8.1 INTRODUCTION TO SECURITY

We start our description of security in distributed systems by taking a look at some general security issues. First, it is necessary to define what a secure system is. We distinguish security *policies* from security *mechanisms*, and take a look at the Globus wide-area system for which a security policy has been explicitly formulated. Our second concern is to consider some general design issues for secure systems. Finally, we briefly discuss some cryptographic algorithms, which play a key role in the design of security protocols.

8.1.1 Security Threats, Policies, and Mechanisms

Security in computer systems is strongly related to the notion of dependability. Informally, a dependable computer system is one that we justifiably trust to deliver its services (Laprie, 1995). As mentioned in Chap. 7, dependability includes availability, reliability, safety, and maintainability. However, if we are to put our trust in a computer system, then confidentiality and integrity should also be taken into account. **Confidentiality** refers to the property of a computer system whereby its information is disclosed only to authorized parties. **Integrity** is the characteristic that alterations to a system's assets can be made only in an authorized way. In other words, improper alterations in a secure computer system should be detectable and recoverable. Major assets of any computer system are its hardware, software, and data.

Another way of looking at security in computer systems is that we attempt to protect the services and data it offers against **security threats**. There are four types of security threats to consider (Pfleeger, 1997):

1. Interception
2. Interruption
3. Modification
4. Fabrication

Interception refers to the situation that an unauthorized party has gained access to a service or data. A typical example of interception is where communication between two parties has been overheard by someone else. Interception also happens when data are illegally copied, for example, after breaking into a person's private directory in a file system.

An example of interruption is when a file is corrupted or lost. In general, interruption refers to the situation in which services or data become unavailable, unusable, destroyed, and so on. In this sense, denial of service attacks by which someone maliciously attempts to make a service inaccessible to other parties is a

security threat that classifies as interruption.

Modifications involve unauthorized changing of data or tampering with a service so that it no longer adheres to its original specifications. Examples of modifications include intercepting and subsequently changing transmitted data, tampering with database entries, and changing a program so that it secretly logs the activities of its user.

Fabrication refers to the situation in which additional data or activity are generated that would normally not exist. For example, an intruder may attempt to add an entry into a password file or database. Likewise, it is sometimes possible to break into a system by replaying previously sent messages. We shall come across such examples later in this chapter.

Note that interruption, modification, and fabrication can each be seen as a form of data falsification.

Simply stating that a system should be able to protect itself against all possible security threats is not the way to actually build a secure system. What is first needed is a description of security requirements, that is, a security policy. A **security policy** describes precisely which actions the entities in a system are allowed to take and which ones are prohibited. Entities include users, services, data, machines, and so on. Once a security policy has been laid down, it becomes possible to concentrate on the **security mechanisms** by which a policy can be enforced. Important security mechanisms are:

1. Encryption
2. Authentication
3. Authorization
4. Auditing

Encryption is fundamental to computer security. Encryption transforms data into something an attacker cannot understand. In other words, encryption provides a means to implement confidentiality. In addition, encryption allows us to check whether data have been modified. It thus also provides support for integrity checks.

Authentication is used to verify the claimed identity of a user, client, server, and so on. In the case of clients, the basic premise is that before a service will do work for a client, the service must learn the client's identity. Typically, users are authenticated by means of passwords, but there are many other ways to authenticate clients.

After a client has been authenticated, it is necessary to check whether that client is authorized to perform the action requested. Access to records in a medical database is a typical example. Depending on who accesses the database, permission may be granted to read records, to modify certain fields in a record, or to

add or remove a record.

Auditing tools are used to trace which clients accessed what, and which way. Although auditing does not really provide any protection against security threats, audit logs can be extremely useful for the analysis of a security breach, and subsequently taking measures against intruders. For this reason, attackers are generally keen not to leave any traces that could eventually lead to exposing their identity. In this sense, logging accesses makes attacking sometimes a riskier business.

Example: The Globus Security Architecture

The notion of security policy and the role that security mechanisms play in distributed systems for enforcing such policies is often best explained by taking a look at a concrete example. Consider the security policy defined for the Globus wide-area system (Chervenak et al., 2000). Globus is a system supporting large-scale distributed computations in which many hosts, files, and other resources are simultaneously used for doing a computation. Such environments are also referred to as computational grids (Foster and Kesselman, 1998). Resources in these grids are often located in different administrative domains that may be located in different parts of the world.

Because users and resources are vast in number and widely spread across different administrative domains, security is essential. To devise and properly use security mechanisms, it is necessary to understand what exactly needs to be protected, and what the assumptions are with respect to security. Simplifying matters somewhat, the security policy for Globus entails the following eight statements, which we explain below (Foster et al., 1998):

1. The environment consists of multiple administrative domains.
2. Local operations (i.e., operations that are carried out only within a single domain) are subject to a local domain security policy only.
3. Global operations (i.e., operations involving several domains) require the initiator to be known in each domain where the operation is carried out.
4. Operations between entities in different domains require mutual authentication.
5. Global authentication replaces local authentication.
6. Controlling access to resources is subject to local security only.
7. Users can delegate rights to processes.
8. A group of processes in the same domain can share credentials.

Globus assumes that the environment consists of multiple administrative

domains, where each domain has its own local security policy. It is assumed that local policies cannot be changed just because the domain participates in Globus, nor can the overall policy of Globus override local security decisions. Consequently, security in Globus will restrict itself to operations that affect multiple domains.

Related to this issue is that Globus assumes that operations that are entirely local to a domain are subject only to that domain's security policy. In other words, if an operation is initiated and carried out within a single domain, all security issues will be carried out using local security measures only. Globus will not impose additional measures.

The Globus security policy states that requests for operations can be initiated either globally or locally. The initiator, be it a user or process acting on behalf of a user, must be locally known within each domain where that operation is carried out. For example, a user may have a global name that is mapped to domain-specific local names. How exactly that mapping takes place is left to each domain.

An important policy statement is that operations between entities in different domains require mutual authentication. This means, for example, that if a user in one domain makes use of a service from another domain, then the identity of the user will have to be verified. Equally important is that the user will have to be assured that he is using a service he thinks he is using. We return to authentication, extensively, later in this chapter.

The above two policy issues are combined into the following security requirement. If the identity of a user has been verified, and that user is also known locally in a domain, then he can act as being authenticated for that local domain. This means that Globus requires that its systemwide authentication measures are sufficient to consider that a user has already been authenticated for a remote domain (where that user is known) when accessing resources in that domain. Additional authentication by that domain should not be necessary.

Once a user (or process acting on behalf of a user) has been authenticated, it is still necessary to verify the exact access rights with respect to resources. For example, a user wanting to modify a file will first have to be authenticated, after which it can be checked whether or not that user is actually permitted to modify the file. The Globus security policy states that such access control decisions are made entirely local within the domain where the accessed resource is located.

To explain the seventh statement, consider a mobile agent in Globus that carries out a task by initiating several operations in different domains, one after another. Such an agent may take a long time to complete its task. To avoid communication with the user on whose behalf the agent is acting, Globus requires that processes can be delegated a subset of the user's rights. Consequently, by authenticating an agent and subsequently checking its rights, Globus should be able to allow an agent to initiate an operation without having to contact the agent's owner.

As a final policy statement, Globus requires that groups of processes running

with a single domain and acting on behalf of the same user may share a single set of credentials. As will be explained below, credentials are needed for authentication. This statement essentially opens the road to scalable solutions for authentication by not demanding that each process carries its own unique set of credentials.

The Globus security policy allows its designers to concentrate on developing an overall solution for security. By assuming that each domain enforces its own security policy, Globus concentrates only on security threats involving multiple domains. In particular, the security policy indicates that the important design issues are the representation of a user in a remote domain, and the allocation of resources from a remote domain to a user or his representative. What Globus therefore primarily needs, are mechanisms for cross-domain authentication, and making a user known in remote domains.

For this purpose, two types of representatives are introduced. A **user proxy** is a process that is given permission to act on behalf of a user for a limited period of time. Resources are represented by resource proxies. A **resource proxy** is a process running within a specific domain that is used to translate global operations on a resource into local operations that comply with that particular domain's security policy. For example, a user proxy will typically communicate with a resource proxy when access to that resource is required.

The Globus security architecture essentially consists of entities such as users, user proxies, resource proxies, and general processes. These entities are located in domains and interact with each other. In particular, the security architecture defines four different protocols, as shown in Fig. 8-1 (see also Foster et al., 1998).

The first protocol describes precisely how a user can create a user proxy and delegate rights to that proxy. In particular, in order to let the user proxy act on behalf of its user, the user gives the proxy an appropriate set of credentials.

The second protocol specifies how a user proxy can request the allocation of a resource in a remote domain. In essence, the protocol tells a resource proxy to create a process in the remote domain after mutual authentication has taken place. That process represents the user (just as the user proxy did), but operates in the same domain as the requested resource. The process is given access to the resource subject to the access control decisions local to that domain.

A process created in a remote domain may initiate additional computations in other domains. Consequently, a protocol is needed to allocate resources in a remote domain as requested by a process other than a user proxy. In Globus, this type of allocation is done via the user proxy, by letting a process have its associated user proxy request the allocation of resources, essentially following the second protocol.

The fourth and last protocol in the Globus security architecture is the way a user can make himself known in a domain. Assuming that a user has an account in a domain, what needs to be established is that the systemwide credentials as held by a user proxy are automatically converted to credentials that are recognized by the specific domain. The protocol prescribes how the mapping between the global

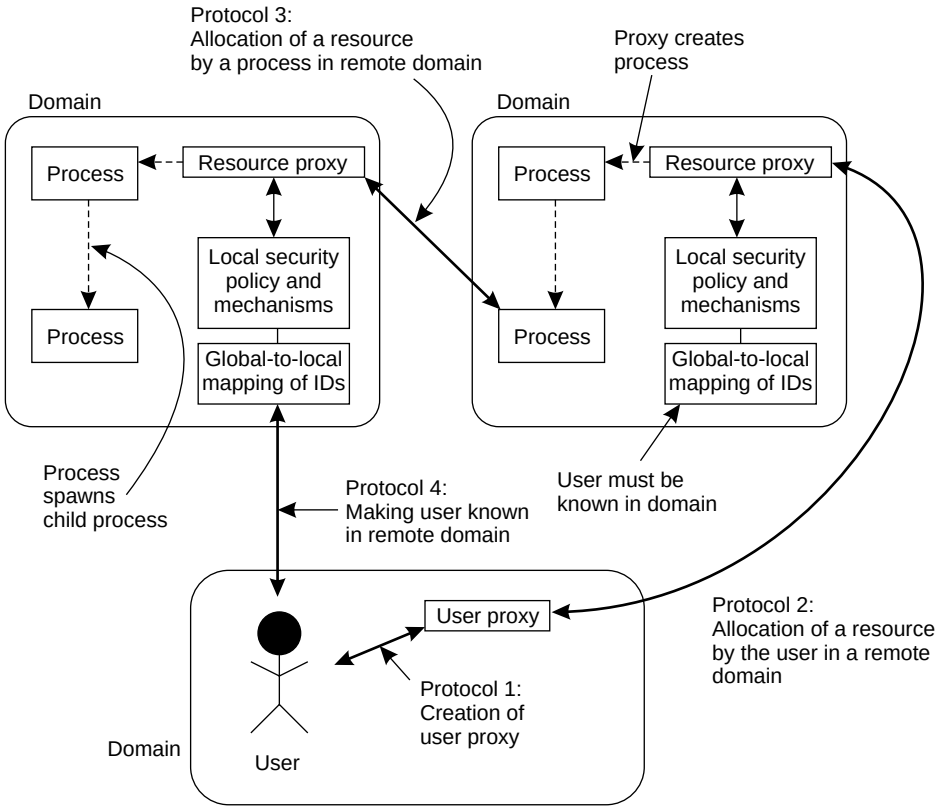


Figure 8-1. The Globus security architecture.

credentials and the local ones can be registered by the user in a mapping table local to that domain.

Specific details of each protocol are described in (Foster et al., 1998). The important issue here is that the Globus security architecture reflects its security policy as stated above. The mechanisms used to implement that architecture, in particular the above mentioned protocols, are common to many distributed systems, and are discussed extensively in this chapter. The difficulty in designing secure distributed systems is not so much caused by security mechanisms, but by deciding on how those mechanisms are to be used to enforce a security policy. In the next section, we consider some of these design decisions.

8.1.2 Design Issues

A distributed system, or any computer system for that matter, must provide security services by which a wide range of security policies can be implemented. There are a number of important design issues that need to be taken into account when implementing general-purpose security services. In the following pages, we

discuss three of these issues: focus of control, layering of security mechanisms, and simplicity (see also (Gollmann, 1999).

Focus of Control

When considering the protection of a (possibly distributed) application, there are essentially three different approaches that can be followed, as shown in Fig. 8-2. The first approach is to concentrate directly on the protection of the data that is associated with the application. By direct, we mean that irrespective of the various operations that can possibly be performed on a data item, the main concern is to ensure data integrity. Typically, this type of protection occurs in database systems in which various integrity constraints can be formulated that are automatically checked each time a data item is modified (see, for example, (Ullman, 1988).

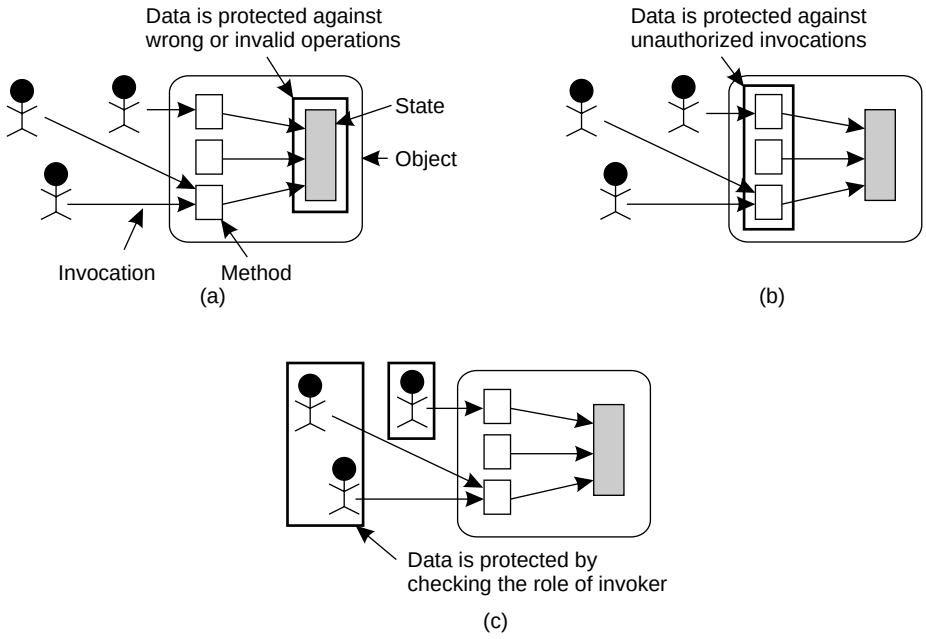


Figure 8-2. Three approaches for protection against security threats. (a) Protection against invalid operations (b) Protection against unauthorized invocations. (c) Protection against unauthorized users.

The second approach is to concentrate on protection by specifying exactly which operations may be invoked, and by whom, when certain data or resources are to be accessed. In this case, the focus of control is strongly related to access control mechanisms, which we discuss extensively later in this chapter. For example, in an object-based system, it may be decided to specify for each method that

is made available to clients which clients are permitted to invoke that method. Alternatively, access control methods can be applied to an entire interface offered by an object, or to the entire object itself. This approach thus allows for various granularities of access control.

A third approach is to focus directly on users by taking measures by which only specific people have access to the application, irrespective of the operations they want to carry out. For example, a database in a bank may be protected by denying access to anyone except higher management. As another example, in many universities, certain data and applications are restricted to be used by faculty and staff members only, whereas access by students is not allowed. In effect, control is focused on defining **roles** that users have, and once a user's role has been verified, access to a resource is either granted or denied. As part of designing a secure system, it is thus necessary to define roles that people may have, and provide mechanisms to support role-based access control. We return to roles later in this chapter.

Layering of Security Mechanisms

An important issue in designing secure systems is to decide at which level security mechanisms should be placed. A level in this context is related to the logical organization of a system into a number of layers. For example, computer networks are often organized into layers following some reference model, as we discussed in Chap. 2. In Chap. 1, we introduced the organization of distributed systems consisting of separate layers for applications, middleware, operating system services, and the operating system kernel. Combining the layered organization of computer networks and distributed systems, leads roughly to what is shown in Fig. 8-3.

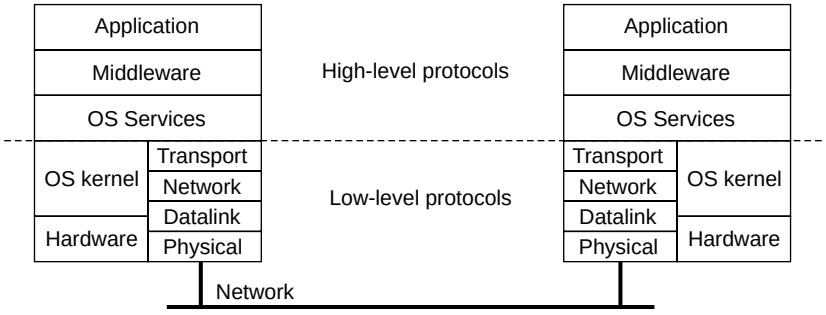


Figure 8-3. The logical organization of a distributed system into several layers.

In essence, Fig. 8-3 separates general-purpose services from communication services. This separation is important for understanding the layering of security in distributed systems and, in particular, the notion of trust. The difference between

trust and security is important. A system is either secure or it is not (taking various probabilistic measures into account), but whether a client considers a system to be secure, is a matter of trust (Pfleeger, 1997). In which layer security mechanisms are placed depends on the trust a client has in how secure the services are in a particular layer.

As an example, consider an organization located at different sites that are connected through a communication service such as **Switched Multi-megabit Data Service** (**SMDS**). An SMDS network can be thought of as a link-level backbone connecting various local-area networks at possibly geographically dispersed sites, as shown in Fig. 8-4 (for more information on SMDS see (Klessig and Tesink, 1995).

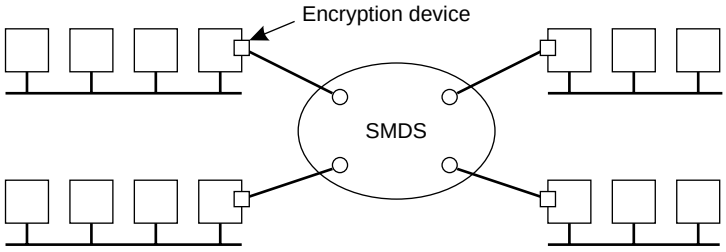


Figure 8-4. Several sites connected through a wide-area backbone service.

Security can be provided by placing encryption devices at each SMDS router, as also shown in Fig. 8-4. These devices automatically encrypt and decrypt packets that are sent between sites, but do not otherwise provide secure communication between hosts at the same site. If Alice at site *A* sends a message to Bob at site *B*, and she is worried about her message being intercepted, she must at least trust that the encryption of intersite traffic to work properly. This means, for example, that she must trust the system administrators at both sites to have taken the proper measures against tampering with the devices.

Now suppose that Alice does not trust the security of intersite traffic. She may then decide to take her own measures by using a transport-level security service such as SSL. SSL stands for **Secure Sockets Layer** and can be used to securely send messages across a TCP connection. We will discuss the details of SSL in Chap. 11 when discussing security in the Web. The important thing to note is that SSL allows Alice to set up a secure connection to Bob. All transport-level messages will be encrypted—and at the SMDS level as well, but that is of no concern to Alice. In this case, Alice will have to put her trust into SSL. In other words, she believes that SSL is secure.

In distributed systems, security mechanisms are often placed in the middleware layer. If Alice does not trust SSL, she may want to use a local secure RPC service. Again, she will have to trust this RPC service to do what it promises, such as not leaking information or properly authenticating clients and servers.

Security services that are placed in the middleware layer of a distributed system can be trusted only if the services they rely on to be secure are indeed secure. For example, if a secure RPC service is partly implemented by means of SSL, then trust in the RPC service depends on how much trust one has in SSL. If the security of SSL is not trusted, then there can be no trust in the security of the RPC service.

Distribution of Security Mechanisms

Dependencies between services regarding trust lead to the notion of a **Trusted Computing Base** (**TCB**). A TCB is the set of all security mechanisms in a (distributed) computer system that are needed to enforce a security policy. The smaller the TCB, the better. If a distributed system is built as middleware on an existing network operating system, its security may depend on the security of the underlying local operating systems. In other words, the TCB in a distributed system may include the local operating systems at various hosts.

Consider a file server in a distributed file system. Such a server may need to rely on the various protection mechanisms offered by its local operating system. Such mechanisms include not only those for protecting files against accesses by processes other than the file server, but also mechanisms to protect the file server from being maliciously brought down.

Middleware-based distributed systems thus require trust in the existing local operating systems they depend on. If such trust does not exist, then part of the functionality of the local operating systems may need to be incorporated into the distributed system itself. Consider a microkernel operating system, in which most operating-system services run as normal user processes. In this case, the file system, for instance, can be entirely replaced by one tailored to the specific needs of a distributed system, including its various security measures. Note that this approach may gradually replace a middleware-based distributed system by a distributed *operating* system such as we have discussed in Chap. 1.

Consistent with this approach is to separate security services from other types of services by distributing services across different machines depending on the required security. For example, for a secure distributed file system, it may be possible to isolate the file server from clients by placing the server on a machine with a trusted operating system, possibly running a dedicated secure file system. Clients and their applications are placed on untrusted machines.

This separation effectively reduces the TCB to a relatively small number of machines and software components. By subsequently protecting those machines against security attacks from the outside, overall trust in the security of the distributed system can be increased. Preventing clients and their applications direct access to critical services is followed in the **Reduced Interfaces for Secure System Components** (**RISSC**) approach, as described in (Neumann, 1995). In the RISSC approach, any security-critical server is placed on a separate machine

isolated from end-user systems using low-level secure network interfaces, as shown in Fig. 8-5. Clients and their applications run on different machines and can access the secured server only through these network interfaces.

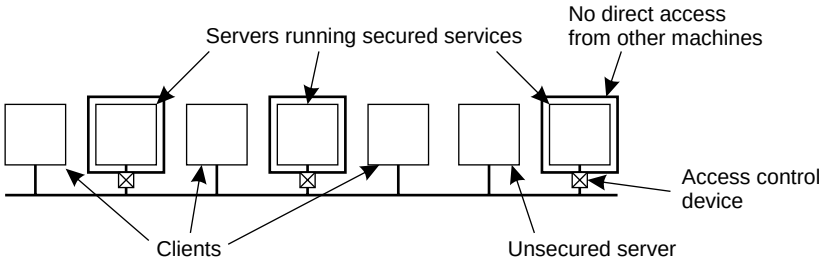


Figure 8-5. The principle of RISSC as applied to secure distributed systems.

Simplicity

Another important design issue related to deciding in which layer to place security mechanisms is that of simplicity. Designing a secure computer system is generally considered a difficult task. Consequently, if a system designer can use a few, simple mechanisms that are easily understood and trusted to work, the better it is.

Unfortunately, simple mechanisms are often not sufficient for implementing security policies. Consider again the situation in which Alice wants to send a message to Bob as discussed above. Link-level encryption is a simple and easy-to-understand mechanism to protect against interception of intersite message traffic. However, much more is needed if Alice wants to be sure that only Bob will receive her messages. In that case, user-level authentication services are needed, and Alice may need to be aware of how such services work in order to put her trust in it. User-level authentication may therefore require at least a notion of cryptographic keys and awareness of mechanisms such as certificates, despite the fact that many security services are highly automated and hidden from users.

In other cases, the application itself is inherently complex and introducing security only makes matters worse. An example application domain involving complex security protocols (as we discuss later in this chapter) is that of digital payment systems. The complexity of digital payment is often caused by the fact that multiple parties need to communicate to make a payment. In these cases, it is important that the underlying mechanisms that are used to implement the protocols are relatively simple and easy to understand. Simplicity will contribute to the trust that end users will put into the application and, more importantly, will contribute to convincing the designers that the system has no security holes.

8.1.3 Cryptography

Fundamental to security in distributed systems is the use of cryptographic techniques. The basic idea of applying these techniques is simple. Consider a sender *S* wanting to transmit message *m* to a receiver *R*. To protect the message against security threats, the sender first **encrypts** it into an unintelligible message *m′*, and subsequently sends *m′* to *R*. *R*, in turn, must **decrypt** the received message into its original form *m*.

Encryption and decryption are accomplished by using cryptographic methods parameterized by keys, as shown in Fig. 8-6. The original form of the message that is sent is called the **plaintext**, shown as *P* in Fig. 8-6; the encrypted form is referred to as the **ciphertext**, illustrated as *C*.

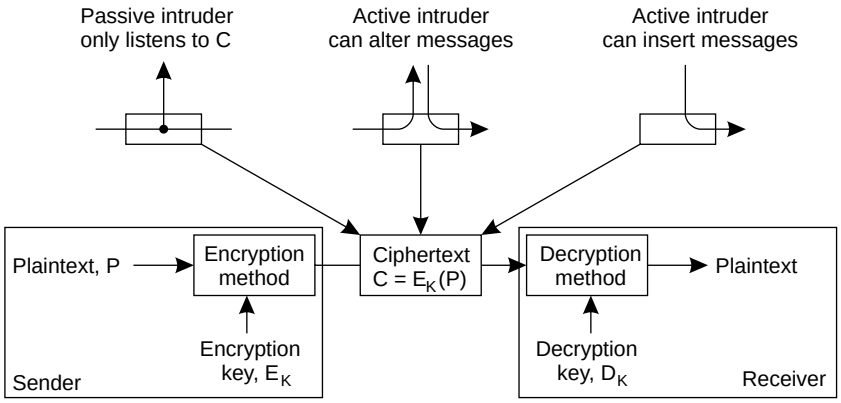


Figure 8-6. Intruders and eavesdroppers in communication.

To describe the various security protocols that are used in building security services for distributed systems, it is useful to have a notation to relate plaintext, ciphertext, and keys. Following the common notational conventions, we will use $C = E_K(P)$ to denote that the ciphertext *C* is obtained by encrypting the plaintext *P* using key *K*. Likewise, $P = D_K(C)$ is used to express the decryption of the ciphertext *C* using key *K*, resulting in the plaintext *P*.

Returning to our example shown in Fig. 8-6, while transferring a message as ciphertext *C*, there are three different attacks that we need to protect against, and for which encryption helps. First, an **intruder** may intercept the message without either the sender or receiver being aware that eavesdropping is happening. Of course, if the transmitted message has been encrypted in such a way that it cannot be easily decrypted without having the proper key, interception is useless: the intruder will see only unintelligible data. (By the way, the fact alone that a message is being transmitted may sometimes be enough for an intruder to draw conclusions. For example, if during a world crisis the amount of traffic into the White House suddenly drops to zero while the amount of traffic going into a certain

mountain in Colorado increases by the same amount, there may be useful information in knowing that.)

The second type of attack that needs to be dealt with is that of modifying the message. Modifying plaintext is easy; modifying ciphertext that has been properly encrypted is much more difficult because the intruder will first have to decrypt the message before it can meaningfully modify it. In addition, he will also have to properly encrypt it again or otherwise the receiver may notice that the message has been tampered with.

The third type of attack is when an intruder inserts encrypted messages into the communication system, attempting to make *R* believe these messages came from *S*. Again, as we shall see later in this chapter, encryption can help protect against such attacks. Note that if an intruder can modify messages, he can also insert messages.

There is a fundamental distinction between different cryptographic systems, based on whether or not the encryption and decryption key are the same. In a **symmetric cryptosystem**, the same key is used to encrypt and decrypt a message. In other words,

$$P = D_K(E_K(P))$$

Symmetric cryptosystems are also referred to as secret-key or shared-key systems, because the sender and receiver are required to share the same key, and to ensure that protection works, this shared key must be kept secret; no one else is allowed to see the key. We will use the notation $K_{A,B}$ to denote a key shared by *A* and *B*.

In an **asymmetric cryptosystem**, the keys for encryption and decryption are different, but together form a unique pair. In other words, there is a separate key K_E for encryption and one for decryption, K_D, such that

$$P = D_{K_D}(E_{K_E}(P))$$

One of the keys in an asymmetric cryptosystem is kept private, the other is made public. For this reason, asymmetric cryptosystems are also referred to as **public-key systems**. In what follows, we use the notation K_A^+ to denote a public key belonging to *A*, and K_A^- as its corresponding private key.

Anticipating the detailed discussions on security protocols later in this chapter, which one of the encryption or decryption keys that is actually made public depends on how the keys are used. For example, if Alice wants to send a confidential message to Bob, she should use Bob's public key to encrypt the message. Because Bob is the only one holding the private decryption key, he is also the only person that can decrypt the message.

On the other hand, suppose that Bob wants to know for sure that the message he just received actually came from Alice. In that case, Alice can keep her encryption key private to encrypt the messages she sends. If Bob can successfully decrypt a message using Alice's public key (and the plaintext in the message has enough information to make it meaningful to Bob), he knows that message must

have come from Alice, because the decryption key is uniquely tied to the encryption key. We return to such algorithms in detail below.

One final application of cryptography in distributed systems is the use of **hash functions**. A hash function H takes a message m of arbitrary length as input and produces a bit string h having a fixed length as output:

$$h = H(m)$$

A hash h is comparable to the extra bits that are added to a message in communication systems to allow for error detection, such a cyclic-redundancy check (CRC).

Hash functions that are used in cryptographic systems have a number of properties. First, they are **one-way functions**, meaning that it is computationally infeasible to find the input m that corresponds to a known output h. On the other hand, computing h from m is easy. Second, they have the **weak collision resistance** property, meaning that given an input m and its associated output $h = H(m)$, it is computationally infeasible to find another, different input $m' \neq m$, such that $H(m) = H(m')$. Finally, cryptographic hash functions also have the **strong collision resistance** property, which means that, when given only H, it is computationally infeasible to find any two different input values m and m', such that $H(m) = H(m')$.

Similar properties apply to any encryption function E and the keys that are used. Furthermore, for any encryption function E, it should be computationally infeasible to find the key K when given the plaintext P and associated ciphertext $C = E_K(P)$. Likewise, analogous to collision resistance, when given a plaintext P and a key K, it should be impossible to find another key K' such that $E_K(P) = E_{K'}(P)$.

The art and science of devising algorithms for cryptographic systems has a long history (Kahn, 1967), and building secure systems is often surprisingly difficult, or even impossible (Schneier, 2000). It is beyond the scope of this book to discuss any of these algorithms in detail. However, to give some impression of cryptography in computer systems, we will now briefly present three representative algorithms. Detailed information on these and other cryptographic algorithms can be found in (Kaufman et al., 1995; Menezes et al., 1996; and Schneier, 1996).

Before we go into the details of the various protocols, Fig. 8-7 summarizes the notation and abbreviations we use.

Notation	Description
$K_{A,B}$	Secret key shared by A and B
K_A^+	Public key of A
K_A^-	Private key of A

Figure 8-7. Notation used in this chapter.

Symmetric Cryptosystems: DES

Our first example of a cryptographic algorithm is the **Data Encryption Standard** (**DES**), which is used for symmetric cryptosystems. DES is designed to operate on 64-bit blocks of data. A block is transformed into an encrypted (64 bit) block of output in 16 rounds, where each round uses a different 48-bit key for encryption. Each of these 16 keys is derived from a 56-bit master key, as shown in Fig. 8-8(a). Before an input block starts its 16 rounds of encryption, it is first subject to an initial permutation, of which the inverse is later applied to the encrypted output leading to the final output block.

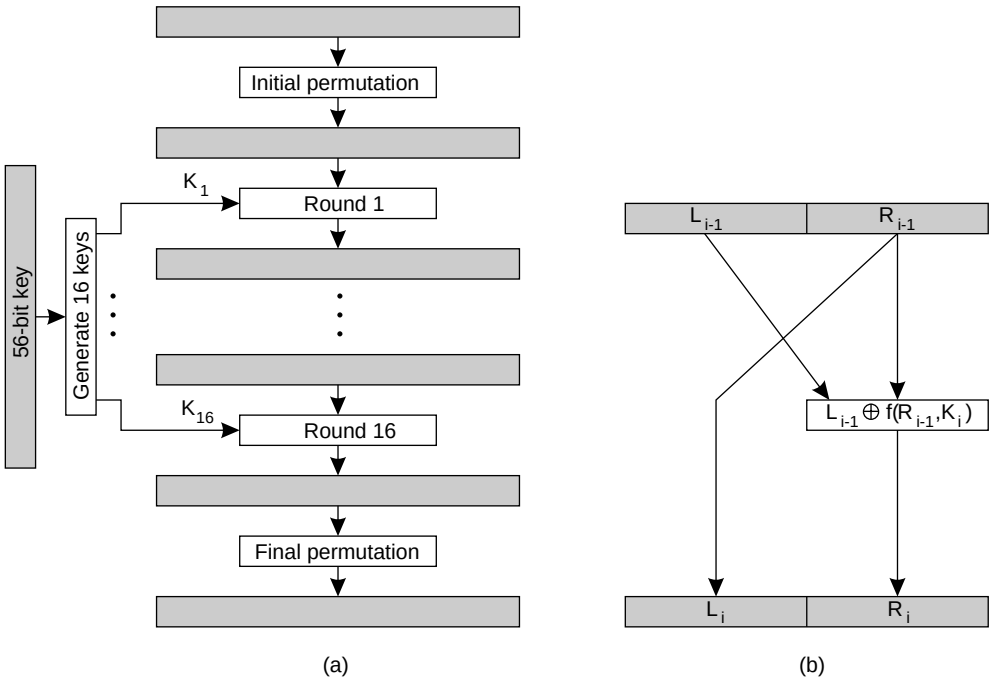


Figure 8-8. (a) The principle of DES. (b) Outline of one encryption round.

Each encryption round i takes the 64-bit block produced by the previous round $i-1$ as its input, as shown in Fig. 8-8(b). The 64 bits are split into a left part L_{i-1} and a right part R_{i-1}, each containing 32 bits. The right part is used for the left part in the next round, that is, $L_i = R_{i-1}$.

The hard work is done in the mangler function f. This function takes a 32-bit block R_{i-1} as input, together with a 48-bit key K_i, and produces a 32-bit block that is XORed with L_{i-1} to produce R_i. (XOR is an abbreviation for the exclusive or operation.) The mangler function first expands R_{i-1} to a 48-bit block and XORs it with K_i. The result is partitioned into eight chunks of six bits each. Each chunk is

then fed into a different **S-box**, which is an operation that substitutes each of the 64 possible 6-bit inputs into one of 16 possible 4-bit outputs. The eight output chunks of four bits each are then combined into a 32-bit value and permuted again.

The 48-bit key K_i for round i is derived from the 56-bit master key as follows. First, the master key is permuted and divided into two 28-bit halves. For each round, each half is first rotated one or two bits to the left, after which 24 bits are extracted. Together with 24 bits from the other rotated half, a 48-bit key is constructed. The details of one encryption round are shown in Fig. 8-9.

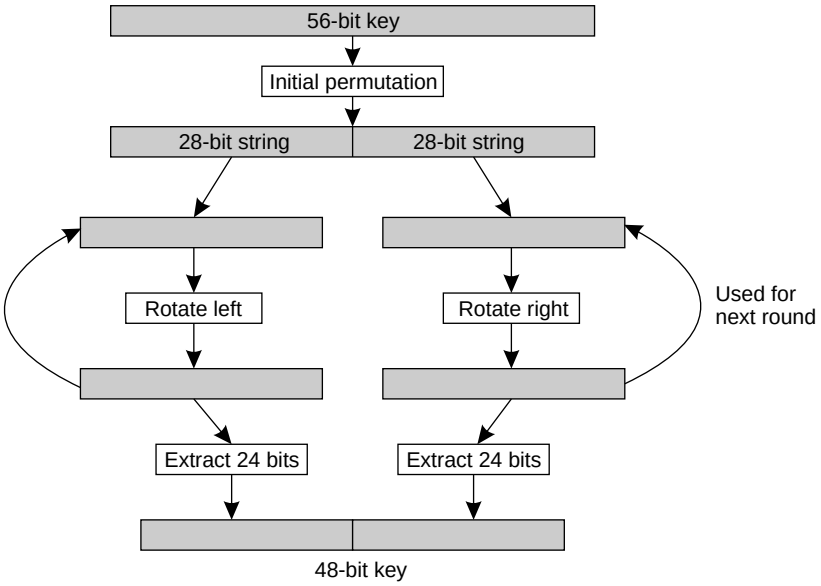


Figure 8-9. Details of per-round key generation in DES.

The principle of DES is quite simple, but the algorithm is difficult to break using analytical methods. Using a brute-force attack by simply searching for a key that will do the job has become easy as has been demonstrated in recent times. Using DES three times in a special encrypt-decrypt-encrypt mode with different keys appears to be safe for the moment.

What makes DES difficult to attack by analysis is that the rationale behind the design has never been clearly explained. For example, it has been shown that taking other S-boxes than are currently used in the standard, makes the algorithm substantially easier to break (see Pfleeger, 1997 for a brief analysis of DES). A rationale for the design and use of the S-boxes was published only after ''new'' attack models had been devised in the 1990s. DES proved to be quite resistant to these attacks, and its designers revealed that the newly devised models had already been known to them when they developed DES in 1974 (Coppersmith,

1994).

DES has been used as a standard encryption technique for years, but is currently in the process of being replaced by the Rijndael algorithm blocks of 128 bits. There are also variants with larger keys and larger data blocks. The algorithm has been designed to be fast enough so that it can even be implemented on smart cards, which form an increasingly important application area for cryptography.

Public-Key Cryptosystems: RSA

Our second example of a cryptographic algorithm is widely used for public-key systems: **RSA**, named after its inventors Rivest, Shamir, and Adleman (1978). The security of RSA comes from the fact that no methods are known to efficiently find the **prime factors** of large numbers. It can be shown that each integer can be written as the product of prime numbers. For example, 2100 can be written as

$$2100 = 2 \times 2 \times 3 \times 5 \times 5 \times 7$$

making 2, 3, 5, and 7 the prime factors in 2100. In RSA, the private and public keys are constructed from very large prime numbers (consisting of hundreds of decimal digits). As it turns out, breaking RSA is equivalent to finding those two prime numbers. So far, this has shown to be computationally infeasible despite mathematicians working on the problem for centuries.

Generating the private and public keys requires four steps:

1. Choose two very large prime numbers, p and q.
2. Compute $n = p \times q$ and $z = (p - 1) \times (q - 1)$.
3. Choose a number d that is relatively prime to z.
4. Compute the number e such that $e \times d = 1 \ mod \ z$.

One of the numbers, say d, can subsequently be used for decryption, whereas e is used for encryption. Only one of these two is made public, depending on what the algorithm is being used for.

Let us consider the case that Alice wants to keep the messages she sends to Bob confidential. In other words, she wants to ensure that no one but Bob can intercept and read her messages to him. RSA considers each message m to be just a string of bits. Each message is first divided into fixed-length blocks, where each block m_i, interpreted as a binary number, should lie in the interval $0 \leq m_i < n$.

To encrypt message m, the sender calculates for each block m_i the value $c_i = m_i^e \ (mod \ n)$, which is then sent to the receiver. Decryption at the receiver's side takes place by computing $m_i = c_i^d \ (mod \ n)$. Note that for the encryption, both e and n are needed, whereas decryption requires knowing the values d and n.

When comparing RSA to symmetric cryptosystems such as DES, RSA has the drawback of being computationally more complex. As it turns out, encrypting

messages using RSA is approximately 100–1000 times slower than DES, depending on the implementation technique used. As a consequence, many cryptographic systems use RSA to exchange only shared keys in a secure way, but much less for actually encrypting ''normal'' data. We will see examples of the combination of these two techniques later in succeeding sections.

Hash Functions: MD5

As a last example of a widely used cryptographic algorithm, we take a look at MD5 (Rivest, 1992). **MD5** is a hash function for computing a 128-bit, fixed length **message digest** from an arbitrary length binary input string. The input string is first padded to a total length of 448 bits (modulo 512), after which the length of the original bit string is added as a 64-bit integer. In effect, the input is converted to a series of 512-bit blocks.

The structure of the algorithm is shown in Fig. 8-10. Starting with some constant 128-bit value, the algorithm proceeds in k phases, where k is the number of 512-bit blocks comprising the padded message. During each phase, a 128-bit digest is computed out of a 512-bit block of data coming from the padded message, and the 128-bit digest computed in the preceding phase.

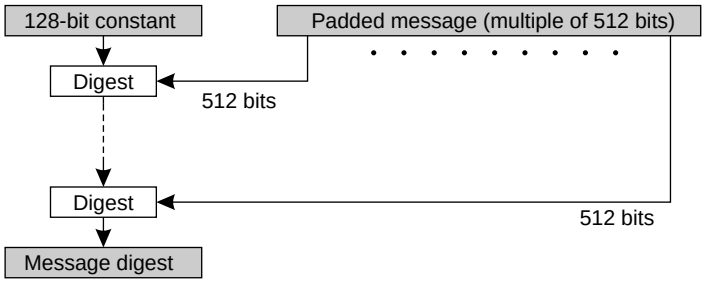


Figure 8-10. The structure of MD5.

A phase in MD5 consists of four rounds of computations, where each round uses one of the following four functions:

$$F(x,y,z) = (x\ AND\ y)\ OR\ ((NOT\ x)\ AND\ z)$$
$$G(x,y,z) = (x\ AND\ z)\ OR\ (y\ AND\ (NOT\ z))$$
$$H(x,y,z) = x\ XOR\ y\ XOR\ z$$
$$I(x,y,z) = y\ XOR\ (x\ OR\ (NOT\ z))$$

Each of these functions operates on 32-bit variables x, y, and z. To illustrate how these functions are used, consider a 512-bit block b from the padded message that is being processed during phase k. Block b is divided into 16 32-bit subblocks $b_0, b_1, ..., b_{15}$. During the first round, function F is used to change four variables (denoted as p, q, r, and s, respectively) in 16 iterations as shown in Fig. 8-11

These variables are carried to each next round, and after a phase has finished, passed on to the next phase. There are a total of 64 predefined constants C_i. The notation $x \lll n$ is used to denote a *left rotate*: the bits in x are shifted n positions to the left, where the bit shifted off the left is placed in the rightmost position.

Iterations 1-8	Iterations 9-16
$p \leftarrow (p + F(q,r,s) + b_0 + C_1) \lll 7$	$p \leftarrow (p + F(q,r,s) + b_8 + C_9) \lll 7$
$s \leftarrow (s + F(p,q,r) + b_1 + C_2) \lll 12$	$s \leftarrow (s + F(p,q,r) + b_9 + C_{10}) \lll 12$
$r \leftarrow (r + F(s,p,q) + b_2 + C_3) \lll 17$	$r \leftarrow (r + F(s,p,q) + b_{10} + C_{11}) \lll 17$
$q \leftarrow (q + F(r,s,p) + b_3 + C_4) \lll 22$	$q \leftarrow (q + F(r,s,p) + b_{11} + C_{12}) \lll 22$
$p \leftarrow (p + F(q,r,s) + b_4 + C_5) \lll 7$	$p \leftarrow (p + F(q,r,s) + b_{12} + C_{13}) \lll 7$
$s \leftarrow (s + F(p,q,r) + b_5 + C_6) \lll 12$	$s \leftarrow (s + F(p,q,r) + b_{13} + C_{14}) \lll 12$
$r \leftarrow (r + F(s,p,q) + b_6 + C_7) \lll 17$	$r \leftarrow (r + F(s,p,q) + b_{14} + C_{15}) \lll 17$
$q \leftarrow (q + F(r,s,p) + b_7 + C_8) \lll 22$	$q \leftarrow (q + F(r,s,p) + b_{15} + C_{16}) \lll 22$

Figure 8-11. The 16 iterations during the first round in a phase in MD5.

The second round uses the function G in a similar fashion, whereas H and I are used in the third and fourth round, respectively. Each step thus consists of 64 iterations, after which the next phase is started, but now with the values that p, q, r, and s have at that point.