CHAPTER 30

Message Security, User Authentication, and Key Management

After studying cryptography in Chapter 29, we discuss some of its applications: message security, user authentication, and key management.

Message security involves confidentiality, integrity, authentication, and finally nonrepudiation.

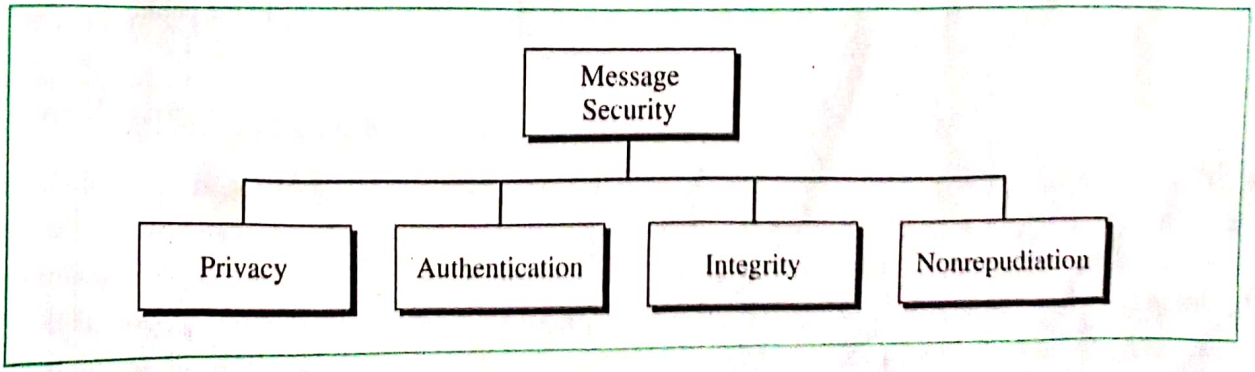
User authentication means verifying the identity of the person or process that wants to communicate with a system. User authentication is also needed for key management.

Finally, we need key management: the distribution of symmetric keys and the certification of the public keys. Section 30.4 explains the methods used in key management.

30.1 MESSAGE SECURITY

Let us first discuss the security measures applied to each single message. We can say that security provides four services: privacy (confidentiality), message authentication, message integrity, and nonrepudiation (see Fig. 30.1).

Figure 30.1 *Message security*



Privacy

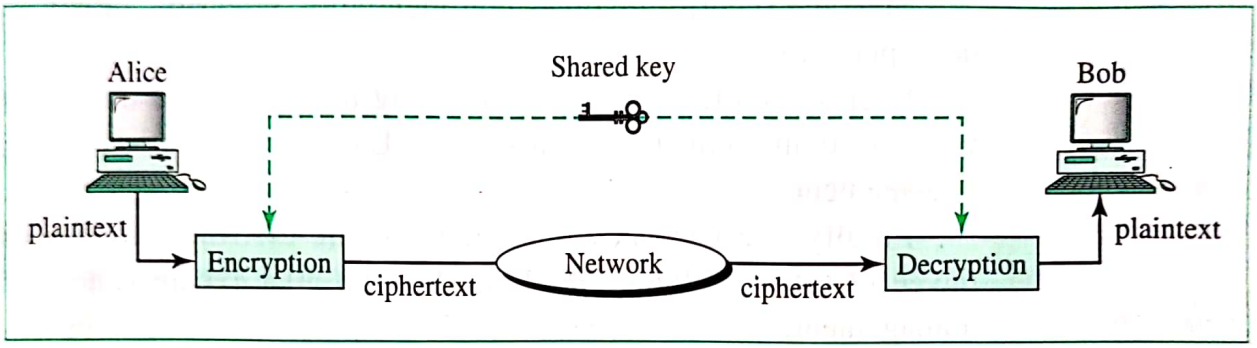
Privacy means that the sender and the receiver expect confidentiality. The transmitted message must make sense to only the intended receiver. To all others, the message must be unintelligible.

The concept of how to achieve privacy has not changed for thousands of years: The message must be encrypted. That is, the message must be rendered unintelligible to unauthorized parties. A good privacy technique guarantees to some extent that a potential intruder (eavesdropper) cannot understand the contents of the message.

Privacy with Symmetric-Key Cryptography

Privacy can be achieved using symmetric-key encryption and decryption, as shown in Figure 30.2. As we discussed in Chapter 29, in symmetric-key cryptography the key is shared between Alice and Bob.

Figure 30.2 *Privacy using symmetric-key encryption*

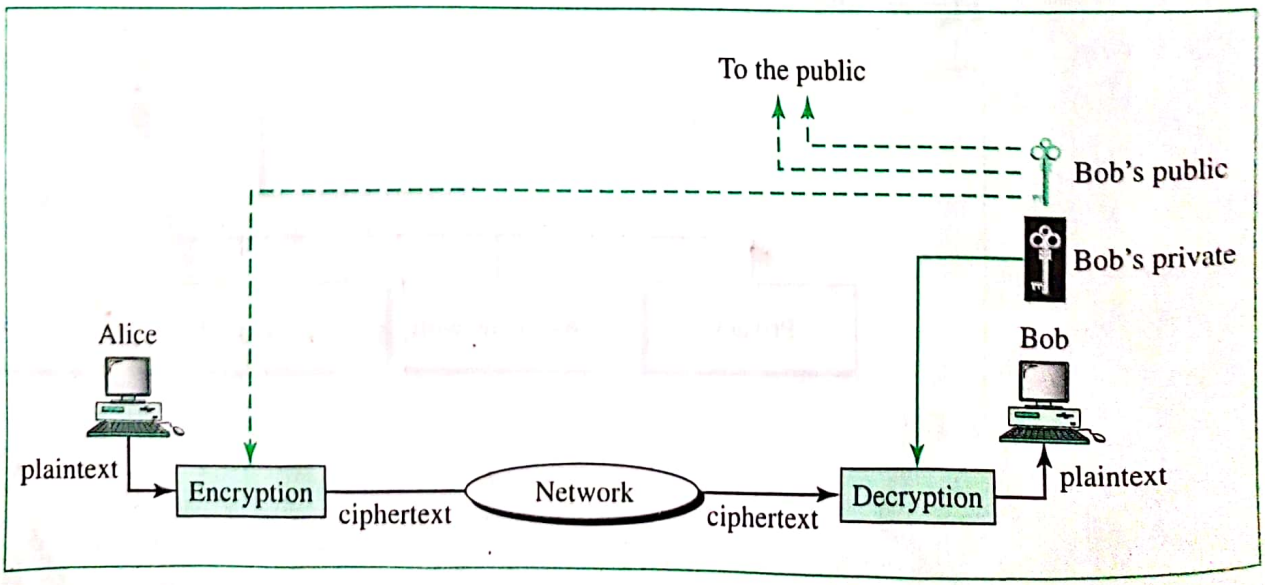


Using symmetric-key cryptography is very common for achieving privacy. Later in this chapter, we will see how to manage the distribution of symmetric keys.

Privacy with Public-Key Cryptography

We can also achieve privacy using public-key encryption. There are two keys: a private key and a public key. The private key is kept by the receiver. The public key is announced to the public. This is shown in Figure 30.3.

Figure 30.3 *Privacy using public-key encryption*



The main problem with public key encryption is its owner must be verified (certified). We will see how to solve this problem shortly.

Message Authentication

Message **authentication** means that the receiver needs to be sure of the sender's identity and that an imposter has not sent the message. We will see how digital signature can provide message authentication.

Integrity

Integrity means that the data must arrive at the receiver exactly as they were sent. There must be no changes during the transmission, either accidental or malicious. As more and more monetary exchanges occur over the Internet, integrity is crucial. For example, it would be disastrous if a request for transferring $100 changed to a request for $10,000 or $100,000. The integrity of the message must be preserved in a secure communication. We will see how digital signature can provide message integrity.

Nonrepudiation

Nonrepudiation means that a receiver must be able to prove that a received message came from a specific sender. The sender must not be able to deny sending a message that he or she, in fact, did send. The burden of proof falls on the receiver. For example, when a customer sends a message to transfer money from one account to another, the bank must have proof that the customer actually requested this transaction. We will see how digital signature can provide nonrepudiation.

30.2 DIGITAL SIGNATURE

We said that security provides four services in relation to a single message: privacy, authentication, integrity, and nonrepudiation. We have already discussed privacy. The other three can be achieved by using what is called **digital signature.**

The idea is similar to the signing of a document. When we send a document electronically, we can also sign it. We have two choices: We can sign the entire document, or we can sign a digest (condensed version) of the document.

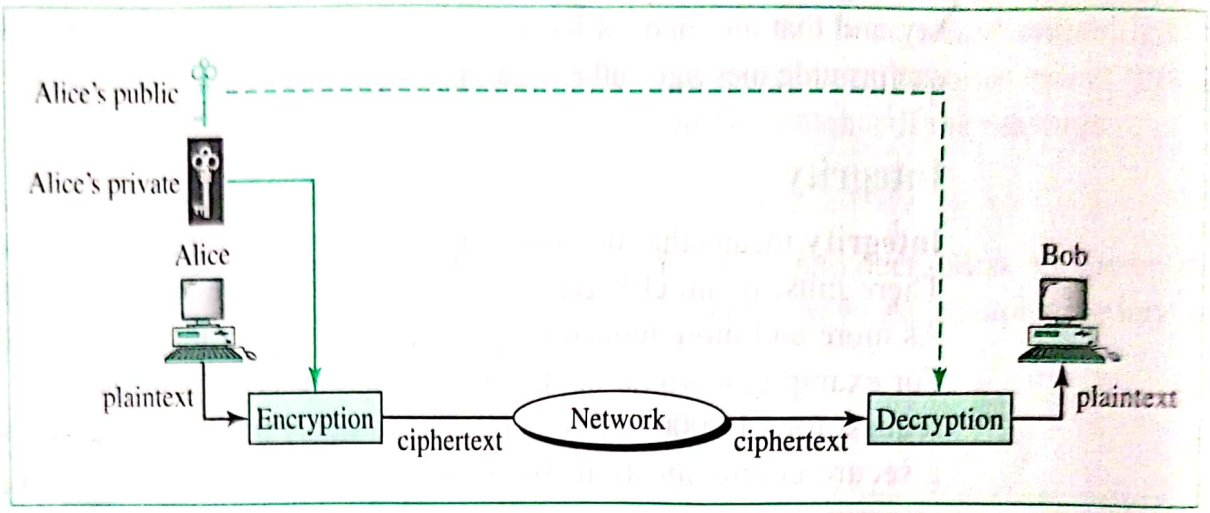
Signing the Whole Document

Public-key encryption can be used to sign a document. However, the roles of the public and private keys are different here. The sender uses her private key to encrypt (sign) the message just as a person uses her signature (which is private in the sense that it is difficult to forge) to sign a paper document. The receiver, on the other hand, uses the public key of the sender to decrypt the message just as a person verifies from memory another person's signature.

In the digital signature, the private key is used for encryption and the public key for decryption. This is possible because the encryption and decryption algorithms used today,

such as RSA, are mathematical formulas and their structures are similar. Figure 30.4 shows how this is done.

Figure 30.4 *Signing the whole document*



Digital signatures can provide integrity, authentication, and nonrepudiation.

Integrity The integrity of a message is preserved because if Eve intercepted the message and partially or totally changed it, the decrypted message would be unreadable.

Authentication We can use the following reasoning to show how a message can be authenticated. If Eve sends a message while pretending that it is coming from Alice, she must use her own private key for encryption. The message is then decrypted with the public key of Alice and will therefore be nonreadable. Encryption with Eve's private key and decryption with Alice's public key result in garbage.

Nonrepudiation Digital signature also provides for nonrepudiation. Bob saves the message received from Alice. If Alice later denies sending the message, Bob can show that encrypting and decrypting the saved message with Alice's private and public key can create a duplicate of the saved message. Since only Alice knows her private key, she cannot deny sending the message.

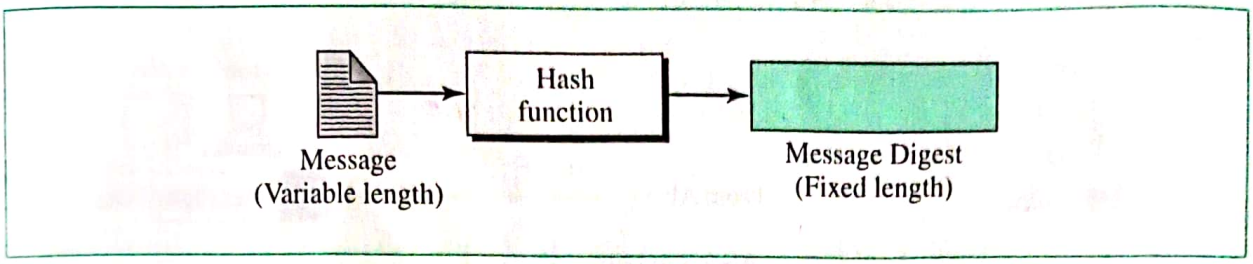
Digital signature does not provide privacy. If there is a need for privacy, another layer of encryption/decryption must be applied.

Signing the Digest

We said before that public-key encryption is efficient if the message is short. Using a public key to sign the entire message is very inefficient if the message is very long. The solution is to let the sender sign a digest of the document instead of the whole document. The sender creates a miniature version or **digest** of the document and signs it; the receiver then checks the signature on the miniature.

To create a digest of the message, we use a **hash function.** The hash function creates a fixed-size digest from a variable-length message, as shown in Figure 30.5.

Figure 30.5 *Signing the digest*

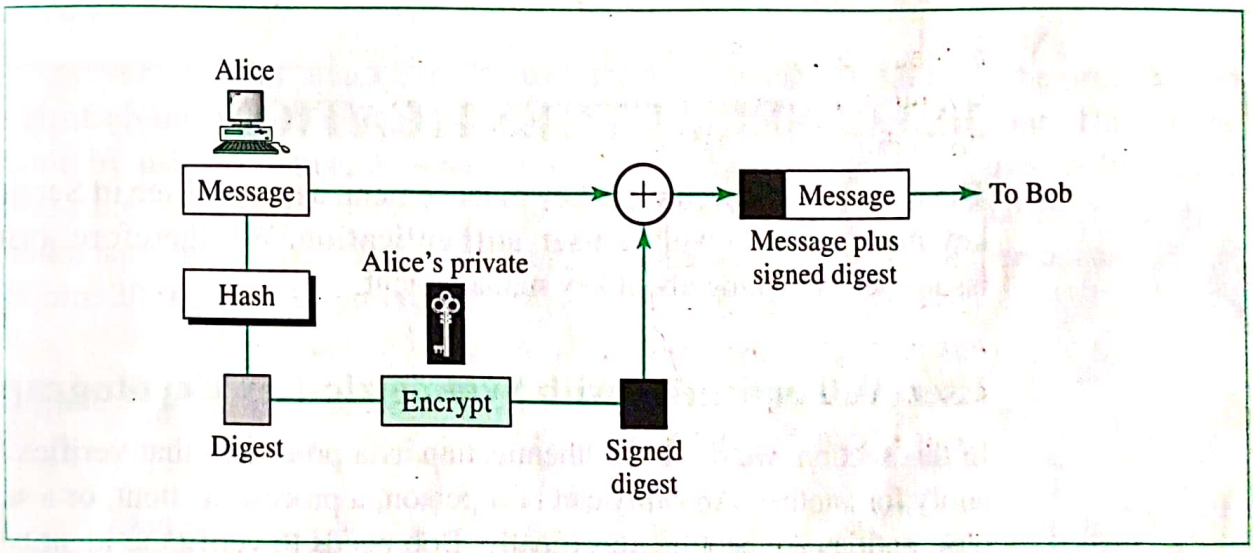


The two most common hash functions are called MD5 (Message Digest 5) and SHA-1 (Secure Hash Algorithm 1). The first one produces a 120-bit digest. The second produces a 160-bit digest.

Note that a hash function must have two properties to guarantee its success. First, hashing is one-way; the digest can only be created from the message, not vice versa. Second, hashing is a one-to-one function; there is little probability that two messages will create the same digest. We will see the reason for this condition shortly.

After the digest has been created, it is encrypted (signed) using the sender's private key. The encrypted digest is attached to the original message and sent to the receiver. Figure 30.6 shows the sender site.

Figure 30.6 *Sender site*

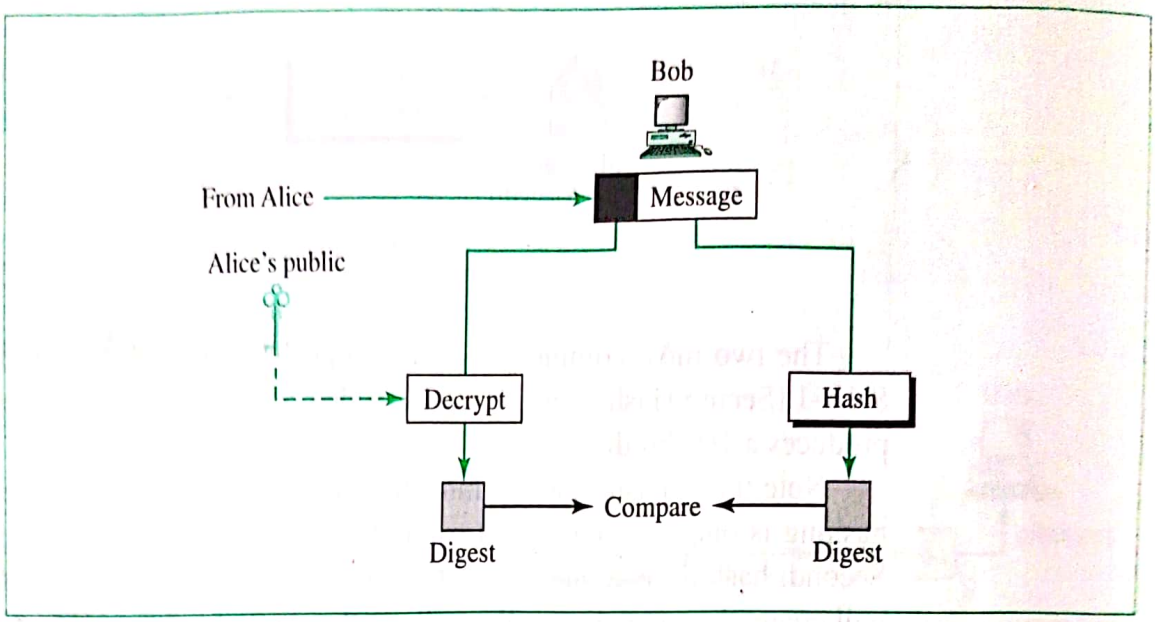


The receiver receives the original message and the encrypted digest. He separates the two. He applies the same hash function to the message to create a second digest. He also decrypts the received digest, using the public key of the sender. If the two digests are the same, all three security measures are preserved. Figure 30.7 shows the receiver site.

According to Section 30.1, we know that the digest is secure in terms of integrity, authentication, and nonrepudiation, but what about the message itself? The following reasoning shows that the message itself is also secured:

1. The digest has not been changed (integrity), and the digest is a representation of the message. So the message has not been changed (remember, it is improbable that two messages create the same digest). Integrity has been provided.

Figure 30.7 *Receiver site*



2. The digest comes from the true sender, so the message also comes from the true sender. If an intruder had initiated the message, the message would not have created the same digest (it is improbable that two messages create the same digest).
3. The sender cannot deny the message since she cannot deny the digest; the only message that can create that digest, with a very high probability, is the received message.

30.3 USER AUTHENTICATION

The main issue in security is key management, as we will see in Section 30.4. However, key management involves **user authentication.** We, therefore, briefly discuss these issues before talking about key management.

User Authentication with Symmetric-Key Cryptography

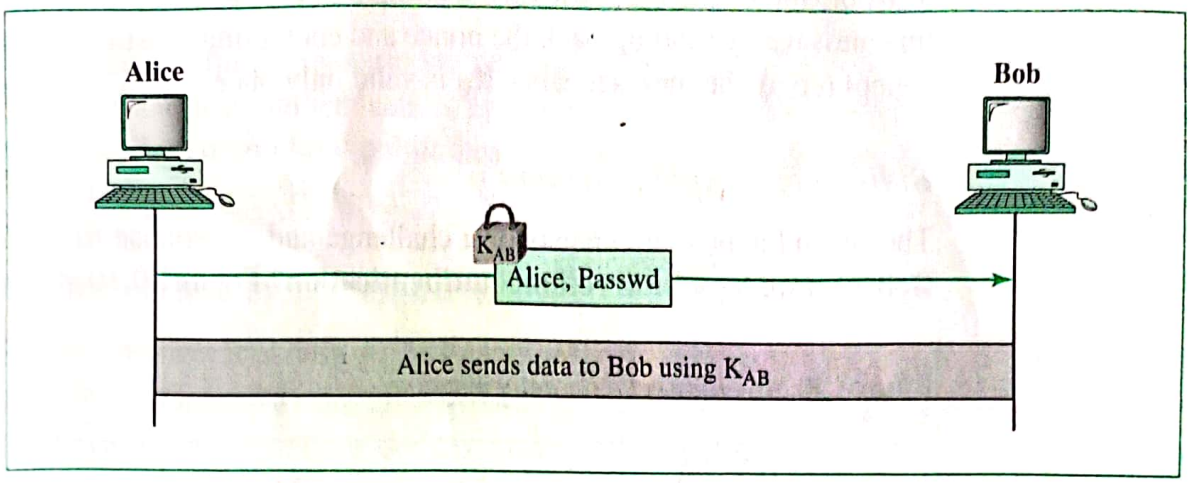
In this section, we discuss authentication as a procedure that verifies the identity of one entity for another. An *entity* can be a person, a process, a client, or a server; in our examples, entities are people. Specifically, Bob needs to verify the identity of Alice and vice versa. Note that entity authentication, as discussed here, is different from the message authentication that we discussed in the previous section. In message authentication, the identity of the sender is verified for each single message. In user authentication, the user identity is verified once for the entire duration of system access.

First Approach

In the first approach, Alice sends her identity and password in an encrypted message, using the symmetric key K_{AB}. Figure 30.8 shows the procedure. We have added the padlock with the corresponding key (shared key between Alice and Bob) to show that the message is encrypted with the key.

Is this a safe approach? Yes, to some extent. Eve, the intruder, cannot decipher the password or the data because she does not know K_{AB}. However, Eve can cause damage

Figure 30.8 *Using a symmetric key only*

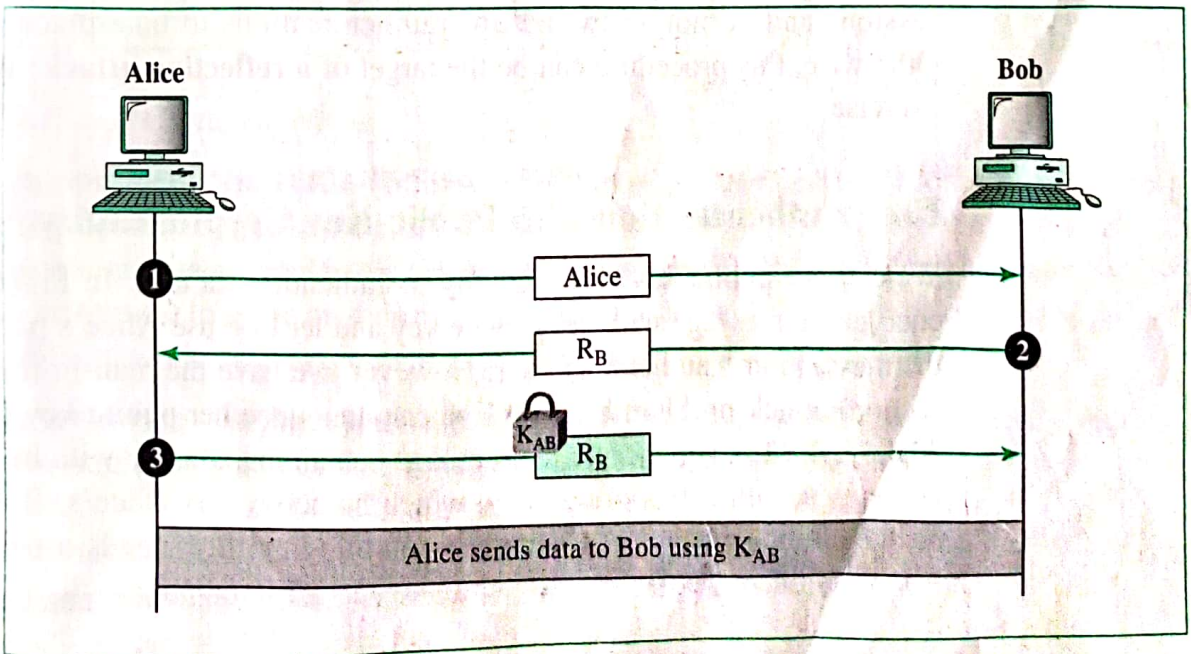


without accessing the contents of the message. If Eve has an interest in the data message sent from Alice to Bob, she can intercept both the authentication message and the data message, store them, and resend them later to Bob. Bob has no way to know that this is a replay of a previous message. There is nothing in this procedure to guarantee the freshness of the message. As an example, suppose Alice's message instructs Bob (as a bank manager) to pay Eve for some job she has done. Eve can resend the message, thereby illegally getting paid twice for the same job. This is called a **replay attack.**

Second Approach

To prevent a replay attack (or playback attack), we add something to the procedure to help Bob distinguish a fresh authentication request from a repeated one. This can be done by using a **nonce.** A nonce is a large random number that is used only once, a one-time number. In this second approach, Bob uses a nonce to challenge Alice, to make sure that Alice is authentic and that someone (Eve) is not impersonating Alice. Figure 30.9 shows the procedure.

Figure 30.9 *Using a nonce*

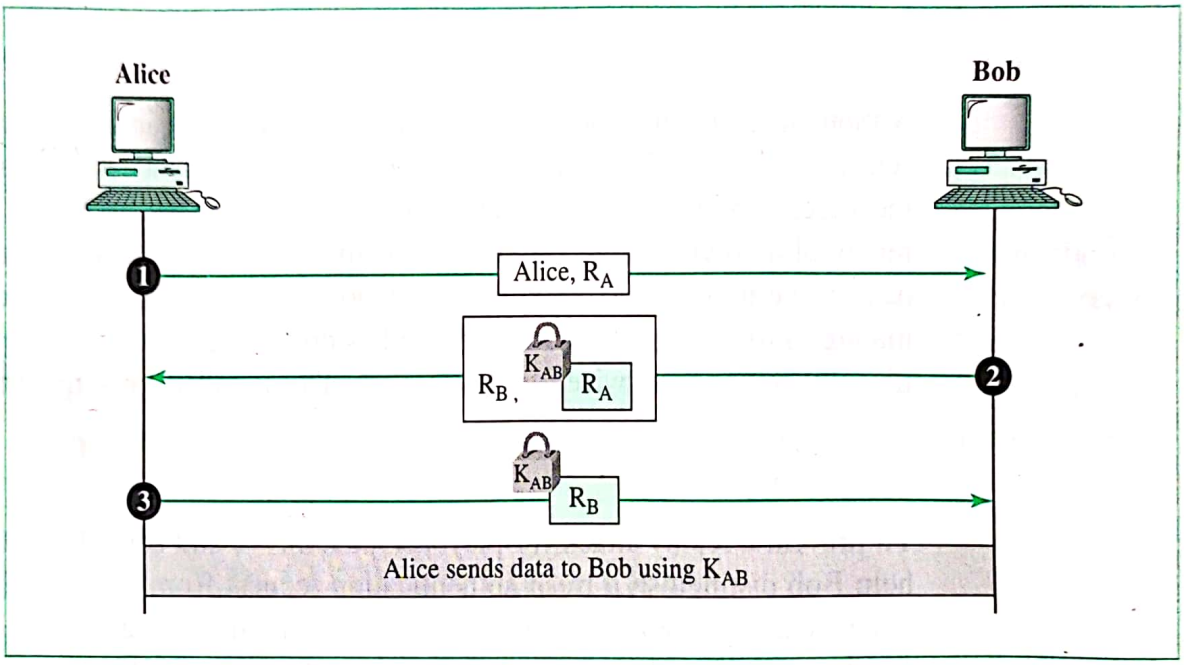


Authentication happens in three steps. First, Alice sends her identity, in plaintext, to Bob. Bob challenges Alice by sending a nonce, R_B, in plaintext. Alice responds to this message by sending back the nonce and encrypting it using the symmetric key. Eve cannot replay the message since R_B is valid only once.

Bidirectional Authentication

The second approach consists of a challenge and a response to authenticate Alice for Bob. Can we have **bidirectional authentication?** Figure 30.10 shows one method.

Figure 30.10 *Bidirectional authentication*



In the first step, Alice sends her identification and her nonce to challenge Bob. In the second step, Bob responds to Alice's challenge by sending his nonce to challenge her. In the third step, Alice responds to Bob's challenge. Is this authentication totally safe? It is on the condition that Alice and Bob use a different set of nonces for different sessions and do not allow multiple authentications to take place at the same time. Otherwise, this procedure can be the target of a **reflection attack;** we leave this as an exercise.

User Authentication with Public-Key Cryptography

We can use public-key cryptography to authenticate a user. In Figure 30.9, Alice can encrypt the message with her private key and let Bob use Alice's public key to decrypt the message and authenticate her. However, we have the man-in-the-middle (see next section) attack problem because Eve can announce her public key to Bob in place of Alice. Eve can then encrypt the message containing a nonce with her private key. Bob decrypts it with Eve's public key, which he believes is Alice's. Bob is fooled. Alice needs a better means to advertise her public key; Bob needs a better way to verify Alice's public key. We discuss public-key certification next.

30.4 KEY MANAGEMENT

We discussed how symmetric-key and public-key cryptography can be used in message security and user authentication. However, we never explained how symmetric keys are distributed and how public keys are certified. We explore these two important issues here.

Symmetric Key Distribution

There are three problems with symmetric keys.

1. First, if n people want to communicate with one another, there is a need for $n(n-1)/2$ symmetric keys. Consider that each of the n people may need to communicate with $n-1$ people. This means that we need $n(n-1)$ keys. However, symmetric keys are shared between two communicating people. Therefore, the actual number of keys needed is $n(n-1)/2$. This is usually referred to as the **n^2 problem.** If n is a small number, this is acceptable. For example, if 5 people need to communicate, only 10 keys are needed. The problem is aggravated if n is a large number. For example, if n is 1 million, almost half a trillion keys are needed.
2. Second, in a group of n people, each person must have and remember $n-1$ keys, one for every other person in the group. This means that if 1 million people want to communicate with one another, each must remember (or store) almost 1 million keys in his or her computer.
3. Third, how can two parties securely acquire the shared key? It cannot be done over the phone or the Internet; these are not secure.

Session Keys

Considering the above problems, a symmetric key between two parties is useful if it is dynamic: created for each session and destroyed when the session is over. It does not have to be remembered by the two parties.

A symmetric key between two parties is useful if it is used only once; it must be created for one session and destroyed when the session is over.

Diffie-Hellman Method

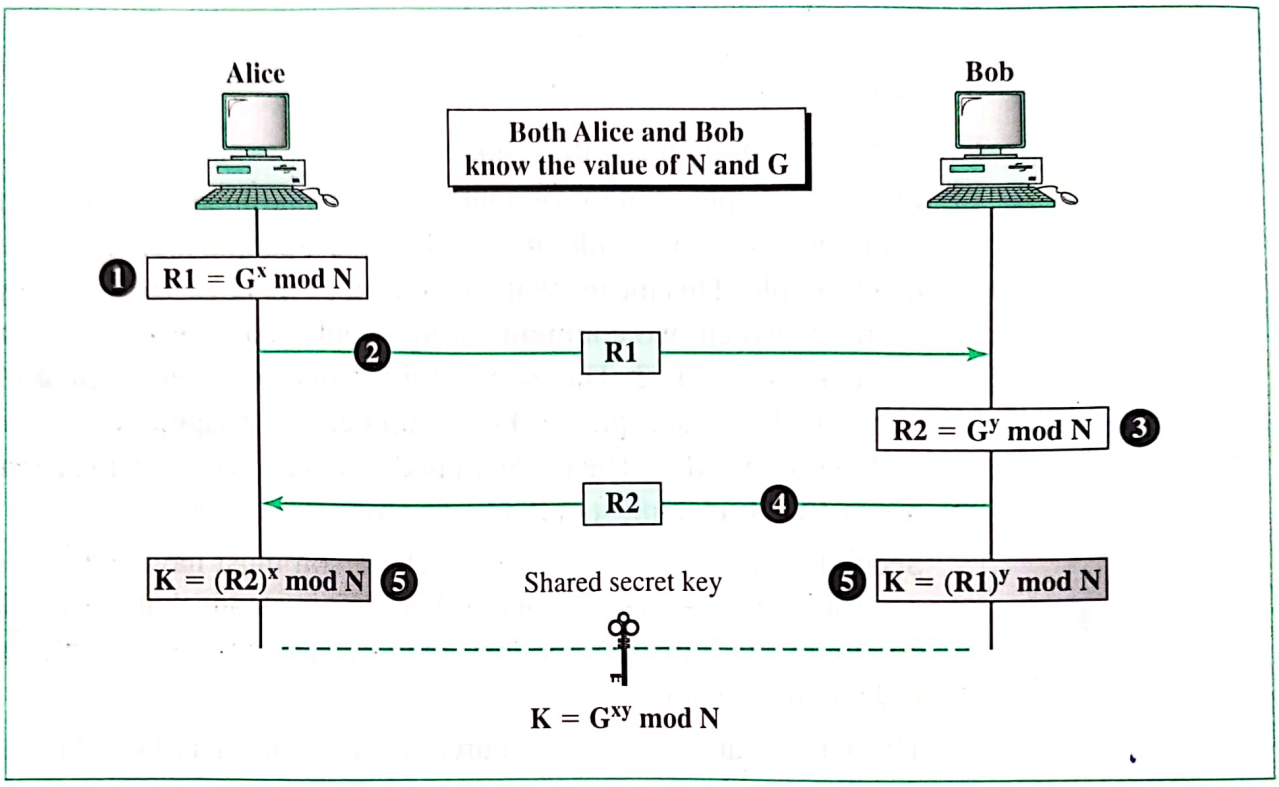
One protocol, the **Diffie-Hellman (DH) protocol,** devised by Diffie and Hellman, provides a one-time session key for two parties. The two parties use the session key to exchange data without having to remember or store it for future use. The parties do not have to meet to agree on the key, it can be done through the Internet. Let us see how the protocol works when Alice and Bob need a symmetric key to communicate.

Prerequisite Before establishing a symmetric key, the two parties need to choose two numbers N and G. The first number, N, is a large prime number with restriction that (N – 1)/2 must also be a prime number; the second number G is also a prime number, but it has more restrictions. These two numbers need not be confidential. They can be sent through the Internet; they can be public. Any two numbers, selected properly,

can serve the entire world. There is no secrecy about these two numbers; both Alice and Bob know these magic numbers.

Procedure Figure 30.11 shows the procedure.

Figure 30.11 *Diffie-Hellman method*



The steps are as follows:

Step 1 Alice chooses a large random number x and calculates $R1 = G^x \bmod N$.

Step 2 Alice sends R1 to Bob. Note that Alice does not send the value of x; she only sends R1.

Step 3 Bob chooses another large number y and calculates $R2 = G^y \bmod N$.

Step 4 Bob sends R2 to Alice. Again, note that Bob does not send the value of y; he only sends R2.

Step 5 Alice calculates $K = (R2)^x \bmod N$. Bob also calculates $K = (R1)^y \bmod N$. And K is the symmetric key for the session.

The reader may wonder why the value of K is the same since the calculations are different. The answer is an equality proved in number theory.

$$(G^x \bmod N)^y \bmod N = (G^y \bmod N)^x \bmod N = G^{xy} \bmod N$$

Bob has calculated $K = (R1)^y \bmod N = (G^x \bmod N)^y \bmod N = G^{xy} \bmod N$. Alice has calculated $K = (R2)^x \bmod N = (G^y \bmod N)^x \bmod N = G^{xy} \bmod N$. Both have reached the same value without Bob knowing the value of x or Alice knowing the value of y.

The symmetric (shared) key in the Diffie-Hellman protocol is $K = G^{xy} \bmod N$.

Example 1

Let us give an example to make the procedure clear. Our example uses small numbers, but note that in a real situation, the numbers are very large. Assume G = 7 and N = 23. The steps are as follows:

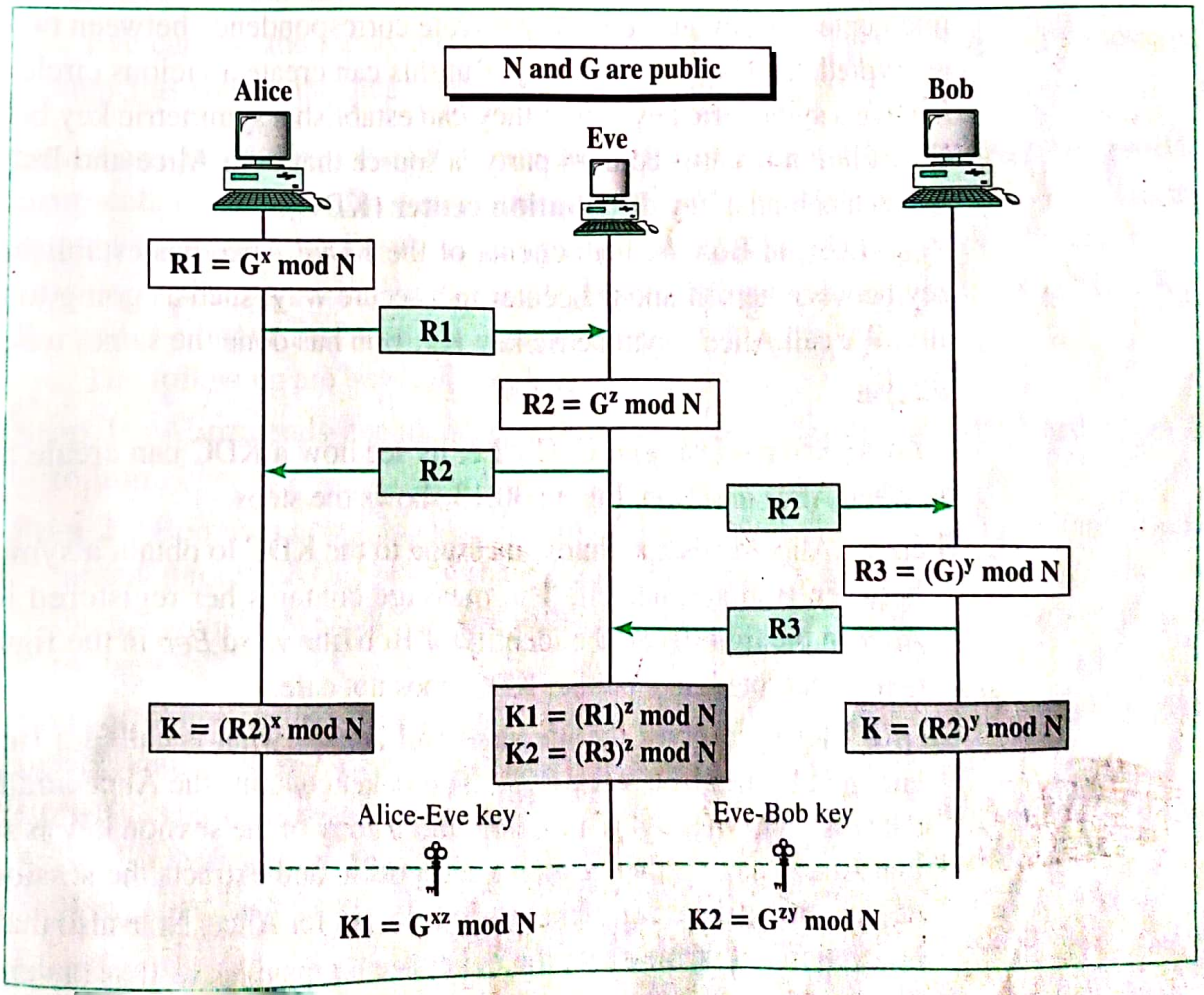
1. Alice chooses x = 3 and calculates R1 = 7^3 mod 23 = 21.
2. Alice sends the number 21 to Bob.
3. Bob chooses y = 6 and calculates R2 = 7^6 mod 23 = 4.
4. Bob sends the number 4 to Alice.
5. Alice calculates the symmetric key K = 4^3 mod 23 = 18.
6. Bob calculates the symmetric key K = 21^6 mod 23 = 18.

The value of K is the same for both Alice and Bob; G^{xy} mod N = 7^{18} mod 23 = 18.

Man-in-the-Middle Attack The Diffie-Hellman protocol is a very sophisticated symmetric-key creation algorithm. If x and y are very large numbers, it is extremely difficult for Eve to find the key knowing only N and G. An intruder needs to determine x and y if R1 and R2 are intercepted. But finding x from R1 and y from R2 are two difficult tasks. Even a sophisticated computer would need perhaps a long time to find the key by trying different numbers. In addition, Alice and Bob change the key the next time they need to communicate.

However, the protocol does have a weakness. Eve does not have to find the values of x and y to attack the protocol. She can fool Alice and Bob by creating two keys: one between herself and Alice and another between herself and Bob. Figure 30.12 shows the situation.

Figure 30.12 *Man-in-the-middle attack*



The following can happen:

1. Alice chooses x, calculates $R1 = G^x \bmod N$, and sends R1 to Bob.
2. Eve, the intruder, intercepts R1. She chooses z, calculates $R2 = G^z \bmod N$, and sends R2 to both Alice and Bob.
3. Bob chooses y, calculates $R3 = G^y \bmod N$, and sends R3 to Alice. R3 is intercepted by Eve and never reaches Alice.
4. Alice and Eve calculate $K1 = G^{xz} \bmod N$, which becomes a shared key between Alice and Eve. Alice, however, thinks that it is a key shared between Bob and herself.
5. Eve and Bob calculate $K2 = G^{zy} \bmod N$, which becomes a shared key between Eve and Bob. Bob, however, thinks that it is a key shared between Alice and himself.

In other words, two keys, instead of one, are created: one between Alice and Eve and one between Eve and Bob. When Alice sends data to Bob encrypted with K1 (shared by Alice and Eve), the data can be deciphered and read by Eve. Eve can send the message to Bob encrypted by K2 (shared key between Eve and Bob); or she can even change the message or send a totally new message. Bob is fooled into believing that the message has come from Alice. The same scenario can happen to Alice in the other direction.

This situation is called a **man-in-the-middle attack** because Eve comes in between and intercepts R1, sent by Alice to Bob, and R3, sent by Bob to Alice. It is also known as a bucket brigade attack because it resembles a short line of volunteers passing a bucket of water from person to person.

Key Distribution Center (KDC)

The flaw in the previous protocol is the sending of R1 and R2 as plaintext which can be intercepted by any intruder. Any private correspondence between two parties should be encrypted using a symmetric key. But this can create a vicious circle. Two parties need to have a symmetric key before they can establish a symmetric key between themselves. The solution is a trusted third party, a source that both Alice and Bob can trust. This is the idea behind a **key distribution center (KDC).**

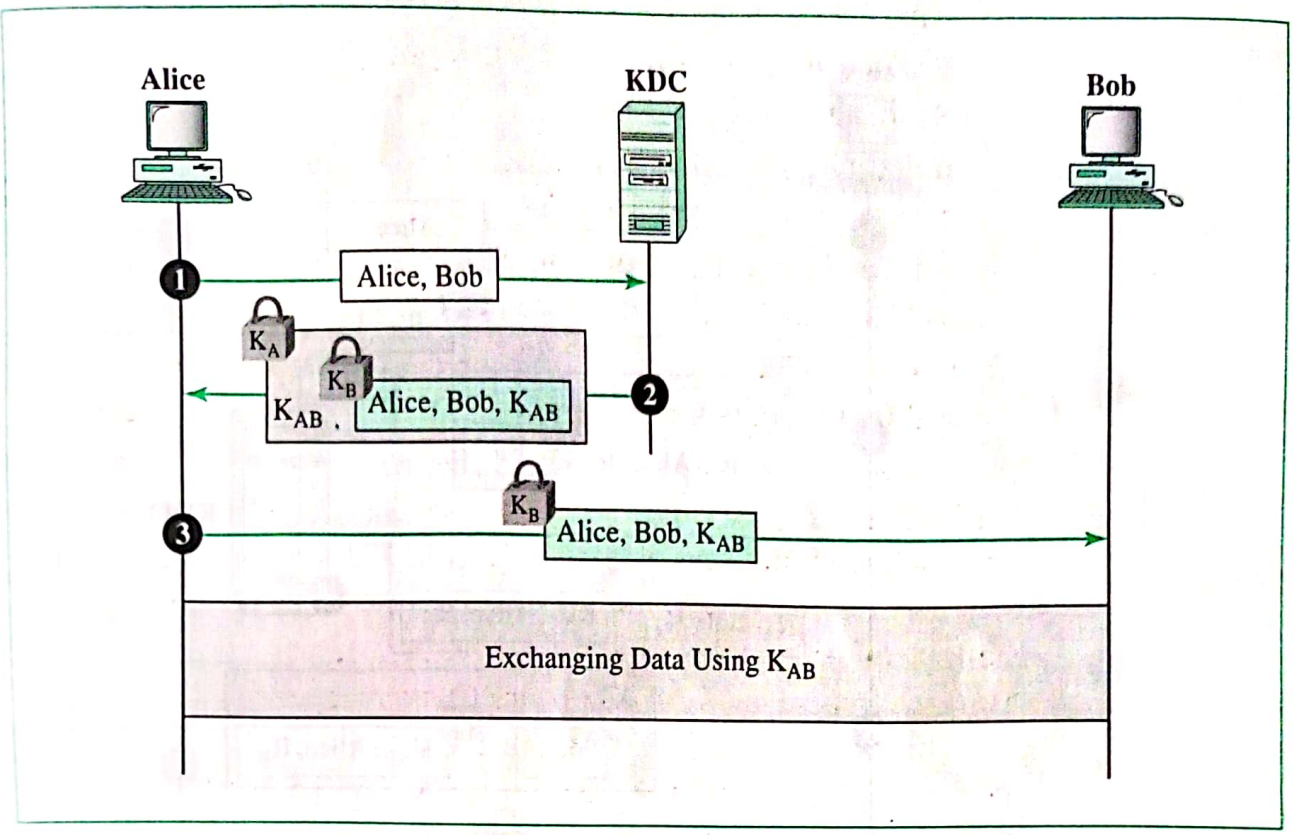
Alice and Bob are both clients of the KDC. Alice has established one symmetric key between herself and the center in a secure way, such as going to the center personally. We call Alice's symmetric key K_A. Bob has done the same; we call his symmetric key K_B.

First Approach Using a KDC Let us see how a KDC can create a session key K_{AB} between Alice and Bob. Figure 30.13 shows the steps.

Step 1 Alice sends a plaintext message to the KDC to obtain a symmetric session key between Bob and herself. The message contains her registered identity (the word *Alice* in the figure) and the identity of Bob (the word *Bob* in the figure). This message is not encrypted; it is public. KDC does not care.

Step 2 KDC receives the message and creates what is called a **ticket.** The ticket is encrypted using Bob's key (K_B). The ticket contains the Alice and Bob identities and the session key (K_{AB}). The ticket with a copy of the session key is sent to Alice. Note that Alice receives the message, decrypts it, and extracts the session key. She cannot decrypt Bob's ticket; the ticket is for Bob, not for Alice. Note also that we have a double encryption in this message; the ticket is encrypted, as well as the entire message.

Figure 30.13 First approach using KDC



Step 3 Alice sends the ticket to Bob. Bob opens the ticket and knows that Alice needs to send messages to him using K_{AB} as the session key.

Sending data. After the third step, Alice and Bob can exchange data using K_{AB} as a one-time session key.

Eve can use the replay attack we discussed previously. She can save the message in step 3 as well as the data messages and replay all.

Needham-Schroeder Protocol Another approach is the elegant **Needham-Schroeder protocol,** a foundation for many other protocols. This protocol uses multiple challenge-response interactions between parties to achieve a flawless protocol. In the latest version of this protocol, Needham and Schroeder use four different nonces: R_A, R_B, R_1, and R_2. Figure 30.14 shows the seven steps of this protocol.

The following are brief descriptions of each step:

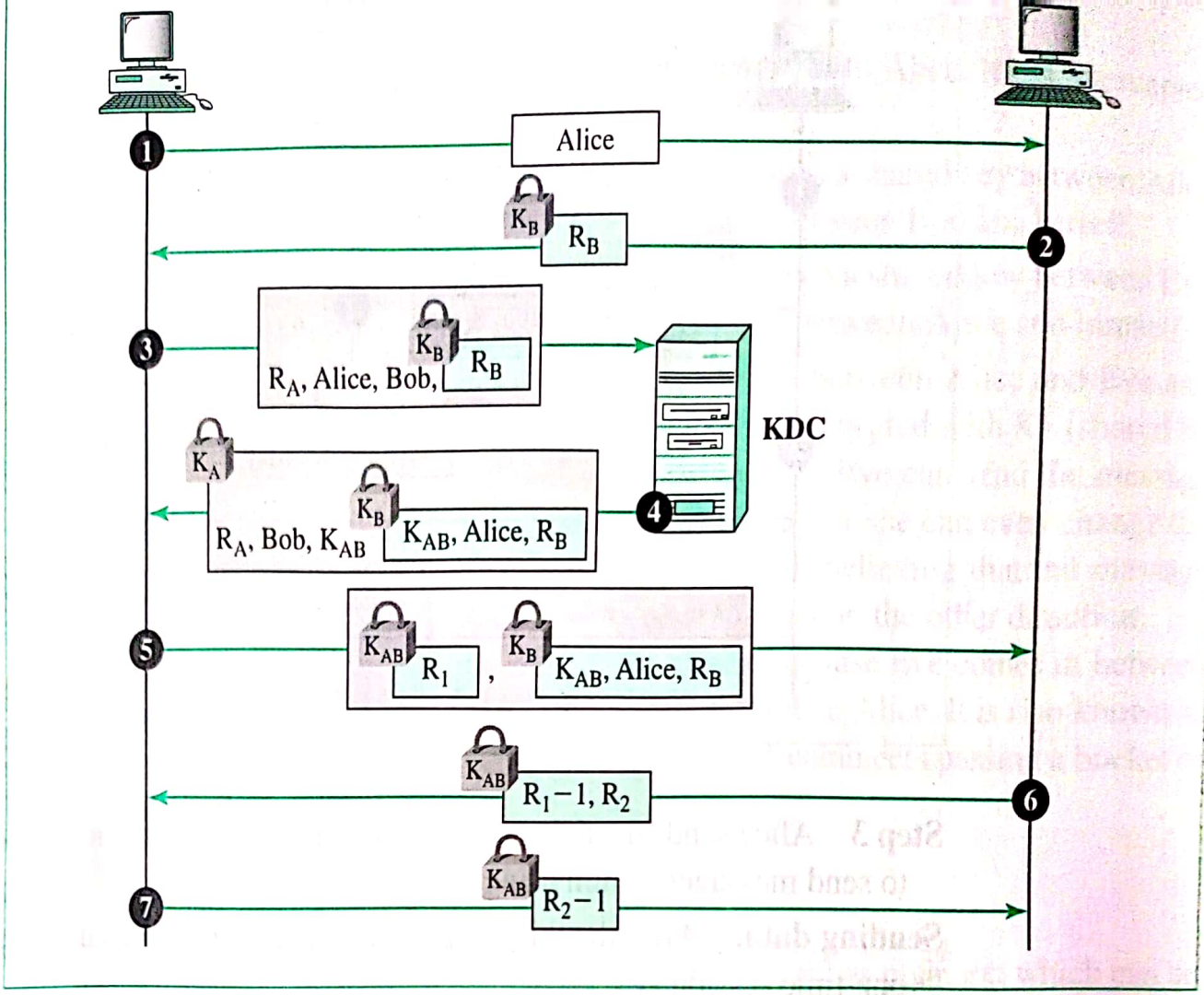
Step 1 Alice sends her identity to Bob, thereby declaring that she needs to talk to him.

Step 2 Bob uses nonce R_B and encrypts it with his symmetric key K_B. Nonce R_B is intended for the KDC, but it is sent to Alice. Alice sends R_B to the KDC to prove that the person who has talked to Bob is the same person (not an imposter) who will talk to the KDC.

Step 3 Alice sends a message to the KDC that includes her nonce, R_A, her identity, Bob's identity, and the encrypted nonce from Bob.

Step 4 The KDC sends an encrypted message to Alice that includes Alice's nonce, Bob's identity, the session key, and an encrypted ticket for Bob that includes his nonce. Now Alice has received the response to her nonce challenge and the session key.

Step 5 Alice sends Bob's ticket to him along with a new nonce, R_1, to challenge him.



Step 6 Bob responds to Alice's challenge and sends his challenge to Alice (R2). Note that the response to Alice's challenge is the value $R_1 - 1$; this ensures that Bob has decrypted the encrypted R_1. In other words, the new encryption ensures that an imposter has not sent the exact encrypted message back.

Step 7 Alice responds to Bob's challenge. Again, note that the response carries $R_2 - 1$ instead of R_2.

Otway-Rees Protocol A third approach is the **Otway-Rees protocol,** another elegant protocol, that has even fewer steps. Figure 30.15 shows this five-step protocol. The following briefly describes the steps.

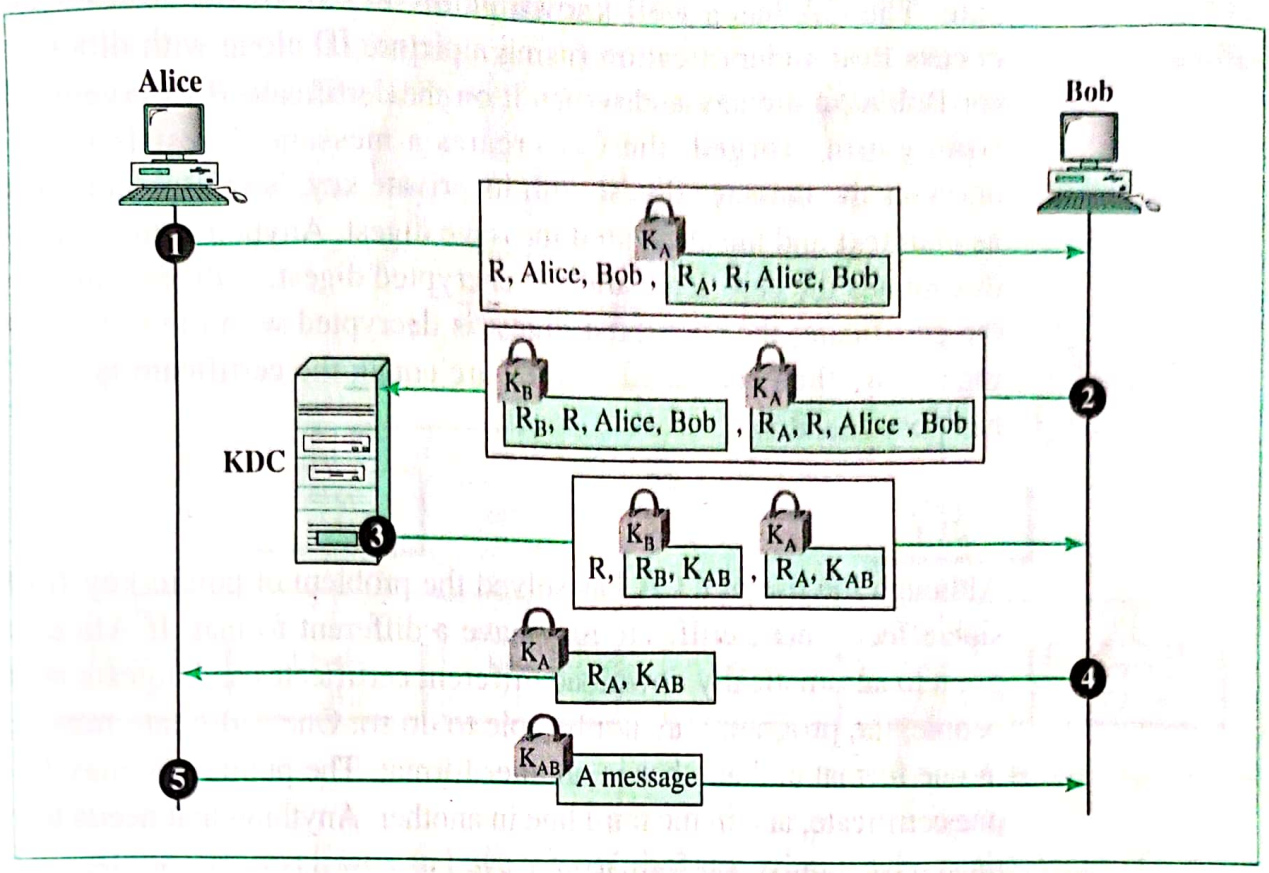
Step 1 Alice sends a message to Bob that includes a common nonce R, the identities of Alice and Bob, and a ticket for KDC that includes Alice's nonce R_A (a challenge for KDC to use), a copy of the common nonce R, and the identities of Alice and Bob.

Step 2 Bob creates the same type of ticket, but with his own nonce R_B; both tickets are sent to KDC.

Step 3 KDC creates a message that contains R, the common nonce, a ticket for Alice and a ticket for Bob; the message is sent to Bob. The tickets contain the corresponding nonce, R_A or R_B, and the session key K_{AB}.

Step 4 Bob sends Alice her ticket.

Step 5 Alice sends a message encrypted with her session key K_{AB}.



Public-Key Certification

In public-key cryptography, people do not need to know a symmetric shared key. If Alice wants to send a message to Bob, she only needs to know Bob's public key, which is open to the public and available to everyone. If Bob needs to send a message to Alice, he only needs to know Alice's public key, which is also known to everyone. In public-key cryptography, everyone shields a private key and advertises a public key.

In public-key cryptography, everyone has access to everyone's public key.

The Problem

In public-key cryptography, everybody who expects to receive a message from someone else needs to somehow advertise his or her public key to the sender of the message. The problem is how to advertise the public key and make it safe from Eve's interference. If Bob sends his public key to Alice, Eve may intercept it and send her (Eve's) own public key to Alice. Alice, assuming that this is Bob's public key, encrypts a message for Bob with this key and sends it to Bob. Eve again intercepts and decrypts the message with her private key and knows what Alice has sent to Bob. Eve can even put her public key online and claim that this is Bob's public key.

Certification Authority

Bob wants two things: He wants people to know his public key, and he wants no one to accept a public key forged as Bob's. Bob can go to a **certification authority (CA)**, a

federal or state organization that binds a public key to an entity and issues a certificate. The CA has a well-known public key itself that cannot be forged. The CA checks Bob's identification (using a picture ID along with other proof). It then asks for Bob's public key and writes it on the certificate. To prevent the certificate itself from getting forged, the CA creates a message digest from the certificate and encrypts the message digest with its private key. Now Bob can upload the certificate as plaintext and the encrypted message digest. Anybody who wants Bob's public key downloads the certificate and the encrypted digest. A digest can then be created from the certificate; the encrypted digest is decrypted with the CA's public key. The two digests are then compared. If they are equal, the certificate is valid and no imposter has posed as Bob.

X.509

Although the use of a CA has solved the problem of public-key fraud, it has created a side effect. Each certificate may have a different format. If Alice wants to use a program to automatically download different certificates and digests belonging to different people, the program may not be able to do so. One certificate may have the public key in one format and another in another format. The public key may be in the first line in one certificate, and in the third line in another. Anything that needs to be used universally must have a universal format.

To remove this side effect, ITU has devised a protocol called **X.509,** which has been accepted by the Internet with some changes. Protocol X.509 is a way to describe the certificate in a structural way. It uses a well-known protocol called ASN.1 (Abstract Syntax Notation 1) that defines fields very familiar to C programmers.

We do not discuss ASN.1 here, but we list some of the fields and their meanings defined by X.509 in Table 30.1.

Table 30.1 *X.509 fields*

Field	*Explanation*
Version	Version number of X.509
Serial number	The unique identifier used by the CA
Signature	The certificate signature
Issuer	The name of the CA defined by X.509
Validity period	Start and end period that certificate is valid
Subject name	The entity whose public key is being certified
Public key	The subject public key and the algorithms that use it

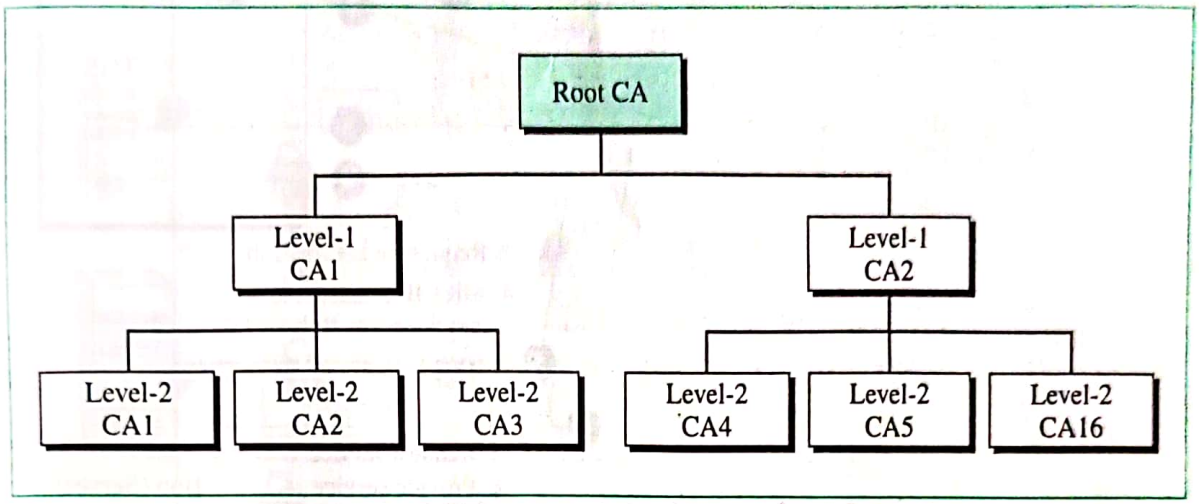
Public-Key Infrastructure (PKI)

When we want to use public keys universally, we have a problem similar to one concerning DNS (Domain Name System) in Chapter 25. We found that we cannot have only one DNS server to answer the queries. We need many servers. In addition, we found that the best solution is to put the servers in a hierarchical relationship. If Alice needs to get Bob's IP address, Alice sends a message to her local server that may or

may not have Bob's IP address. The local server can consult its parent server, up to the root, until the IP address is found.

Likewise, a solution to public-key queries is a hierarchical structure called a **public-key infrastructure (PKI).** Figure 30.16 shows an example of this hierarchy.

Figure 30.16 *PKI hierarchy*



At the first level, we can have a root CA that can certify the performance of CAs in the second level; these level-1 CAs may operate in a large geographic area or logical area. The level-2 CAs may operate in smaller geographic areas.

In this hierarchy, everybody trusts the root. But people may or may not trust intermediate CAs. If Alice needs to get Bob's certificate, she may find a CA somewhere to issue the certificate. But Alice may not trust that CA. In a hierarchy Alice can ask the next-higher CA to certify the original CA. The inquiry may go all the way to the root.

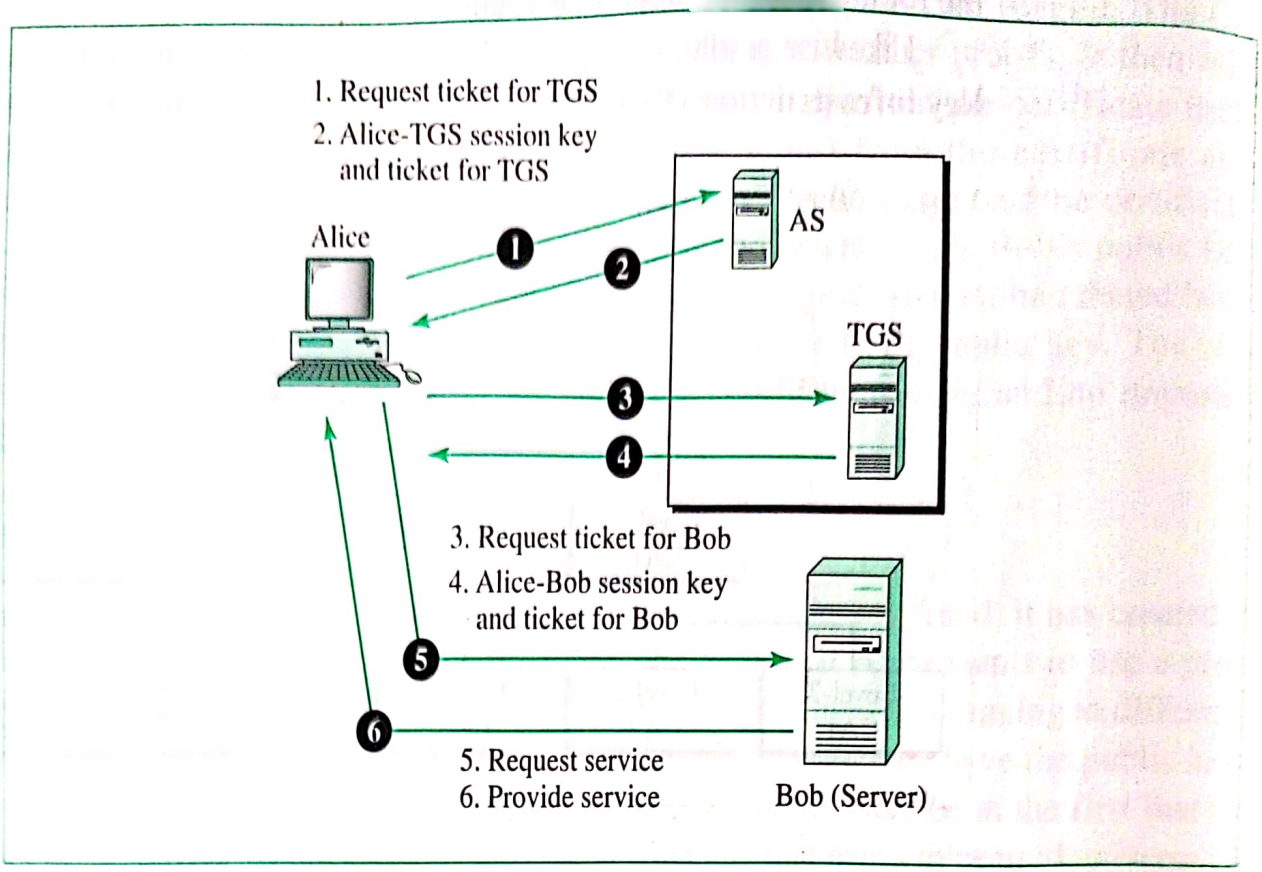
PKI is a new issue in the Internet. It will undoubtedly broaden in scope and change in the next few years.

30.5 KERBEROS

Kerberos is an authentication protocol, and at the same time a KDC, that has become very popular. Several systems including Windows 2000 use Kerberos. Kerberos is named after the three-headed dog in Greek mythology that guards the gates of Hades. Originally designed at MIT, it has gone through several versions. We discuss only version 4, the most popular, and we briefly explain the difference between version 4 and version 5, the latest.

Servers

Three servers are involved in the Kerberos protocol: an **authentication server (AS),** a **ticket-granting server (TGS),** and a real (data) server that provides services to others. In our examples and figures, *Bob* is the real server and *Alice* is the user requesting service. Figure 30.17 shows the relationship between these three servers.



Authentication Server (AS)

The AS is the KDC in the Kerberos protocol. Each user registers with the AS and is granted a user identity and a password. The AS has a database with these identities and the corresponding passwords. The AS verifies the user, issues a session key to be used between Alice and the TGS, and sends a ticket for the TGS.

Ticket-Granting Server (TGS)

The TGS issues a ticket for the real server (Bob). It also provides the session key (K_{AB}) between Alice and Bob. Kerberos has separated the user verification from ticket issuing. In this way, although Alice verifies her ID just once with AS, she can contact TGS multiple times to obtain tickets for different real servers.

Real Server

The real server (Bob) provides services for the user (Alice). Kerberos is designed for a client-server program such as FTP, in which a user uses the client process to access the server process. Kerberos is not used for person-to-person authentication.

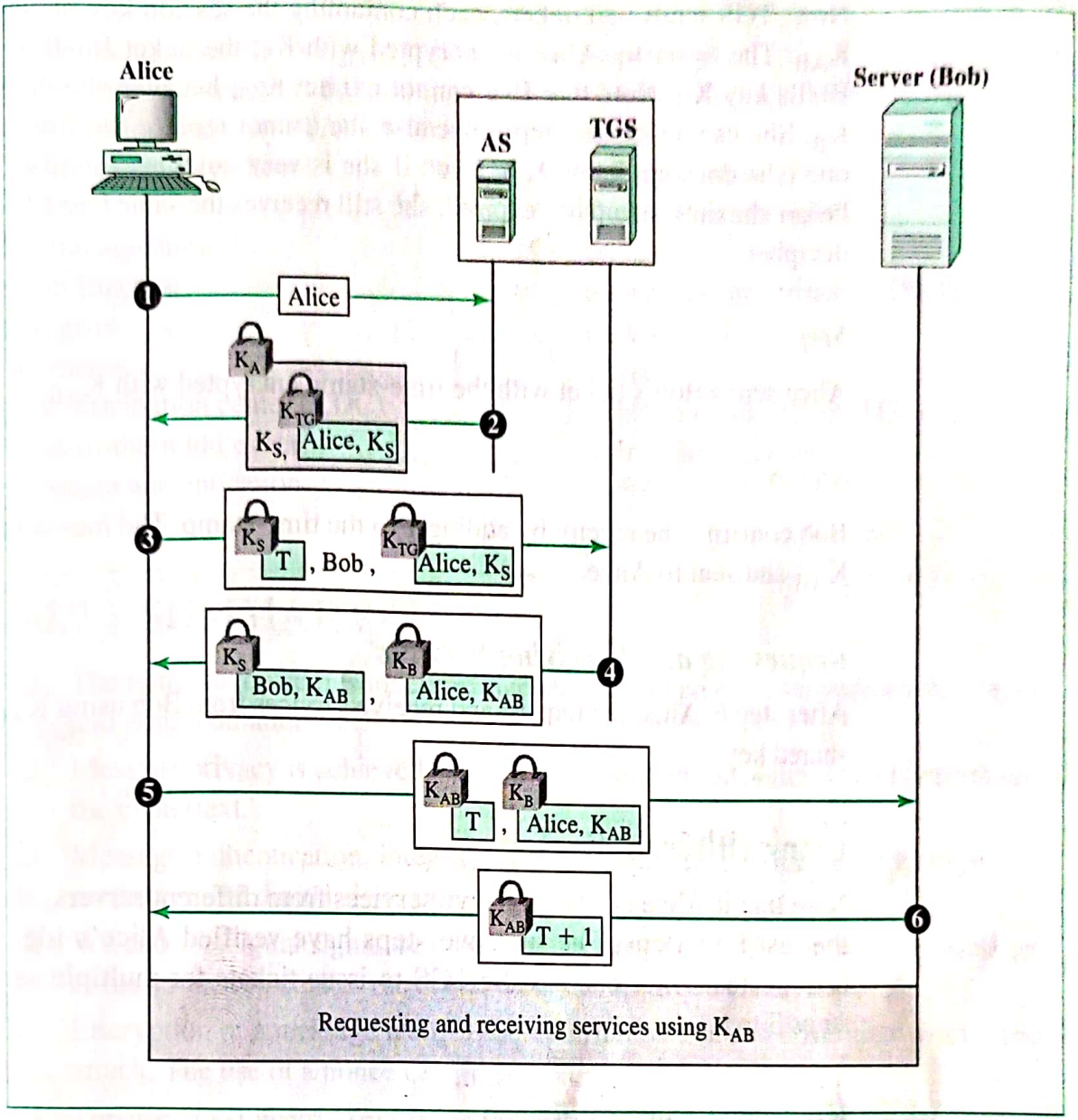
Operation

A client process (Alice) can receive a service from a process running on the real server (Bob) in six steps, as shown in Figure 30.18.

Step 1

Alice sends her request to AS in plaintext, using her registered identity.

Figure 30.18 *Kerberos example*



Step 2

The AS sends a message encrypted with Alice's symmetric key K_A. The message contains two items: a session key K_S that is used by Alice to contact TGS and a ticket for TGS that is encrypted with the TGS symmetric key K_{TG}. Alice does not know K_A, but when the message arrives, she types her password. The password and the appropriate algorithm together create K_A if the password is correct. The password is then immediately destroyed; it is not sent to the network, and it does not stay in the terminal. It is only used for a moment to create K_A. The process now uses K_A to decrypt the message sent; K_S and the ticket are extracted.

Step 3

Alice now sends three items to the TGS. The first is the ticket received from AS. The second is the name of the real server (Bob), and the third is a timestamp which is encrypted by K_S. The timestamp prevents a replay by Eve.

Step 4

Now, TGS sends two tickets, each containing the session key between Alice and Bob K_{AB}. The ticket for Alice is encrypted with K_S; the ticket for Bob is encrypted with Bob's key K_B. Note that Eve cannot extract K_{AB} because she does not know K_S or K_B. She cannot replay step 3 because she cannot replace the time-stamp with a new one (she does not know K_S). Even if she is very quick and sends the step 3 message before the time-stamp has expired, she still receives the same two tickets that she cannot decipher.

Step 5

Alice sends Bob's ticket with the time-stamp encrypted with K_{AB}.

Step 6

Bob confirms the receipt by adding 1 to the time-stamp. The message is encrypted with K_{AB} and sent to Alice.

Requesting and Receiving Services

After step 6, Alice can request and receive services from Bob using K_{AB} as the symmetric shared key.

Using Different Servers

Note that if Alice needs to receive services from different servers, she need repeat only the last four steps. The first two steps have verified Alice's identity and need not be repeated. Alice can ask the TGS to issue tickets for multiple servers by repeating steps 3 to 6.

Kerberos Version 5

The minor differences between version 4 and version 5 are briefly listed below:

1. Version 5 has a longer ticket lifetime.
2. Version 5 allows tickets to be renewed.
3. Version 5 can accept any symmetric-key algorithm.
4. Version 5 uses a different protocol for describing data types.
5. Version 5 has more overhead than version 4.

Realms

Kerberos allows the global distribution of ASs and TGSs, with each system called a realm. A user may get a ticket for a local server or a distant server. In the second case, for example, Alice may ask her local TGS to issue a ticket that is accepted by a distant TGS. The local TGS can issue this ticket if the distant TGS is registered with the local one. Then Alice can use the distant TGS to access the distant real server.