstrates both the absence of malicious code and that the mechanisms to enforce policy are complete and correct. Construction of a secure system involves careful attention not only to the security architecture and its implementation, but also to lifecycle management issues. To avoid construction of what might be deemed a "secure brick," system designers must account for the various services it will provide and performance obligations the system will meet.

Although the formality of the Common Criteria (ISO/IEC, 2004) may not be needed for ad hoc systems, its systematic presentation of issues related to system assurance can be quite helpful in creating a set of requirements system developers must address. Consider, for example, the principle sections of a typical protection profile. After providing a set of definitions and conventions, a protection profile starts with a high-level description of the system to be constructed. Here, the developer is able to state whether a full, general-purpose operating system will be built or some less all-encompassing special purpose component. This leads to a presentation of the threats to the system through out its entire lifetime and the security policies it is expected to enforce. Based on threats, policies, and various usage assumptions, it is possible to develop a set of security objectives that both counter the threats and address the policies and assumptions. The objectives drive both the functional and assurance requirements for the system. Security functional requirements might include audit, identification and authentication, access control, enforcement of flow control in support of mandatory policies, administrative interfaces, and other functions. It is interesting to note at this point that not all of the functional requirements will map to the formal security policy model, which is called for in the assurance requirements of high assurance systems.

Assurance requirements dictate how the system will be constructed and may influence the way various mechanisms support functional requirements. Because system security must be addressed for the entire system lifecycle, assurance includes many activities beyond design and coding. It is necessary to ensure that the tools used to construct the system are protected, so that they do not become vectors for the insertion of malicious software by a highly skilled adversary (Karger & Schell, 1974). Also, it is necessary to ensure that the system is not modified en route to its end users and that unscrupulous installers or maintenance personnel do not corrupt the system. Before starting to construct the system, the development team must put into place all of the necessary mechanisms and procedures to ensure that the system is built in a manner that both identifies and eliminates flaws and prevents the inclusion of malicious functionality (Irvine et al., 2004). This high assurance development framework will include standards for system specification and design, review processes, configuration management, distribution procedures, user and administrator documentation, and maintenance and flaw remediation procedures. Separation of duty and multiperson oversight are an integral part of the effort.

Secure System Development

As described in previous sections, dependencies are of great importance in designing a secure system. Consider the system to be organized as a set of hierarchical layers. Then, it must be organized so that each layer of the system depends only on layers that are of equal or higher assurance and that enforce equivalent or stronger policies. The raw resources of the hardware layer can be organized by low-level system software to export virtualized resources that are subject to a set of low-level policies. Primitives provided by the system might include memory, interrupt handling, and low-level scheduling, as well as synchronization. In a minimized system enforcing a separation policy, these may be the abstract data types exported at the system interface. For more traditional operating system kernels, additional operating system layers may be constructed that contain mechanisms to enforce the intended mandatory policies. The number and extent of these layers depend on whether the mandatory policy is void or richly populated, but it is important to recall that the dependencies must be such that the mandatory policy enforcement mechanisms do not depend on discretionary, viz. modifiable, policy components. It is important to note that a minimized kernel does not usually present a typical user-friendly application programming interface. A set of code libraries is usually superimposed on the operating system to hide the primitives described previously from typical programmers and users.

Once the system architecture has been delineated and policy has been allocated to its various layers, it is possible to focus on the construction of each layer. Here, the techniques used to develop a high assurance, low-level layer are sketched.

Because the objective in constructing the lowest layers of the system is to develop a coherent mechanism for the enforcement of the system's most critical policy, a combination of hardware and software is used to create the abstract machine that will be exported at this interface. Once the overall objectives of the system have been described, a formal security policy model is developed. The model provides a proof that if the system starts in a secure state then all operations will maintain that secure state. The formal model serves two important purposes: first, it demonstrates that the intended policy is logically self-consistent and not flawed, and second, it provides a mathematical description of the system to which the implementation can be mapped. The objective of this mapping is to demonstrate that everything in the implementation is both necessary and sufficient for the enforcement of the policy and that no unspecified functionality, for example, possible subversion, is present.

The formal model is highly abstract, so two other documents are produced. One is a formal description of the system interface; the other is a high-level system interface description. Both describe the system interface in terms of inputs, outputs, effects, and exceptions. For the former, a formal proof is generated showing that the formal description maps to the formal security policy model. Thus, by transitivity, a proof that the formal interface also describes a system that maintains secure state is achieved. The latter is used as the starting point for the concrete implementation of the system.

Rigorous security engineering techniques employing the concepts of layering, modularity, and data hiding are used during development to ensure that the system has a coherent loop-free design and provides abstractions so that it is understandable. Within the system itself, the principle of least privilege can be applied as part of the engineering process. Ultimately, both the formal and informal efforts provide a mapping of the implementation to the formal security policy model as well as evidence that the system is correct and complete.

Because information flow is a concern, all of the processes described previously contribute to the ability of the developers to conduct a covert channel analysis of the system. Covert channels result from the manipulation of system interfaces in ways that cause unintended information flow and result from incomplete resource virtualization by the underlying protection mechanism. This means that some abstract data type presented at the system interface involves operating system constructs that can be manipulated to allow signaling to take place in violation of the system security policy. Covert channels fall into two classes: timing channels and storage channels. An effective technique for covert channel analysis is the shared-resource matrix method (Kemmerer, 1982). Using this technique, the effects of each system call on operating system–level data structures are analyzed and their visibility, perhaps through exceptions or timing delays, to other processes is identified. When the effects could result in unintended transmission of information, either a system flaw or a covert channel is present.

Although testing cannot prove that a system is secure, it is important to include testing in the development process. Traditional testing demonstrates that each function and module performs as specified. At the system level, traditional testing is supplemented by penetration testing. Here, the tester behaves as an adversary and attempts to abuse the system interfaces in an unexpected manner. A useful approach to penetration testing is the Flaw Hypothesis Methodology (Linde, 1975). This testing technique organizes testing in a way that allows the testing team to set goals by working with the customer. The team studies the target system and conducts extensive nonjudgmental brainstorming to hypothesize system flaws. Then, the flaws are prioritized according to a preestablished guideline. Desk checking or live testing allows the team to determine the feasibility of flaw exploitation. The process is iterative and encourages the generalization of flaws into broad categories from which additional flaws may be hypothesized. It is important to emphasize that this form of testing augments the careful development process described previously. No testing is exhaustive, so it cannot demonstrate the absence of flaws or subversion but merely lessen the likelihood of obvious flaws.

Once the system is built, its administrators and users must be provided with the documentation and training necessary to use it securely. When a system that contains useful security mechanisms is configured and used improperly, a false sense of security may result that may have consequences far worse than when management believed security was inadequate.

The processes sketched here must be thoroughly documented so that the system can be assessed with respect to its security requirements. Third-party evaluation provides a way for those who acquire the system to understand whether the system does, in fact, meet its security objectives. To date, no viable alternative to either the reference monitor concept or the need for third-party evaluation has been proposed. Future research may result in more streamlined approaches to secure system construction and assessment.

Future Challenges

Many organizations enforce mandatory policies; however, operationally, they also require mechanisms where exceptions to these policies can be implemented. For example, consider a system that encrypts information before transmitting it on the network. The process of encryption can transform bits that represent proprietary sensitive information into bits that represent no information and, thus, can be seen by anyone (Shannon, 1949). In essence, encryption is *downgrading* the information, viz. changing its sensitivity level from high to low. Modern systems may require additional downgrading functions that move certain information from high networks or repositories to low ones. Sometimes the information is scanned for certain sensitive words that are then expunged from the data. This is called *sanitization*. All of these activities must be conducted using systems for which there is a very high confidence that only the correct actions will be taken. A danger in systems that do not involve human review is that of *steganography* (Kurak & McHugh, 1992). Steganography, the art of hidden writing, involves a secret encoded by malicious code in seemingly innocuous data so that it is not visible to the casual observer.

The components within a system that perform encryption, downgrading, and other operations that span mandatory sensitivity levels must be trusted to perform their tasks and nothing more—they must not contain unspecified functionality, for example, steganography, that would violate the intent of the system owners. The construction of *trusted systems* that, for most processes, enforce mandatory policies using underlying operating system controls and at the same time permit certain *trusted* applications to be subject to relaxed mandatory constraints represents one of the great challenges in security modeling and engineering.

CONCLUSION

Security policies are essential for computer and network security. Without a policy, security mechanisms are merely vacuous ad hoc functions that are combined to "do something," but what they might achieve, if anything, cannot be determined. At the management level, users must determine information assets that must be protected and must understand whether the authorizations for access to those assets are static or dynamic. This permits mandatory, discretionary, and supporting policies to be differentiated.

The nature of the policy will determine the mechanisms to be used for its enforcement. How those mechanisms are constructed addresses both developmental and operational threats. Assurance is derived from the rigorous security engineering process applied to its development and to the controls maintained over the system throughout its entire lifecycle. Independent assessment provides confidence that claims made regarding the correctness and completeness of the security policy enforcement mechanisms are valid.

GLOSSARY

Assurance Basis for confidence that a system meets its security requirements. Increasing levels of assurance provide increasing confidence of the absence of flaws and malicious artifices.

Covert Channel A means to pass information in violation of the mandatory policy of a system through the manipulation of a system-internal object for which no explicit system interfaces are presented.

Discretionary Security Policy A security policy that may be modified through a functional interface presented at the run-time system interface.

Least Privilege The notion that an active system entity should operate with the privileges necessary to complete its job but no more.

Object A passive entity in a system that contains information.

Nondiscretionary (Mandatory) Policy A policy that is global and persistent, and that cannot be modified via run-time interfaces presented to applications.

Security Policy Rules, laws, and similar constraints used by an organization to define how its information is managed and disseminated.

Subject An active entity in a system that makes references to objects.

Supporting Policy Nonaccess control policies that must be adhered to in order to protect the information of an organization, including, but not limited to, audit, identification and authentication, regrading, and sanitization.

CROSS REFERENCES

See *Access Control: Principles and Solutions; Security Policy Enforcement Information Assurance; Security Policy Guidelines*.

REFERENCES

Anderson, E. A., Irvine, C. E., & Schell, R. R. (2004). Subversion as a threat in information warfare. *Journal of Information Warfare, 3*(2), 52–65.

Anderson, J. P. (1972). *Computer security technology planning study* (Tech. Rep. ESD-TR-73-51, Vols. 1 and 2, NTIS Document No. AD758206). Hanscom, MA: Hanscom Air Force Base, Air Force Electronic Systems Division.

Bell, D. E., & LaPadula, L. (1973) *Secure computer systems: Mathematical foundations and model* (Tech. Rep. No. M74-244). Bedford, MA: MITRE Corp.

Biba, K. J. (1977). *Integrity considerations for secure computer systems* (Tech. Rep. No. ESD-TR-76-372). Bedford, MA: MITRE Corp.

Blaze, M., Feigenbaum, J., Ioannidis, J., & Keromytis, A. D. (1999, September). *The KeyNote trust management system version 2* (RFC 2704). Internet Engineering Task Force. Retrieved November, 15 2002, from http://www.apps.ietf.org/rfc/rfc2704.html

Brewer, D. F. C., & Nash, M. J. (1989). The Chinese wall security policy. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy* (pp. 219–230). Los Alamitos, CA: IEEE Computer Society Press.

Brinkley, D. L., & Schell, R. R. (1995). Concepts and terminology for computer security. In M. Abrams, S. Jajodia, & H. Podell (Eds.), *Information security: An integrated collection of essays* (pp. 40–97). Los Alamitos, CA: IEEE Computer Society Press.

Clark, D., & Wilson, D. (1987). A comparison of commercial and military security policies. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy* (pp. 184–194). Los Alamitos, CA: IEEE Computer Society Press.

Denning, D. E. (1976). A lattice model of secure information flow. *Communications of the ACM, 19*(5), 236–243.

Department of Defense. (1987). *A guide to understanding discretionary access control in trusted systems*. Fort George Meade, MD: National Computer Security Center.

Department of Defense. (1994). *Final evaluation report of Gemini Computers*. Incorporated Gemini Trusted Network Processor, version 1.01. Fort George Meade, MD: National Computer Security Center.

Ferraiolo, D. F., & Kuhn, D. R. (1992). Role-based access control. In *Proceedings of the 15th national computer security conference* (pp. 554–563). Fort George Meade, MD: National Security Agency.

Grossman, G. (1995). Immediacy in distributed trusted systems. In *Proceedings of the eleventh annual computer security applications conference* (pp. 75–79). Los Alamitos, CA: IEEE Computer Society Press.

Harrison, M., Ruzzo, W., & Ullman, J. (1976). Protection in operating systems. *Communications of the ACM, 19*(8), 461–471.

Hennessy, J. L., & Patterson, D. A. (1996). *Computer architecture: A quantitative approach*. San Francisco, CA: Morgan Kaufmann.

Horton, J. D., Harland, R., Ashby, E., Cooper, R. H., Hyslop, W. F., Nickerson, B. G., Stewart, W. M., & Ward, O. K. (1993). The cascade vulnerability problem. In *Proceedings of the IEEE Symposium on Research in Security and Privacy* (pp. 110–116). Los Alamitos, CA: IEEE Computer Society Press.

Irvine, C. E., & Levin, T. (2002). A cautionary note regarding the data integrity capacity of certain secure systems. In M. Gertz, E. Guldentops, & L. Strous (Eds.), *Integrity, internal control and security in information systems* (pp. 3–25). Norwell, MA: Kluwer Academic.

Irvine, C. E., Levin, T., Wilson, J. D., Shifflett, D., & Pereira, B. (2002). An approach to security requirements engineering for a high assurance system. *Requirements Engineering, 7*(4), 192–208.

Irvine, C. E., Levin, T. E., Nguyen, T. D., & Dinolt, G. W. (2004). The trusted computing exemplar project. In *Proceedings of the 2004 IEEE systems, man and cybernetics information assurance workshop* (pp. 109–115). West Point, NY, & Los Alamitos, CA: IEEE Computer Society Press.

ISO/IEC. (2004, January). *15408—Common criteria for information technology security evaluation* (Rep. No. CCIMB-2004-01-001, Ver. 2.2, Rev. 256). Geneva, Switzerland: International Organization for Standardisation.

Karger, P. A., & Schell, R. R. (1974). *Multics security evaluation: Vulnerability analysis*. Bedford, MA: Hanscom Air Force Base, Information Systems Technology Application Office Deputy for Command and Management Systems Electronic Systems Division (AFSC).

Karger, P. A., & Schell, R. R. (2002). Thirty years later: The lessons from the Multics security evaluation. In *Proceedings of the annual computer security application conference* (pp. 119–126). Los Alamitos, CA: IEEE Computer Society Press.

Kemmerer, R. (1982). A practical approach to identifying storage and timing channels. In *Proceedings of the IEEE Symposium on Security and Privacy* (pp. 66–73). Los Alamitos, CA: IEEE Computer Society Press.

Kurak, C., & McHugh, J. (1992). A cautionary note on image downgrading. In *Proceedings of the eighth annual computer security applications conference* (pp. 153–59). Los Alamitos, CA: IEEE Computer Society Press.

Lampson, B. W. (1971). Protection. In *Fifth Princeton conference on information sciences and systems* (pp. 437–443). Reprinted in *ACM SIGOPS Operating Systems Review, 8*(1),18–24.

Lampson, B. W. (1973). A note on the confinement problem. *Communications of the ACM, 16*(10), 613–615.

Levin, T., & Clark, P. C. (2004). A note regarding covert channels. In *Proceedings of the sixth workshop on computer security education* (pp. 11–15). Monterey, CA: Naval Postgraduate School.

Levin, T. E., Irvine, C. E., & Nguyen, T. D. (2004). *A least privilege model for static separation kernels* (Tech. Rep. NPS-CS-05-003). Monterey, CA: Naval Postgraduate School.

Linde, R. R. (1975). Operating system penetration. In *Proceedings of the national computer conference* (pp. 36–368). Montvale, NJ: AFIPS Press.

Lipner, S. (1982). Non-discretionary controls for commercial applications. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy* (pp. 2–20). Los Alamitos, CA: IEEE Computer Society Press.

Lunt, T. F. (1989). Access control policies: Some unanswered questions. *Computers and Security, 8*, 43–54.

Lunt, T. F., Neumann, P. G., Denning, D. E., Schell, R. R., Heckman, M., & Shockley, W. R. (1989, December). *Secure distributed data views security policy and interpretation for DMBS* for a Class A1 DBMS (RADC-TR-89-313, Vol 1). Griffiss Air Force Base, NY: Rome Air Development Center.

Maekawa, M., Oldehoft, A. E., & Oldehoft, R. R. (1987). *Operating systems*. Menlo Park, CA: Benjamin Cummings.

MITRE Corp. (2004). *Common vulnerabilities and exposures*. Retrieved December 22, 2004, from http://www.cve.mitre.org/

Myers, P. (1980). *Subversion: The neglected aspect of computer security*. Unpublished master's thesis, Naval Postgraduate School Monterey, CA.

NIST. (2004). *The common criteria evaluation and validation scheme*. Retrieved December 22, 2004, from http://niap.nist.gov/cc-scheme/index.html

Oasis. (2003). *Extensible access control markup language (XACML), version 1.0, Oasis standard*. In S. Godik & T. Moses (Eds.). Retrieved December 22, 2004, from http://www.oasis-open.org/committees/xacml/repository/

Oasis. (2004). *Conformance requirements for the OASIS security assertion markup language (SAML) v2.0* (Committee Draft 03). In P. Mishra, R. Philpott, & E. Maler (Eds.). Retrieved December 22, 2004, from http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM, 15*(12), 1053–1058.

Rushby, J., & Randell, B. (1983). A distributed secure system. *IEEE Computer*, *16*(5), 55–67.

Saltzer, J. H., & Schroeder, M. D. (1975). The protection of information in computer systems. *Proceedings of the IEEE, 63*(9), 1278–1308.

Sandhu, R., Coyne, E. J., Feinstein, H. L., & Youman, C. E. (1996). Role-based access control models. *IEEE Computer, 29*(2), 38–47.

Schroeder, M. D., & Saltzer, J. H. (1972). A hardware architecture for implementing protection rings. *Communications of the ACM, 15*(3), 157–170.

Shannon, C. (1949). Communication theory of secrecy systems. *Bell Systems Technical Journal, 28*, 656–715.

Shirley, L. J., & Schell, R. R. (1981). Mechanism sufficiency validation by assignment. In *Proceedings of the IEEE Symposium on Security and Privacy* (pp. 26–32). Oakland, CA: IEEE Computer Society Press.

Shockley, W. R. (1988). Implementing the Clark/Wilson integrity policy using current technology. In *Proceedings of the 11th national computer security conference* (pp. 29–37). Fort George Meade, MD: National Security Agency.

Sterne, D. F. (1991). On the buzzword "security policy." In *Proceedings of the IEEE Symposium on Research in Security and Privacy* (pp. 219–230). Los Alamitos, CA: IEEE Computer Society Press.

Tannenbaum, A. (2001). *Modern operating systems* (2nd ed., pp. 81–100). Upper Saddle River, NJ: Prentice Hall.

Weissman, C. (2003). MLS-PCA: A high assurance security architecture for future avionics. In *Proceedings of the annual computer security application conference* (pp. 2–12). Los Alamitos, CA: IEEE Computer Society Press.

Wilkes, M. V., & Needham, R. M. (1980). The Cambridge model distributed system. *ACM SIGOPS Operating Systems Review, 14*(1), 21–29.