Module II

OS Security – Protection Mechanisms –Authentication & Access control – Discretionary and Mandatory access control – Authentication mechanisms – Official levels of computer security (DoD) - Security breaches – Concept of a hole - Types of a holes – Study of the security features for authentication, access control and remote execution in UNIX, WINDOWS 2000

OS Security

The term security means the overall security of the system including protection of files. The protection mechanism is the mechanism used by specific operating system to safeguard information in a computer system.

Protection Mechanism

In some systems, protection is enforced by a program called a reference monitor. Every time an access to a potentially protected resource is attempted, the system first asks the reference monitor to check its legality. The reference monitor then looks at its policy tables and makes a decision.

Protection Domains

A computer system contains many ―objects‖ that need to be protected. These objects can be hardware (e.g., CPUs, memory segments, disk drives, or printers), or they can be software (e.g., processes, files, databases, or semaphores). Each object has a unique name by which it is referenced, and a finite set of operations that processes are allowed to carry out on it. The read and write operations are appropriate to a file.

It is obvious that a way is needed to prohibit processes from accessing objects that they are not authorized to access.

In order to discuss different protection mechanisms, it is useful to introduce the concept of a domain. A **domain** is a set of (object, rights) pairs. Each pair specifies an object and some subset of the operations that can be performed on it.
A right in this context means permission to perform one of the operations. Often a domain corresponds to a single user, telling what the user can do and not do.

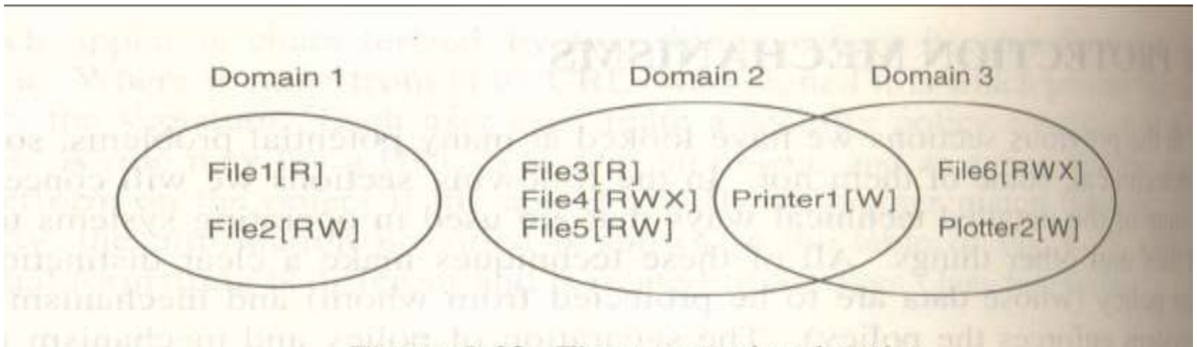
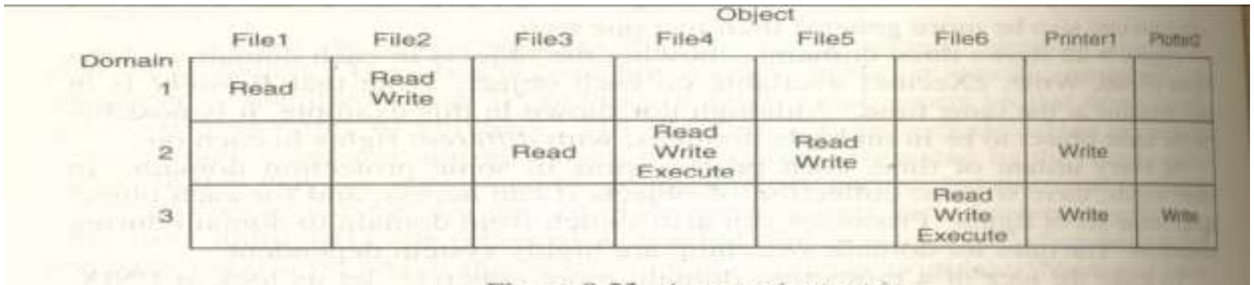


Figure 9-22. Three protection domains.

Figure shows three domains, showing the objects in each domain and the rights [Read, Write, eXecute] available on each object. Note that *Printer* is in two domains at the same time. Although not shown in this example, it is possible for the same object to be in multiple domains, with *different* rights in each one. At every instant of time, each process runs in some protection domain. In other words, there is some collection of objects it can access, and for each object it has some set of rights. Processes can also switch from domain to domain during execution. The rules for domain switching are highly system dependent.

In UNIX, the domain of a process is defined by its UID and GID. Given any (UID, GID) combination, it is possible to make a complete list of all objects that can be accessed and whether they can be accessed for reading, writing, or executing. Two process with the same (UID, GID) combination will have access to exactly the same set of objects. Processes with different (UID, GID) values will have access to a different set of files. Furthermore, each process in UNIX has two halves: the user part and the kernel part. When the process does a system call, it switches from the user part to the kernel part. The kernel part has access to a different set of objects from the user part.



	Object							
Domain	File1	File2	File3	File4	File5	File6	Printer1	Plotter2
1	Read	Read Write						
2			Read	Read Write Execute	Read Write		Write	
3						Read Write Execute	Write	Write

Figure 9-23. A protection matrix.

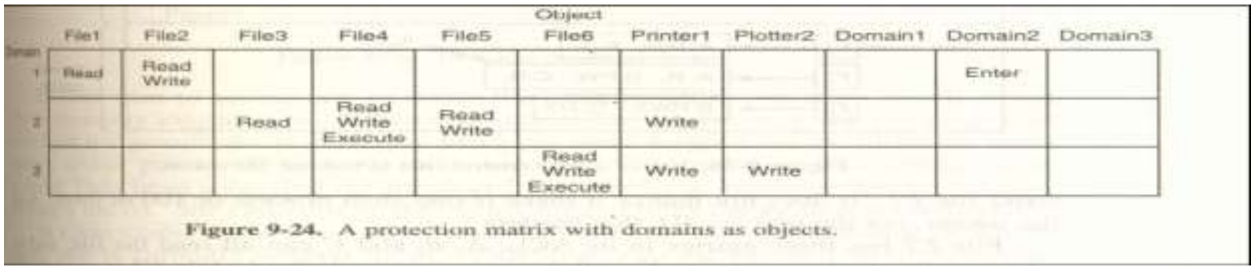
When a process does an exec on a file with the SETUID or SETGID bit on, acquires a new effective UID or GID. With a different (UID, GID) combination, it has a different set of files and operations available. Running a program with SETUID or SETGID is also a domain switch, since the rights available change.
This can be represented in the form of a large matrix, with rows being domains and the columns being objects. Each box lists the right, if any, that the domain contains for the object.

Domain switching itself can be easily included in the matrix model by realizing that a domain is itself an object. In practice, actually storing the matrix is rarely done because it is large and sparse.
We use two methods to store the matrix either by rows or by columns. They are

- Access Control List
- Capability List

Access Control List

The first technique consists of associating with each object an (ordered) list containing all the domains that may access the object, and how. This list is called the Access Control List or **ACE** and is illustrated in Figure. Here we see three processes, each belonging to a different domain. *A, B,* and *C,* and three files *F1,* F2, and *F3.* For simplicity, we will assume that each domain corresponds to exactly one user, in this case, users *A, B,* and *C.* Often in the security literature,



	Object									
Domain	File1	File2	File3	File4	File5	File6	Printer1	Plotter2	Domain1	Domain2	Domain3
1	Read	Read Write								Enter	
2			Read	Read Write Execute	Read Write		Write				
3						Read Write Execute	Write	Write			

Figure 9-24. A protection matrix with domains as objects.

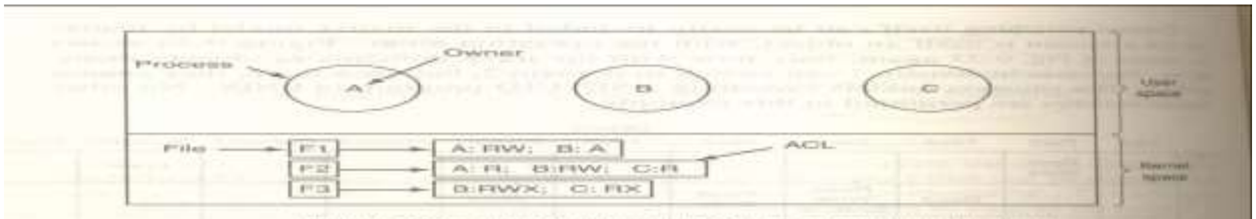


Figure 9-25. Use of access control lists to manage file access.

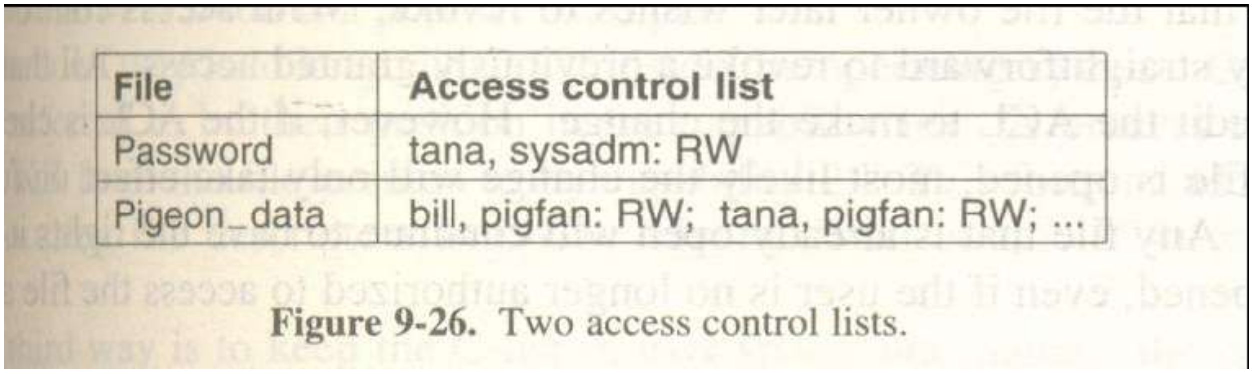
Each file has an ACL associated with it. File *Fl* has two entries in its ACL (separated by a semicolon). The first entry says that any process owned by user *A* may read and write the file. The second entry says that any process owned by user *B* may read the file. All other accesses by these users and all accesses by other users are forbidden. Note that the rights are granted by user, not by process. As far as the protection system goes, any process owned by user *A* can read and write file *F1*

File *F2* has three entries in its ACL: *A*, *B*, and *C* can all read the file, and in addition *B* can also write it. No other accesses are allowed. File *F3* is apparently an executable program, since *B* and *C* can both read and execute it. *B* can also write it.

Many systems sup port the concept of a **group** of users. Groups have names and can be included in ACLs. Two variations on the semantics of groups are possible. In some systems, each process has a user ID (UID) and group ID (GID). In such systems, an ACL entry contains entries of the form

UD1, GID1: rightsl; UD2, G)D2: rights2;

Under these conditions, when a request is made to access an object, a check is made using the caller‘s UID and GID. If they are present in the ACL, the right listed are available. If the (UID, GID) combination is not in the list, the access is not permitted.



File	Access control list
Password	tana, sysadm: RW
Pigeon_data	bill, pigfan: RW; tana, pigfan: RW; ...

Figure 9-26. Two access control lists.

Using groups this way effectively introduces the concept of a role. Consider an installation in which Tana is system administrator, and thus in the group *sysadm.* However, suppose that the company also has some clubs for employees Tana is a member of the pigeon fanciers club. Club members belong to the group *pigfan* and have access to the company‘s computers for managing their pigeon database.

If Tana tries to access one of these files, the result depends on which group is currently logged in as. When she logs in, the system may ask her to choose *which of* her groups she is currently using, or there might even be different login names and/or passwords to keep them separate. The point of this scheme is to prevent Tana from accessing the password file when she currently has her pigeon group. She can only do that when logged in as the system administrator.

In some cases, a user may have access to certain files independent of which, group she is currently logged in as. That case can be handled by introducing wildcards, which mean everyone.

For example, the entry

tana, * :RW

The password file would give Tana access no matter which group she was currently in.

Capabilities

The other way of‘slicing up the matrix of Figure is by rows. When this method is used, associated with each process is a list of objects that may be accessed, along with an indication of which operations are permitted on each domain. This list is called a capability list or C-list and the individual items on it are called capabilities. A set of three processes and their capability lists is shown in Figure.

When capabilities are used, each process has a capability list. Each capability grants the owner certain rights on a certain object. In Figure the process owned by user *A* can read files *F1* and *F2*, for example. Usually, a capability consists of a file identifier and a bitmap for the various rights. In a UNIX-like system, the file identifier would probably be the i-node number. Capability lists are themselves objects and may be pointed to from other capability lists, thus facilitating sharing of subdomains.

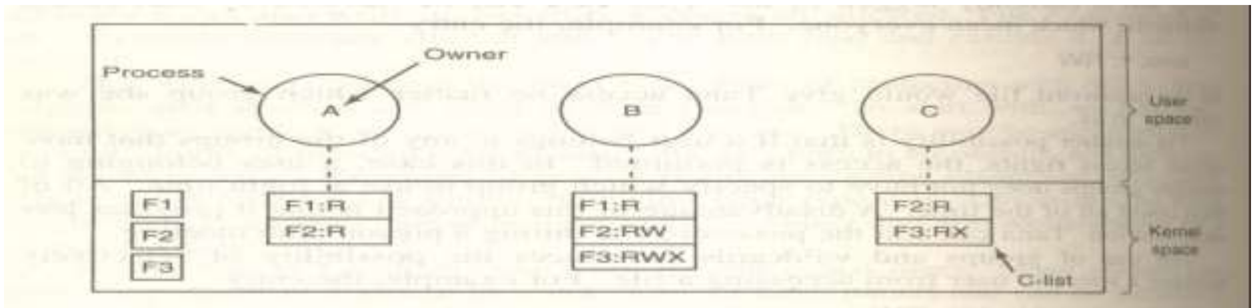


Figure 9-27. When capabilities are used, each process has a capability list.

The capability lists must be protected from user tampering. Three methods of protecting them are known.

The first way requires a tagged **architecture,** a hardware design in which each memory word has an extra (or tag) bit that tells whether the word contains a capability or not. The tag bit is not used by arithmetic, comparison, or similar ordinary instructions and it can be modified by programs running in kernel mode. The IBM *ASI400* is a popular example.

The second way is to keep the C-list inside the operating system. Capabilities are referred to by their position in the capability list. Hydra worked this way.

The third way is to keep the C-list in user space, but manage the capabilities graphically so that users cannot tamper with them. This approach is particularly suited to distributed systems and works as follows. When a client process sends a message to a remote server to create an object for it may, the server creates the object and generates a long random number. A slot in the server‘s file table is reserved for the object and the check field is stored there along with the addresses of the disk blocks. In UNIX terms, the check field is stored on the server in the i-node. It is not sent back to the user and never put on the network. The server then generates and returns a capability to the user of the form shown in Figure.

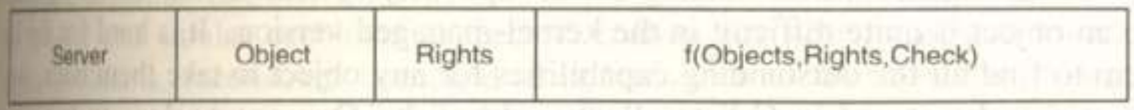


Figure 9-28. A cryptographically-protected capability.

The capability returned to the user contains the server‘s identifier, the object number, and the rights, stored as a bitmap. For a newly created object, all the rights bits are turned on. The last field consists of the concatenation of the object, rights, and check field run through a cryptographically-secure one-way function, f.

When the user wishes to access the object, it sends the capability to the server as part of the request. The server then extracts the object number to index into its tables to find the object. It then computes *f(Object, Rights, Check)* taking the first two parameters from the capability itself and the third one from its own tables. If the result agrees with the fourth field in the capability, the request is honored; otherwise, it is rejected. If a user tries to access someone else‘s object, he will not be able to fabricate the fourth field correctly since he does not know the check field, and the request will be rejected.

A user can ask the server to produce a weaker capability, for read-only access. First the server verifies that the capability is valid. If so, if computes *f (Object,New_rights, Check)* and generates a new capability putting this value in the fourth field. Note that the original *Check* value is used because its code has other outstanding capabilities depend on it**.**

This new capability is sent back to the requesting process. The user can now give this to a friend by just sending it in a message. If the friend turns on rights bits that should be off, the server will detect this when the capability is used since the *f* value will not correspond to the false rights field. Since the friend does not pointed to know

the true check field, he cannot fabricate a capability that corresponds to the false rights bits. This scheme was developed for the Amoeba system.

In addition to the specific object-dependent rights, such as read and execute, On the other capabilities usually have generic rights which are applicable to all objects.

Examples of generic rights are

- Copy capability: create a new capability for the same object.
- Copy object: create a duplicate object with a new capability.
- Remove capability: delete an entry from the C-list; object unaffected,
- Destroy object: permanently remove an object and a capability,

Running code with as few access rights as possible is known as the **principle of least privilege** and is a powerful guideline for producing secure systems.

Briefly summarized, ACLs and capabilities have somewhat complementary properties. Capabilities are very efficient because if a process says " Open the file pointed to by capability 3" no checking is needed. With ACLs, a search of the ACL may be needed. If groups are **not** supported, then granting everyone read access to a file requires enumerating all users in the ACL. Capabilities also allow a process to be encapsulated easily, whereas ACLs do not. On the other hand, ACLs allow selective revocation of rights, which capabilities do not. Finally, if an object is removed and the capabilities are not or the capabilities are removed and an object is not, problems arise. ACLs do not suffer from this problem.

Authentication and Access Control

Multilevel Security

There are two types of access control methods

- Discretionary
- Mandatory

Discretionary Access Control

Most operating systems allow individual users to determine who may read and write their files and other objects. This policy is called **discretionary** access system control.

It leaves a certain amount of access control to the discretion of the object‘s owner or to anyone else who is authorized to control the object‘s access. The owner can determine who should have access right to an object and what those rights should be. DAC access rights can change dynamically.

The security policy of UnixWare prescribes a relationship between access rules and access attributes. The access attributes allow the system to define several distinct levels of authorization, and the access rules provide the mechanism for the system to prevent unauthorized access to sensitive information.

Access to a file is determined by the file's absolute pathname. The kernel determines whether or not to allow a process the kind of file access requested (read, write, execute/search) based on

the user and group IDs associated with the process

the privileges (if any) associated with the process

the discretionary controls (in the form of permission bits, and, possibly, Access Control Lists (ACLs)) associated with the file and all the directories that make up the absolute pathname of the file

These access checks are performed at the time the file is opened, rather than at the time a read or write is actually attempted.

For example, if the file `/usr/src/cmd/mv.c` is readable by a user, but the directory `/usr/src/cmd` (or any other directory in the path) is not searchable by the user (that is, the user does not have search permission on `/usr/src/cmd`), then `mv.c` cannot be read.

The system enforces access control by means of the Discretionary Access Control (DAC) mechanism

Mandatory Access Controls

Mandatory access controls are needed to ensure that the stated security policies are enforced by the system, in addition to the standard discretionary access controls. It regulates the flow of information, to make sure that the information does not leak out.

The Bell-La Padula Model

The most widely used multilevel security model is the **Bell-La Padula model**. This model was designed for handling military security. In the military world, documents (objects) can have a security level, such as unclassified, confidential, secret, and top secret. People are also assigned these levels, pending on which documents they are allowed to see. A process running on behalf of a user acquires the user‘s security level. Since there are multiple security levels, this state of scheme is called a **multilevel security system.**

The Bell-La Padula model has rules about how information can flow: The simple security property

A process running at security level is this: can read only objects at its level or lower. For example, a general that the can read a lieutenant‘s documents but a lieutenant cannot read a general‘s documents.

The * property

A process running at security level *k* can write only objects at its level or higher. For example, a lieutenant can append a message to a general‘s mailbox telling everything he knows, but a general cannot append a message to a lieutenant‘s mailbox telling everything he knows because the general may have seen top secret documents that may not be disclosed to a lieutenant.

The Bell-La Padula model is illustrated graphically in Figure.

In this figure a (solid) arrow from an object to a process indicates that the process is reading the object, that is, information is flowing from the object to the process. Similarly, a (dashed) arrow from a process to an object indicates that the process is writing into the object, that is, information is flowing from the process to the object. Thus all information flows in the direction of the arrows.

The simple security property says that all solid (read) arrows go sideways or up. The * property says that all dashed (write) arrows also go sideways or up. Since information flows only horizontally or upward, any information that starts out at level *k* can never appear at a lower level. In other words, there is never a path that moves information downward, thus guaranteeing the security of the model.

The Biba Model

The problem with the Bell-La Padula model is that it was devised to keep secrets, not guarantee the integrity of the data. To guarantee the integrity of the data, we need precisely the reverse properties.

The simple integrity principle: A process running at security level *k* can write only objects at its level or lower (no write up).

The integrity * property: A process running at security level *k* can read only objects at its level or higher (no read down).

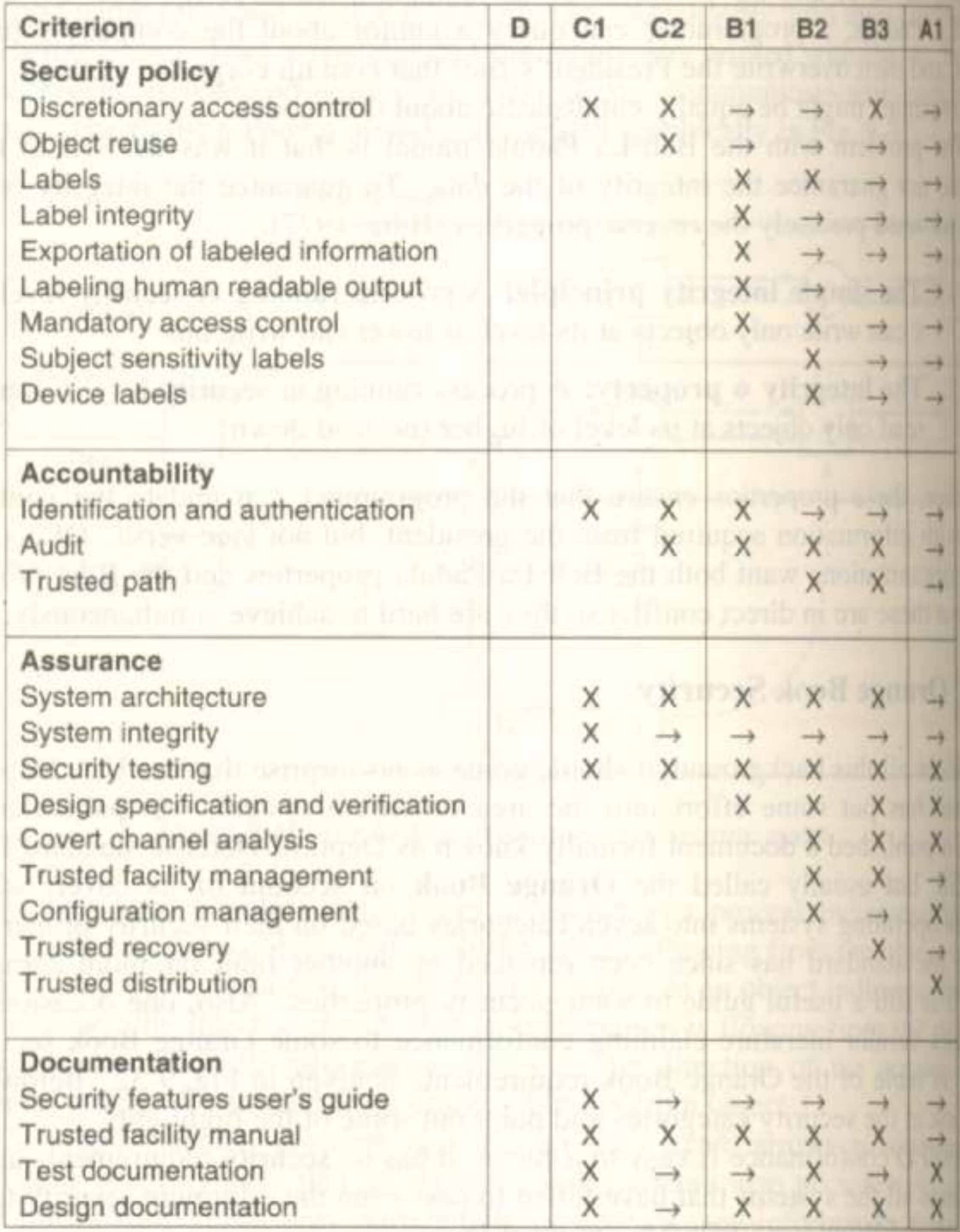
Orange Book Security

It divides the operating systems into seven categories based on their security properties. A table of the Orange Book requirements is given in Figure.

Level D conformance is easy to achieve: it has no security requirements at all. It collects all the systems that have failed to pass even the minimum security tests. MS-DOS and Windows 95/98/Me are level D.

Level C is intended for environments with cooperating users. C1 requires a protected mode operating system, authenticated user login, and the ability for users to specify which files can be made available to other users and how (discretionary access control). Minimal security testing and documentation are also required.

C2 adds the requirement that discretionary access control is down to the level of the individual user. It also requires that objects (e.g., files, virtual memory general‘s pages) given to users must be initialized to all zeros, and a minimal amount of auditing is needed. The UNIX *rwx* scheme meets Cl but does not meet C2.



Criterion	D	C1	C2	B1	B2	B3	A1
Security policy							
Discretionary access control		X	X	→	→	X	→
Object reuse			X	→	→	→	→
Labels				X	X	→	→
Label integrity				X	→	→	→
Exportation of labeled information				X	→	→	→
Labeling human readable output				X	→	→	→
Mandatory access control				X	X	→	→
Subject sensitivity labels					X	→	→
Device labels					X	→	→
Accountability							
Identification and authentication		X	X	X	→	→	→
Audit			X	X	X	X	→
Trusted path					X	X	→
Assurance							
System architecture		X	X	X	X	X	→
System integrity		X	→	→	→	→	→
Security testing		X	X	X	X	X	X
Design specification and verification				X	X	X	X
Covert channel analysis					X	X	X
Trusted facility management					X	X	→
Configuration management					X	→	X
Trusted recovery						X	→
Trusted distribution							X
Documentation							
Security features user's guide		X	→	→	→	→	→
Trusted facility manual		X	X	X	X	X	→
Test documentation		X	→	→	X	→	X
Design documentation		X	→	X	X	X	X

Figure 9-32. Orange Book security criteria. The symbol X means that there are new requirements here. The symbol → means that the requirements from the next lower category also apply here.

The B and A levels require all controlled users and objects to be assigned a security label, such as unclassified, secret, or top secret. The system must be capable of enforcing the Bell-La Padula information flow model.
B2 adds to this requirement that the system has been designed top-down in a modular way. The design must be presented in such a way that it can be verified. The possible covert channels must be analyzed.
B3 contains all of B2‘s features plus there must be ACLs with users and groups, a formal TCB must be presented, adequate security auditing must be presented, and secure crash recovery must be included.
Al requires a formal model of the protection system and a proof that the model is correct. It also requires a demonstration that the implementation conforms to the model. Covert channels must be formally analyzed.

Covert Channels
Even in a system which has a proper security model security leaks can still occur. In this section we discuss ho information can still leak out even when it has been rigorously proven that such leakage is mathematically impossible. These ideas are due to Lampson.
Lampson‘s model it involves three processes on some protected machine.
The first process is the client, which wants some work performed by the second one, the server. The client and the server do not entirely trust each other.
The second is the server is worried that the clients will try to steal the valuable tax program.
The third process is the collaborator, which is conspiring with the server to indeed steal the client‘s confidential data. The collaborator and server are typically owned by the same person.
These three processes are shown in Figure. The object of this exercise is to design a system in which it is impossible for the server process to leak to the collaborator process the information that it has legitimately received from the client process. Lampson called this the **confinement** problem.

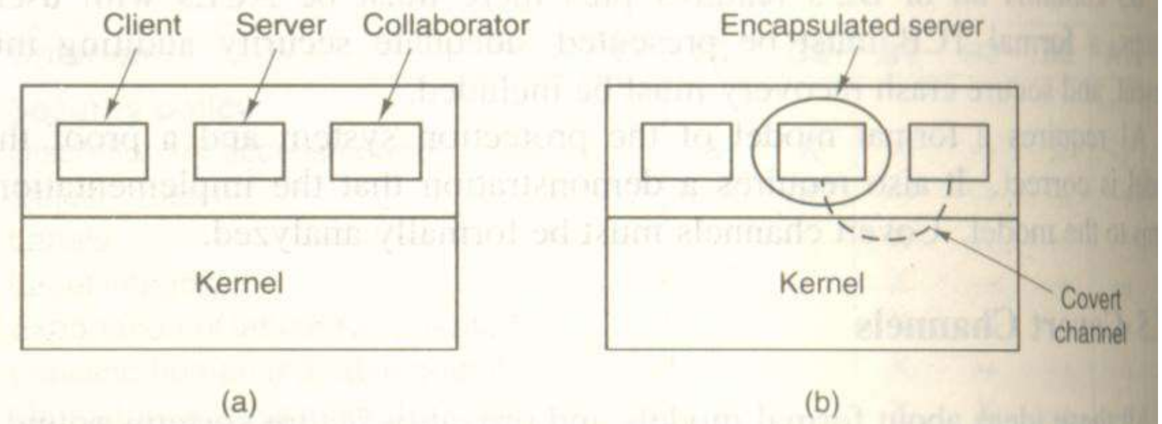


Figure 9-33. (a) The client, server, and collaborator processes. (b) The encapsulated server can still leak to the collaborator. via covert channels.

From the system designer‘s point of view, the goal is to encapsulate or confine the server in such a way that it cannot pass information to the collaborator. Using a protection matrix scheme we can easily guarantee that the server cannot communicate with the collaborator by writing a file to which the collaborator has designed a read access. We can probably also ensure that the server cannot communicate the collaborator using the system‘s inter process communication mechanism.

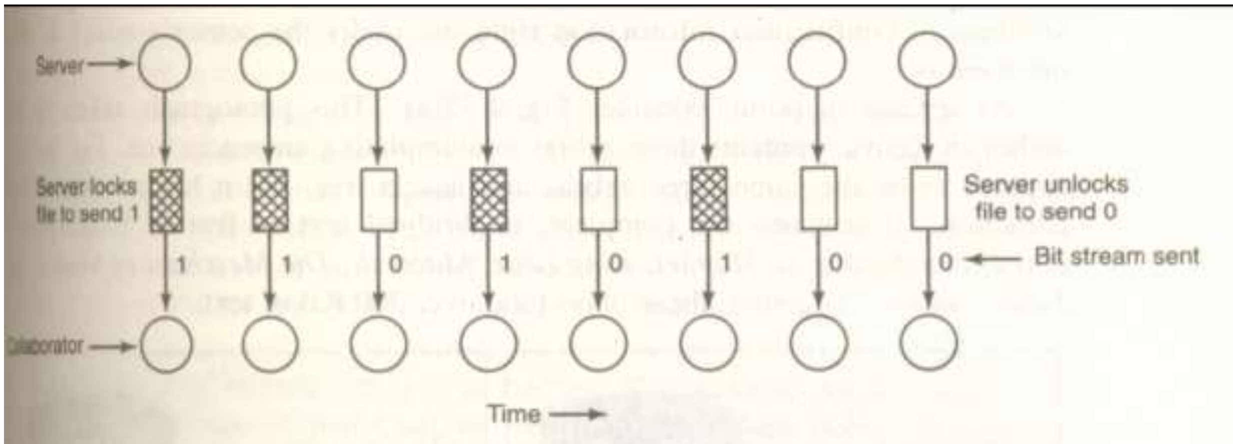


Figure 9-34. A covert channel using file locking.

The covert channel is **a** noisy channel, containing a lot of extraneous information, but information can be reliably sent over a noisy channel by using an error-correcting code.

Authentication Mechanism

Determining the user of the system is called as authentication. This can be provided in three different ways.

- Using password
- Using physical object
- Using biometrics

Authentication using passwords

The widest form of authentication is by typing a login name and password. The simplest implementation just keeps a central list of (login name, password) pairs. The login name typed is looked in the list and password typed is compared to the stored one. If they match login succeeds.

To provide security for the passwords the password will encrypt and save in the list so that even if the cracker breaks in he cannot identify the exact password.

We can do the following steps to improve password security

- Passwords should be a minimum of seven characters
- They should contain both upper and lower case letters
- They should contain at least one digit or special character
- They should not be dictionary words

Authentication using physical objects

Nowadays, the physical object used is often a plastic card that is inserted into a reader associated with the terminal or computer. The user must not only insert the card, but also type in a password, to prevent someone from using a lost or stolen card. Example is ATM card.

Information bearing plastic cards comes in two varieties:

- Magnetic stripe cards
- Chip cards

Magnetic stripe cards hold about 140 bytes of information written on a piece of magnetic tape glued to the back of the card. This information can be read out by the terminal and sent to the central computer. Often the information contains the user‘s password (e.g., PIN code) so the terminal can do an identity need to check even if the link to the main computer is down. Typically the password is encrypted by a key known only to the bank.

Chip cards contain an integrated circuit (chip) on them. These cards can be further subdivided into two categories:

- Stored value cards
- Smart cards

Stored value cards contain a small amount of memory (usually less than 1 KB) using EEPROM technology to allow the value to be remembered when the card is removed from the reader and thus the power turned off. There is no CPU on the card, so the value stored must be changed by an external CPU (in the reader). Example is prepaid telephone card.

However, nowadays, much security work is being focused on the **smart cards** back, which currently have something like a 4-MHz 8-bit CPU, 16 KB of ROM, 4 KB of EEPROM. 512 bytes of scratch RAM, and a 9600-bps communication channel to the reader.

Smart cards can be used to hold money but with much better security and universality. The cards can be loaded with money at an ATM machine or at home over the telephone using a special reader supplied by the bank. When inserted into a merchant‘s reader, the user can authorize the card to deduct a certain amount of money from the card, causing the card to send a little encrypted message to the merchant. The merchant can later turn the message over to a bank to be credited for the amount paid.

The big advantage of smart cards over, say, credit or debit cards, is that they do not need an online connection to a bank.

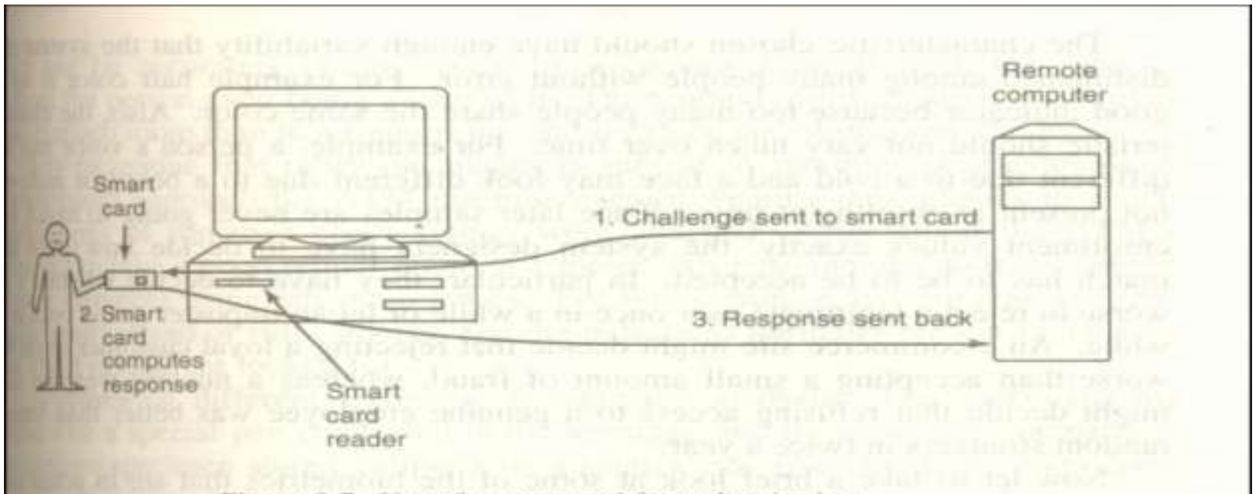


Figure 9-7. Use of a smart card for authentication.

KB of EEPROM. 512 bytes of scratch RAM, and a 9600-bps communication channel to the reader.

Smart cards can be used to hold money but with much better security and universality. The cards can be loaded with money at an ATM machine or at home over the telephone using a special reader supplied by the bank. When inserted into a merchant‘s reader, the user can authorize the card to deduct a certain amount of money from the card, causing the card to send a little encrypted message to the merchant. The merchant can later turn the message over to a bank to be credited for the amount paid.

The big advantage of smart cards over, say, credit or debit cards, is that they do not need an online connection to a bank

A**uthentication Using Biometrics**

The third authentication method measures physical characteristics of the user that are hard to forge. These are called biometrics. For simple example, a fingerprint or a voiceprint reader in the terminal could verify the user‘s identity.

A typical biometrics system has two parts:

- ➢ Enrollment
- ➢ Identification

During enrollment, the user‘s characteristics are measured and the results digitized. Then significant features are extracted and stored in a record associated with the

user. The record can be kept in a central database or stored on a smart card that the user carries around and inserts into a remote reader (e.g., at an ATM machine).
The other part is identification. The user shows up and provides a login name. Then the system makes the measurement again. If the new values match the ones sampled at enrollment time, the login is accepted; otherwise it is rejected. The login name is
needed because the measurements are not exact, so it is difficult to a index them and then search the index. Also, two people may have the same characteristics, so requiring the measured characteristics to match those of a specific user is stronger than just requiring it to match those of any user. The characteristic chosen should have enough variability that the system can distinguish among many people without error.

Biometrics that are in use

Finger length analysis

When this is used, each terminal has a device like the one of Figure. The user inserts his hand into it, and the length of all his fingers is measured and checked against the database. Finger length measurements are not perfect however. The system can he attacked with hand molds made out of plaster of Paris or some other material, possibly with adjustable fingers to allow some experimentation.

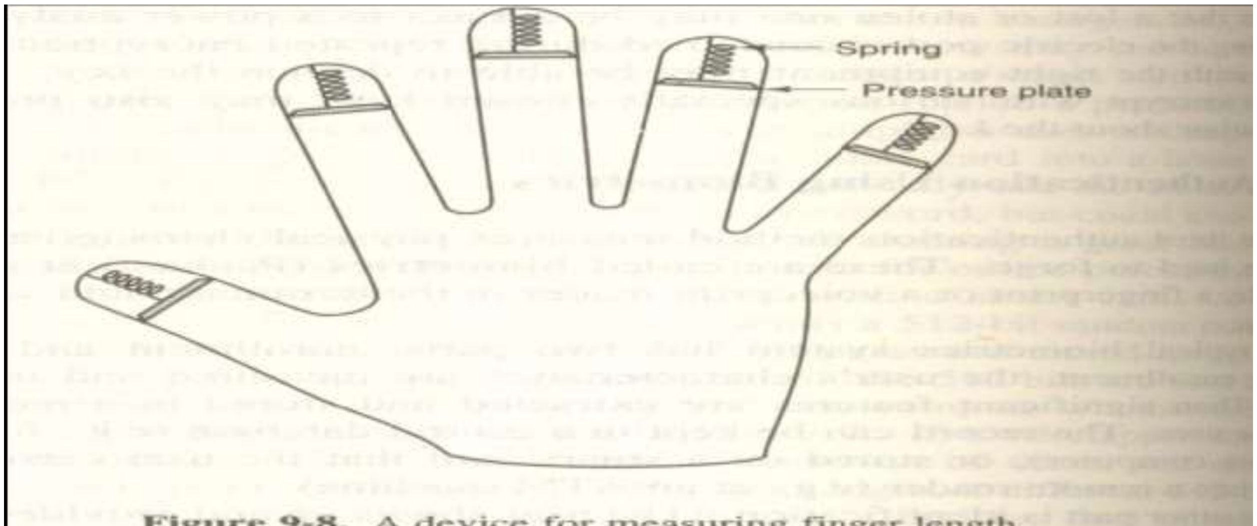


Figure 9-8. A device for measuring finger length.

Retinal pattern analysis

Every person has a different pattern of retinal blood vessels, even identical twins. They can be accurately photographed by a camera 1 meter from the subject, without the person even being aware of it. The amount of information in a retinal scan may be much more than in a fingerprint, and it can be coded in about *256* bytes.

Signature analysis

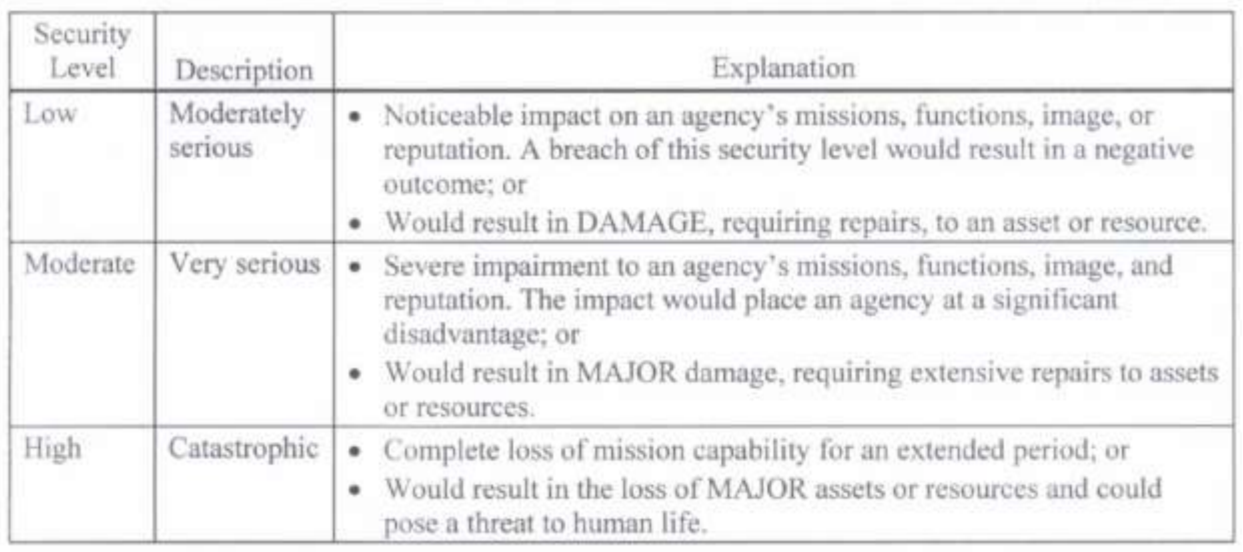
The user signs his same with a special pen connected to the terminal, and the computer compares it to a known specimen stored online or on a smart card. Even better is not to compare the signature, but compare the pen motions and pressure made while writing to it. A good forger may be able to copy the signature, but will not have a clue as to exact order in which the strokes were made or at what speed and what pressure.

Voice biometrics

In this we need a microphone (or even a telephone); the rest is software. In contrast to voice recognition systems, which try to determine what the speaker is saying, these systems try to determine who the speaker is. Some systems just require the user to say a secret password, but these can be defeated by an eavesdropper who can tape record passwords and play them back later.

Official levels of computer security

System owners must determine the appropriate system security level based on the confidentiality, integrity and availability of the information, as well as its criticality to the agency‘s business mission. This is the basis for assessing the risks to CMS operations and assets and in selecting appropriate security controls and techniques. CMS Standard for Information Security Levels establishes common criteria for security levels by information category. The first table defines the information security levels. The second table lists security levels for the various information categories. (Note: that mission critical information is its own category). In other words, the system owner locates his information
category to find the appropriate system security level. In the cases where information of varying security levels is combined, the highest security level takes precedence. Where system availability or data integrity are of high importance, see the table footnote.



Information Security Levels

Security Level	Description	Explanation
Low	Moderately serious	• Noticeable impact on an agency's missions, functions, image, or reputation. A breach of this security level would result in a negative outcome; or • Would result in DAMAGE, requiring repairs, to an asset or resource.
Moderate	Very serious	• Severe impairment to an agency's missions, functions, image, and reputation. The impact would place an agency at a significant disadvantage; or • Would result in MAJOR damage, requiring extensive repairs to assets or resources.
High	Catastrophic	• Complete loss of mission capability for an extended period; or • Would result in the loss of MAJOR assets or resources and could pose a threat to human life.

This level is based on data sensitivity requirements for the information system. The low system security level may be increased to moderate (not to high) if‘ the information system has significant integrity and/or availability requirements. The moderate level cannot be increased to high.

Security Breaches

The report on security breaches shows that the major attack is on access controls, which ensure that only authorized individuals can read, alter, or delete data; and configuration management controls, which provide assurance that only authorized software programs are implemented. Organizations can reduce the risks associated with intrusions and misuse if they take steps to detect and respond to incidents before significant damage occurs, analyze the causes and effects of incidents and apply the lessons learned.

The recommendations by the Identity Theft Task Force to reduce the security breaches are given below.

1. Reduce the unnecessary use of Social Security numbers by federal agencies,
2. Establish national standards that require private sector entities to safeguard the personal data they compile and maintain and to provide notice to consumers when a breach occurs that poses a
3. Significant risk of identity theft.

4. Implement a broad, sustained awareness campaign by federal agencies to educate consumers, the private sector, and the public sector on methods to deter, detect, and defend against identity theft
5. Create a National Identity Theft Law Enforcement Center to allow law enforcement agencies to coordinate their efforts and information more efficiently, and investigate and prosecute identity thieves
6. Most of the security breaches are in educational institutions and others are in financial institutions.

Holes

A hole is any feature of hardware or software that allows unauthorized users to gain access or increase their level of access without authorization. A hole could be virtually anything. For example, many peculiarities of hardware or software commonly known to all users qualify as holes. One such peculiarity is that CMOS passwords on IBM compatibles are lost when the CMOS battery is shorted, disabled, or removed. Even the ability to boot into single-user mode on a workstation could be classified as a hole. This is so because it will allow a malicious user to begin entering interactive command mode, seizing control of the machine.

So a hole is nothing more than some form of vulnerability. Every platform has holes, whether in hardware or software. In short, nothing is absolutely safe. You might draw the conclusion that no computer system is safe and that the entire Net is nothing but one big hole.

The Vulnerability Scale

There are different types of holes, including

- ➢ Holes that allow denial of service
- ➢ Holes that allow local users with limited privileges to increase those privileges without authorization
- ➢ Holes that allow outside parties (on remote hosts) unauthorized access to the network

These types of holes and attacks can be rated according to the danger they pose to the victim host.

Types of Holes

1. Holes that allow denial of service (Class C)
2. Holes That Allow Local Users Unauthorized Access (Class B)

3. Holes That Allow Remote Users Unauthorized Access (Class **A)**

Holes That Allow Denial of Service

Holes that allow denial of service are in category C, and are of low priority. These attacks are almost always operating-system based. That is, these holes exist within the *networking portions of the operating system* itself. When such holes exist, they must generally be corrected by the authors of the software or by patches from the vendor.

For large networks or sites, a denial-of-service attack is of only limited significance. It amounts to a nuisance and no more. Smaller sites, however, may suffer in a denial-of-service attack. This is especially so if the site maintains only a single machine. These occur most often in the form of attacks like syn_flooding. An excellent definition of denial-of-service attacks is given in a popular paper called "Protecting Against TCP SYN Denial of Service Attacks":

Denial of Service attacks are a class of attack in which an individual or individuals exploit aspects of the Internet Protocol suite to deny other users of legitimate access to systems and information. The TCP SYN attack is one in which connection requests are sent to a server in high volume, causing it to become overwhelmed with requests. The result is a slow or unreachable server, and upset customers.

The syn_flooder attack is instigated by creating a high number of half-open connections. Because each connection opened must be processed to its ultimate conclusion, the system is temporarily bogged down. This appears to be a problem inherent in the design of the TCP/IP suite, and something that is not easily remedied.

This hole, then, exists within the heart of the networking services of the UNIX operating system. Thus, although efforts are underway for fixes, I would not classify this as a high priority. This is because in almost all cases, denial-of-service attacks represent no risk of penetration.

There are other forms of denial-of-service attacks. Certain denial-of-service attacks can be implemented against the individual user as opposed to a network of users. These types of attacks do not really involve any bug or hole rather, these attacks take advantage of the basic design of the WWW.

For example, suppose I harbored ill feelings toward users of Netscape Navigator. Using either Java or JavaScript, I could effectively undertake the following actions:

1. Configure an inline or a compiled program to execute on load, identifying the type of browser used by the user.

2. If the browser is Netscape Navigator, the program could spawn multiple windows, each requesting connections to different servers, all of which start Java applets on load.

In fewer than 40 seconds, the target machine would come to a grinding halt.
One particularly irritating denial-of-service attack is the dreaded CHARGEN attack. CHARGEN is a service that runs on port 19. It is a character generator (hence the name) used primarily in debugging. Many administrators use this service to determine whether packets are being inexplicably dropped or where these packets disappear before the completion of a given TCP/IP transaction.

Holes That Allow Local Users Unauthorized Access

Still higher in the hole hierarchy (class B) are those holes that allow local users to gain increased and unauthorized access. These types of holes are typically found within applications on this or that platform.
A *local user* is someone who has an account on the target machine or network. A typical example of a local user is someone with shell access to his ISP's box. If he has an e-mail address on a box and that account also allows shell access, that "local" user could be thousands of miles away.

sendmail

A fine example of a hole that allows local users increased and unauthorized access is a well-known sendmail problem. sendmail is perhaps the world's most popular method of transmitting electronic mail. It is the heart of the Internet's e-mail system. Typically, this program is initiated as a daemon at boot time and remains active as long as the machine is active. In its active state, sendmail listens (on port 25) for deliveries or other requests from the void.
When sendmail is started, it normally queries to determine the identity of the user because only root is authorized to perform the startup and maintenance of the sendmail program. Other users with equivalent privileges may do so, but that is the extent of it. However, according to the CERT advisory titled "Sendmail Daemon Mode Vulnerability":

Unfortunately, due to a coding error, sendmail can be invoked in daemon mode in a way that bypasses the built-in check. When the check is bypassed, any local user is able to start sendmail in daemon mode. In addition, as of version 8.7, sendmail will restart itself when it receives a SIGHUP signal. It does this restarting operation by re-executing itself using the exec(2) system call. Re-executing is done as the root user. By manipulating the sendmail environment, the user can then have sendmail execute an arbitrary program with root privileges.

Thus, a local user can gain a form of root access. These holes are quite common. One surfaces every month or so. sendmail is actually renowned for such holes, but has no monopoly on the phenomenon. These types of holes represent a serious threat for one reason: If a local user successfully manages to exploit such a hole, the system administrator may never discover it. Also, leveraged access is far more dangerous in the hands of a local user than an outsider. This is because a local user can employ basic system utilities to learn more about the local network. Such utilities reveal far more than any scanner can from the void. Therefore, a local user with even fleeting increased access can exploit that access to a much greater degree.

Other Class B Holes

Most class B holes arise from some defect within an application. There are some fairly common programming errors that lead to such holes. One such error concerns the character buffer in programs written in C.

What happens when you try to stuff more data into a buffer (holding area) than it can handle. This may be due to a mismatch in the processing rates of the producing and consuming processes or because the buffer is simply too small to hold all the data that must accumulate before a piece of it can be processed.

Programs written in C often use a *buffer*. Flatly stated, a buffer is an abstraction, an area of memory in which some type of text or data will be stored. Programmers make use of such a buffer to provide pre-assigned space for a particular block or blocks of data. For example, if one expects the user to input his first name, the programmer must decide how many characters that first name buffer will require. This is called the size of the character buffer. Thus, if the programmer writes:

```
char first_name[20];
```

This is allowing the user 20 characters for a first name. But suppose the user's first name has 35 characters. What happens to the last 15 characters? They overflow the character buffer. When this overflow occurs, the last 15 characters are put somewhere in memory, at another address

Crackers, by manipulating where those extra characters end up, can cause arbitrary commands to be executed by the operating system. Most often, this technique is used by local users to gain access to a root shell.

Holes That Allow Remote Users Unauthorized Access (Class A)

Class A holes are the most threatening of all and not surprisingly, most of them stem from either poor system administration or misconfiguration. Vendors rarely overlook those holes that allow remote users unauthorized access. At this late stage of the game, even vendors that were previously not security minded have a general grasp of the terrain.

The typical example of a misconfiguration (or configuration failure) is any sample script that remains on the drive, even though the distribution docs advise that it be removed. One such hole has been rehashed innumerable times on the Net. It involves those files included within Web server distributions.

Most Web server software contains fairly sparse documentation. A few files may exist, true, and some may tout themselves as tutorials. Nonetheless, as a general rule, distributions come with the following elements:

- Installation instructions
- The binaries
- In some rare cases, the source
- Sample configuration files with comments interspersed within them, usually commented out within the code
- Sample CGI scripts

Naturally, these holes pose a significant danger to the system from outside sources. In many cases, if the system administrator is running only minimal logs, these attacks may go unrecorded.

Other Holes

In the preceding paragraphs, I named only a few holes. This might give you the erroneous impression that only a handful of programs have ever had such holes. This is untrue. Holes have been found in nearly every type of remote access software at one stage or another. The list is very long indeed. Here is a list of some programs that have been found (over the years) to have serious class A holes:

- FTP
- Gopher
- Telnet
- sendmail
- NFS
- ARP
- Portmap
- finger

In addition to these programs having class A holes, all of them have had class B holes as well.

SECURITY IN UNIX

UNIX has been a multiuser system almost from the beginning.

Fundamental Concepts

The user community for a UNIX system consists of some number of registered users, each of whom has a unique **UID (User ID). A** UID is an integer between 0 and 65,535. Files (but also processes, and other resources) are marked with the UID of their owner. By default, the owner of a file is the person who created the file, though there is a way to change ownership.

Users can be organized into groups, which are also numbered with 16-bit integers called GIDs **Group IDs.** Assigning users to groups is done manually, by the system administrator, and consists of making entries in a system database telling which user is in which group. Originally, a user could be in only one group. UNIX now allows a user to be in more than one group at the same time.

The basic security mechanism in UNIX is simple. Each process carries the UID and GID of its owner. When a file is created, it gets the UID and GID of the creating process. The file also gets a set of permissions determined by the creating process. These

permissions specify what access the owner, the other members of the owner‘s group, and the rest of the users have to the file. For each of these three categories, potential accesses are read, write, and execute, designated by the letters *r, w,* and x, respectively. The ability to execute a file makes sense only if that file is an executable binary program, of course. An attempt to execute a file that has

executed permission but which is not executable will fail with an error. Since there are three categories of users and 3 bits per category, 9 bits are sufficient to represent the access rights.
The first two entries in Figure are clear, allowing the owner and the owner‘s group full access, respectively. The next one allows the owner‘s group to read the file but not to change it, and prevents outsiders from any access. The fourth entry is common for a data file the owner wants to make public. Similarly, the fifth entry is the usual one for a publicly available program. The sixth entry denies all access to all users. This mode is sometimes used for dummy files used for mutual exclusion because an attempt to create such a file will fail if one already exists. Thus if multiple processes simultaneously attempt to create such a file as a lock, only one of them will succeed. The last example is strange indeed, since it gives the rest of the world more access than the owner. However, its existence follows from the protection rules. Fortunately, there is a way for the owner to subsequently change the protection mode, even without having any access to the file itself.
The user with UTD 0 is special and is called the **superuser** (or root). The super user has the power to read and write all files in the system, no matter who owns them and no matter how they are protected. Processes with UID 0 also have the ability to make a small number of protected system calls denied to ordinary users. Normally, only the system administrator knows the superuser‘s password. Directories are files and have the same protection modes that ordinary files do except that the x bits refer to search permission instead of execute permission, Thus a
directory with mode *rwxr—xr—x* allows its owner to read, modify, and search the directory, but allows others only to read and search it, but not add or inove files from it.
Special files corresponding to the **I/O** devices have the same protection bits as regular files.
The problems associated with I/O files are solved by allowing controlled access to all I/O devices and other system resources. This problem was solved by adding a new protection bit, the **SETUID bit** to the 9 protection bits. When a program with the SETUID bit on is executed, the effective **UID** for that process becomes the UID of the executable file‘s owner instead of the UID of the user who invoked it. When a process attempts to open a file, it is the effective UID that is checked, not the underlying real UID. By making the program that accesses the printer be

owned by daemon but with the SETUID bit on, any user could execute it, and have the power of daemon but only to run that program
In addition to the SETUID bit there is also a SETGID bit that works analogously, temporarily giving the user the effective GID of the program. In practice, this bit is rarely used.

Security System Calls in UNIX

There are only a small number of system calls relating to security. The most important ones are listed in Figure. The most heavily used security system call is chmod. It is used to change the protection mode. For example,
s = chmod(‖/usr/ast/newgame, 0755);
sets *newgame* to *rwxr—xr—x* so that everyone can run it. Only the owner of a file and the superuser can change its protection bits.
The access call tests to see if a particular access would be allowed using the real UID and GID. This system call is needed to avoid security breaches in programs that are SETUID and owned by the root. Such a program can do anything, and it is sometimes needed for the program to figure out if the user is allowed to perform a certain access. The program cannot just try it, because the access will always succeed. With the access call the program can find out if the access is allowed by the real UID and real GID.
The next four system calls return the real and effective UIDs and GIDs. The last three are only allowed for the superuser. They change a file‘s owner, and a process‘ UID and GID.

Implementation of Security in UNIX

When a user logs in, the login program, *login* (which is SETUID root) asks **for a login name and a** password. It hashes the password and then looks in the password file, */etc/passwd,* to see if the hash matches the one there (networked systems work slightly differently). The reason for using hashes is to prevent the password from being stored in unencrypted form anywhere in the system. If the password is correct, the login program looks in */etc/passwd* to see the name of the user‘s preferred shell, possibly *sh,* but possibly some other shell such as *csh* or *ksh.* The login program then uses setuid and setgid to give itself the user‘s UID and GID. Then it opens the keyboard for standard input (file descriptor 0), the screen for standard output (file descriptor 1), and the screen for standard error (file descriptor 2). Finally, it executes the preferred shell, thus terminating itself.

At this point the preferred shell is running with the correct UID and GID and standard input, output, arid error all set to their default devices. All processes that it forks off automatically inherit the shell‘s UID and GID, so they also will have the correct owner and group. All files they create also get these values. When any process attempts to open a file, the system first checks the protection bits in the file‘s i-node against the caller‘s effective UID and effective GID to *see* if the access is permitted. If so, the file is opened and a file descriptor returned. If not, the file is not opened and -1 is returned. No checks are made on subsequent read or write calls. As a consequence, if the protection mode changes after a file are already open, the new mode will not affect processes that already have the file open. Security in Linux is essentially the same as in UNIX.

Security in Windows 2000

The security features inherited from Windows NT by Windows 2000 is given below,

- ➢ Secure login with antispoofing measures.
- ➢ Discretionary access controls.
- ➢ Privileged access controls.
- ➢ Address space protection per process.
- ➢ New pages must be zeroed before being mapped in.
- ➢ Security auditing.

Secure login means that the system administrator can require all users to have a password in order to log in. Spoofing is when a malicious user writes a program that displays the login prompt or screen and then walks away from the computer in the hope that an innocent user will sit down and enter a name and password. The name and password are then written to disk and
the user is told that login has failed. Windows 2000 prevents this attack by instructing users to hit CTRLALT-DEL to log in. This key sequence is .always captured by the keyboard driver, which then invokes a system program that puts up the genuine login screen. This procedure works because there is no way for user processes to disable CTRL-ALT-DEL processing in the keyboard driver.

Discretionary access controls allow the owner of a file or other object to say who can use it and in what way. Privileged access controls allow the system administrator (superuser) to override them when needed. Address space protection means that each process has its own protected virtual space accessible by an unauthorized process. The next item means that when a stack grows, the pages mapped in are mi ma ized to zero so processes cannot find any old information put there by the previous owner. Finally, security auditing allows the administrator to produce a log of certain security related events.

Fundamental Concepts

Every Windows 2000 user (and group) is identified by **a SID (Security ID).** SIDs are binary numbers with **a** short header followed by **a** long random component. Each SID is intended to be unique worldwide. When a user starts up a process, the process and its threads run under the user‘s SID. Most of the security system is designed to make sure that each object can be accessed only by threads a with authorized SIDs. Each process has an access **token** that specifies its SID and other properties. **0** It is normally assigned at login time by *winlogon*, although processes should call *GetTokeninformation* to acquire this information since it may change in the future. The header contains some administrative information. The expiration time field could tell when the token ceases to be valid, but it is currently not used. The *Groups* fields specify the groups to which the process belongs; this is needed for POSIX conformance. The default **DACL** (Discretionary **ACE)** is the access control list assigned to objects created by the process if no other ACL is specified. The user SID tells who owns the process. The restricted SIDS are to allow untrustworthy processes to take part in jobs with trustworthy processes but with less power to do damage.

Finally, the privileges listed, if any, give the process special powers, such as the right to shut the machine down or access files to which access would otherwise be denied. In effect, the privileges split up the power of the superuser into several rights that can be assigned to processes individually. In this way, a user can be given some superuser power, but not all of it. The access token tells who owns the process and which defaults and powers are associated with it.

Structure of an access token When a user logs in, *winlogon* gives the initial process an access token. Subsequent processes normally inherit this token on down the line. A process‘ access token initially applies to all the threads in the process.

However, a thread can acquire a different access token during execution, in which case the thread‘s access token overrides the process‘ access token. In particular, a client thread can pass its access token to a server thread to allow the server to access the client‘s protected files and other objects. This mechanism is called **impersonation.**

Another basic concept is the security **descriptor.** Every object has a security descriptor associated with it that tells who can perform which operations on it. A security descriptor consists of a header followed by a DACL with one or more ACEs (Access **Control** Elements). The two main kinds of elements are Allow and Deny. An allow element specifies a S1D and a bitmap that specifies which operations processes with that SID may perform on the object. A deny element works the same way, except a match means the caller may not perform the operation.

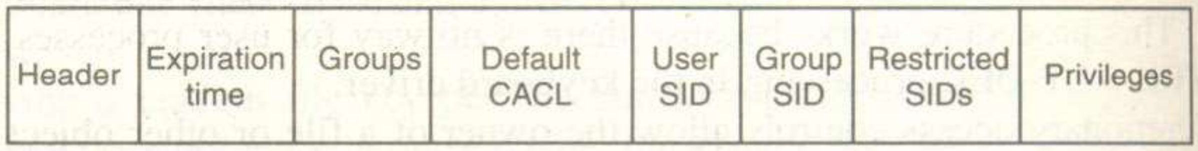


Figure 11-42. Structure of an access token .

Structure of an access token When a user logs in, *winlogon* gives the initial process an access token. Subsequent processes normally inherit this token on down the line. A process‘ access token initially applies to all the threads in the process.

However, a thread can acquire a different access token during execution, in which case the thread‘s access token overrides the process‘ access token. In particular, a client thread can pass its access token to a server thread to allow the server to access the client‘s protected files and other objects. This mechanism is called **impersonation.**

Another basic concept is the security **descriptor.** Every object has a security descriptor associated with it that tells who can perform which operations on it. A security descriptor consists of a header followed by a DACL with one or more ACEs (Access **Control** Elements). The two main kinds of elements are Allow and Deny. An allow element specifies a S1D and a bitmap that specifies which operations processes with that SID may perform on the object. A deny element

works the same way, except a match means the caller may not perform the operation.

For example, Ida has a file whose security descriptor specifies that everyone has read access, Elvis has no access. Cathy has read/write access, and Ida herself has full access. This simple example is illustrated in Figure. The SID Everyone refers to the set of all users, but it is overridden by any explicit ACEs that follow.

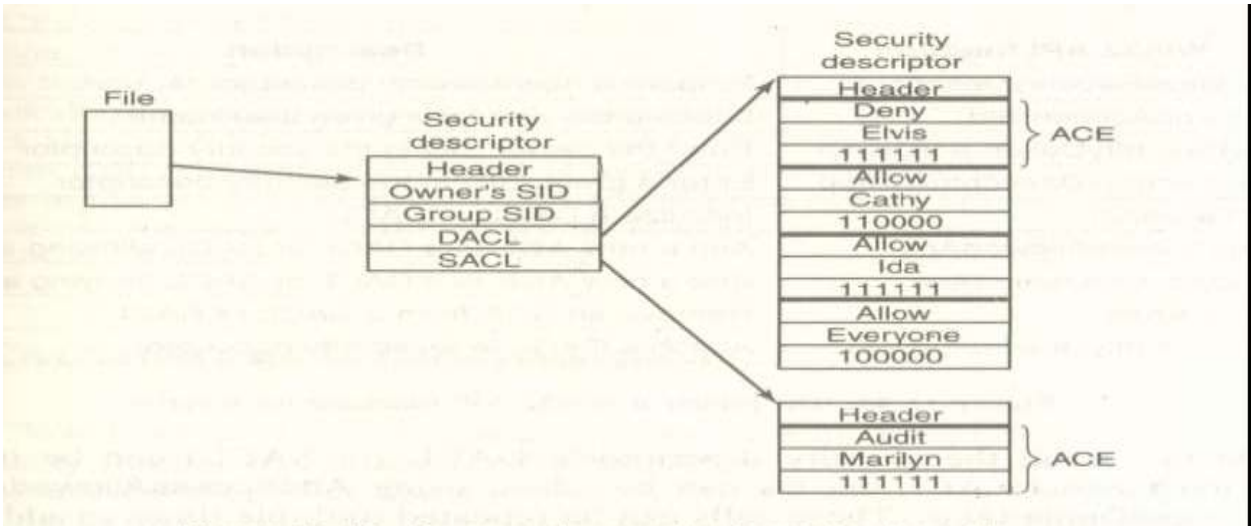
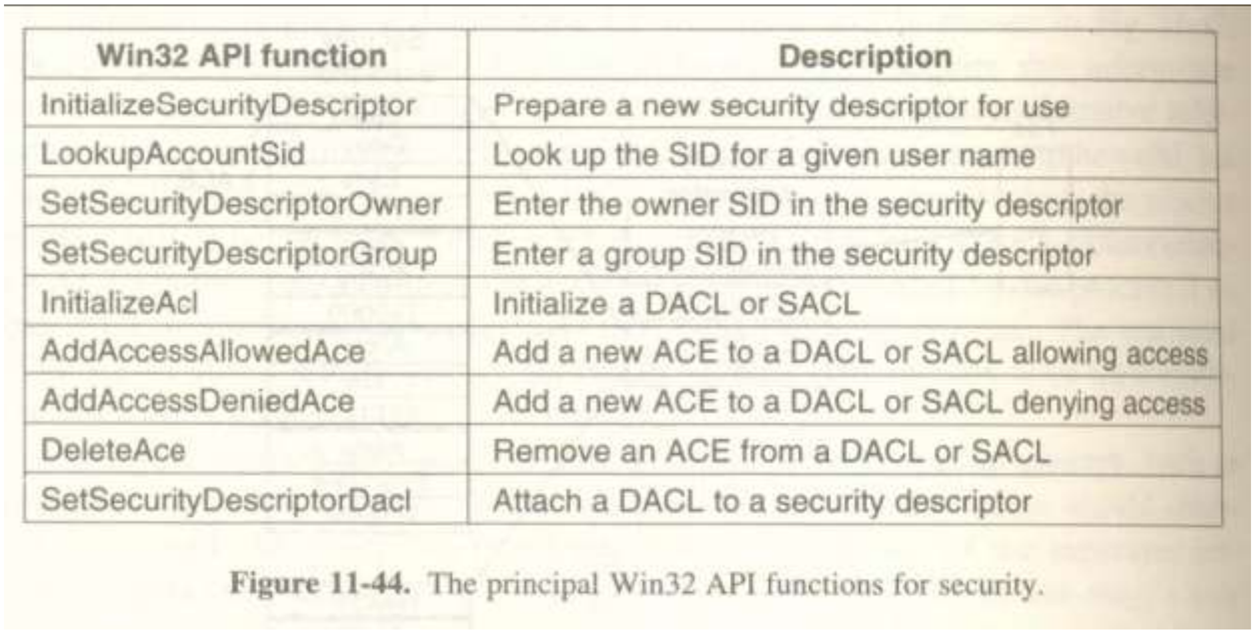


Figure 11-43. An example security descriptor for a file.

In addition to the DACL, a security descriptor also **has a SACL (System** Access **Control** list), which is like a DACL except that it specifies not who may use the object, but which operations on the object are recorded in the system-wide security event log.

Security API Calls

Most of the Windows 2000 access control mechanism is based on security descriptors. The usual pattern is that when a process creates an object, it provides a security descriptor as one of the parameters to the CreateProcess, CreateRe, or other object creation call. This security descriptor then becomes the security descriptor attached to the object. If no security descriptor is provided in the object creation call, the default security in the caller‘s access token is used instead. Many of the Win32 API security calls relate to the management of security



Win32 API function	Description
InitializeSecurityDescriptor	Prepare a new security descriptor for use
LookupAccountSid	Look up the SID for a given user name
SetSecurityDescriptorOwner	Enter the owner SID in the security descriptor
SetSecurityDescriptorGroup	Enter a group SID in the security descriptor
InitializeAcl	Initialize a DACL or SACL
AddAccessAllowedAce	Add a new ACE to a DACL or SACL allowing access
AddAccessDeniedAce	Add a new ACE to a DACL or SACL denying access
DeleteAce	Remove an ACE from a DACL or SACL
SetSecurityDescriptorDacl	Attach a DACL to a security descriptor

Figure 11-44. The principal Win32 API functions for security.

descriptors, so we will focus on those here. The most important calls are listed in Figure. To create a security descriptor storage for it is first allocated and then initialized using using InitializeSecurityDescriptor. This call fills in the header. If the **owner SID is** not known, it can be looked up by name using LookupAccountSid. **It** can **then be** inserted into the security descriptor. The same holds for the group **SID,** if any. Normally, these will be the caller‘s own SID and one of the caller‘s groups, but the system administrator can fill in any SIDs.

At this point the security descriptor‘s DACL (or SACL) can be initialized with InitializeAcl. ACL entries can be added using AddAccessAllowedAce, and AddAccessDeniedAce. These calls can be repeated multiple times to add as many ACE entries as are needed. DeleteAce can be used to remove an entry, more like on an existing ACL than on one being constructed for the first time. When the ACL is ready, SetSecurityDescriptorDaci can be used to attach it to the security descriptor. Finally, when the object is created, the newly minted security descriptor can be passed as a parameter to have it attached to the object.

Implementation of Security

Security in a standalone Windows 2000 system is implemented by a number of components, most of which we have already seen. Logging in is handled by *winlogon* and authentication is handled by *isass* and *msgina.dll.* The result of a successful login is a new shell with its associated access token. This process uses the SECURITY and SAM keys in the registry. The former sets the general security policy and the latter contains the security information for the individual users. Once a user is logged in, security operations happen when an object is opened for access. Every OperiXXX call requires the name of the object being opened and the set of rights needed. During processing of the open, the security manager checks to see if the caller has all the rights required. It performs this check by looking at the caller‘s access token and the DACL associated with the object. It goes down the list of entries in the ACL in order. As soon as it finds an entry that matches the caller‘s SID or one of the caller‘s groups, the access found there is taken as definitive. If all the rights the caller needs are available, the open succeeds; otherwise it fails.

DACLs can have Deny entries as well as Allow entries, as we have seen. For this reason, it is usual to put entries denying access ahead of entries granting access in the ACL, so that a user who is specifically denied access cannot get in via a back door by being a member of a group that has legitimate access.

After an object has been opened, a handle to it is returned to the caller. On subsequent calls, the only check that is made is whether the operation now being tried was in the set of operations requested at open time, to prevent a caller from opening a file for reading and then trying to write on it. Any log entries required by the SACL are made.