CHAPTER 11

Security

11.1 INTRODUCTION

Computer systems store large amounts of information, some of which is highly sensitive and valuable to their users. Users can trust the system and rely on it only if the various resources and information of a computer system are protected against destruction and unauthorized access. Obviously, the security requirements are different for different computer systems depending on the environment in which they are supposed to operate. For example, security requirements for systems meant to operate in a military environment are different from those for systems that are meant to operate in an educational environment. The security goals of a computer system are decided by its security policies, and the methods used to achieve these goals are called security mechanisms. While designing the security of a system, it is often useful to distinguish between security policies and security mechanisms because security policies can be decided independent of the available technology but security mechanisms are influenced by the available technology. It may be difficult to implement the desired security policies with a selected set of security mechanisms. But new security mechanisms can be later added to the set to implement the desired security policies.

Irrespective of the operation environment, some of the common goals of computer security are as follows [Mullender 1985]:

1. *Secrecy.* Information within the system must be accessible only to authorized users.
2. *Privacy.* Misuse of information must be prevented. That is, a piece of information given to a user should be used only for the purpose for which it was given.
3. *Authenticity.* When a user receives some data, the user must be able to verify its authenticity. That is, the data arrived indeed from its expected sender and not from any other source.
4. *Integrity.* Information within the system must be protected against accidental destruction or intentional corruption by an unauthorized user.

A total approach to computer security involves both external and internal security. *External security* deals with securing the computer system against external factors such as fires, floods, earthquakes, stolen disks/tapes, leaking out of stored information by a person who has access to the information, and so on. For external security, the commonly used methods include maintaining adequate backup copies of stored information at places far away from the original information, using security guards to allow the entry of only authorized persons into the computer center, allowing the access to sensitive information to only trusted employees/users, and so on.

Internal security, on the other hand, mainly deals with the following two aspects:

1. *User authentication.* Once a user is allowed physical access to the computer facility, the user's identification must be checked by the system before the user can actually use the facility.

2. *Access control.* A computer system contains many resources and several types of information. Obviously, not all resources and information are meant for all users. Therefore, even when a user passes the authentication phase and is allowed to use the computer facility, a way is needed to prohibit the user from accessing those resources/information that he or she is not authorized to access. In fact, a secure system requires that at any time a subject (person or program) should be allowed to access only those resources that it currently needs to complete its task. This requirement is commonly referred to as the *need-to-know principle* or the *principle of least privilege.*

We saw that the security needs of a computer system are intricately linked with that system's environment, use, and implementation. Therefore, in addition to the two aspects mentioned above, internal security in distributed systems has a third aspect called communication security.

3. *Communication security.* In a distributed system, the communication channels that are used to connect the computers are normally exposed to attackers who may try to breach the security of the system by observing, modifying, or disrupting the communications. Wireless networks are even more vulnerable to monitoring by intruders

because anyone with a scanner can pluck the radio signals out of the air without being detected. Communication security safeguards against unauthorized tampering of information while it is being transmitted from one computer to another through the communication channels. Two other aspects of communication security are *authenticity of communicating entities* and *integrity of messages*. That is, the sender of a message wants to know that the message was received by the intended receiver, and the receiver wants to know that the message was sent by the genuine sender. Obviously, both the sender and the receiver also want to be guaranteed that the contents of the message were not changed while it was in transfer.

Providing both external and internal security is more difficult in distributed systems than in centralized systems because of the lack of a single point of control and the use of insecure networks for data communication. Although external security is as important as internal security, the policies and mechanisms used at the operating system level for computer security deal only with the internal security aspects. Therefore, this chapter deals mainly with the commonly used mechanisms for providing different types of internal security in distributed systems.

11.2 POTENTIAL ATTACKS TO COMPUTER SYSTEMS

The first step in the provision of appropriate computer security is to identify the potential threats/attacks to computer systems. The term *intruder* or *attacker* is commonly used to refer to a person or program trying to obtain unauthorized access to data or a resource of a computer system. An intruder may be a threat to computer security in many ways that are broadly classified into two categories—passive attacks and active attacks. A *passive attack* does not cause any harm to the system being threatened, whereas an *active attack* does. Therefore, passive attacks are inherently undetectable by the system and can only be dealt with by using preventive measures. On the other hand, active attacks are combated by a combination of prevention, detection, and recovery techniques. A description of these attacks and some other related problems are presented below.

11.2.1 Passive Attacks

In passive attacks, an intruder somehow tries to steal unauthorized information from the computer system without interfering with the normal functioning of the system. Some commonly used methods of passive attack are described below:

1. *Browsing.* In this method, intruders attempt to read stored files, message packets passing by on the network, other processes' memory, and so on, without modifying any data. Access control mechanisms are used to prevent unauthorized reading of stored files and other processes' memory contents, and message encryption is used to prevent eavesdropping of messages transmitted over network links.

2. *Leaking*. In this method, an intruder uses an accomplice (a legitimate user having authority to access the information to be stolen) who leaks the information to him or her. Prevention of leaking is a difficult problem to solve and requires preventing all types of communication between the accomplice and the intruder. The problem of ensuring that it is impossible for a potential accomplice to leak any information to the outside world is called the *confinement problem* [Lampson 1973]. As described later, leaking of information between processes that in theory cannot communicate at all is relatively straightforward. Therefore, the confinement problem is in general unsolvable.

3. *Inferencing*. In this method, an intruder tries to draw some inference by closely observing and analyzing the system's data or the activities carried out by the system. For example, if information is encrypted to protect unauthorized access, an intruder may try to derive the encryption key by analyzing several pieces of encrypted data. Since the derived key can be used for stealing information from the system, it is valuable and may be sold to other intruders. Another example of inferencing is traffic analysis in distributed systems. In this case, an intruder observes when and where interprocess messages flow in the system, and by analyzing the frequency of message exchanges between various communicating partners, the intruder tries to draw some inference. For example, in a business environment, traffic analysis may provide useful clues to negotiations taking place between different organizations.

4. *Masquerading*. In this method, an intruder masquerades as an authorized user or program in order to gain access to unauthorized data or resources. For instance, many systems have mechanisms for allowing programs written by users to be used by other users. These programs can improperly use the access rights of an executing user and leak information. For example, an intruder may write an editor program that works perfectly as an editor but also creates a copy of the edited file to a special area accessible to the intruder. This editor program is then compiled and read into the *bin* directory of a user, whose files the intruder is interested in. From then on, the intruder gets a copy of all the files edited by the user. The user is ignorant of the theft being made because the editor program performs all his or her editing jobs in a perfectly normal fashion.

Penetrating computer security in this manner is known as the *Trojan horse attack*. That is, a *Trojan horse program* is a program that consists of clandestine code to do nasty things in addition to its usual function but appears to be benign. It is often offered as a gift or sometimes for a nominal price to prevent suspicion. A user normally accepts it into his or her system because of the useful function performed by it. However, once inside the user's computer, code hidden in the program becomes active and either executes malicious acts or creates a way of subverting system security so that unauthorized personnel can gain access to the system resources. Note that a Trojan horse attack may either be passive or active depending on the activities performed by the clandestine code. For example, if the clandestine code simply steals information, then it is of the passive type. But if it does something more harmful like destroying/corrupting files, then it is of the active type.

An intruder can also masquerade as a trusted server to a client requesting a service from the system. This action is known as *spoofing*.

11.2.2 Active Attacks

Active intruders are more malicious than passive intruders. Unlike passive attacks, active attacks interfere with the normal functioning of the system and often have damaging effects. The most common types of damage that active attacks cause are corrupting files, destroying data, imitating hardware errors, slowing down the system, filling up memory or disk space with garbage, causing the system to crash, confounding a receiver into accepting fabricated messages, and denial/delay of message delivery. Some commonly used forms of active attacks are described below. In the following, the description of viruses, worms, and logic bombs is based on the material presented in [Bowles and Pelaez 1992].

Viruses

A *computer virus* is a piece of code attached to a legitimate program that, when executed, infects other programs in the system by replicating and attaching itself to them. In addition to this replicating effect, a virus normally does some other damage to the system, such as corrupting/erasing files. Therefore, due to its spreading nature, a virus can cause severe damage to a system. Notice that virus attacks are active-type Trojan horse attacks.

A typical virus works as follows. The intruder writes a new program that performs some interesting or useful function (such as some game or utility) and attaches the virus to it in such a way that when the program is executed the viral code also gets executed. The intruder now uploads this infected program to a public bulletin board system or sends it by mail to other users of the system or offers it for free or for a nominal charge on floppy disks. Now if anyone uses the infected program, its viral code gets executed. When the viral code of the infected program executes, it randomly selects an executable file on the hard disk and checks to see if it is already infected. Most viruses include a string of characters that acts as a marker showing that the program has been infected. If the selected file is already infected, the virus selects another executable file. When an uninfected program is found, the virus infects it by attaching a copy of itself to the end of that program and replacing the first instruction of the program with a jump to the viral code. When the viral code is finished executing, it executes the instruction that had previously been first and then jumps to the second instruction so that the program now performs its intended function. Notice that a virus spreads because every time an infected program is executed, it tries to infect more programs. Also notice that a virus does not infect an already infected file in order to prevent an object file from growing ever longer. This allows the virus to infect many programs without noticeably increasing disk space usage.

Recovery from a virus infection is a difficult task that often requires partial or complete shutdown for long periods of time of the computer system under attack. Therefore, it is always better to take necessary precautions to prevent virus problems. Some precautionary steps include (a) buying software only from respectable stores, (b) refusing to accept software in unsealed packages or from untrusted sources, (c) avoiding borrowing programs from someone whose security standards are less rigorous than one's own, and (d) avoiding uploading of free software from public domain, bulletin boards, and programs sent by electronic mail.

When a computer system suffers from virus infection, it has to be cured. The simplest way to cure a computer from virus infection is to shut it down, purge its memory and all its disks, and rebuild its files from scratch using the original manufacturer's copy. Disinfection utilities may also be used to cure a computer from virus infection. These utilities first identify the virus type with which the computer is infected by matching its marker against the markers of well-known viruses. Once the type is known, the original programs are restored from their infected versions by applying a detailed knowledge of the infection method used by the virus. For example, in viruses that modify jump instructions at the beginning of the host program, recovering can be done simply by restoring the original jump to the start of the host program code. However, notice that these disinfection utilities can only cure specific known viruses. They cannot cure a newly encountered type of virus. A good disinfection utility can normally cure several hundred types of viruses and its power can be regularly improved by frequently updating it as new viruses are discovered.

Notice that the longer a virus remains in a system, the more time it has to spread and the tougher recovery from it becomes. Therefore, it is important to detect a virus as soon as possible. An effective method to detect viruses is to use a snapshot program and a check routine. The snapshot program is used to log all critical system information at the time of the initial installation and the check routine is periodically executed to compare the system's current state with the original snapshot. If signs of infection are detected, the affected area of the computer is identified and the user is notified.

Curing a distributed system from virus infection is much more difficult because if the infection is not removed from every workstation at the same time, reinfection will occur. This is because an infected file on the network server can infect every workstation on the network.

Worms

Worms are programs that spread from one computer to another in a network of computers. They spread by taking advantage of the way in which resources are shared on a computer network and, in some cases, by exploiting flaws in the standard software installed on network systems. A worm program may perform destructive activities after arrival at a network node. Even when not directly destructive, worms often cripple a network by subverting the operation of computers on the network to their own purposes, monopolizing their resources, and saturating the communications links in a network. Often, it is necessary to shut down the entire system (all computers of the network) to recover from a worm problem.

To illustrate how a worm propagates in a network, the famous Internet Worm attack by a Cornell graduate student, Robert Tappan Morris, on November 2, 1988, that infected thousands of UNIX machines all over the Internet is described here. This worm program had two types of code, the bootstrap code and the code forming the main body of the worm. The bootstrap code was compiled and executed on the system under attack. When executed, the bootstrap code established a communications link between its new host and the machine from which it came, copied the main body of the worm to the new host, and executed it. Once installed on a new host, the worm's first few actions were to hide its

existence. For this, it unlinked the binary version of itself, killed its parent process, read its files into memory and encrypted them, and deleted the files created during its entry into the system. After finishing its hiding operations, the worm's next job was to look into its host's routing tables to collect information about other hosts to which its current host was connected. Using this information, it then attempted to spread its bootstrap code to those machines by trying the following three methods one by one:

1. The first method was to try to spawn a remote shell on the target machine using the *rsh* command of UNIX. This method sometimes works because some machines trust other machines and willingly run *rsh* without authenticating the remote machine. If successful, the remote shell uploaded the worm program on the target machine and continued spreading to new machines from there.

2. If the first method failed, the second method was tried. This method took advantage of a bug in the *finger* program that runs as a daemon process at every BSD site. A user anywhere on the Internet can type

finger user_name@host_name

to obtain general information about a person at a particular site, such as the person's real name, home address, office address, telephone number, and so on. The *finger* utility uses the C library function *gets* to read input data. A problem with *gets* is that it reads the entire input string without checking for buffer overflows. The worm exploited this flaw and called *finger* with a specially constructed 536-byte string as a parameter. The overflow caused an area of the system stack to be overwritten, allowing the worm to put its own suitable instructions (a procedure to execute */bin/sh*) on the stack. Now when the *finger* daemon returned from the procedure it was in at the time it got the request, it returned to and executed the procedure inside the 536-byte string on the stack instead of returning to *main*. If this method succeeded, the worm had a shell running on the target machine.

3. If the first two methods failed, the worm tried a third method that takes advantage of a loophole in the UNIX electronic mail utility *sendmail*. The *sendmail* program has a *DEBUG* option, which allows program testers to verify that mail has arrived at a site without having to invoke the mailer's address resolution routines. Many vendors and site administrators leave the debug option compiled into the *sendmail* code to facilitate configuring the mailer for local conditions. What the worm did was to execute the *sendmail* program with the *DEBUG* option and then enact a sequence of commands to mail the bootstrap code to the target machine.

Once established on a new machine, the worm tried to break into user accounts by exploiting the accessibility of the UNIX password file and the tendency of users to choose common words as their passwords. Each broken password allowed the worm to masquerade as the user corresponding to that password and gain access to any remote machine where that user had an account.

The worm was designed to act intelligently to prevent being spotted. It periodically *forked* itself and killed its parent, so that its process ID was constantly changing. This

prevented any one process from accumulating a large amount of CPU time that might create suspicion or cause its scheduling priority to be degraded. Furthermore, after every 12 hours, the worm erased its record of the machines it had infected, so that already infected hosts were put back on the list of potential targets. Whenever the worm gained access to a new machine, it first checked to see if the machine already had a copy of the worm. If so, the new copy exited, except one time in seven. The use of one in seven possibly was to allow the worm to spread even on a machine on which the system administrator might have started its own version of the worm to fool the real worm.

Although viruses and worms both replicate and spread themselves, the two differ in the following aspects:

1. A virus is a program fragment whereas a worm is a complete program in itself.
2. Since a virus is a program fragment, it does not exist independently. It resides in a host program, runs only when the host program runs, and depends on the host program for its existence. On the other hand, a worm can exist and execute independently.
3. A virus spreads from one program to another whereas a worm spreads from one computer to another in a network.

Logic Bombs

A logic bomb is a program that lies dormant until some trigger condition causes it to explode. On explosion, it destroys data and spoils system software of the host computer. A trigger condition may be an event such as accessing a particular data file, a program being run a certain number of times, the passage of a given amount of time, or the system clock reaching some specific date (for instance, Friday the 13th or April Fool's Day). The trigger condition is normally selected so that the logic bomb explodes at the moment when it can do maximum damage to the system. Logic bombs can be embedded in a Trojan horse or carried about by a virus.

Active Attacks Associated with Message Communications

In a distributed system, communication channels are used to carry information from one node to another in the system in the form of messages. These communication channels may be exposed to attackers who may try to breach the security of the system by observing, modifying, deleting, inserting, delaying, redirecting, or replaying the messages that travel through the communication channels. The commonly known active attacks associated with message communications are of the following types:

1. *Integrity attack.* For secure communication, the integrity requirement specifies that every message is received exactly as it was sent or a discrepancy is detected. However, an intruder may change the contents of a message while it is traveling through a communication channel and the receiver may not be aware of this and accept it as the original message.

2. *Authenticity attack.* An intruder may illegally connect his or her own computer system to a communication channel and impersonate a legal network site. The intruder can then synthesize and insert bogus messages with valid addresses into the system so that they are delivered as genuine messages. If an integrity attack is possible, an intruder may also cause an authenticity attack by changing the protocol control information (addresses) of the messages so that they are delivered to wrong destinations.

3. *Denial attack.* In this case, the intruder either completely blocks the communication path between two processes so that the two processes cannot communicate at all or observes all messages exchanged between the two processes and prevents only selected messages from delivery. That is, the intruder causes complete or partial denial of message delivery.

4. *Delay attack.* Several messages have time value. Therefore, instead of using a denial attack, an intruder may simply delay the delivery of message passing in an association between two communicating processes to fulfill his or her motive.

5. *Replay attack.* In this case, an intruder retransmits old messages that are accepted as new messages by their recipients.

Cryptography deals with the encryption of sensitive data to prevent its comprehension and is the only practical means for protecting information sent over an insecure channel, be it telephone line, microwave, satellite, or any other transmission media. This is because an encrypted message provides no information regarding the original message, hence guaranteeing secrecy; and an encrypted message, if tampered with, would not decrypt into a legal message, hence guaranteeing integrity. Cryptography can also be used for secure identification of communicating entities, hence guaranteeing authenticity. Furthermore, encryption, in conjunction with protocols, can also be used to prevent denial, delay, and replay of messages. For instance, replay of old messages can be countered by using nonces or timestamps. A *nonce* is an information that is guaranteed to be fresh; that is, it has not been used or appeared before. Therefore, a reply that contains some function of a recently sent nonce should be considered timely because the reply could have been generated only after the nonce was sent. Perfect random numbers are suitable for use as nonces. In summary, the only way to prevent attacks associated with message communications is by the application of cryptographic techniques.

11.2.3 Confinement Problem

In a client-server model, a single server program may be shared by multiple clients. In situations where programs are shared, a security problem is considerably more complex than if only data objects are shared. One reason that we have already seen is a Trojan horse. A Trojan horse is just one way in which a shared program could leak classified information to other unclassified subjects. There may be several other ways in which a shared program could leak confidential information to unauthorized subjects.

A program that cannot retain or leak confidential information is said to be *memoryless* or *confined*, and the prevention of such leakage is called the *confinement*

problem [Lampson 1973]. That is, the confinement problem deals with the problem of eliminating every means by which an authorized subject can release any information contained in the object to which it has access to some subjects that are not authorized to access that information. According to Lampson, as long as a program does not have to retain or output any information, confinement can be implemented by restricting the access rights of the program. But if a program must retain or output information, access control alone is not sufficient to ensure security.

Lampson identified the following kinds of channels that can be used by a program to leak information:

1. *Legitimate channels.* Legitimate channels are those that the program uses to convey the results of its computation, such as messages or printed output. The program may hide additional information in these channels along with the actual result. Some form of encoding that is meaningful to the person or process receiving the result is used to convey the additional information. For example, in a printed output, two different space lengths that are not discernible to normal persons but visible if observed minutely may be used between words to mean 0 and 1 bits. This type of printed output can be used to convey additional information to a person who knows about the hidden bits between two words.

2. *Storage channels.* Storage channels are those that utilize system storage such as shared variables or files to leak information to other processes. Notice that when a process (A) wants to leak information to another process (B) by using a storage channel, it is not necessary that both processes must have access rights to the shared object. For example, if the system provides a way of locking files, process A can lock some file to indicate a 1 and unlock it to indicate a 0. It may be possible for process B to detect the status of a lock even on a file that B cannot access. Similarly, in UNIX, process A could create a file to indicate a 1 and remove it to indicate a 0. Even though process B has no permission to access the file created by A, it can use the *access* system call to see if the file exists.

3. *Covert channels.* Covert channels are paths that are not normally intended for information transfer at all but could be used to send some information. For example, a process may use one of the following methods to leak information to some other process that is carefully monitoring its activities [Tanenbaum 1992]:

- *By modulating paging rate.* For example, during a fixed time period, many page faults caused by the process may be used to convey a 1, and no page faults for a 0.
- *By modulating CPU usage.* For example, usage of CPU for a fixed time period by the process may be used to convey a 1, and sleeping of the process for the same period may be used to convey a 0.
- *By acquiring and releasing dedicated resources.* For example, the process may acquire the resource to convey a 1 and release it to convey a 0.

To solve the confinement problem, it is important to block all channels that a program may use to communicate with other processes. However, finding all such

channels and trying to block them is extremely difficult. In practice, there is little that can be done. Therefore, the confinement problem is in general unsolvable.

11.3 CRYPTOGRAPHY

Cryptography is a means of protecting private information against unauthorized access in those situations where it is difficult to provide physical security. The basic idea behind this security technique is that if it is not possible to prevent copying of information, it is better to prevent comprehension.

11.3.1 Basic Concepts and Terminologies

Two primitive operations employed by cryptography are *encryption* and *decryption*. Encryption (also called *enciphering*) is the process of transforming an intelligible information (called *plaintext* or *cleartext*) into an unintelligible form (called *ciphertext*). Decryption (also called *deciphering*) is the process of transforming the information back from ciphertext to plaintext. When cryptography is employed for protecting information transmitted through communication channels, plaintext is also called a *message*.

Encryption is basically a mathematical function (encryption algorithm) having the following form:

$$C = E\ (P,\ K_e)$$

where P is the plaintext to be encrypted, K_e is an encryption key, and C is the resulting ciphertext. Decryption of C is performed by a matching function (decryption algorithm) that has the following form:

$$P = D\ (C,\ K_d)$$

where K_d is the decryption key. Note that the decryption function D is the inverse of the encryption function E. Therefore we have

$$D\ (E\ (P,\ K_e),\ K_d) = P$$

To prevent the plaintext from being easily revealed, it must be possible to transform a given plaintext into a large variety of possible ciphertexts selected by a specific parameter. The keys K_e and K_d serve as this parameter. That is, the function parts remain the same but the keys are changed as often as necessary.

The above described general structure of a cryptosystem is illustrated with an example in Figure 11.1, where a message is encrypted for secure transmission over an insecure channel from a sender node to a receiver node.

11.3.2 Basic Requirements

To be practically useful, a cryptosystem must fulfill the following basic requirements:

1. It must be easy to use and its encryption and decryption algorithms should be efficient for computer application.

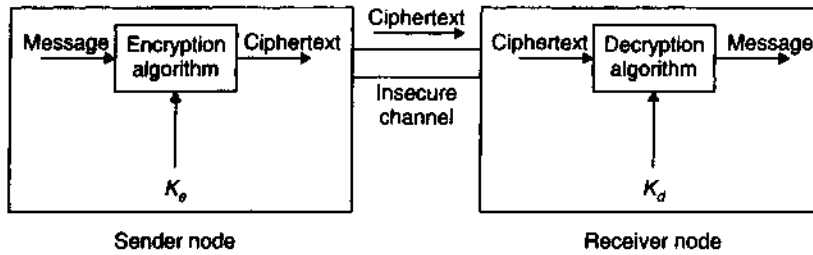


Fig. 11.1 General structure of a cryptosystem.

2. There are two methods to achieve security. In the first method, the encryption algorithm is kept secret and is rather complex to make it difficult to guess. In the second method, the encryption algorithm is made public but the keys are kept secret and they are long enough to make it practically impossible to guess a key. The second method is preferred for practically useful systems. That is, the security of the system should depend only on the secrecy of the keys and not on the secrecy of the algorithms.

3. The system must be computationally (practically) secure. That is, the determination of K_d must be computationally infeasible for an attacker (also called a *cryptanalyst*). Note that the strength of a cryptosystem is measured by the level of difficulty (usually measured either by the time or number of elementary operations) of determining K_d. Depending on the amount of information available to an intruder, in a cryptosystem, attacks are mainly of three types—ciphertext only, known plaintext, and chosen plaintext [Bright 1977].

In *ciphertext-only* attack, an intruder is able to intercept ciphertext and tries to derive K_d from the ciphertext. A system whose security is not resistant to a ciphertext-only attack is considered to be totally insecure and is useless.

In *known-plaintext* attack, an intruder has considerable amount of both ciphertext and corresponding plaintext and tries to derive K_d from them. A system that can resist a known-plaintext attack is considered to be secure.

In *chosen-plaintext* attack, an intruder has access to ciphertext for any plaintext of his or her choice. The intruder tries to derive K_d by examining several ciphertexts for the carefully thought plaintexts of his or her choice. It is most appropriate nowadays to evaluate cryptosystems by their ability to withstand chosen-plaintext attacks.

Another important way by which a cryptosystem is demonstrated to be secure is the test of time. If no known successful attacks have been reported since a system is published and in use for a significant amount of time (measured in years), the cryptosystem is considered to probably provide pretty good security.

11.3.3 Symmetric and Asymmetric Cryptosystems

There are two broad classes of cryptosystems, symmetric and asymmetric. In a *symmetric cryptosystem*, either both the encryption key (K_e) and decryption key (K_d) are the same or one is easily derivable from the other. Usually, a common key (K) is used for both

enciphering and deciphering. For security, it is important that the key of a symmetric cryptosystem be easily alterable and must always be kept secret. This implies that the key is known only to authorized users. Symmetric cryptosystems are also known as *shared-key* or *private-key cryptosystems*.

Symmetric cryptosystems are useful in those situations where both encryption and decryption of information are performed by a trusted subsystem. For example, a password-based user authentication system may use this scheme for saving passwords in encrypted form. When a user declares a password, the operating system uses the encryption key for encrypting the password before storing it internally. At the time of authentication, the operating system again uses the same key to decrypt the stored password to compare it to the password supplied by the user.

In an *asymmetric cryptosystem*, on the other hand, the decryption key (K_d) is not equal to the encryption key (K_e). Furthermore, it is computationally impractical to derive K_d from K_e. Because of this property, only K_d needs to be kept secret and K_e is made publicly known. Asymmetric cryptosystems are also known as *public-key cryptosystems*.

Public-key cryptosystems are computationally expensive and hence are not suitable for bulk data encryption. A typical use of a public-key cryptosystem in distributed systems is for establishing connection between two communicating entities (A and B) for the exchange of messages using a symmetric cryptosystem. Let us suppose that A and B want to establish a connection between themselves for initiating message transfers using a symmetric cryptosystem whose key is K. Note that it is insecure to send the key K over a normal communication channel for the purpose of sharing it with A and B. Therefore, a public-key cryptosystem is first used to establish a connection between A and B in the following manner:

- Entity A posts the encryption key (K_e) of a public-key cryptosystem on, say, an electronic bulletin board, so that it is obtainable by B.
- Entity B uses A's public key to encrypt the key K and transmits it to A.
- Entity A decrypts B's message using the decryption key (K_d) of the public-key cryptosystem. Only A can decrypt this message because K_d is available only with A.
- Now that A also has the key K, both A and B can safely communicate with each other using the symmetric cryptosystem scheme. Any eavesdropper would only have the public key (K_e), the encrypted form of key K, and the encrypted form of the messages being communicated between A and B.

One pitfall here is that someone else could masquerade as A or B. This can be overcome by using digital signatures (described later in this chapter).

The relative advantages and disadvantages of symmetric and asymmetric cryptosystems are as follows:

1. Symmetric cryptosystems require that both encryption and decryption of information be performed by a trusted subsystem, whereas this is not necessary with asymmetric cryptosystems. Therefore, general security policies need asymmetric cryptosystems.

2. When employed for the security of messages in a communication system, a symmetric cryptosystem requires a secure channel by which the sender can inform the receiver of the key used to encipher the messages. The encrypted messages may, however, be transmitted through an insecure channel. On the other hand, in an asymmetric cryptosystem, both the public-key and the messages can be transmitted through an insecure channel. Therefore, there is no need for a special secure channel for key transmission. Due to this reason, asymmetric cryptosystems are considered to be more secure than symmetric cryptosystems.

3. In general, asymmetric cryptosystems are computationally much more expensive than symmetric cryptosystems and are inefficient for bulk data encryption. Hence, in practice, for data communications, asymmetric cryptosystems are often used only for initialization/control functions, while symmetric cryptosystems are used for actual data transfer.

The Data Encryption Standard (DES) cryptosystem [NBS 1977, Seberry and Pieprzyk 1989] is the best known and most widely used symmetric cryptosystem today. On the other hand, the Rivest–Shamir–Adleman (RSA) cryptosystem [Rivest et al. 1978, Seberry and Pieprzyk 1989] is the first published and practically the most satisfactory asymmetric cryptosystem today.

11.3.4 Key Distribution Problem

When cryptography is employed for secure communications in distributed systems, a need for key distribution arises because two communicating entities can securely communicate only when they obtain matching keys for encryption and decryption of the transmitted messages. A matching pair of keys held by two communicating entities forms an independent, private logical channel between them. The key distribution problem deals with how to securely supply the keys necessary to create these logical channels. The key distribution problem in symmetric and asymmetric cryptosystems are described below.

Key Distribution in Symmetric Cryptosystems

When two users (persons or programs) of two different nodes want to communicate securely by using a symmetric cryptosystem, they must first share the encryption/decryption key. For this, the key must be transmitted from one of the two users to the other user. However, there is no special transmission medium for the key transfer and the key must be transmitted using the same insecure physical medium by which all exchanged messages are transmitted. This requires that the key must itself be encrypted before transmission because if the key is compromised by an intruder while being transmitted over the insecure medium, the intruder can decrypt all encrypted messages exchanged between the two users. Therefore, a circularity exists in symmetric cryptosystems. This circularity can only be broken through prior distribution of a small number of keys by some secure means. The usual approach is to use a server process that performs the job of a *key distribution center* (*KDC*). Each user in the system shares with the KDC a prearranged pair of unique keys.

The KDC is a generally trusted entity and is shared by all communicating users of the system. On request by a user, it generates a new secret key to be used by the user to communicate with another user. In actual implementation, there may be several KDCs in the system. The three commonly used implementation approaches are as follows:

- Centralized approach
- Fully distributed approach
- Partially distributed approach

Below we describe how key distribution takes place in each approach between two users who want to communicate securely with each other.

Centralized Approach. In this approach, a single centralized KDC is used that maintains a table of secret keys for each user (see Fig. 11.2). A user's secret key is known only to the user and KDC. Suppose that the secret keys of users A and B are K_a and K_b, respectively, and that a secure logical communication channel is to be established for exchanging encrypted messages between them. The following protocol was proposed in [Needham and Schroeder 1978] for performing this task (see Fig. 11.2):

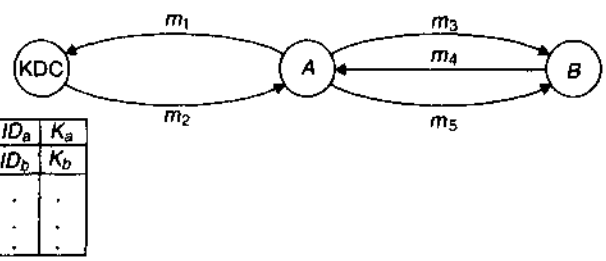


Table of secret keys for each user

$m_1 = (R_a, ID_a, ID_b)$
where R_a = code for the request made by user A
ID_a = identifier of user A
ID_b = identifier of user B

$m_2 = E\ ((R_a, ID_a, K_{ab}, C_1), K_a)$
where K_{ab} = secret key generated by the KDC for secure communications between users A and B
$C_1 = E\ ((K_{ab}, ID_a), K_b)$
where K_b = private key of user B
K_a = private key of user A

$m_3 = C_1$

$m_4 = C_2 = E\ (N_t, K_{ab})$
where N_t = a random number generated by user B

$m_5 = C_3 = E\ (N_t, K_{ab})$
where $N_t = f(N_t)$ and f is a previously defined function

Fig. 11.2 The method of key distribution in the centralized approach.

1. User A sends a request message (m_1) to the KDC indicating that it wants to establish a secure logical communication channel with user B. The message contains a code for the request (R_a), the user identifier of A (ID_a), and the user identifier of B (ID_b). This message is transmitted from user A to KDC in plaintext form.

2. On receiving m_1, the KDC extracts from its table the keys K_a and K_b, which correspond respectively to the user identifiers ID_a and ID_b in the message. It then creates a secret key K_{ab} for secure communications between users A and B. By using key K_b, the KDC encrypts the pair (K_{ab}, ID_a) to generate a ciphertext $C_1 = E((K_{ab}, ID_a), K_b)$. Finally, it sends a message (m_2) to user A that contains R_a, ID_a, K_{ab}, and C_1. The message m_2 is encrypted with the key K_a so that only user A can decrypt it.

3. On receiving m_2, user A decrypts it with its private key K_a and checks whether R_a and ID_a of the message match with the originals to get confirmed that m_2 is the reply for m_1. If so, user A keeps the key K_{ab} with it for future use and sends a message m_3 to user B. This message contains the ciphertext C_1. Note that only user B can decrypt C_1 because it was generated using key K_b.

4. On receiving m_3, user B decrypts C_1 with its private key K_b and retrieves both K_{ab} and ID_a. At this stage, both users A and B have the same key K_{ab} that can be used for secure communications between them because no other user has this key. At this point, user B needs to verify if user A is also in possession of the key K_{ab}. Therefore, user B initiates an authentication procedure that involves sending a nonce to user A and receiving a reply that contains some function of the recently sent nonce. For this, user B generates a random number N_r, encrypts N_r by using the key K_{ab} to generate a ciphertext $C_2 = E(N_r, K_{ab})$, and sends C_2 to user A in a message m_4. The random number N_r is used as a nonce.

5. On receiving m_4, user A decrypts C_2 with the key K_{ab} and retrieves N_r. It then transforms N_r to a new value N_t by a previously defined function (f). User A encrypts N_t by using the key K_{ab} to generate a ciphertext $C_3 = E(N_t, K_{ab})$, and sends C_3 to user B in a message m_5.

6. On receiving m_5, user B decrypts C_3, retrieves N_t, and applies the inverse of function f to N_t to check if the value obtained is N_r. If so, user B gets confirmed that a secure channel can be created between users A and B by using the key K_{ab}. This is sufficient to achieve mutual confidence, and from now on, the exchange of actual messages encrypted with key K_{ab} can take place between users A and B.

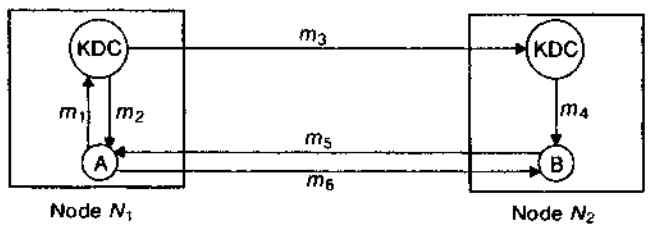
That there is a problem in the protocol was pointed out by Denning and Sacco [1981]. They observed that during the transfer of message m_3, if an intruder copies C_1 and by unspecified means came to know K_{ab}, that intruder can in future always pretend to B that it was A. The basic problem here is that B never had the chance to offer a nonce to the KDC and, therefore, has no means of deducing the freshness of the quantity (K_{ab}) that came from the KDC. Note that only the KDC and A know that K_{ab} is fresh. One method to correct this problem is to add a timestamp (T) to the ciphertext C_1, so that it becomes $E((K_{ab}, ID_a, T), K_b)$. User B decrypts this message

and checks that T is recent. This solution is adopted in the Kerberos system (described later in this chapter).

Notice that in the centralized KDC approach, if there are n users in the system, n prearranged key pairs are needed to provide secure communications. The approach is simple and easy to implement. However, it suffers from the drawbacks of poor reliability and performance bottleneck of the single KDC. That is, fresh key distributions cannot take place if the node on which the KDC resides crashes, and the KDC may get overloaded in a large system with too many users. Two other approaches described below may be used to overcome these drawbacks of the centralized approach.

Fully Distributed Approach. In this approach, there is a KDC at each node of the distributed system. The prior distribution of secret keys allows each KDC to communicate securely with all other KDCs. That is, each KDC has a table of secret keys having private keys of all other KDCs. Therefore, in a system having n nodes, each KDC keeps $n-1$ keys, resulting in a total of $n(n-1)/2$ key pairs in the system.

Suppose that a secure logical communication channel is to be established between user A of node N_1 and user B of node N_2. Also suppose that K_1 and K_2 are the private keys of the KDCs of nodes N_1 and N_2, respectively. The desired connection can be established in the following manner (see Fig. 11.3):



$m_1 = (R_a, ID_a, ID_b)$
where R_a = code for the request made by user A
ID_a = identifier of user A
ID_b = identifier of user B

$m_2 = (R_a, ID_a, K_{ab})$
where K_{ab} = secret key generated by the KDC of node N_1 for secure communications between users A and B

$m_3 = C_1 = E((K_{ab}, ID_a, ID_b), K_2)$
where K_2 = private key of KDC of node N_2

$m_4 = (K_{ab}, ID_a)$

$m_5 = C_2 = E(N_r, K_{ab})$
where N_r = a random number generated by user B

$m_6 = C_3 = E(N_t, K_{ab})$
where $N_t = f(N_r)$ and f is a previously defined function

Fig. 11.3 The method of key distribution in the fully distributed approach.

1. User A sends a request message (m_1) to its local KDC. The message contains a code for the request (R_a), the user identifier of A (ID_a), and the user identifier of B (ID_b). It is assumed that all local communications are secure and hence local messages can be securely transmitted in plaintext form.

2. On receiving m_1, the KDC of node N_1 consults the name server to get the location (N_2) of the user having identifier ID_b. It then extracts the private key (K_2) of the KDC of node N_2 from its table. Next, it creates a secret key K_{ab} for secure communications between users A and B. By using key K_2, it encrypts the triplet (K_{ab}, ID_a, ID_b) to generate a ciphertext $C_1 = E((K_{ab}, ID_a, ID_b), K_2)$. Finally, it sends a message (m_2) to user A that contains R_a, ID_a, K_{ab} and a message (m_3) to the KDC of node N_2 that contains C_1.

3. On receiving m_2, user A checks whether R_a and ID_a of the message match with the originals to get confirmed that m_2 is the reply for m_1. If so, user A keeps the key K_{ab} with it for future use.

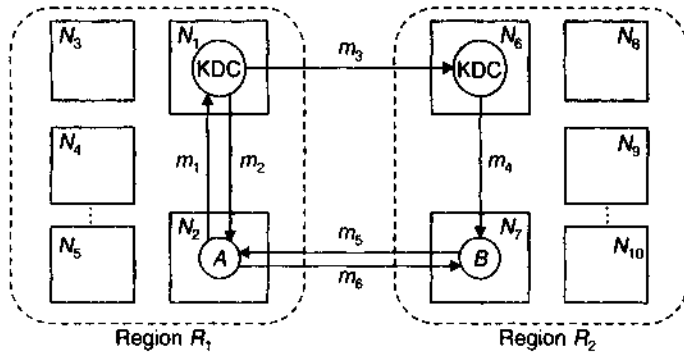
4. On the other hand, on receiving message m_3, the KDC of node N_2 decrypts it with its private key K_2 and forwards the pair (K_{ab}, ID_a) to the user with identifier ID_b in the form of message m_4.

5. On receiving m_4, user B initiates an authentication procedure and authenticates user A exactly in the same way as done in the centralized approach.

Notice that in this approach the number of prearranged key pairs needed to provide secure communications is independent of the number of users and depends only on the number of nodes in the system. Another major advantage of this approach is that for successful distribution of a key for secure communications between two users, only the nodes of the two users must be properly functioning. Therefore, the approach is highly reliable.

Partially Distributed Approach. In this approach, the nodes of the system are partitioned into regions and, instead of having a KDC for each node, there is a KDC only for each region that resides on one of the nodes of that region. The prior distribution of secret keys allows each KDC to communicate securely with each user of its own region and with the KDCs of all other regions. That is, each KDC has a table of secret keys that contains private keys of all users of its own region and of all other KDCs.

In this approach, the distribution of a key for the establishment of a secure logical communication channel between two users A and B depends on the locations of the two users. If both the users A and B reside on nodes that belong to the same region, the key is distributed exactly in the same manner as is done in the centralized approach. In this case, the KDC of that region plays the role of the centralized KDC. On the other hand, if the users A and B reside on nodes belonging to different regions, the key distribution is performed in a manner similar to that in the case of the fully distributed approach. The only difference is that, in this case, messages m_2 and m_4 are also encrypted because they are transmitted from one node to another. That is, m_2 is encrypted with the private key of user A, and m_4 is encrypted with the private key of user B. The complete process of key distribution in this approach is shown in Figure 11.4. Notice that, in this approach, the



Let there be 10 nodes in the system that are partitioned into two regions R_1 and R_2 as shown above. The KDCs of regions R_1 and R_2 are located on nodes N_1 and N_6, respectively.

$m_1 = (R_a, ID_a, ID_b)$
where R_a = code for the request made by user A
ID_a = identifier of user A
ID_b = identifier of user B

$m_2 = C_1 = E\ ((R_a, ID_a, K_{ab}), K_a)$
where K_{ab} = secret key generated by the KDC of region R_1 for secure communications between users A and B
K_a = private key for user A

$m_3 = C_2 = E\ ((K_{ab}, ID_a, ID_b), K_2)$
where K_2 = private key of KDC of region R_2

$m_4 = C_3 = E\ (K_{ab}, ID_a), K_b)$
where K_b = private key of user B

$m_5 = C_4 = E\ (N_r, K_{ab})$
where N_r = a random number generated by user B

$m_6 = C_5 = E\ (N_t, K_{ab})$
where $N_t = f(N_r)$ and f is a previously defined function.

Fig. 11.4 The method of key distribution in the partially distributed approach.

failure of the KDC of a particular region will only disrupt key distribution activities that involve a user of that region. Key distribution involving users of other regions can still be carried out successfully. Therefore, the reliability of this approach lies in between the reliabilities of the centralized and the fully distributed approaches.

Key Distribution in Asymmetric Cryptosystems

In an asymmetric cryptosystem only public keys are distributed. Since public keys need not be kept secret, there is no problem of key transmission through an insecure physical medium. Therefore, one might conclude that the key distribution problem does not exist in asymmetric cryptosystems. However, this is not correct since the safety of an

asymmetric cryptosystem depends critically on the correct public key being selected by a user. A user who receives a public key wants to be sure that the received key is genuine. Thus, in asymmetric cryptosystems, the key distribution process involves an authentication procedure to prevent an intruder from generating a pair of keys and sending a public key to another user for establishing a logical communication channel with that user. The authentication procedure allows the two users of a logical communication channel to identify themselves before starting the actual exchange of data.

The commonly used key distribution approach in asymmetric cryptosystems is to use a *public-key manager* (*PKM*) process that maintains a directory of public keys of all users in the system. There is a key pair (P_k, S_k) for the PKM also. The public key P_k is known to all users and the secret key S_k is known exclusively to the PKM. The protocol based on this approach for establishing a logical communication channel between two users is described below.

Let us assume that a logical communication channel is to be established between users A and B. Also assume that (P_a, S_a) is the key pair of user A and (P_b, S_b) is the key pair of user B. The public keys P_a and P_b are stored with the PKM and the secret keys S_a and S_b are known exclusively to users A and B, respectively. The desired connection can be established in the following manner (see Fig. 11.5):

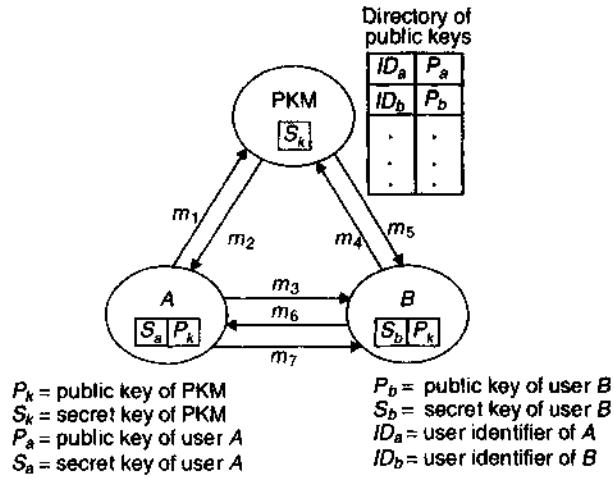
1. User A sends a request message (m_1) to the PKM indicating that it wants to establish a secure logical communication channel with user B. The message contains a code for the request (R_a), a timestamp (T_1), the user identifier of A (ID_a), and the user identifier of B (ID_b). This message is encrypted by using P_k. That is, m_1 contains the ciphertext $C_1 = E((R_a, T_1, ID_a, ID_b), P_k)$.

2. On receiving m_1, the PKM decrypts C_1 by using S_k. It then extracts the public keys P_a and P_b from its table that correspond respectively to the user identifiers ID_a and ID_b of the message. By using key P_a, the PKM encrypts the triplet (R_a, T_1, P_b) to generate a ciphertext $C_2 = E((R_a, T_1, P_b), P_a)$. Finally, it sends C_2 to user A in the form of a message (m_2).

3. On receiving m_2, user A decrypts C_2 by using S_a and retrieves its contents. The retrieved values of R_a and T_1 are compared with the originals to get confirmed that m_2 is the reply for m_1 and that it is not a replay of an old message. User A then generates a random number N_a, encrypts the pair (ID_a, N_a) by using user B's public key P_b to generate a ciphertext $C_3 = E((ID_a, N_a), P_b)$, and sends C_3 to user B in a message (m_3).

4. On receiving m_3, user B decrypts C_3 by using S_b and retrieves its contents. By seeing ID_a in the message, user B knows that user A wants to establish a logical communication channel with it. In order to get the authentic public key of user A, user B contacts the PKM. For this, it sends a message (m_4) to the PKM requesting for user A's public key. The message contains a code for the request (R_b), a timestamp (T_2), the user identifier of A (ID_a), and the user identifier of B (ID_b). This message is encrypted by using P_k. That is, m_4 contains the ciphertext $C_4 = E((R_b, T_2, ID_a, ID_b), P_k)$.

5. On receiving m_4, the PKM decrypts C_4 by using S_k. It then extracts the public keys P_a and P_b that correspond respectively to the user identifiers ID_a and ID_b of the



$m_1 = C_1 = E\ ((R_a, T_1, ID_a, ID_b), P_k)$
where R_a = code for the request made by user A
T_1 = a timestamp

$m_2 = C_2 = E\ ((R_a, T_1, P_b), P_a)$

$m_3 = C_3 = E\ ((ID_a, N_a), P_b)$
where N_a = a random number generated by user A

$m_4 = C_4 = E\ ((R_b, T_2, ID_a, ID_b), P_k)$
where R_b = code for the request made by user B
T_2 = a timestamp

$m_5 = C_5 = E\ ((R_b, T_2, P_a), P_b)$

$m_6 = C_6 = E\ ((N_a, N_b), P_a)$
where N_b = a random number generated by user B

$m_7 = C_7 = E\ (N_b, P_b)$

Fig. 11.5 The method of key distribution in an asymmetric cryptosystem.

message. By using key P_b, the PKM encrypts the triple (R_b, T_2, P_a) to generate a ciphertext $C_5 = E\ ((R_b, T_2, P_a), P_b)$. Finally it sends C_5 to user B in the form of a message (m_5).

6. On receiving m_5, user B decrypts C_5 by using S_b, retrieves its contents, and compares R_b and T_2 with the originals to get confirmed that m_5 is really the reply of m_4. User B then generates a random number N_b, encrypts the pair (N_a, N_b) by using the key P_a to generate a ciphertext $C_6 = E\ ((N_a, N_b), P_a)$, and sends C_6 to user A in a message (m_6).

7. On receiving m_6, user A decrypts C_6 by using S_a, retrieves its contents, and compares the received N_a with the original. If they match, user A becomes sure that user

B is authentic. User A then encrypts N_b by using the key P_b to generate a ciphertext $C_7 = E(N_b, P_b)$ and sends C_7 to user B in a message (m_7).

8. On receiving m_7, user B decrypts C_7 by using S_b and compares the received N_b with the original. If they match, user B also gets confirmed that user A is authentic. This is sufficient to achieve mutual confidence and allows regular communication to start.

Note that the protocol paradigms described above for key distribution in symmetric and asymmetric cryptosystems illustrate basic design principles only. A realistic protocol is necessarily a refinement of these basic paradigms.

11.4 AUTHENTICATION

Authentication deals with the problem of verifying the identity of a user (person or program) before permitting access to the requested resource. That is, an authentication mechanism prohibits the use of the system (or some resource of the system) by unauthorized users by verifying the identity of a user making a request.

Authentication basically involves identification and verification. *Identification* is the process of claiming a certain identity by a user, while *verification* is the process of verifying the user's claimed identity. Thus, the correctness of an authentication process relies heavily on the verification procedure employed.

The main types of authentication normally needed in a distributed system are as follows:

1. *User logins authentication.* It deals with verifying the identity of a user by the system at the time of login.
2. *One-way authentication of communicating entities.* It deals with verifying the identity of one of the two communicating entities by the other entity.
3. *Two-way authentication of communicating entities.* It deals with mutual authentication, whereby both communicating entities verify each other's identity.

A description of the authentication mechanisms that are commonly used to perform these types of authentication is presented below. Note that the authentication protocol paradigms described below illustrate basic design principles only. A realistic protocol is necessarily a refinement of these basic paradigms and addresses weaker environment assumptions, stronger postconditions, or both.

11.4.1 Approaches to Authentication

The basic approaches to authentication are as follows [Shankar 1977, Woo and Lam 1992]:

1. *Proof by knowledge.* In this approach, authentication involves verifying something that can only be known by an authorized principal. Authentication of a user

based on the password supplied by him or her is an example of proof by knowledge. Authentication methods based on the concept of proof by knowledge are again of two types—direct demonstration method and challenge-response method. In the *direct demonstration* method, a user claims his or her identity by supplying information (like typing in a password) that the verifier checks against prestored information. On the other hand, in the *challenge-response* method, a user proves his or her identity by responding correctly to the challenge questions asked by the verifier. For instance, when signing up as a user, the user picks a function, for example, $x + 18$. When the user logs in, the system randomly selects and displays a number, say 105, in which case the user must type 123 for authentication to be successful. For further security improvement, several functions may be used by the same user. At the time of login, the function to be used will depend on when the login is made. For example, a user may use seven different functions, one for each day of the week. In another variation of this method, a list of questions such as what is the name of your father, what is the name of your mother, what is the name of your street on which your house is located, is maintained by the system. When signing up as a user, the user has to reply to all these questions and the system stores the answers. At login, the system asks one of these questions at random and verifies the answer supplied by the user.

2. *Proof by possession.* In this approach, a user proves his or her identity by producing some item that can only be possessed by an authorized principal. The system is designed to verify the produced item to confirm the claimed identity. For example, a plastic card with a magnetic strip on it that has a user identifier number written on it in invisible, electronic form may be used as the item to be produced by the user. The user inserts the card in a slot meant for this purpose in the system's terminal, which then extracts the user identifier number from the card and checks to see if the card produced belongs to an authorized user. Obviously, security can be ensured only if the item to be produced is unforgeable and safely guarded.

3. *Proof by property.* In this approach, the system is designed to verify the identity of a user by measuring some physical characteristics of the user that are hard to forge. The measured property must be distinguishing, that is, unique among all possible users. For example, a special device (known as a biometric device) may be attached to each terminal of the system that verifies some physical characteristic of the user, such as the person's appearance, fingerprints, hand geometry, voice, signature. In deciding the physical characteristic to be measured, an important factor to be considered is that the scheme must be phycologically acceptable to the user community. Biometric systems offer the greatest degree of confidence that a user actually is who he or she claims to be, but they are also generally the most expensive to implement. Moreover, they often have user acceptance problems because users see biometric devices as unduly intrusive.

Of the three authentication approaches described above, proof by knowledge and possession can be applied to all types of authentication needs in a secure distributed system, while proof by property is generally limited to the authentication of human users by a system equipped with specialized measuring instruments. Moreover, in practice, a system may use a combination of two or more of these authentication methods. For

example, the authentication mechanism used by automated cash-dispensing machines usually employ a combination of the first two approaches. That is, a user is allowed to withdraw money only if he or she produces a valid identification card and specifies the correct password corresponding to the identification number on the card.

11.4.2 User Login Authentication

As in centralized systems, a user gains access to a distributed system by logging in a host in the system. User identity is established at login, and all subsequent user activities are attributed to this identity. Correct user identification at the time of login is crucial to the functioning of a secure system because all access-control decisions and accounting functions are based on this identity. Although any of the three basic approaches to authentication can be employed for user login authentication, the proof by knowledge is the most widely used method. In particular, most systems employ the direct demonstration method based on passwords.

In the authentication scheme based on passwords, the system maintains a table of authorized users' login names and their corresponding passwords. When a user wants to log in, the system asks the user to type his or her name and password. If the password supplied by the user matches the password stored in the system against his or her name, it is assumed that the user is legitimate and login is permitted; otherwise it is refused.

To provide good security and be practically useful, a password-based authentication system must have mechanisms for the following:

1. Keeping passwords secret
2. Making passwords difficult to guess
3. Limiting damages done by a compromised password
4. Identifying and discouraging unauthorized user logins
5. Single sign-on for using all resources in the system

The commonly used mechanisms for dealing with these issues are described below.

Keeping Passwords Secret

A user is responsible for keeping his or her password secret outside the computer system (external world). How to keep passwords secret in the external world is not of concern to operating system designers except for the fact that while a password is being typed in, it should not be displayed on the terminal, to prevent it from being seen by prying eyes near the terminal. The main concern, however, is to prevent an intruder from obtaining somebody's password by having access to the system's password table. The password table is of course protected and is accessible only to the authentication program. However, there is a great chance that the password table is exposed by accident or that the system administrator has access to the table. Therefore, instead of storing the names and passwords in plaintext form, they are encrypted and stored in ciphertext form in the table. In this case, instead of directly using a user-specified name and password for table lookup,

they are first encrypted and then the results are used for table lookup. Notice that for implementing this scheme, the main requirement is that even if both the encryption function and the password table are known, it is impossible to find the original password. That is, a noninvertible function is needed for encryption because in the scheme we never need to decrypt a ciphertext. This is an example of the use of *one-way cipher*. In cryptography we use the function

$$C = E(P, K)$$

where K is a key, P is a plaintext, and C is the resulting ciphertext. A one-way cipher can be implemented by letting the plaintext P serve as the key to E such that

$$C = E(P, P)$$

In a one-way cipher, it is very difficult to invert the encryption function (E) even if the intruder knows the key. Furthermore, if the key is not known (as in the case of password security scheme in which a password itself is used as the key), the inversion becomes much more difficult because the intruder cannot know which functions (and how many times of them) are used.

To keep passwords secret in a distributed environment, it is important that passwords should never be sent across the network in plain text form. Moreover, they should not be stored on normal servers but should only be stored on trusted servers that are well protected. Notice that due to these requirements, authentication of a user by simply sending his or her password to an authentication server for approval does not work in a distributed environment. The Kerberos authentication system (described later in this chapter) provides an interesting solution to this problem.

Making Passwords Difficult to Guess

Use of mechanisms to keep passwords secret does not guarantee that the system's security cannot be broken. It only says that it is difficult to obtain passwords. The intruder can always use a trial-and-error method. Virtually, in case of passwords, break-ins usually consist of guessing a user name and password combination. How successful an intruder can be in guessing passwords is obvious from the study made by Morris and Thompson [1979] of passwords on UNIX systems. They compiled a list of likely passwords that contained first and last names of persons, names of cities and streets, words from a small dictionary spelled correctly and backward, short strings of random characters, and license plate numbers. These passwords were then encrypted using the known password encryption algorithm to obtain a list of encrypted guessed passwords. Then another list of encrypted passwords was prepared by using entries in password tables of various UNIX systems. The entries in the two lists were then compared to find matching entries. Surprisingly, it was found that over 86% of actual passwords had a match in the list of guessed passwords.

A test of only a limited set of potential strings tends to reveal most passwords because there is a strong tendency for people to choose relatively short and simple passwords that they can remember. Some techniques that may be used to make the task of guessing a password difficult are as follows:

1. *Longer passwords.* The length of a password determines the ease with which a password can be found by exhaustion. For example, a three-digit password provides 1000 variations whereas a four digit password provides 10,000 variations. Longer passwords are less susceptible to enumeration because the work involved in enumerating all possible passwords increases by increasing the length of the password. Use of longer passwords can be enforced or encouraged by providing a password entry program that asks a user to enter a longer password if he or she enters a short one.

2. *Salting the password table.* Another technique to make the task of guessing passwords difficult is to artificially lengthen passwords by associating an n-bit random number with each password when it is first entered. The random number is changed whenever the password is changed. Instead of just storing the encrypted password in the password table, the password and the random number are first concatenated and then encrypted together. This encrypted result is stored in the password table. In this scheme, the password table has an additional field that contains the random number in its unencrypted form. At the time of login, this random number is concatenated with the entered password, encrypted, and then compared with the stored encrypted value. Guessing of a password becomes difficult because if an intruder suspects that *Tokyo* might be the password of a user, it is no longer enough to just encrypt *Tokyo* and compare the encrypted result with the value stored in the password table. Rather, for each guess made, the intruder has to try out 2^n strings, such as *Tokyo0000*, *Tokyo0001*, *Tokyo0002*, and so forth, for $n = 4$. This increases the encryption and comparison time for each guess made. UNIX uses this method with $n = 12$.

3. *System assistance in password selection.* A password can be either system generated or user selected. User-selected passwords are often easy to guess. A system can be designed to assist users in using passwords that are difficult to guess. This can be done in two ways. One way is to store a list of easy-to-guess passwords within the system and to first compare a user-selected password with the entries in the list. If a match is found, the system refuses to accept the selected password and asks the user to select another password informing him or her that the selected password is easy to guess. Another way of providing system assistance is to have a password generator program that generates random, easy-to-remember, meaningless words, such as *mounce*, *bulbul*, *halchal*, that can be used as passwords. The user selects a password suggested by the system and uses it with his or her own choice of a mixture of upper- and lowercase letters of that word, such as *hAlCHal*.

Identifying and Discouraging Unauthorized User Logins

Some management techniques should be used to improve the security of a system against unauthorized user logins. Three such techniques are as follows:

1. *Threat monitoring.* This technique detects security violations by checking for suspicious patterns of activity. For example, the system may count the number of incorrect passwords given when a user is trying to log in, and after, say, three failed login attempts, the system notifies security personnel by setting an alarm.

2. *Audit logs.* In this technique, the system maintains a record of all logins. That is, the time at which login was done, the duration of login, the accessed objects and the types of accesses made, etc., are recorded for each login. When a user logs in successfully, the system reports to the user some of the recorded information (such as time and duration) of the previous login. This may be helpful for the user to detect possible break-ins. After a security violation has been detected, the audit log can be used to detect which objects were accessed by the intruder and the amount of damage done. This information can be useful for recovery from the damages done by the intruder.

3. *Baited traps.* Intruders may also be caught by laying baited traps. For example, the system may maintain some special login names with easy passwords, such as login name: user, password: user. Whenever anyone logs in using any of these names, the system security personnel are immediately notified.

Single Sign-on

In a distributed client-server environment, a user might have several client programs running on his or her host node that access different server programs on remote nodes. In such an environment, the servers must authenticate that the clients run on behalf of a legitimate user. If a simple password-based authentication scheme is used, the user's password must be presented each time a server wants to authenticate a client program running on behalf of the user. This is cumbersome and inconvenient because a user will certainly not want to enter a password each time he or she accesses a new service. This is also not desirable from the point of view of transparency, which aims to provide a virtual single system image. A simple way to solve this problem is to cache the user's password on his or her host computer and use it from the cache every time a new service is accessed. However, this solution does not work because it is dangerous to keep passwords in cache from the point of view of keeping passwords secret. Once again, the Kerberos authentication system (described later in this chapter) provides an interesting solution to this problem.

Limiting Damages Done by a Compromised Password

It is suggested that a user should change his or her password frequently so that an intruder who has been successful in guessing the current password will have to guess a new one after the current password is changed by the user. In this way, the damage done by the intruder can be reduced. The extreme form of this approach is the use of *one-time passwords*. There are three different methods to implement the idea of one-time passwords. In the first method, the user gets a book containing a list of passwords. For each login, the user must use the next password in the list. Obviously, the user must keep the password book in a secure place. In the second method, every time a user logs out, the system asks to select a new password for the next login and replaces the old password with the new one. The user either remembers the new password or notes it down somewhere for use at the time of next login. The third method relies on the use of a special equipment such as smart cards or synchronized password generators. For example, a synchronized

password generator device generates a pseudorandom alphanumeric word or number that changes every minute or so and is time synchronized to a database stored in the computer. To log in, a user types in the word or number displayed on the card at the time of login. This results in a one-time password that is good only at that particular point in time and for only one login. One such device is the SecureID from Security Dynamics (Cambridge, Massachusetts). This method has not been very successful thus far, primarily due to the inconvenience and cost of the additional hardware required. It requires that all users purchase the hardware device. However, it has the advantage that once set up, it is fairly easy to administer (as opposed to a manual password list that requires frequent updating for all users).

11.4.3 One-Way Authentication of Communicating Entities

When an entity A wants to communicate with another entity B, B may like to verify the identity of A before allowing A to communicate with it. For example, a server may be designed to first verify the identity of any client that wants to communicate with it. The commonly used authentication protocols for one-way authentication of communicating entities are described below.

The following description is based on the material presented in [Woo and Lam 1992]. Since the authentication protocol paradigms directly use cryptosystems, their basic design principles also follow closely the type of cryptosystem used. Therefore, the protocols can be broadly classified into two categories—those based on symmetric cryptosystems and those based on asymmetric cryptosystems. Authentication protocols of both categories are based on the proof-by-knowledge principle.

Protocols Based on Symmetric Cryptosystems

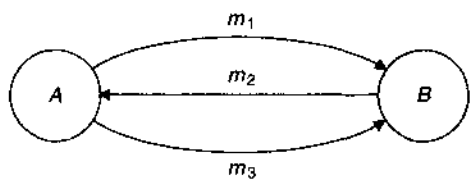
In a symmetric cryptosystem, the knowledge of the shared key allows an entity to encrypt or decrypt arbitrary messages. Without such knowledge, it is not possible for the entity to encrypt a message or to decrypt an encrypted message. Hence, in authentication protocols based upon symmetric cryptosystems, the verifier verifies the identity of a claimant by checking if the claimant can correctly encrypt a message by using a key that the verifier believes is known only to an entity with the claimed identity (outside the entities used in the verification process).

Let us assume that user A wants to communicate with user B but B wants to authenticate A before starting the communication. Also assume that K is the key of a symmetric cryptosystem that is shared between users A and B. The authentication protocol for this consists of the following steps:

1. User A encrypts its identifier (ID_a) by using key K to obtain a ciphertext $C_1 = E(ID_a, K)$. It then sends a message m_1 to user B, which contains ID_a and C_1.
2. On receiving m_1, user B decrypts C_1 by using key K and compares the obtained result with ID_a of the message. If they match, user A is accepted; otherwise it is rejected.

One major weakness of this protocol is its vulnerability to replays. That is, an intruder could masquerade as A by recording the message m_1 and later replay it to B. Replay attacks can be countered by using nonces or timestamps. A nonce-based challenge-response protocol that overcomes the problem of replay of messages works as follows (see Fig. 11.6):

1. User A sends its identifier (ID_a) to user B in plaintext form in a message m_1.
2. On receiving m_1, user B generates a random number N_r and sends N_r to user A in plaintext form in a message m_2.
3. On receiving m_2, user A encrypts N_r by using key K to obtain a ciphertext $C_1 = E(N_r, K)$. It then sends C_1 to B in a message m_3.
4. On receiving m_3, user B decrypts C_1 by using key K and compares the obtained result with the original value of N_r. If they are equal, user A is accepted; otherwise it is rejected.



$m_1 = ID_a$ = identifier of user A

$m_2 = N_r$ = a random number grenerated by user B

$m_3 = C_1 = E(N_r, K)$
where K is the symmetric key shared between users A and B

B decrypts C_1 using K and compares the result with original N_r.
If they are equal, A isaccepted, otherwise rejected.

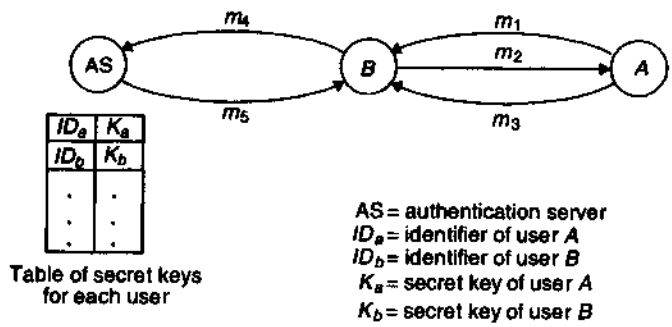
Fig. 11.6 One-way authentication protocol based on symmetric cryptosystem.

In this protocol, the freshness of N_r (whose value is different for each conversation) guarantees that an intruder cannot masquerade as A by replaying a recording of an old authentication conversation between A and B.

Although the above-described protocol functions correctly, it is impractical for a general large-scale system due to the following reasons:

1. Notice that the scheme requires that each user must store the secret key for every other user it would ever want to authenticate. This may not be practically feasible in a large system having too many users and in which the number of users keeps changing frequently.
2. The compromise of one user can potentially compromise the entire system.

To overcome these problems, the use of a centralized *authentication server (AS)* was proposed in [Needham and Schroeder 1978]. Each user in the system shares with the AS a prearranged secret key. The AS is a generally trusted entity and is shared by all communicating users of the system. When the AS is used, the authentication protocol takes the form shown in Fig. 11.7 (it has been assumed that K_a and K_b are the secret keys of users A and B, respectively, that are shared with the AS).



$m_1 = ID_a$

$m_2 = N_r$ = a random number generated by user B

$m_3 = C_1 = E\ (N_r, K_a)$

$m_4 = C_2 = E\ ((ID_a, C_1), K_b)$

The AS retrieves N_r by first decrypting C_2 with K_b and then decrypting C_1 with K_a.

$m_5 = C_3 = E\ (N_r, K_b)$

B decrypts C_3 with K_b and compares the result with original N_r. If they are equal, then A isaccepted, otherwise rejected.

Fig. 11.7 One-way authentication protocol based on symmetric cryptosystem and the use of a centralized authentication server.

1. User A sends its identifier (ID_a) to user B in plaintext form in a message m_1.
2. On receiving m_1, user B generates a random number N_r and sends N_r to user A in plaintext form in a message m_2.
3. On receiving m_2, user A encrypts N_r by using its secret key K_a to obtain a ciphertext $C_1 = E\ (N_r, K_a)$. It then sends C_1 to B in a message m_3.
4. On receiving m_3, user B encrypts the pair (ID_a, C_1) by using its secret key K_b to generate a ciphertext $C_2 = E\ ((ID_a, C_1), K_b)$. It then sends C_2 to the AS in a message m_4.
5. On receiving m_4, the AS decrypts C_2 with key K_b and retrieves the pair (ID_a, C_1). It then extracts from its database the key (K_a) that corresponds to ID_a and decrypts

C_1 with key K_a and retrieves N_r. Next, it encrypts N_r by using key K_b to generate a ciphertext $C_3 = E(N_r, K_b)$. Finally, it sends C_3 to B in a message m_5.

6. On receiving m_5, user B decrypts it by using its secret key K_b, retrieves N_r, and compares it with the original value of N_r. If they are equal, user A is accepted; otherwise it is rejected.

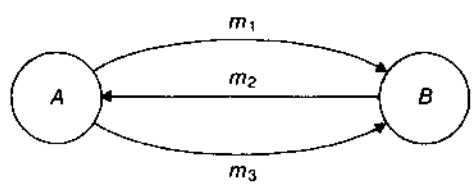
As compared to the previous protocol, the key distribution and storage problems are greatly alleviated because now each user needs to keep only one key. Moreover, the system's security can be greatly improved simply by tightening security for the AS because the risk of compromise is mostly shifted to the AS. The centralized AS, however, suffers from the same drawbacks as the centralized KDC. These drawbacks can also be solved by using fully/partially distributed ASs in a similar manner as described for the key distribution problem in Section 11.3.4.

Protocols Based on Asymmetric Cryptosystems

In an asymmetric cryptosystem, the public key of each user is published while the secret key of each user is known only to the user and no one else. Hence, in authentication protocols based upon asymmetric cryptosystems, the verifier verifies the identity of a claimant by checking if the claimant can correctly encrypt a message by using the secret key of the user whose identity is being claimed.

Let us assume that user A wants to communicate with user B but B wants to authenticate A before starting the communication. Also assume that P_a and S_a are the public and secret keys of user A, and P_b and S_b are the public and secret keys of user B. The authentication protocol for this consists of the following steps (see Fig. 11.8):

1. User A sends its identifier (ID_a) to user B in plaintext form in a message m_1.
2. On receiving m_1, user B generates a random number N_r and sends N_r to user A in plaintext form in a message m_2.



$m_1 = ID_a$ = identifier of user A

$m_2 = N_r$ = a random number generated by user B

$m_3 = C_1 = E(N_r, S_a)$

where S_a = secret key of user A

B decrypts C_1 by using the public key of user A and compares the result with original N_r. If they are equal, A isaccepted, otherwise rejected.

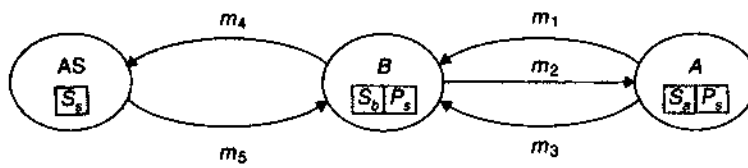
Fig. 11.8 One-way authentication protocol based on asymmetric cryptosystem.

3. On receiving m_2, user A encrypts N_r by using its secret key S_a to obtain a ciphertext $C_1 = E(N_r, S_a)$. It then sends C_1 to B in a message m_3.
4. On receiving m_3, user B decrypts C_1 by using the public key of user A (P_a) and compares the obtained result with the original value of N_r. If they are equal, user A is accepted; otherwise it is rejected.

As in the case of protocols based upon symmetric cryptosystems, in this case also a centralized AS may be used to greatly alleviate the key distribution and storage problems. In this case, the AS maintains a database of all published public keys and each user in the system keeps a copy of the public key (P_s) of the AS. When the AS is used, the authentication protocol takes the following form (see Fig. 11.9):

1. User A sends its identifier (ID_a) to user B in plaintext form in a message m_1.

2. On receiving m_1, user B generates a random number N_r and sends N_r to user A in plaintext form in a message m_2.



ID_a	P_a
ID_b	P_b
.	.
.	.
.	.

Database of public keys

P_a = public key of A
S_a = secret key of A
P_b = public key of B
S_b = secret key of B
P_s = public key of AS
S_s = secret key of AS
ID_a = identifier of A
ID_b = identifier of B

$m_1 = ID_a$

$m_2 = N_r$ = a random number generated by B

$m_3 = C_1 = E(N_r, S_a)$

$m_4 = (R_b, ID_a)$
where R_b = code for requesting the public key of a user

$m_5 = C_2 = E((ID_a, P_a), S_s)$

B retrieves N_r by first decrypting C_2 with P_s and then decrypting C_1 with P_a. The value of N_r obtained in this way is compared with the original value of N_r. If they are equal, then A is accepted, otherwise rejected.

Fig. 11.9 One-way authentication protocol based on asymmetric cryptosystem and the use of a centralized authentication server.

3. On receiving m_2, user A encrypts N_r by using its secret key S_a to obtain a ciphertext $C_1 = E(N_r, S_a)$. It then sends C_1 to B in a message m_3.

4. On receiving m_3, user B sends the pair (R_b, ID_a) to the AS in plaintext form in a message m_4, where R_b is a request code for requesting the public key of the user whose identifier is specified in the second element of the message.

5. On receiving m_4, the AS extracts from its database the public key (P_a) of the user whose identifier is ID_a. It then encrypts the pair (ID_a, P_a) by using its own secret key S_s to generate a ciphertext $C_2 = E((ID_a, P_a), S_s)$. Finally, it sends C_2 to B in a message m_5.

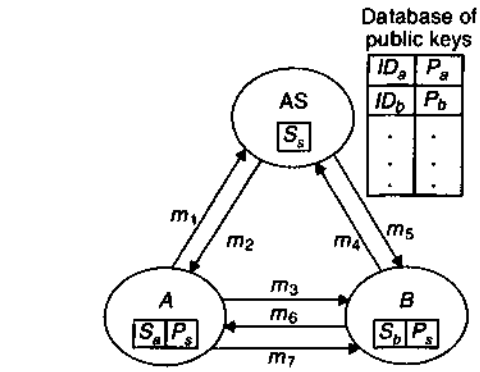
6. On receiving m_5, user B decrypts C_2 by using the public key of the AS (P_s) and retrieves the pair (ID_a, P_a). Now by using the key P_a, it decrypts C_1 and compares the obtained result with the original value of N_r. If they are equal, user A is accepted; otherwise it is rejected.

Notice that in this case each user needs to keep only one public key, the public key of the AS. Also notice that in the above protocol it has been assumed that the asymmetric cryptosystem is commutative. That is, the public and secret keys function in either order. Therefore, if the public key is used for encryption, then the secret key can be used for decryption, whereas if the secret key is used for encryption, then the public key can be used for decryption.

11.4.4 Two-Way Authentication of Communicating Entities

In a distributed system, tasks are often distributed over multiple hosts to achieve a higher throughput or more balanced utilization of resources than centralized systems. Correctness of such a distributed task depends on whether peer processes participating in the task can correctly identify each other. Two-way authentication protocols allow both communicating entities to verify each other's identity before establishing a secure logical communication channel between them.

Obviously, mutual authentication can be achieved by performing one-way authentication twice. That is, if two communicating users A and B want to authenticate each other, A can first authenticate B by performing one-way authentication, and then B can authenticate A by repeating the same process, but with the roles of A and B reversed. However, this may turn out to be costlier than a protocol designed specially for two-way authentication. For example, if the protocol of Figure 11.9 is repeated twice for performing two-way authentication, a total of 10 messages will be required. However, a protocol for mutual authentication of communicating entries that requires only seven messages and that is also based upon an asymmetric cryptosystem and uses a centralized AS is described below. In the description, it has been assumed that two users A and B want to authenticate each other. Here, P_a and S_a are the public and secret keys of user A, and P_b and S_b are the public and secret keys of user B. Moreover, P_s and S_s are the public and secret keys of the AS. The authentication protocol consists of the following steps (see Fig. 11.10):



$m_1 = (R_a, ID_a, ID_b)$
where R_a = code for the request made by user A

$m_2 = C_1 = E\ ((ID_b, P_b), S_s)$

$m_3 = C_2 = E\ ((ID_a, N_a), P_b)$
where N_a is a random number generated by user A

$m_4 = C_3 = E\ ((R_b, ID_a, ID_b, N_a), P_s)$

$m_5 = (C_4, C_6)$
where $C_4 = E\ ((ID_a, P_a), S_s)$ and
$C_6 = E\ (C_5, P_b)$
where $C_5 = (ID_b, K, N_a), S_s)$
where K is a session key generated by the AS for users A and B.

$m_6 = C_7 = E\ ((C_5, N_b), P_a)$
where N_b is a random number generated by user B

$m_7 = C_8 = E\ (N_b, K)$

B decrypts C_8 with the session key K and compares the result with the original value of N_b. If they are equal, this is sufficient to prove that the logical communication channel established between A and B with key K is a newly established channel and is secure.

Fig. 11.10 Two-way authentication protocol based on asymmetric cryptosystem and the use of a centralized authentication server.

1. User A sends a request message (m_1) to the AS indicating that it wants to establish a secure logical communication channel with user B. The message contains a code for the request (R_a), the identifier of user A (ID_a), and the identifier of user B (ID_b). This message is sent in plaintext form.

2. On receiving m_1, the AS extracts from its database the public key P_b that corresponds to the user identifier ID_b of the message. By using its secret key S_s, the AS encrypts the pair (ID_b, P_b) to generate a ciphertext $C_1 = E((ID_b, P_b), S_s)$. It then sends C_1 to A in a message m_2.

3. On receiving m_2, user A decrypts C_1 by using the public key of the AS (P_s) and retrieves its contents. It then generates a random number N_a, encrypts the pair (ID_a, N_a) by using the public key of user B (P_b) to generate a ciphertext $C_2 = E((ID_a, N_a), P_b)$, and sends C_2 to B in a message m_3.

4. On receiving m_3, user B decrypts C_2 by using its secret key S_b and retrieves its contents. It then sends a message m_4 to the AS requesting for the public key of user A and a session key for the secure logical communication channel between A and B. The message contains the request code (R_b), ID_a, ID_b, and N_a. Before being sent, the message is encrypted with the public key of the AS (P_s). That is, the message contains the ciphertext $C_3 = E((R_b, ID_a, ID_b, N_a), P_s)$.

5. On receiving m_4, the AS decrypts C_3 with its secret key (S_s) and retrieves its contents. It generates a new session key K for A and B. Next it generates three ciphertexts: $C_4 = E((ID_a, P_a), S_s)$, $C_5 = E((ID_b, K, N_a), S_s)$, and $C_6 = E(C_5, P_b)$. Finally, it sends C_4 and C_6 to B in a message m_5.

6. On receiving m_5, user B decrypts C_4 and C_6 with P_s and S_b, respectively, and retrieves their contents. It then generates a random number N_b and creates a ciphertext $C_7 = E((C_5, N_b), P_a)$. Finally, it sends C_7 to A in a message m_6. Notice that C_5 is obtained by decrypting C_6.

7. On receiving m_6, user A first decrypts C_7 by using its secret key S_a and then decrypts C_5 by using the public key of the AS (P_s). Now both users A and B have the session key K. User A next generates a ciphertext $C_8 = E(N_b, K)$ and sends C_8 to B in a message m_7.

8. On receiving m_7, user B decrypts C_8 by using the session key K and compares the result with the original value of N_b. If they are equal, this is sufficient to prove that the logical communication channel established between A and B with key K is a newly established channel and is secure. Both users A and B are now sure of each other's identity.

Notice that although the authentication protocol is based upon an asymmetric cryptosystem, the actual communications between users A and B, after the secure logical communication channel is established between them, take place by a symmetric cryptosystem.

11.4.5 Case Study: Kerberos Authentication System

The need for secure authentication in distributed computing systems has led to the design of authentication standards and systems for this purpose. For example, X.509 identifies the CCITT X.500 directory authentication framework standard [Smart 1994], Kerberos is a network authentication system developed at MIT [Neuman and Theodore 1994, Stallings 1994], and SPX is an experimental authentication system developed by the Digital Equipment Corporation [Khanna 1994]. Of several authentication systems developed to date, the Kerberos system is the most popular one and is continuing to evolve. It has been declared the Internet standard and has become the de facto standard for remote authentication in networked client-server environments. Therefore, a description of this system is presented here.

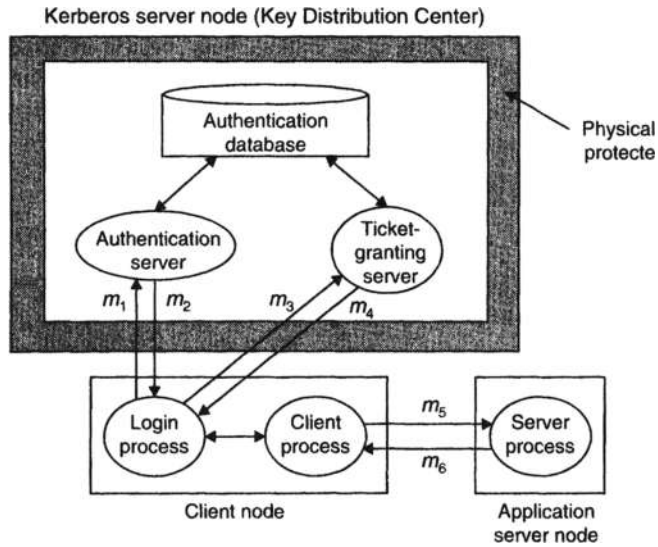
Kerberos was developed at MIT as part of its project Athena [Champine et al. 1990]. It is named after the three-headed dog of Greek mythology that guards the entrance to Hades. Its design is based on the ideas of Needham and Schroeder [1978] for key distribution and authentication that we have already seen in Sections 11.3.4 and 11.4.3, respectively. It is now available in both commercial and public-domain implementations and is widely used at MIT and elsewhere to provide secure access to resources in distributed environments. It is used by Transarc's AFS file system and is the underlying component of the OSF's DCE Security Server (described later in this chapter). Several vendors, including DEC, Hewlett-Packard, and IBM, now offer Kerberos implementations as part of their standard middleware offerings on commercial UNIX and midrange server platforms. Version 5, the most recent version of Kerberos, is described here.

Kerberos System Architecture

The system architecture of Kerberos is shown in Figure 11.11. It consists of the following basic components:

1. *Kerberos server.* The key component of a Kerberos system is a Kerberos server that acts as a key distribution center. Each Kerberos server has an authentication database, an authentication server, and a ticket-granting server. The authentication database has the user ID and password of all users of the system. Moreover, the Kerberos server shares a unique secret key with each server in the system. Therefore, the authentication database also has the server ID and secret key for all servers in the system. The passwords and secret keys are distributed physically or in some other secure manner as part of the Kerberos installation. Kerberos uses the DES algorithm to generate the keys and encrypt messages, but this is implemented as a separate module that can be easily replaced by any other suitable algorithm.

The authentication server performs the task of verifying user's identity at the time of login without requiring the password to travel over the network. Kerberos has single sign-on facility. Therefore, a user has to enter his or her password only once at the time of login no matter how many different resources are accessed by the user after that.



ID_g = ticket-granting server's identifier
ID_c = client's identifier
ID_s = applicationserver's identifier
N_i = a nonce
K_c = client's secret key
K_s = applicationserver's secret key
K_g = ticket-granting server's secret key
K_1 = ticket-granting ticket session key
K_2 = service-granting ticket session key
T_{si} = starting time of validity of ticket
T_{ei} = ending time of validity of ticket
T_i = a timestamp

$m_1 = (ID_c, N_1)$

$m_2 = C_2 = E((N_1, K_1, C_1), K_c)$, where $C_1 = E((ID_c, ID_g, T_{s1}, T_{e1}, K_1), K_g)$

$m_3 = (ID_s, N_2, C_1, C_3)$, where $C_3 = E((ID_c, T_1), K_1)$

$m_4 = C_5 = E((N_2, K_2, C_4), K_1)$, where $C_4 = E((ID_c, ID_s, T_{s2}, T_{e2}, K_2), K_s)$

$m_5 = (C_4, C_6)$, where $C_6 = E((ID_c, T_2), K_2)$

$m_6 = C_7 = E(T_3, K_2)$, where $T_3 = T_2 + 1$

Fig. 11.11 Kerberos authentication protocol.

The ticket-granting server performs the task of supplying tickets to clients for permitting access to other servers in the system. These tickets are used to establish secure logical communication channels between clients and servers by performing mutual authentication.

Since the Kerberos server has valuable information in its authentication database that must be kept secret, it is extremely important that it be installed on a carefully protected and physically secure machine. Although there is no technical problem in installing the Kerberos server on the same machine that has another application, it is always better for security reasons to have a dedicated machine for the Kerberos server. The number of users having access permission to this machine should be extremely limited.

2. *Client.* The second component of a Kerberos system is comprised of client processes that usually run on workstations located in effectively public places where their consoles are available to whatever user happens to be physically in front of them. Therefore, they are completely untrusted. Users (on whose behalf client processes run) must first get their identification verified by the Kerberos server before attempting to access any other server in the system. Once a user's identity has been verified, each client process running on his or her behalf must obtain a ticket from the ticket-granting server for communicating with a server that it wants to access.

3. *Application server.* The third component of a Kerberos system is the application server, also known simply as the server. A server provides a specific type of service to a client upon request only after verifying the authenticity of the client. A server usually runs on a machine that is located in a moderately secure room. Therefore, Kerberos ensures that a compromise of one server does not compromise another server.

Kerberos Authentication Protocol

With the idea of the basic components and their functions, we will now see the Kerberos protocol that explains how these components interact with each other to perform user login authentication and mutual authentication of client and server processes. The protocol is described below and is summarized in Figure 11.11. In the description, A, G, C, and S stand respectively for the authentication server, the ticket-granting server, the client, and the application server. Moreover, let K_a, K_g, and K_s be the secret keys of A, G, and S, respectively, and K_c be the secret key of C (this key is generated from the user's password by using a one-way function).

1. When a user logs on to a workstation by typing his or her login name, the login program sends a request to the authentication server for what is known as a ticket-granting ticket in a message m_1. Message m_1 contains the user's ID (login name) (ID_c) and a nonce N_1 that is used to check the validity of the reply. This message is sent in plaintext form.

2. On receiving m_1, the authentication server extracts the password of this user from the authentication database. It then generates a random number for use as a session key (K_1). After this, it creates a ticket-granting ticket that contains the user's ID (ID_c), the ticket-granting server's ID (ID_g), the starting time for validity of the ticket (T_{s1}), the ending time for validity of the ticket (T_{e1}) (typically on the order of 8 hours), and a copy of the session key (K_1). Making the ticket-granting ticket time sensitive prevents an unauthorized user from capturing it and using it at a later time. Now it encrypts this ticket by using the ticket-granting server's secret key (K_g) to generate a ciphertext $C_1 = E((ID_c, ID_g, T_{s1}, T_{e1}, K_1), K_g)$. This encryption ensures that no one (not even the client) can tamper with the ticket-granting ticket and only the Kerberos server can decode it. Next it uses the client's secret key (K_c) (generated from the user's password) to generate another ciphertext $C_2 = E((N_1, K_1, C_1), K_c)$. It then returns C_2 to the login program in a message m_2.

3. On receiving m_2, the login program prompts the user for his or her password. The entered password is run through a one-way function that generates the client's secret key (K_c) from the password. Immediately after obtaining K_c, the password is removed from the computer's memory to minimize the chance of password disclosure in the event of a client crash. The login program then attempts to decrypt C_2 by using K_c. If the user supplied the correct password, C_2 is successfully decrypted and the login program obtains the nonce, the session key, and the encrypted ticket from the message. It checks the nonce for validity of the reply and stores the session key and the encrypted ticket for subsequent use when communicating with the ticket-granting server. When this has been done, the client's secret key can also be erased from memory, since the ticket now serves to authenticate the user. A login session is then started for the user on the user's workstation.

Notice that user authentication is done without requiring the password to travel over the network. Moreover, if an intruder intercepts the reply message, it will be unable to decrypt it and thus be unable to obtain the session key and the ticket inside it.

4. Now when a client process running on the client workstation on behalf of the authenticated user wants to access the application server, it requests the ticket-granting server for a service-granting ticket that can be used to communicate with the application server. For this, the client creates an authenticator that contains the client's ID (ID_c) and a timestamp (T_1). It encrypts this authenticator by using the session key (K_1) to obtain a ciphertext $C_3 = E((ID_c, T_1), K_1)$. Unlike the ticket-granting ticket, which is reusable, this authenticator is intended for one-time use and has a very short life span (typically on the order of a few minutes). The client next sends the encrypted authenticator (C_3), the encrypted ticket-granting ticket (C_1), the ID of the application server (ID_s), and a nonce (N_2) to the ticket-granting server in a message m_3.

5. On receiving m_3, the ticket-granting server decrypts C_1 by using its secret key (K_g) and makes sure that it has not expired by comparing T_{e1} with the current time. It extracts the session key (K_1) from it and uses it to decrypt C_3 to obtain ID_c and T_1. The obtained ID_c is compared with the value of ID_c in the ticket-granting ticket to authenticate the source of the request. On the other hand, T_1 is used to ascertain the freshness of the request. If all verifications pass successfully, the ticket-granting server gets assured that the sender of the ticket is indeed the ticket's real owner.

Notice here that it is the authenticator and not the ticket-granting ticket that proves the client's identity. Therefore, the use of a ticket-granting ticket is merely a way to distribute keys securely. Moreover, since the authenticator can be used only once and has a very short life span, it is nearly impossible for an intruder to steal both the ticket-granting ticket and an authenticator for later use. Each time a client applies to the ticket-granting server for a new service-granting ticket, it sends its reusable ticket-granting ticket plus a fresh authenticator. Also notice that the reusability of the ticket-granting ticket allows support of a single sign-on facility in which a user need not enter his or her password every time it needs to access a new server.

After successful authentication of the client, the ticket-granting server generates a new random session key (K_2) and then creates a reusable service-granting ticket for access to the requested server. This ticket contains the client's ID (ID_c), the application server's

ID (ID_s), the starting time for validity of the ticket (T_{s2}), the ending time for validity of the ticket (T_{e2}), and a copy of the new session key (K_2). It then encrypts the service-granting ticket with the secret key of the application server (K_s) to obtain a ciphertext $C_4 = E((ID_c, ID_s, T_{s2}, T_{e2}, K_2), K_s)$. This encryption ensures that no one (not even the client) can tamper with the service-granting ticket and only the application server or the Kerberos server can decode it. Next it uses the old session key (K_1) to generate another ciphertext $C_5 = E((N_2, K_2, C_4), K_1)$. It then returns C_5 to the client in a message m_4.

6. On receiving m_4, the client decrypts C_5 by using the old session key (K_1) and obtains the nonce, the new session key, and the encrypted service-granting ticket. It checks the nonce for validity of the reply and stores the new session key and the encrypted ticket for subsequent use when communicating with the application server. The client is now ready to issue request messages to the application server. However, if mutual authentication is desired, before proceeding with its transaction or request for service, the client creates an authenticator that contains the client's ID (ID_c) and a timestamp (T_2). It encrypts this authenticator by using the session key (K_2) to obtain a ciphertext $C_6 = E((ID_c, T_2), K_2)$. The client next sends to the application server, in a message m_5, the encrypted authenticator (C_6), the encrypted service-granting ticket (C_4), and a request that the server reply with the value of the timestamp from the authenticator, incremented by 1, and encrypted in the session key.

7. On receiving m_5, the application server decrypts C_4 by using its secret key (K_s) and extracts the copy of the session key (K_2). It then uses it to decrypt C_6 to obtain ID_c and T_2. Next, it increments T_2 by 1 and encrypts the obtained value (T_3) with the session key K_2 to obtain a ciphertext $C_7 = E(T_3, K_2)$. It returns C_7 to the client in a message m_6.

8. On receiving m_6, the client decrypts C_7 by using the session key K_2 to obtain T_3. If $T_3 = T_2 + 1$, the client gets assured that the server is genuine. This mutual authentication procedure prevents any possibility of an intruder impersonating a server in attempt to gain access information from a client.

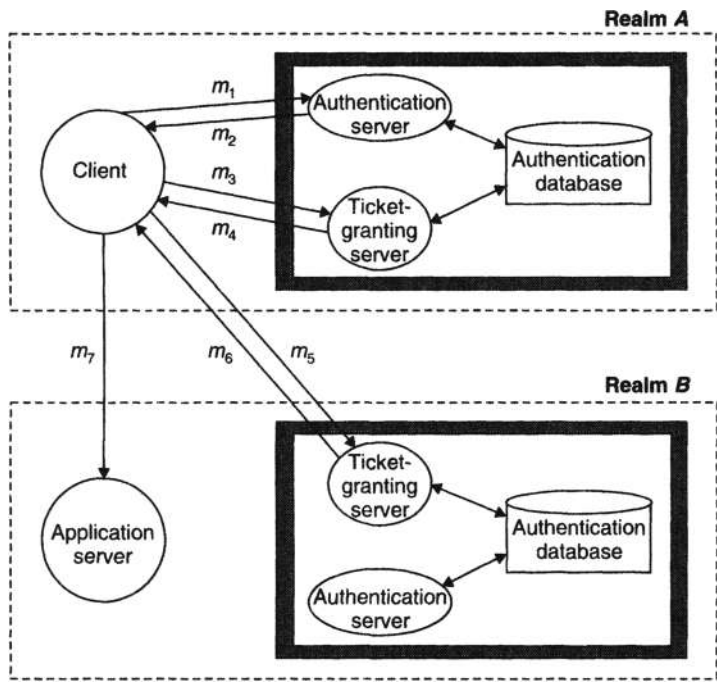
At the conclusion of this process, the client and server are assured of the establishment of a secure communication channel in between them. They also now share a session key (K_2), which they can use (if required) to encrypt future messages.

Interrealm Authentication in Kerberos

In a system that crosses organizational boundaries, it is not appropriate for all users and servers to be registered with a single Kerberos server. Therefore, in such an environment, multiple Kerberos servers exist, each responsible for a subset of the users and servers in the system. A subset of users and servers along with the Kerberos server with which they are registered is called a *realm* in Kerberos. In a typical implementation, networks of clients and servers belonging to different organizations usually constitute different realms. In such an environment, interrealm authentication facility is needed to allow a client to securely interact with a server belonging to a different realm. For this, a Kerberos server

of one realm is registered (shares a secret key) with the Kerberos servers of all other realms. The interrealm authentication protocol is shown in Figure 11.12 and is described below:

1. A client that wants to access a server in a different realm first makes a request for a service-granting ticket from its own ticket-granting server to access the ticket-granting server of the other realm.
2. With the obtained service-granting ticket, the client then makes a request for another service-granting ticket from the ticket-granting server of the other realm to access the application server of that realm.



m_1 = request for ticket-granting ticket

m_2 = reply for m_1

m_3 = request for service-granting ticket to access the remote ticket-granting server

m_4 = reply for m_3

m_5 = request for service-granting ticket to access the remote application server

m_6 = reply for m_5

m_7 = access request

Fig. 11.12 Interrealm authentication protocol of Kerberos.

3. With the newly obtained service-granting ticket, the client now sends its request to the desired application server of the other realm. The remote server then chooses whether to honor the client's request.

In the above approach for interrealm authentication, if it is desired that each Kerberos realm be able to operate with all other Kerberos realms, in a system having n realms, there must be $n(n-1)/2$ secure key exchanges. Therefore, this approach does not scale well. It creates a lot of overhead on the network and the Kerberos servers themselves. Thus, it is suggested that Kerberos implementations should use a few relatively large realms rather than too many small realms. The latest version of Kerberos (Version 5) also supports multihop interrealm authentication, allowing keys to be shared hierarchically [Neuman and Theodore 1994].

Within a single realm, the Kerberos server is a critical component for smooth functioning. Therefore, to ensure reliability of the Kerberos server, Kerberos supports replication of the Kerberos server. When replicated, a simple master-slave technique is used to keep the authentication databases of all the replicas of a Kerberos server consistent. That is, all changes are only applied to the master copy by a single Kerberos database management server (KDBMS) that runs only on the master Kerberos server's machine. Administrative operations such as adding or deleting users or requests for password change from users are handled by the KDBMS. Changes made to the master Kerberos server's authentication database are periodically propagated to the authentication databases of slave Kerberos servers.

Some Limitations of Kerberos

For all its popularity, however, Kerberos is not a complete security solution because of the following limitations [Bellovin and Merritt 1990, Neumann and Theodore 1994]:

1. Kerberos is not effective against password-guessing attacks. If a user chooses a password that is easy to guess, an intruder guessing that password can impersonate the user. Another way by which an intruder can get to know a user's password is by modifying the login program (a Trojan horse) that resides on the user's workstation. In this case, also, the intruder may obtain sufficient information to impersonate the user. To address these limitations, it is suggested that Kerberos should be combined with a one-time password technique. Commercial products that combine a one-time password technique with Kerberos are available.

2. The Kerberos protocol depends upon loose synchronization of clocks of the nodes present in a system. It may not be very difficult to meet this requirement, but the problem is that the synchronization protocol used for this purpose must itself be secure against security attacks. A Kerberos protocol that does not rely on synchronized clocks has been presented in [Kehne et al. 1992].

3. Another problem with Kerberos is that it is difficult to decide the appropriate lifetime of the tickets, which is generally limited to a few hours. For better security, it is desirable that this lifetime should be short so that users who have been unregistered or

downgraded will not be able to continue to use the resources for a long time. However, for user transparency, the lifetime should be as long as the longest possible login session, since the use of an expired ticket will result in the rejection of service requests. Once rejected, the user must reauthenticate the login session and then request new server tickets for all the services in use. Interrupting an application at an arbitrary point for reauthentication might not be acceptable in commercial environments.

4. Finally, client-server applications must be modified to take advantage of Kerberos authentication. This process is called *Kerberization*. Kerberizing an application is the most difficult part of installing Kerberos. Many large organizations may find it almost impossible to Kerberize their applications and turn to other solutions. Fortunately, the availability of Kerberized applications has improved with time and is expected to improve further. More and more vendors are now producing new Kerberized versions of their popular products.

11.5 ACCESS CONTROL

Once a user or a process has been authenticated, the next step in security is to devise ways to prohibit the user or the process from accessing those resources/information that he or she or it is not authorized to access. This issue is called *authorization* and is dealt with by using access control mechanisms. Access control mechanisms (also known as protection mechanisms) used in distributed systems are basically the same as those used in centralized systems. The main difference is that since all resources are centrally located in a centralized system, access control can be performed by a central authority. However, in a distributed client-server environment, each server is responsible for controlling access to its own resources.

When talking about access control in computer systems, it is customary to use the following terms:

1. *Objects*. An object is an entity to which access must be controlled. An object may be an abstract entity, such as a process, a file, a database, a semaphore, a tree data structure, or a physical entity, such as a CPU, a memory segment, a printer, a card reader, a tape drive, a site of a network.

Each object has a unique name that differentiates it from all other objects in the system. An object is referenced by its unique name. In addition, associated with each object is a "type" that determines the set of operations that may be performed on it. For example, the set of operations possible on objects belonging to the type "data file" may be *Open*, *Close*, *Create*, *Delete*, *Read*, and *Write*, whereas for objects belonging to the type "program file," the set of possible operations may be *Read*, *Write*, and *Execute*. Similarly, for objects of type "semaphore," the set of possible operations may be *Up* and *Down*, and for objects of type "tape drive," the set of possible operations may be *Read*, *Write*, and *Rewind*.

2. *Subjects*. A subject is an active entity whose access to objects must be controlled. That is, entities wishing to access and perform operations on objects and to which access authorizations are granted are called subjects. Examples of subjects are processes and

users. Note that subjects are also objects since they too must be protected. Therefore, each subject also has a unique name.

3. *Protection rules.* Protection rules define the possible ways in which subjects and objects are allowed to interact. That is, protection rules govern the subjects' access to objects. Therefore, associated with each (subject, object) pair is an *access right* that defines the subset of the set of possible operations for the object type that the subject may perform on the object. The complete set of access rights of a system defines which subjects can perform what operations on which objects. At any particular instance of time, this set defines the *protection state* of the system at that time.

The exact manner in which the protection rules are imposed to control the subjects' access to objects depends on the access control model used by the system. The following access control models have been proposed in the literature:

1. The access matrix model [Lampson 1971, Graham and Denning 1972, Harrison et al. 1976]
2. The information flow control model [Denning 1976, Bell and LaPadula 1973]
3. The security kernel model [Ames et al. 1983, Rushby and Randell 1983]

Of these, the access matrix model is the most popular one and is widely used in existing centralized and distributed systems. The other two models are mainly of theoretical interest. A description of the access control mechanisms based on the access matrix model is presented below.

11.5.1 Protection Domains

We saw that the principle of least privilege requires that at any time a subject should be able to access only those objects that it currently requires to complete its task. Therefore, from time to time, subjects may need to change the set of access rights they have to objects, depending on the particular task that they have to do at any time. The concept of a domain is commonly used to provide this type of flexibility of access control in a security system.

A *domain* is an abstract definition of a set of access rights. It is defined as a set of (object, rights) pairs. Each pair specifies an object and one or more operations that can be performed on the object. Each one of the allowed operations is called a *right.*

A security system based on the concept of domain defines a set of domains with each domain having its own set of (object, rights) pairs. At any instance of time, each process executes in one of the protection domains of the system. Therefore, at a particular instance of time, the access rights of a process are equal to the access rights defined in the domain in which it is at that time. A process can also switch from one domain to another during execution.

Figure 11.13 shows an example of a system having three protection domains D_1, D_2, D_3. The following points may be observed from this example:

1. Domains need not be disjoint. That is, the same access rights may simultaneously exist in two or more domains. For instance, access right (*Semaphore-1*, {*Up*, *Down*}) is in both domains D_1 and D_2. This implies that a subject in either of the two domains can perform *Up* and *Down* operations on *Semaphore-1*.

2. The same object can exist in multiple domains with different rights in each domain. For instance, rights for *File-1* and *File-2* are different in D_1 and D_2, and rights for *TapeDrive-1* are different in D_2 and D_3. Considering *File-1*, a subject in domain D_1 can *Read*, *Write*, and *Execute* it, but a subject in domain D_2 can only perform *Read* and *Write* operations on it. Therefore, in order to execute *File-1*, a subject must be in domain D_1.

3. If an object exists only in a single domain, it can be accessed only by the subjects in that domain. For instance, *File-3* can only be accessed by the subjects in domain D_3.

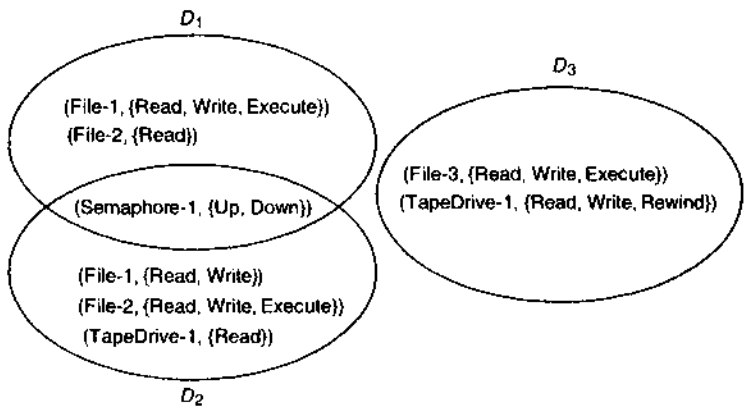


Fig. 11.13 A system having three protection domains.

Since a domain is an abstract concept, its realization and the rules for domain switching are highly system dependent. For instance, some ways of realizing a domain may be the following:

1. *Each user as a domain.* In this case, processes are assigned to domains according to the identity of the user on whose behalf they are executed. A domain switching occurs when a user logs out and another user logs in.
2. *Each process as a domain.* In this case, the access rights of a process are limited to the access rights of its own domain. A domain switching occurs when one process sends a message to another process and then waits for a response.
3. *Each procedure as a domain.* In this case, each procedure has its own set of access rights and a domain switching occurs when a procedure calls another procedure during the course of its execution.

In the protection scheme of UNIX, both the concepts of user-oriented domains and procedure-oriented domains are employed. Let us first see the use of the concept of user-oriented domains. In UNIX, a *user-id* (*uid*) and a *group-id* (*gid*) are associated with each user. A (*uid*, *gid*) pair identifies the list of objects and the types of operations that can be performed on these objects. The domain of a process depends on the user on whose behalf it is executed. Therefore, the domain of a process is defined by its (*uid*, *gid*) pair. That is, two processes with the same (*uid*, *gid*) pair will have exactly the same access rights and two processes with different (*uid*, *gid*) pairs will have different access rights. In the latter case, several of the access rights may be the same for the two processes. A process switches domains by acquiring a new *uid* or *gid* by executing the *setuid* or *setgid* commands. Furthermore, it is also worth mentioning here that the *superuser* in a UNIX system is not a person but a name for a special domain within which the most privileged processes, needing access to all of the objects in the system, may run.

Now let us see the use of the concept of procedure-oriented domains in UNIX. In UNIX, all the procedures are grouped into two classes—user procedures and kernel procedures. These two classes form two domains called the *user mode* and the *kernel mode*. A process running in the kernel mode has a different set of access rights as compared to a process running in the user mode. For example, in the kernel mode, a process can access all pages in the physical memory, the entire disk area, and all other protected resources. In this two-domain architecture, when a user process does a system call, it switches from the user mode to the kernel mode. Therefore, at a particular instance of time, the domain of a process and hence its access rights depend on whether the process is executing a user procedure or a kernel procedure at that time.

Multics [Schroeder et al. 1977] used a more generalized form of the procedure-oriented domains. Unlike UNIX, which uses two domains (user mode and kernel mode), the Multics architecture had the flexibility to support up to 64 domains. Each domain of Multics was called a *ring*. As shown in Figure 11.14, the rings were concentric and the procedures in the innermost ring, the operating system kernel, were most powerful, having maximum access rights. Moving outward from the innermost ring, the rings became successively less powerful having fewer access rights. In fact, the access privileges of ring i were a subset of those for ring j for all $i > j$ when the ring numbers started from the innermost ring and increased for each ring as we moved outward. That is, $j < i$ implied that the procedures in ring j had more access rights than the procedures in ring i. A process could operate in multiple domains (rings) during its lifetime. A domain switching occurred when a procedure in one domain (ring) made a call to a procedure in another domain (ring). Obviously, domain switching was done in a controlled manner; otherwise, a process could start executing in the innermost ring and no protection would be provided.

11.5.2 Access Matrix

In the domain-based protection approach, the system must keep track of which rights on which objects belong to a particular domain. In the access matrix model, this information is represented as a matrix, called an *access matrix*, that has the following form:

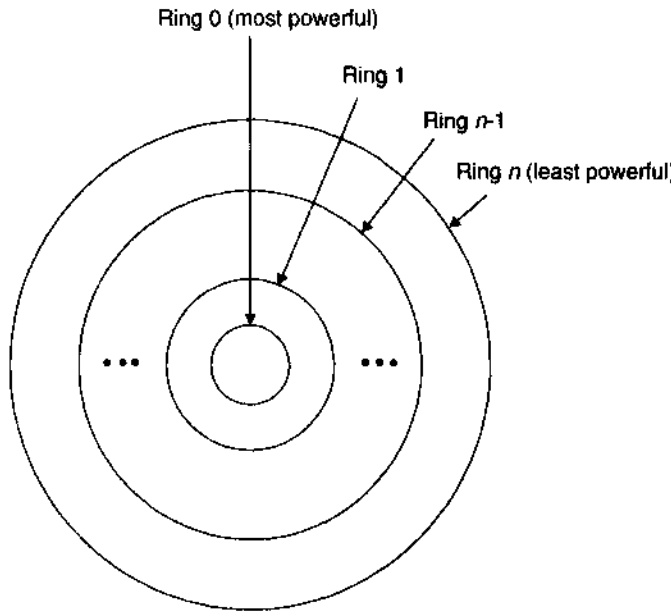


Fig. 11.14 The ring architecture of Multics protection domains.

1. The rows represent domains.
2. The columns represent objects.
3. Each entry in the access matrix consists of a set of access rights.
4. The (i, j)th entry of the access matrix defines the set of operations that a process, executing in domain D_i, can perform on object O_j.

At any instance of time, the *protection state* of the system is defined by the contents of the access matrix. The access matrix of Figure 11.15 shows the protection state of the system in Figure 11.13.

When an access matrix is used to represent the protection state of a system, the following issues must be resolved:

1. How to decide the contents of the access matrix entries.
2. How to validate access to objects by subjects.
3. How to allow subjects to switch domains in a controlled manner.
4. How to allow changes to the protection state of the system in a controlled manner.

The normally used methods to handle these issues are described below.

Object / Domain	F_1 (File-1)	F_2 (File-2)	F_3 (File-3)	S_1 (Semaphore-1)	T_1 (Tape drive-1)
D_1	Read Write Execute	Read		Up Down	
D_2	Read Write	Read Write Execute		Up Down	Read
D_3			Read Write Execute		Read Write Rewind

Fig. 11.15 The access matrix for the protection state of the system in Figure 11.13.

Deciding the Contents of the Access Matrix Entries

Policy decisions concerning which rights should be included in the (i, j)th entry are system dependent. However, in general, the contents of the access matrix entries corresponding to user-defined objects are decided by the users, whereas the contents of the entries corresponding to system-defined objects are decided by the system. For example, when a user creates a new object O_j, column j is added to the access matrix with suitable entries as decided by the user's specification of access control for the object.

Validating Access to Objects by Subjects

Given the access matrix, how can access to objects by subjects be allowed only in the manner permitted by the protection state of the matrix? For this, an *object monitor* is associated with each type of object, and every attempted access by a subject to an object is validated in the following manner:

1. A subject S in domain D initiates r access to object O, where r belongs to the set of operations that may be performed on O.
2. The protection system forms the triple (D, r, O) and passes it to the object monitor of O.
3. The object monitor of O looks for the operation r in the (D, O)th entry of the access matrix. If present, the access is permitted; otherwise, a protection violation occurs.

Allowing Controlled Domain Switching

The principle of least privilege requires that a process should be allowed to switch from one domain to another during its lifetime so that at any instance of time the process is given only as many rights as are necessary for performing its task at that time. However, domain switching by processes must be done in a controlled manner; otherwise, a process may switch to a powerful domain and violate the protection policies of the system.

Domain switching can be controlled by treating domains as objects on which the only possible operation is *switch*. Therefore, for allowing domain switching in a controlled manner, the domains are also included among the objects of the access matrix.

Figure 11.16 shows the access matrix of Figure 11.15 with the three domains as objects themselves. In the modified access matrix, domain switching from domain D_i to domain D_j is permitted if and only if the right *switch* is present in the (D_i, D_j)th entry of the access matrix. Thus in Figure 11.16, a process executing in domain D_1 can switch to domain D_2 or to domain D_3, a process executing in domain D_2 can switch only to domain D_3, and a process executing in domain D_3 cannot switch to any other domain.

Object / Domain	F_1	F_2	F_3	S_1	T_1	D_1	D_2	D_3
D_1	Read Write Execute	Read		Up Down			Switch	Switch
D_2	Read Write	Read Write Execute		Up Down	Read			Switch
D_3			Read Write Execute		Read Write Rewind			

Fig. 11.16 The access matrix of Figure 11.15 with the domains included as objects.

Allowing Controlled Change to the Protection State

In a flexible design, the protection system should also allow the content of a domain to be changed. However, this facility is not essential because if the content of a domain cannot be changed, the same effect can be provided by creating a new domain with the changed contents and switching to that new domain when we want to change the domain contents. Although not essential, the facility of allowing controlled change to the protection state is normally provided in a protection system for greater flexibility.

In an access matrix model, this facility can be provided by treating the access matrix itself as an object to be protected. In fact, since each entry in the access matrix may be individually modified, each entry must be considered as an object to be protected. For allowing controlled change, the possible rights defined for this new object are *copy*, *owner*, and *control*. To simplify the description, we will categorize the changes to be allowed to the content of the access matrix entries into two types—allowing changes to the column entries and allowing changes to the row entries.

Allowing Changes to the Column Entries. The *copy* and *owner* rights allow a process to change the entries in a column. The ability to copy an access right from one domain (row) to another is denoted by appending an asterisk (*) to the access right. For example, Figure 11.17 shows the access rights of only the first three objects (F_1, F_2, and F_3) of the access matrix of Figure 11.15 with some copy rights. A process executing in domain D_1 can copy the *Read* and *Write* operations on F_1 to any other domain (any other entry in column F_1), whereas a process executing in domain D_2 can copy the *Read* operation on F_1 and *Read* and *Execute* operations on F_2 to any other domain. None of the operations on F_3 can be copied to any other domain.

Object / Domain	F_1	F_2	F_3
D_1	Read* Write* Execute	Read	
D_2	Read* Write	Read* Write Execute*	
D_3			Read Write Execute

Fig. 11.17 An access matrix with copy rights.

The copy right may have the following three variants:

1. *Transfer.* In this case, when a right is copied from the (i, j)th entry to the (k, j)th entry of the access matrix, it is removed from the (i, j)th entry.
2. *Copy with propagation not allowed.* In this case, when the right R^* is copied from the (i, j)th entry to the (k, j)th entry, only the right R (not R^*) is created in the (k, j)th entry. The implication of this is that a process executing in domain D_k cannot further copy the right R.

3. *Copy with propagation allowed.* In this case, when the right R^* is copied from the (i, j)th entry to the (k, j)th entry, the right R^* is created in the (k, j)th entry, so that a process executing in domain D_k can further copy the right R^* or R.

On the other hand, the *owner* right is used to allow the adding/deleting of rights to column entries in a controlled manner. If the *owner* right is included in the (i, j)th entry of the access matrix, a process executing in domain D_i can add and delete any right in any entry in column j. Figure 11.18 shows the access matrix of Figure 11.17 with *owner* rights included for the three objects. In the figure, domains D_1, D_2, and D_3 have the *owner* rights for objects F_1, F_2, and F_3, respectively. Therefore, a process executing in domain D_1 can add and delete any valid right of F_1 in any entry in column F_1. Similarly, a process executing in domain D_2 can add and delete any valid right of F_2 in any entry in column F_2, and a process executing in domain D_3 can add and delete any valid right of F_3 in any entry in column F_3.

Domain \ Object	F_1	F_2	F_3
D_1	Read* Write* Execute Owner	Read	
D_2	Read* Write	Read* Write Execute* Owner	
D_3			Read Write Execute Owner

Fig. 11.18 An access matrix with owner rights.

Allowing Changes to the Row Entries. The *control* right, which is only applicable to domain objects, is used to allow a process to change the entries in a row. If the control right is present in the (D_i, D_j)th entry of the access matrix, a process executing in domain D_i can remove any access right from row D_j. For example, Figure 11.19 shows the access matrix of Figure 11.16 with *control* rights included. In this figure, since the entries (D_1, D_2) and (D_1, D_3) have *control* rights, a process executing in domain D_1 can delete any right from rows D_2 and D_3. Similarly, since the entry (D_2, D_3) has *control* right, a process executing in domain D_2 can delete any right from row D_3.

Object / Domain	F_1	F_2	F_3	S_1	T_1	D_1	D_2	D_3
D_1	Read Write Execute	Read		Up Down			Switch Control	Switch Control
D_2	Read Write	Read Write Execute		Up Down	Read			Switch Control
D_3			Read Write Execute		Read Write Rewind			

Fig. 11.19 An access matrix with control rights.

11.5.3 Implementation of Access Matrix

In practice, an access matrix is large and sparse. Most domains have no access at all to most objects, that is, most of the entries are empty. Therefore, a direct implementation of an access matrix as a two-dimensional matrix would be very inefficient and expensive (wastage of disk space). Moreover, in distributed systems, subjects and objects may be located on different sites, which further complicates the implementation issue. The two most commonly used methods that have gained popularity in contemporary distributed systems for implementing an access matrix are access control lists (ACLs) and capabilities. For instance, Andrew [Satyanarayanan 1989], Apollo [Levine 1986], and Butler [Dannenberg and Hibbard 1985] use ACLs and Accent [Rashid and Robertson 1981], Amoeba [Mullender and Tanenbaum 1986], and Mach [Sansom et al. 1986] use capabilities. These two methods are described below.

Access Control Lists

In this method, the access matrix is decomposed by columns, and each column of the matrix is implemented as an access list for the object corresponding to that column. The empty entries of the matrix are not stored in the access list. Therefore, for each object, a list of ordered pairs (*domain, rights*) is maintained, which defines all domains with a nonempty set of access rights for that object. Further details of the working and properties of a security system based on ACLs are presented below.

Access Validation. Whenever a subject in domain D executes an operation r on an object O, the access list for object O is first searched for the list element whose domain field is D. Then the rights field of this element is searched for r. If found, the operation is allowed to continue; otherwise, a protection violation occurs.

In this method, the access list is checked on every access. This is a very desirable feature from a security point of view. However, consulting the access list on every access

could cause substantial overhead, especially when the access list is long. This drawback can be overcome by maintaining a cache for the access list entries of only the active domains.

Granting Rights. Access right r for object O is granted to domain D in the following manner:

1. The access list for object O is first searched for the list element whose domain field is D.
2. If found, right r is added to the rights field of this list element. Otherwise, a new list element is added to the access list for the object O. The domain field and rights field of this list element are set to D and r, respectively.

Passing Rights. Access right r for object O is passed (propagated) from a domain D_1 to another domain D_2 in the following manner:

1. Access list for object O is first checked to ensure that D_1 possesses either *owner* right for object O or *copy* right for access right r.
2. If D_1 possesses any of the above two rights, access right r for object O is granted to domain D_2 in the manner described above. Otherwise, a protection violation occurs.

Rights Revocation. Access right r for object O is revoked from domain D simply by deleting r from the rights set of domain D in the access list for O.

For file protection, several systems use user-oriented domains like that of UNIX in which a (*uid*, *gid*) pair forms a domain. Revocation of a user's access right becomes more complicated in these systems because access revocation requires the deletion of the user's *uid* (if present) from the access list of the object in question and also the cancellation of the user's membership from all the groups that belong to a domain that has access to that object. In a large distributed system, the process of discovering all groups that the user should be removed from and performing the actual removal operation may take a significant amount of time that may be unacceptable in emergencies. The concept of *negative rights* is used to overcome this problem [Satyanarayanan 1989, 1990]. This concept is based on the idea that to revoke a user's access right to an object, the user can be given negative rights on that object. Negative rights indicate denial of the specified rights, with denial overriding possession in case of conflict. With this extension, the ACLs may contain negative rights, so that it is possible to express facts, such as every user of a domain except *user-i* and *user-j* may exercise right r on object O. The union of all the negative rights specified for a user subtracted from his or her positive rights gives his or her actual total rights. Negative rights thus act as a mechanism for rapid and selective revocation and are particularly valuable in a large distributed system.

The main advantage of the method of ACLs is that for a given object the set of domains from which it can be accessed can be determined efficiently. However, the main drawback of the method is that for a given domain the set of access rights cannot be determined efficiently.

Capabilities

Rather than decomposing the access matrix by columns, in this method the access matrix is decomposed by rows, and each row is associated with its domain. Obviously, the empty entries are discarded. Therefore, for each domain, a list of ordered pairs (*object, rights*) is maintained, which defines all objects for which the domain possesses some access rights. Each (*object, rights*) pair is called a *capability* and the list associated with a domain is called a *capability list.*

A capability is used for the following two purposes:

- To uniquely identify an object
- To allow its holder to access the object it identifies in one or more permission modes

Therefore, as shown in Figure 11.20, a capability is composed of two basic parts—an object identifier part and a rights information part. The object identifier part usually contains a pointer to the object and acts as a globally unique system-oriented name for the object. On the other hand, the rights information part is usually a set of bits that determine which operations are allowed on the object with this capability. Further details of the working and properties of a security system based on capabilities are presented below. For ease of presentation, in the following description we will assume that each user/process forms a domain of the system.

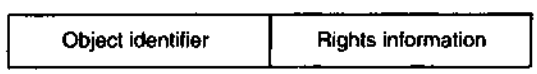


Fig. 11.20 The two basic parts of a capability.

Access Validation. A capability is considered as an unforgeable ticket that allows its holder to access the object (identified by its object identifier part) in one or more permission modes (specified by its rights information part). A process that possesses a capability can access the object identified by it in the modes permitted by it. There are usually several capabilities for the same object. Each one confers different access rights to its holders. The same capability held by different holders provides the same set of access rights to all of them.

Since simple possession of a capability means that access is permitted in the modes associated with the capability, to execute an operation r on an object O, a process executes the operation r, specifying the capability for object O as a parameter. When the monitor for object O receives the access request (with the capability), it need only verify that the rights information part of the capability has permission for operation r.

Notice that in a capability-based security system, there is no need to search a list to verify that access is allowed. Rather, the security system need only verify that the capability supplied by a process is valid for the desired operation on the object. Therefore, once a process establishes the possession of a capability for an object, it can access the object in one of the modes allowed by the capability without any further check by the

security system. For these reasons, capability-based security systems are more efficient than security systems based on ACLs. Also notice that in the capability-based approach, there is no checking of user identity. If this is required for access validation, some user authentication mechanism must be used.

Granting and Passing of Rights. In the capability-based security scheme, each user maintains a list of capabilities that identifies all objects that the user can access and the associated access permissions. But how does a user get a capability in the first place?

In a capability-based system, there are usually one or more object managers for each type of object. A request to create an object or to perform some operation on an object is sent to one of the object managers of that object type. When a new object is created, the object manager that creates the object generates (as a part of the object creation process) a capability with all rights of access for the object. The generated capability is returned to the owner for use. Now the owner may give the capability to other users with whom the object is to be shared. However, the owner may want to restrict the access modes in a different manner for different users who share the object. Therefore, before the owner gives the capability to other users, it may be necessary to restrict the capability by removing some of the rights. The usual way to do this is to have a function in the object manager for restricting capabilities. When called, this function generates a new capability for the object with only the desired access permissions and returns the newly generated capability to the caller. This capability is then given to the user(s) for whom it was generated.

Protecting Capabilities against Unauthorized Access. To ensure that in a capability-based security system the objects are protected against unauthorized access, the following basic requirements must be met:

1. A capability must uniquely identify an object in the entire system. Even after the object associated with a given capability is deleted, it is important that the capability is not reused because some users may retain obsolete capabilities. Use of an obsolete capability should produce an error instead of allowing access to a different object.

2. Capabilities must be protected from user tampering. For this, it is necessary that capabilities be treated as unforgeable protected objects that are maintained by the operating system and only indirectly accessed by the users.

3. Guessing of a valid capability should be impossible or at least very difficult. This is because in the capability-based scheme the degree of protection is probabilistic and proportional to the difficulty of guessing a valid capability.

The following methods are normally used to meet these requirements:

1. *Tagged architecture.* In this method, each object has a tag to denote its type as either a capability or an ordinary accessible data such as integer, pointer, character, or instruction. Tags are normally implemented in hardware by associating a tag with units of memory, usually words. In this case, each memory word has a tag field that tells whether

the word contains a capability or not. The tag field is not directly accessible by an application program, and it can be modified only by programs running in the kernel mode (i.e., the operating system). Although only one bit is necessary for the tag field to distinguish between capabilities and other objects, tagged architecture machines typically use n bits for the tag field to allow the software to distinguish among 2^n different types of objects (memory contents).

2. *Partitioned implementation.* Another method to preserve the integrity of capabilities is to store them separately from data in special segments that can only be accessed by the operating system. One way to implement this is to partition the address space of a user process into two parts—one accessible to the process and the other accessible only to the operating system. The former contains the process's normal data and instructions, whereas the latter contains its capabilities.

3. *Encryption of sparse capabilities.* The tagged and partitioned methods of protecting capabilities from unauthorized access are oriented toward centralized computer systems. These methods are not suitable for use in a distributed system because in a distributed system the security mechanism used should allow capabilities to be safely transferred from one node to another. The third method, which does not require capabilities to be distinguished from other objects either by separation or by tagging, is particularly suited to distributed systems. In addition to preventing capabilities from user tampering, this method also makes capabilities unique and difficult to guess.

In this method, a large name space that is sufficiently sparse is used for the capabilities. Uniqueness is achieved by using the methods described in Section 10.4 for the creation of unique system-oriented identifiers that form the object identifier part of the capabilities. On the other hand, to make the capabilities difficult to guess or forge, the rights information part of each capability is combined with an extra field containing a random number, thereby rending the task of a malicious user wishing to generate any valid capability so lengthy as to be impractical. Furthermore, the rights information part and the random-number part are encrypted by the object manager before a capability is issued to a user. The secret key is available only with the object manager. When a process presents the capability along with a request for object accessing, the object manager uses the key to decrypt the encrypted part of the capability before using it. In this way, the rights information part of a capability that confers restricted permissions cannot be forged by its possessors to convert it to one with more permissions.

Rights Amplification. The concept of rights amplification was introduced in Hydra [Cohen and Jefferson 1975]. We saw that in a capability-based system objects are typed and recognize a set of predefined operations. The set of predefined operations for an object type is known as *auxiliary rights* in Hydra. In addition to the auxiliary rights, each object type has a set of *kernel rights*, such as *get*, *put*, and *add* to manipulate the data part of an object and *load*, *store*, *append*, *delete*, *copy*, and *create* to manipulate the capability list part of an object. The kernel rights are implemented within the kernel and are transparent to the user processes.

A request to perform an operation on an object is sent to the object manager of that object type. The request contains the capability for the object as a parameter. This

capability may include an auxiliary right to invoke some operation on the object but would not include any of the kernel rights for the object. A problem arises here because the object manager is itself an ordinary program. It is essential that the object manager be able to invoke kernel operations in order to perform the requested operation successfully. The rights amplification technique of Hydra solves this problem by giving a rights template to object managers that gives them more rights to an object than the capability itself allows. Additional rights given to the object managers allow them to perform kernel operations on the objects.

When a process P invokes an operation r on an object O, the capability C supplied by P may get amplified to C_a when the object manager M starts performing the operation r on the object. This may be necessary in order to allow M to access the storage segment representing O for performing operation r on it. That is, M is allowed to perform kernel operations on O directly, even though the calling process P cannot. After completion of the operation r on O, the capability C_a for O is restored to its original, unamplified state C. This is a typical case in which the rights held by a process for access to a protected segment must change dynamically, depending on the task to be performed.

Notice that in the rights amplification scheme, the object managers are treated as "trustworthy" procedures and are allowed to perform kernel operations on the objects of a specified type, on behalf of any process that holds an auxiliary right for an object of that type. Therefore, the rights held by an object manager are independent of and normally exceed the rights held by the subjects that access the object. However, an object manager is not a universally trustworthy procedure because it is not allowed to act on other types of objects and cannot extend its rights to any other procedure.

Rights Revocation. In a security system based on ACLs, revocation of rights is easy because for a given object it is possible to easily and efficiently determine which subjects have what rights for the object. However, in a security system based on capabilities, revocation of rights is a much more difficult problem because for a given object it is difficult to determine which subjects have what rights for the object. This is because the capabilities for an object may be stored in several capability lists that are distributed throughout the system, and we must first find them before we can revoke them. Some commonly used methods for implementing revocation for capabilities are described below.

1. *Back pointers.* One way is to keep track of all the capabilities for an object and to change/delete them selectively depending on the desired revocation of access rights. A simple method to keep track of all the capabilities for an object is to maintain a list of pointers with the object, pointing to all capabilities associated with the object. This method has been used in the Multics system. The method is quite general but very costly to implement.

2. *Indirection.* Another approach is to use indirect addressing. In this method each capability points to an indirect object (such as a table entry) rather than to the object itself. The indirect object in turn points to the real object. Revocation is implemented by deleting the indirect object to break the connection between the real object and the capabilities for it. When an access is attempted with a capability whose indirect object has been deleted,

access operation fails because the real object is unknown due to the broken connection. A drawback of this method is that it does not allow selective revocation.

3. *Use of keys.* In this method, in addition to the object identifier and rights information fields, each capability has a field that contains a unique bit pattern. The contents of this field is called a "key." A capability's key is defined when the capability is created and it cannot be modified or inspected by a process owning that capability. Each object has a *master key* associated with it that can be dynamically defined or changed with a special *set_key* operation. Normally, only the owner of an object is given the right to invoke the *set_key* operation for changing the master key of the object.

When a new capability for an object is created, the key field of the capability is set to the current master key for the object. When an access is attempted with a capability, the capability's key is compared to the master key of the corresponding object. If the two keys match, access to the object is allowed; otherwise, a protection violation occurs. Revocation involves replacement of the master key with a new value by using the *set_key* operation, invalidating all previous capabilities for this object.

This revocation scheme is used in Amoeba, in which random numbers are used as keys. A drawback of this scheme is that it does not allow selective revocation, since only one master key is associated with each object. However, this drawback can be overcome by associating a list of keys with each object.

Hybrid Approach

As compared to the capability-based scheme, the scheme based on ACLs is more suited to the implementation of security systems because ACLs correspond directly to the needs of the users. When users create objects, they can specify which domains can access the objects as well as the operations allowed. However, we saw that a security system based on ACLs is normally less efficient than capability-based security systems because of the need to search the access list on every access. To overcome the drawbacks of the two schemes and to combine their advantages, most systems use a hybrid approach for designing their security system.

In the hybrid approach, both ACLs and capabilities are employed along with the concept of a session. A *session* is a logical concept of a period during which a process accesses an object. When a process first tries to start a session for accessing an object, it specifies the access modes (types of operations) that it may perform on the object during the session. The ACL of the object is searched for the types of operations desired. If access is denied, a protection violation occurs. Otherwise, the system creates a new capability for the object and attaches it to the process. After this, all accesses to the object by this process during the session are made using the capability so that an access control check can be performed efficiently. After the session is over, the capability is destroyed.

The UNIX file system employs this scheme, with a session being the period between the *open* and *close* operations of a file by a process. Each file has an associated ACL. When a process opens a file, its ACL is checked for the mode specified in the *open* command. If access in the specified mode is permitted, a new entry is allocated in a file

table, the access mode is recorded in this entry, and an index to this entry is returned to the process. All subsequent operations on the file are made by this process by specifying the index into the file table. The entry in the file table points to the file. When the process executes the *close* operation on the file, the session is closed by deleting the file table entry. A new session must be started after this if the process wants to access the same file at some later time.

The file table is maintained by the operating system, so that it cannot be corrupted by the user. Security is ensured because access is validated at the time a session is started and a process can access only those files for which a session has been started but not yet closed.

11.6 DIGITAL SIGNATURES

Message integrity, which guarantees that the contents of a message were not changed when it was in transfer, is also an important security requirement in a distributed system. The concept of digital signature, which is based upon asymmetric cryptosystems, is the most commonly used method to handle this issue.

Recall that in an asymmetric cryptosystem the secret key of a user is known only to that user and no one else. Therefore, the sender of a message can use its secret key for signing the message by encrypting it with the key. That is, the sender can uniquely "seal" the message with his or her own signature (secret key). The sealed message can be sent to anyone with the corresponding public key. Using digital signatures assures the receiver of the message not only that the message content has not been manipulated but also that the message was indeed sent by the claimed sender. Thus, digital signatures are applicable to both user authentication and message integrity.

A *digital signature* is basically a code, or a large number, that is unique for each message and to each message originator. It is obtained by first processing the message with a hash function (called a *digest function*) to obtain a small digest dependent on each bit of information in the message and then encrypting the digest by using the originator's secret key. To avoid duplicity problems, a digest function (D) must have the property that $D(M)$ is different from $D(M')$ for all possible pairs of M and M'. Rivest [1992] proposed a message digest function (known as *MD5*) for use in secure mail and other applications on the Internet.

To illustrate how the digest of a message can be obtained from the message, the example given in [Adam 1992] is presented here. In this example, it is assumed that a message is a digital string of 0's and 1's and is divided into blocks of 64 bits. The bitwise Exclusive-OR of the first two blocks is performed to obtain a new block of 64 bits. The newly obtained block is again Exclusive-ORed with the third block of the message, again resulting in a new block of 64 bits. This process is continued one by one with all the other blocks of the message. The end result is a 64-bit digest that depends on each bit of data in the whole message stream. In other words, to alter the message, even by 1 bit, would alter the 64-bit digest. Moreover, it should be essentially impossible to forge a message that would result in the same digest.

A protocol based on a digital signature for ensuring message integrity works as follows:

1. A sender (A) computes the digest (D) of a message (M). It then encrypts the digest D by using its secret key (S_a) to obtain a ciphertext $C_1 = E(D, S_a)$. A signed message is then created that consists of the sender's identifier, the message M in its plaintext form, and the ciphertext C_1. The signed message, which has the form (ID_a, C_1, M), is then sent to a receiver.

2. On receiving the signed message, the receiver decrypts C_1 by using the public key of the sender to recover the digest D. It then calculates a digest for M (by using the same digest function) and compares the calculated digest with the digest recovered by decrypting C_1. If the two are equal, message M is considered to be correct; otherwise it is considered incorrect.

Notice that the protocol does not require a message to be hidden from unauthorized users. Rather, it allows a message to be read openly by anyone who receives or intercepts it. But a forged message is successfully detected by the protocol.

An application may require that the first receiver retransmit the signed message to another receiver, which may have to subsequently retransmit it to other receivers. In such a situation, it is important that each of the recipients should be able to verify that the signed message indeed originated from the claimed originator and that its contents were not changed by any of the intermediate recipients or by an intruder. A digitally signed message meets these requirements because it has the originator's identifier included in it and the digest of the message can only be decrypted by using the originator's public key.

In the actual implementation, a key distribution server may be used that maintains a database of the public keys of all users. If the receiver of a digitally signed message does not already have the public key of the message originator, it can request it from the key distribution server. This avoids the need to send a new user's public key to all other user's in the system. The new user's public key can be simply registered with the key distribution server.

Privacy Enhanced Mail (*PEM*) scheme, designed for adding privacy to Internet mail applications, is a good example of use of cryptography and digital signature techniques. PEM offers confidentiality, authentication, and message integrity. These features are intended to provide sufficient trust so that the general Internet user population will feel comfortable using the Internet for business correspondence and sending messages that contain sensitive information. PEM is completely implemented at the application level by end systems so that it can be incorporated on a site-by-site or user-by-user basis. This approach imposes no special requirements on message transfer systems at intermediate relay sites or endpoints. That is, network routers and mail relays treat PEM messages as an ordinary piece of mail. How PEM provides privacy to electronic mails is briefly described below. Readers interested in its detail description may refer to [Linn 1993, Kent 1993b, Balenson 1993, Kaliski 1993, Kent 1993a].

PEM assumes that the network is not trusted but that each user of PEM trusts his or her own local computer. Mail users obtain a public/secret key pair from a local PEM program and publish their public keys with their mail addresses. The PEM program maintains a database of the secret keys of its local users and the public keys of remote users. Currently, the Rivest-Shamir-Adleman (RSA) algorithm is used to generate the public/secret key pairs for users. PEM provides the following types of facilities:

1. *Confidentiality.* Sending a message in encrypted form so that sensitive information within it cannot be read by an intruder.
2. *Message integrity.* Sending a signed message so that the receiver can be ensured that the contents of the message were not changed.

Both facilities also possess an authentication feature because the encryption and decryption of the message or the digital signature can only be done by a user having the proper key.

Let us first see how PEM sends a secret message (M) for ensuring confidentiality:

1. The PEM program of the sender's computer first generates a random secret key (K) and encrypts the message (M) by using this key to obtain a ciphertext $C_1 = E(M, K)$. Currently, the DES algorithm is used for this purpose, but others may as well be used in the future. The secret key (K) is then encrypted by using the recipient's public key (say, P_r) to obtain a ciphertext $C_2 = E(K, P_r)$. Now C_1 and C_2 are sent to the recipient in a message m_1.

2. On receiving m_1, the PEM program of the recipient's computer fetches the recipient's secret key (S_r) from its database and decrypts C_2 by using S_r to obtain K. Now by using K, it decrypts C_1 to obtain the original message M, which it then stores in the recipient's mailbox.

Notice that the PEM scheme retains the efficiency of symmetric cryptography for the bulk encryption but avoids the need for a secure key distribution server.

Now let us see how PEM sends a signed message (M) for ensuring message integrity:

1. The PEM program of the sender's computer computes the digest (D) of the message (M) by using a message digest function. The digest (D) is then encrypted by using the sender's secret key (S_s) to obtain a ciphertext $C_1 = (D, S_s)$. The sender's ID, C_1, and M are then sent to the recipient in a message m_1.

2. On receiving m_1, the PEM program of the recipient's computer fetches the sender's public key (P_s) from its database and uses it to decrypt C_1 to obtain the digest D. It then applies the same message digest function to M and compares the result with D. If the two are equal, message M is considered to be correct; otherwise it is considered incorrect. The message M is then stored in the recipient's mailbox with a proper note from the PEM program's side about the result of its integrity check.

11.7 DESIGN PRINCIPLES

Based on their experience with Multics, Saltzer, and Schroeder [1975] identified some design principles that can be used as a guide to designing secure systems. Although these design principles were proposed for centralized systems, they hold good for distributed systems as well [Kent 1981]. These and some other design principles are summarized below. Designers of security components of a distributed operating system should use them as basic guidelines.

1. *Least privilege.* The principle of least privilege (also known as the need-to-know principle) states that any process should be given only those access rights that enable it to access, at any time, what it needs to accomplish its function and nothing more and nothing less. That is, the security system must be flexible enough to allow the access rights of a process to grow and shrink with its changing access requirements. This principle serves to limit the damage when a system's security is broken. For example, if an editor is given the right to access only the file that has to be edited, even if the editor has a Trojan horse, it will not be able to access other files of the user and hence cannot do much damage.

2. *Fail-safe defaults.* Access rights should be acquired by explicit permission only and the default should be no access. This principle requires that access control decisions should be based on why an object should be accessible to a process rather than on why it should not be accessible.

3. *Open design.* This principle requires that the design of the security mechanisms should not be secret but should be public. It is a mistake on the part of a designer to assume that the intruders will not know how the security mechanism of the system works.

4. *Built in to the system.* This principle requires that security be designed into the systems at their inception and be built in to the lowest layers of the systems. That is, security should not be treated as an add-on feature because security problems cannot be resolved very effectively by patching the penetration holes detected in an existing system.

5. *Check for current authority.* This principle requires that every access to every object must be checked using an access control database for authority. This is necessary to have immediate effect of revocation of previously given access rights. For instance, in some file systems, a check for access permission is made only when a file is opened and subsequent accesses to the file are allowed without any check. In these systems, a user can keep a file open for several days and continue to have access to its contents, even if the owner of the file changes the access permission and revokes the user's right to access its contents.

6. *Easy granting and revocation of access rights.* For greater flexibility, a security system must allow access rights for an object to be granted or revoked dynamically. It should be possible to restrict some of the rights and to grant to a user only those rights that are sufficient to accomplish its functions. On the other hand, a good security

system should allow immediate revocation with the flexibility of selective and partial revocation. With *selective revocation* facility, it is possible to revoke access rights to an object only from a selected group of users rather than from all users who posses access rights for the object. And with *partial revocation* facility, only a subset of the rights granted to a user for an object can be revoked instead of always revoking all its rights for the object.

7. *Never trust other parties.* For producing a secure distributed system, the system components must be designed with the assumption that other parties (people or programs) are not trustworthy until they are demonstrated to be trustworthy. For example, clients and servers must always be designed to view each other with mutual suspicion.

8. *Always ensure freshness of messages.* To avoid security violations through the replay of messages, the security of a distributed system must be designed to always ensure freshness of messages exchanged between two communicating entities.

9. *Build firewalls.* To limit the damage in case a system's security is compromised, the system must have firewalls built into it. One way to meet this requirement is to allow only short-lived passwords and keys in the system. For example, a shared secret key used to build a logical communication channel between a client and a server should be fairly short-lived, perhaps being changed with every communication session between them.

10. *Efficient.* The security mechanisms used must execute efficiently and be simple to implement.

11. *Convenient to use.* To be psychologically acceptable, the security mechanisms must be convenient to use. Otherwise, they are likely to be bypassed or incorrectly used by the users.

12. *Cost effective.* It is often the case that security needs to be traded off with other goals of the system, such as performance or ease of use. Therefore, in designing the security of a system, it is important to come up with the right set of trade-offs that take into account the likelihood that the system will be compromised with the cost of providing the security, both in terms of money and personnel experience.

11.8 CASE STUDY: DCE SECURITY SERVICE

As a case study of how the various security concepts described in this chapter can be integrated to provide security in a single system, the DCE Security Service is briefly described below.

In DCE, a user or a process (client or server) that needs to communicate securely is called a *principal*. For convenience of access control, principals are assigned membership in one or more *groups* and *organizations*. All principals of the same group or organization have the same access rights. Groups generally correspond to work groups or departments, and organizations typically include multiple groups having some common properties. Typically, a principal is a member of one organization but may simultaneously be a member of multiple groups. Each principal has a unique identifier associated with it.

Together, a principal's identifier, group, and organization membership are known as the principal's *privilege attributes.*

The main components of the DCE Security Service for a single cell are shown in Figure 11.21. These components collectively provide authentication, authorization, message integrity, and security administration services. Let us consider these services one by one.

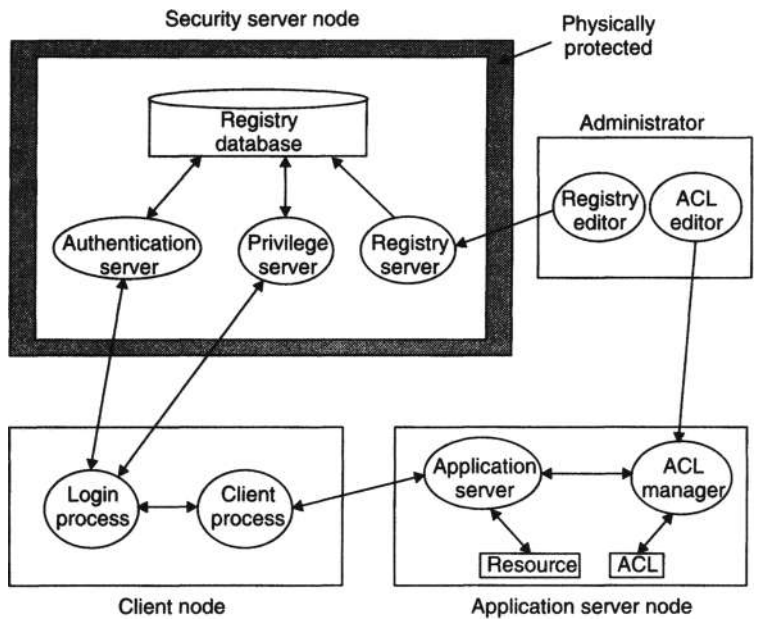


Fig. 11.21 Main components of DCE Security Service for a single cell.

11.8.1 Authentication in DCE

The DCE authentication service uses the Kerberos system described in Section 11.4.5. The authentication server, ticket-granting server, and authentication database of Kerberos are respectively called authentication server, privilege server, and registry database in DCE. The information registered in the registry database includes each principal's secret key and privilege attributes. The protocols for authenticating a user at the time of login and for mutual authentication of a client and a server are the same as that of Kerberos (the intercell client-server authentication protocol in DCE is the same as the interrealm authentication protocol of Kerberos). The only difference is that the service-granting ticket in DCE also contains the group and organization membership information of a client. This information is used by the application server to verify the access rights of the client before providing the requested service.

The establishment of a secure logical communication channel between a client and a server by using the authentication protocol is known as *authenticated RPC* in DCE. This is because in DCE clients and servers communicate by using RPCs. Once authenticated RPC has been established, it is up to the client and the server to determine how much security is desired. That is, subsequent RPC messages may or may not be encrypted depending on the security needs of the application.

11.8.2 Authorization in DCE

Authorization in DCE is based on ACLs. Associated with each application server is an ACL and an ACL manager. The ACL contains complete information about which principals have what rights for the resources managed by the server. When a client's request comes to the server, it extracts the client's ID and its group and organization membership information from the received encrypted ticket. It then passes the client's ID, membership, and the operation desired to the ACL manager. Using this information, the ACL manager checks the ACL to make a decision if the client is authorized to perform the requested operation. It returns an access granted or denied reply to the server, after which the server acts accordingly.

Note that in DCE groups are effective only within cells. Therefore, a principal belonging to a different cell can be granted access based solely on its unique identifier, not on group membership. That is, if access is to be granted to principals of remote cells, their unique identifiers have to be entered in the ACL along with the access rights.

11.8.3 Message Integrity in DCE

As already mentioned above, once authenticated RPC has been established, it is up to the client and server to determine how much security is desired. Therefore, if message integrity is desired, it can be ensured by the use of a digital signature technique. That is, applications can ensure data integrity by including an encrypted digest of the message data passed between clients and servers. The digest must be encrypted and decrypted by using the session key that a client and a server share for secure communication between them.

11.8.4 Security Administration in DCE

The administrator, registry server, and ACL manager jointly perform security administration tasks. Two programs are used by the administrator for performing administration tasks. One is the registry editor program and the other is the ACL editor program.

The registry editor program may be used by the system administrator to view, add, delete, and modify information in the registry database. Even system administrators do not have direct access to the registry database, and they access the registry database only by making requests to the registry server. This is much safer, for although an administrator can change any password, he or she cannot obtain the password of any user.

On the other hand, the ACL editor program may be used by an application administrator to view, add, delete, and modify entries in ACLs for applications or ACLs

for objects (resources) controlled by them. Once again, all requests for updates to ACLs are sent to the ACL manager and not performed directly on ACLs.

In DCE, system administrators use organization membership to apply global security policies, such as deciding the lifetime of tickets and passwords of different principals. For example, the lifetime of tickets and passwords is kept smaller for principals of an organization that handles highly sensitive information as compared to the lifetime of tickets and passwords of principals of an organization that does not handle sensitive information.

11.9 SUMMARY

Computer security deals with protecting the various resources and information of a computer system against destruction and unauthorized access. The main goals of computer security are secrecy, privacy, authenticity, and integrity.

A total approach to computer security involves both external and internal security. The three main aspects of internal security in distributed systems are authentication, access control, and communication security.

An intruder is a person or program that tries to obtain unauthorized access to data or a resource of a computer system. An intruder may be a threat to computer security in many ways that are broadly classified into two categories—passive attacks and active attacks. In passive attacks, an intruder somehow tries to steal unauthorized information from the computer system without interfering with the normal functioning of the system. Some commonly used methods of passive attack are browsing, leaking, inferencing, and masquerading. On the other hand, active attacks interfere with the normal functioning of the system and often have damaging effects. Some commonly used forms of active attacks are viruses, worms, and logic bombs. Active attacks associated with message communications are integrity attack, authenticity attack, denial attack, delay attack, and replay attack.

Three kinds of channels that can be used by a program to leak information are legitimate channels, storage channels, and convert channels. The confinement problem deals with the problem of eliminating every means by which an authorized subject can release any information contained in the object to which it has access to some subjects that are not authorized to access that information. The confinement problem is in general unsolvable.

Cryptography is a means of protecting private information against unauthorized access in those situations where it is difficult to provide physical security. There are two broad classes of cryptosystems—symmetric and asymmetric. When cryptography is employed for secure communications in distributed systems, a need for key distribution arises. The mechanisms and protocols for key distribution in symmetric and asymmetric cryptosystems have been described in the chapter.

An authentication mechanism prohibits the use of the system (or some resource of the system) by unauthorized users by verifying the identity of a user making a request. The main types of authentication normally needed in a distributed system are user login authentication, one-way authentication of communicating entities, and two-way authenti-

cation of communicating entities. The three basic approaches to authentication are proof by knowledge, proof by possession, and proof by property. The proof-by-knowledge method based on passwords is the most widely used method for user login authentication. For one-way and two-way authentication of communicating entities, the protocols based on cryptosystems have been described in the chapter. The Kerberos authentication system has also been described as a case study.

Access control deals with the ways that are used in a computer system to prohibit a user (or a process) from accessing those resources/information that he or she is not authorized to access. The three access control models proposed in the literature are the access matrix model, the information flow control model, and the security kernel model. Of these, the access matrix model is the most popular one and is widely used in existing centralized and distributed systems.

In the access matrix model, the access rights of each subject to each object are defined as entries in a matrix, called the access matrix. The two most widely used methods that have gained popularity in contemporary distributed systems for implementing an access matrix are ACLs and capabilities.

The concept of digital signatures, which is based upon asymmetric cryptosystems, is the most commonly used method to handle the issue of message integrity in distributed systems.

Some design principles that can be used as a guide to designing secure systems are least privilege, fail-safe defaults, open design, security built in to the system, checking for current authority, easy to grant and revoke access rights, not to trust other parties, always ensuring freshness of messages, building of firewalls, cost effective, efficient, convenient to use, and right set of trade-offs.

EXERCISES

11.1. List some of the common goals of computer security.

11.2. What are the additional security problems that a distributed operating system designer must deal with as compared to the designer of an operating system for a centralized time-sharing system? Can we ensure the same degree of security in a distributed system as we have in a centralized time-sharing system? Give reasons for your answer.

11.3. What is the "need-to-know" principle in computer security? Think of some security problems that may occur if this principle is not taken care of in the design of the security component of a computer system.

11.4. Differentiate between passive and active attacks. Which of the two is more harmful and why?

11.5. What are some of the commonly used methods for passive attack? Comment on the relative complexity of each of these methods from the point of view of the following:
(a) An intruder
(b) The designer of a security system

11.6. What is a Trojan horse program? Give an example (in pseudocode) of both a passive type and an active type Trojan horse program.