CHAPTER 15

Key Management

Objectives

This chapter has several objectives:

- ❏ To explain the need for a key-distribution center (KDC)
- ❏ To show how a KDC can create a session key between two parties
- ❏ To show how two parties can use a symmetric-key agreement protocol to create a session key between themselves without using the services of a KDC
- ❏ To describe Kerberos as a KDC and an authentication protocol
- ❏ To explain the need for certification authorities (CAs) for public keys and how X.509 recommendation defines the format of certificates
- ❏ To introduce the idea of a Public-Key Infrastructure (PKI) and explain some of its duties

Previous chapters have discussed symmetric-key and asymmetric-key cryptography. However, we have not yet discussed how secret keys in symmetric-key cryptography, and public keys in asymmetric-key cryptography, are distributed and maintained. This chapter touches on these two issues.

We first discuss the distribution of symmetric keys using a trusted third party. Second, we show how two parties can establish a symmetric key between themselves without using a trusted third party. Third, we introduce Kerberos as both a KDC and an authentication protocol. Fourth, we discuss the certification of public keys using certification authorities (CAs) based on the X.509 recommendation. Finally, we briefly discuss the idea of a Public-Key Infrastructure (PKI) and mention some of its duties.

15.1 SYMMETRIC-KEY DISTRIBUTION

Symmetric-key cryptography is more efficient than asymmetric-key cryptography for enciphering large messages. Symmetric-key cryptography, however, needs a shared secret key between two parties.

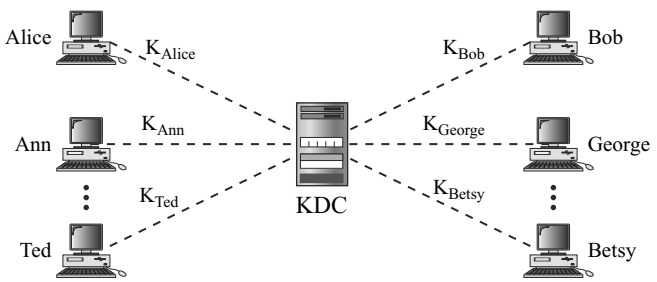
If Alice needs to exchange confidential messages with N people, she needs N different keys. What if N people need to communicate with each other? A total of $N(N-1)$ keys is needed if we require that Alice and Bob use two keys for bidirectional communication; only $N(N-1)/2$ keys are needed if we allow a key to be used for both directions. This means that if one million people need to communicate with each other, each person has almost one million different keys; in total, almost one trillion keys are needed. This is normally referred to as the N^2 problem because the number of required keys for N entities is N^2.

The number of keys is not the only problem; the distribution of keys is another. If Alice and Bob want to communicate, they need a way to exchange a secret key; if Alice wants to communicate with one million people, how can she exchange one million keys with one million people? Using the Internet is definitely not a secure method. It is obvious that we need an efficient way to maintain and distribute secret keys.

Key-Distribution Center: KDC

A practical solution is the use of a trusted third party, referred to as a **key-distribution center (KDC)**. To reduce the number of keys, each person establishes a shared secret key with the KDC, as shown in Figure 15.1.

Figure 15.1 *Key-distribution center (KDC)*



A secret key is established between the KDC and each member. Alice has a secret key with the KDC, which we refer to as K_{Alice}; Bob has a secret key with the KDC, which we refer to as K_{Bob}; and so on. Now the question is how Alice can send a confidential message to Bob. The process is as follows:

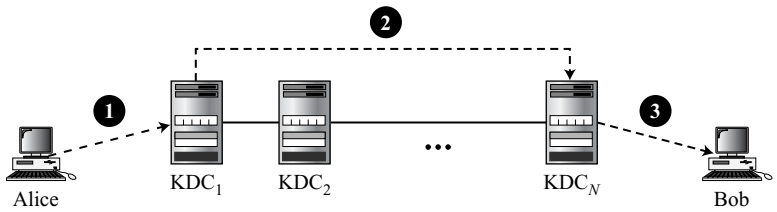
1. Alice sends a request to the KDC stating that she needs a session (temporary) secret key between herself and Bob.
2. The KDC informs Bob about Alice's request.
3. If Bob agrees, a session key is created between the two.

The secret key between Alice and Bob that is established with the KDC is used to authenticate Alice and Bob to the KDC and to prevent Eve from impersonating either of them. We discuss how a session key is established between Alice and Bob later in the chapter.

Flat Multiple KDCs

When the number of people using a KDC increases, the system becomes unmanageable and a bottleneck can result. To solve the problem, we need to have multiple KDCs. We can divide the world into domains. Each domain can have one or more KDCs (for redundancy in case of failure). Now if Alice wants to send a confidential message to Bob, who belongs to another domain, Alice contacts her KDC, which in turn contacts the KDC in Bob's domain. The two KDCs can create a secret key between Alice and Bob. Figure 15.2 shows KDCs all at the same level. We call this flat multiple KDCs.

Figure 15.2 *Flat multiple KDCs*



Hierarchical Multiple KDCs

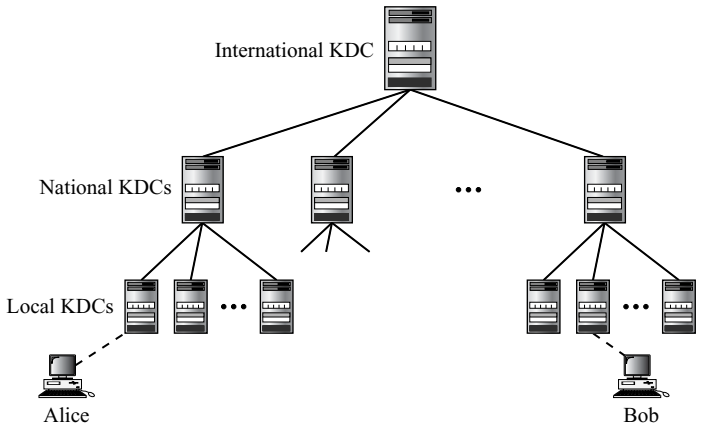
The concept of flat multiple KDCs can be extended to a hierarchical system of KDCs, with one or more KDCs at the top of the hierarchy. For example, there can be local KDCs, national KDCs, and international KDCs. When Alice needs to communicate with Bob, who lives in another country, she sends her request to a local KDC; the local KDC relays the request to the national KDC; the national KDC relays the request to an international KDC. The request is then relayed all the way down to the local KDC where Bob lives. Figure 15.3 shows a configuration of hierarchical multiple KDCs.

Session Keys

A KDC creates a secret key for each member. This secret key can be used only between the member and the KDC, not between two members. If Alice needs to communicate secretly with Bob, she needs a secret key between herself and Bob. A KDC can create a **session key** between Alice and Bob, using their keys with the center. The keys of Alice and Bob are used to authenticate Alice and Bob to the center and to each other before the session key is established. After communication is terminated, the session key is no longer useful.

A session symmetric key between two parties is used only once.

Figure 15.3 *Hierarchical multiple KDCs*

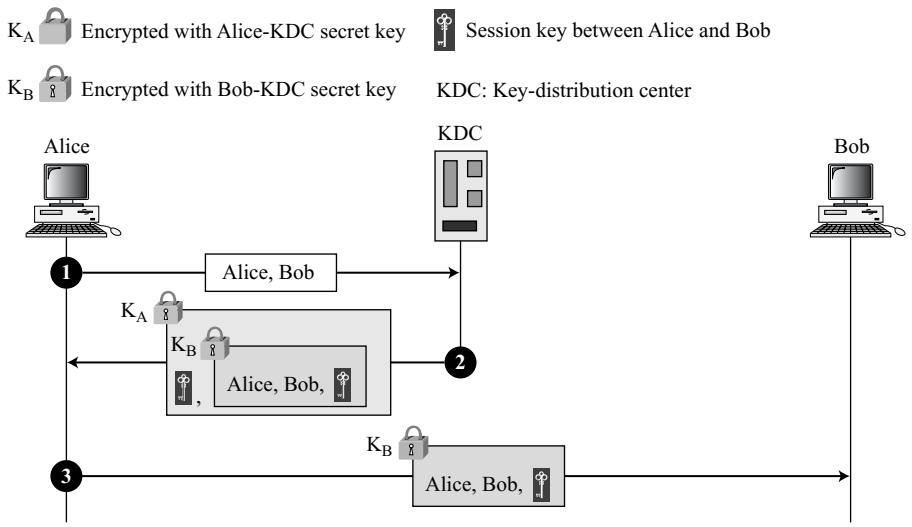


Several different approaches have been proposed to create the session key using ideas discussed in Chapter 14 for entity authentication.

A Simple Protocol Using a KDC

Let us see how a KDC can create a session key K_{AB} between Alice and Bob. Figure 15.4 shows the steps.

Figure 15.4 *First approach using KDC*



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. The KDC does not care.
2. The 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 identities of Alice and Bob and the session key (K_{AB}). The ticket with a copy of the session key is sent to Alice. 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 that this message contains a double encryption; the ticket is encrypted, and the entire message is also encrypted. In the second message, Alice is actually authenticated to the KDC, because only Alice can open the whole message using her secret key with KDC.
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. Note that in this message, Bob is authenticated to the KDC because only Bob can open the ticket. Because Bob is authenticated to the KDC, he is also authenticated to Alice, who trusts the KDC. In the same way, Alice is also authenticated to Bob, because Bob trusts the KDC and the KDC has sent Bob the ticket that includes the identity of Alice.

Unfortunately, this simple protocol has a flaw. Eve can use the replay attack discussed previously. That is, she can save the message in step 3 and replay it later.

Needham-Schroeder Protocol

Another approach is the elegant **Needham-Schroeder protocol,** which is a foundation for many other protocols. This protocol uses multiple challenge-response interactions between parties to achieve a flawless protocol. Needham and Schroeder uses two nonces: R_A and R_B. Figure 15.5 shows the five steps used in this protocol.

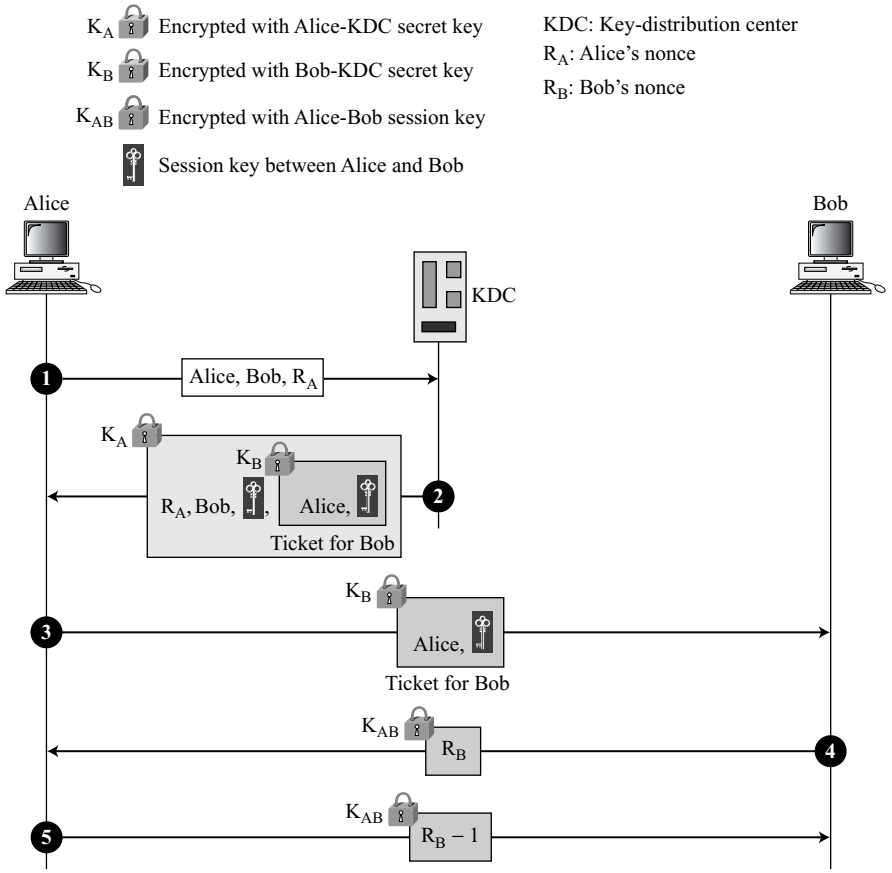
We briefly describe each step:

1. Alice sends a message to the KDC that includes her nonce, R_A, her identity, and Bob's identity.
2. 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. The whole message is encrypted with Alice's key.
3. Alice sends Bob's ticket to him.
4. Bob sends his challenge to Alice (R_B), encrypted with the session key.
5. Alice responds to Bob's challenge. Note that the response carries $R_B - 1$ instead of R_B.

Otway-Rees Protocol

A third approach is the **Otway-Rees protocol,** another elegant protocol. Figure 15.6 shows this five-step protocol.

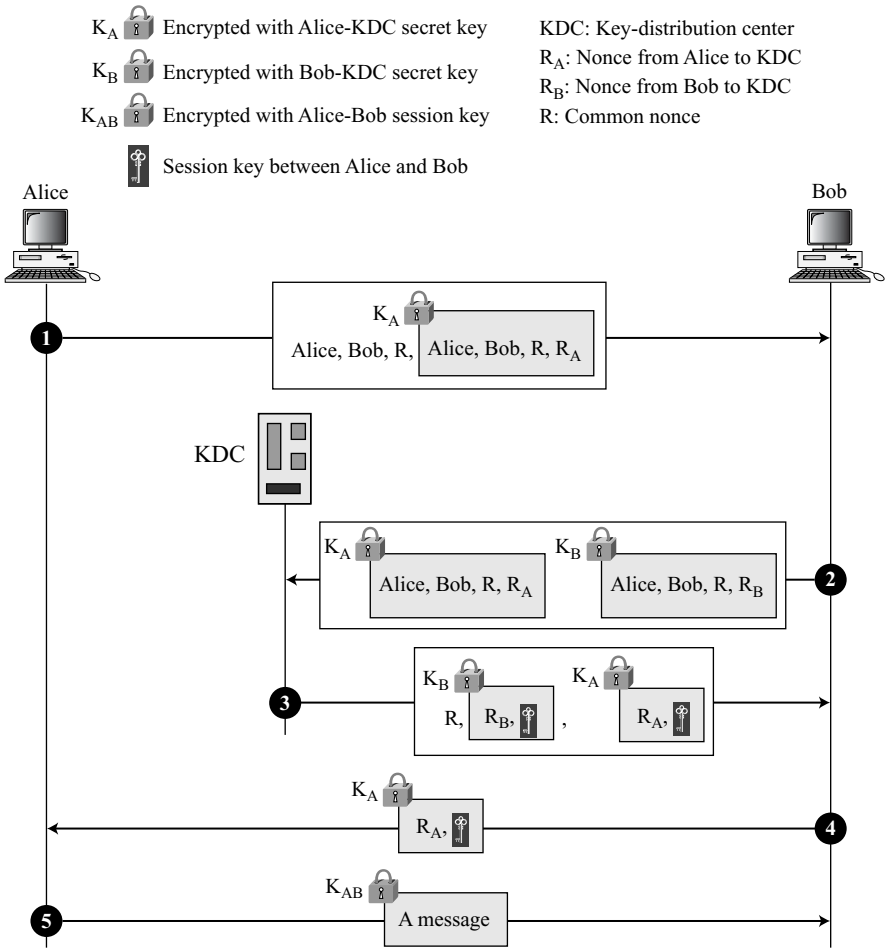
Figure 15.5 *Needham-Schroeder protocol*



The following briefly describes the steps.

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 the KDC to use), a copy of the common nonce, R, and the identities of Alice and Bob.
2. Bob creates the same type of ticket, but with his own nonce R_B. Both tickets are sent to the KDC.
3. The 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}.
4. Bob sends Alice her ticket.
5. Alice sends a short message encrypted with her session key K_{AB} to show that she has the session key.

Figure 15.6 *Otway-Rees protocol*



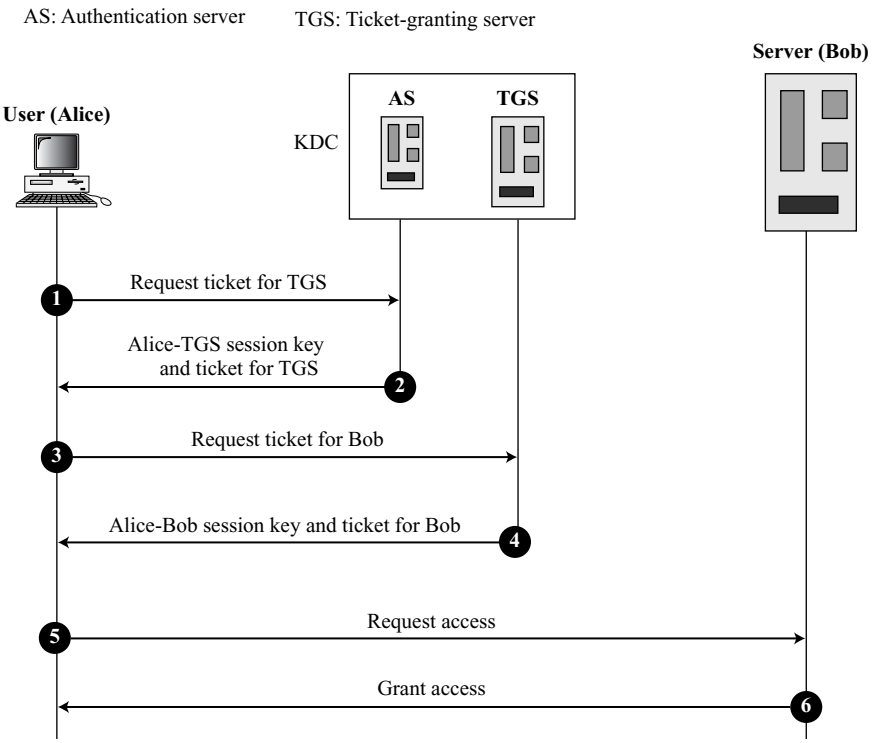
15.2 KERBEROS

Kerberos is an authentication protocol, and at the same time a KDC, that has become very popular. Several systems, including Windows 2000, use Kerberos. It 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 only discuss 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 15.7 shows the relationship between these three servers.

Figure 15.7 *Kerberos servers*



Authentication Server (AS)

The **authentication server (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 **ticket-granting server (TGS)** issues a ticket for the real server (Bob). It also provides the session key (K_{AB}) between Alice and Bob. Kerberos has separated user

verification from the issuing of tickets. In this way, though Alice verifies her ID just once with the AS, she can contact the 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

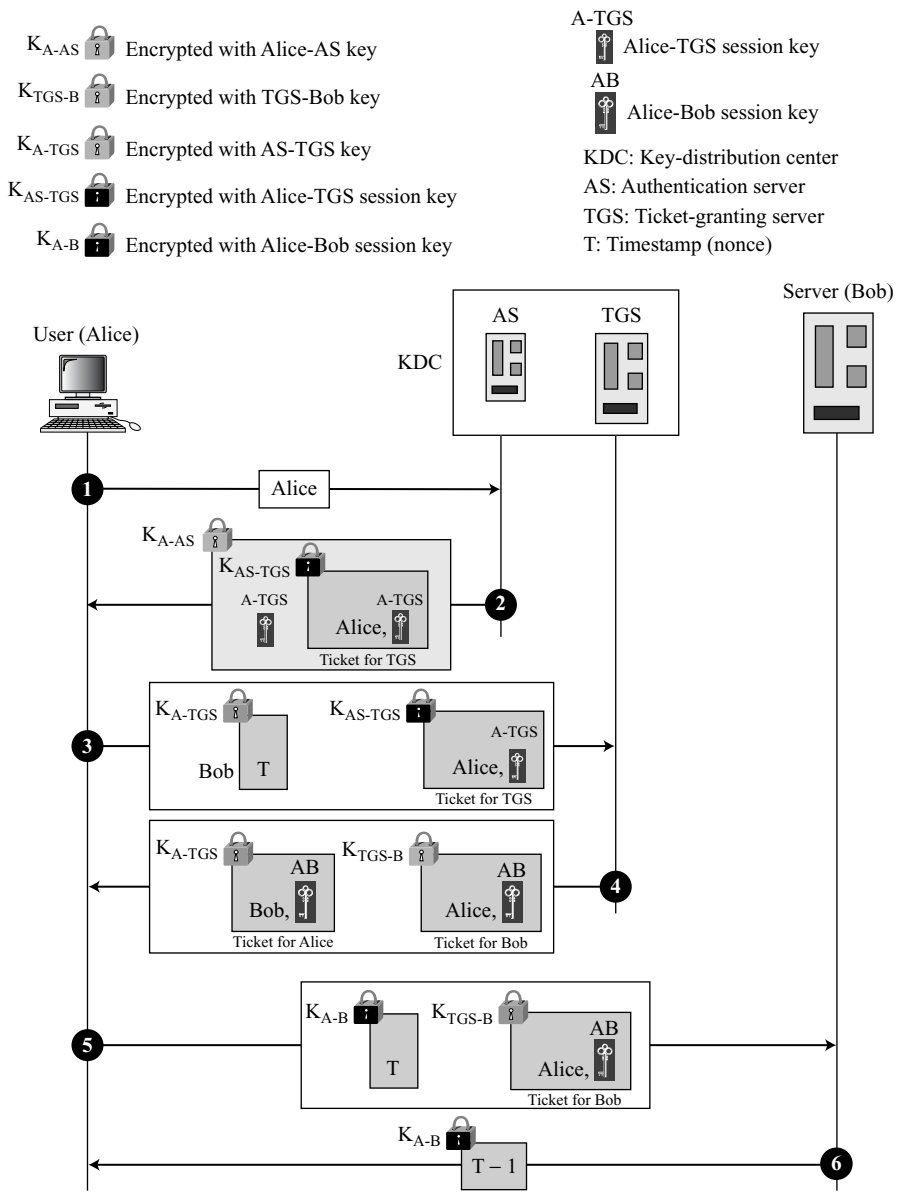
A client process (Alice) can access a process running on the real server (Bob) in six steps, as shown in Figure 15.8.

1. Alice sends her request to the AS in plain text using her registered identity.
2. The AS sends a message encrypted with Alice's permanent symmetric key, $K_{A\text{-}AS}$. The message contains two items: a session key, $K_{A\text{-}TGS}$, that is used by Alice to contact the TGS, and a ticket for the TGS that is encrypted with the TGS symmetric key, $K_{AS\text{-}TGS}$. Alice does not know $K_{A\text{-}AS}$, but when the message arrives, she types her symmetric password. The password and the appropriate algorithm together create $K_{A\text{-}AS}$ 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 used only for a moment to create $K_{A\text{-}AS}$. The process now uses $K_{A\text{-}AS}$ to decrypt the message sent. $K_{A\text{-}TGS}$ and the ticket are extracted.
3. Alice now sends three items to the TGS. The first is the ticket received from the AS. The second is the name of the real server (Bob), the third is a timestamp that is encrypted by $K_{A\text{-}TGS}$. The timestamp prevents a replay by Eve.
4. Now, the TGS sends two tickets, each containing the session key between Alice and Bob, $K_{A\text{-}B}$. The ticket for Alice is encrypted with $K_{A\text{-}TGS}$; the ticket for Bob is encrypted with Bob's key, $K_{TGS\text{-}B}$. Note that Eve cannot extract K_{AB} because Eve does not know $K_{A\text{-}TGS}$ or $K_{TGS\text{-}B}$. She cannot replay step 3 because she cannot replace the timestamp with a new one (she does not know $K_{A\text{-}TGS}$). Even if she is very quick and sends the step 3 message before the timestamp has expired, she still receives the same two tickets that she cannot decipher.
5. Alice sends Bob's ticket with the timestamp encrypted by $K_{A\text{-}B}$.
6. Bob confirms the receipt by adding 1 to the timestamp. The message is encrypted with $K_{A\text{-}B}$ and sent to Alice.

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 TGS to issue tickets for multiple servers by repeating steps 3 to 6.

Figure 15.8 *Kerberos example*



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 remote server. In the second case, for example, Alice may ask her local TGS to issue a ticket that is accepted by a remote TGS. The local TGS can issue this ticket if the remote TGS is registered with the local one. Then Alice can use the remote TGS to access the remote real server.

15.3 SYMMETRIC-KEY AGREEMENT

Alice and Bob can create a session key between themselves without using a KDC. This method of session-key creation is referred to as the symmetric-key agreement. Although there are several ways to accomplish this, only two common methods, Diffie-Hellman and station-to-station, are discussed here.

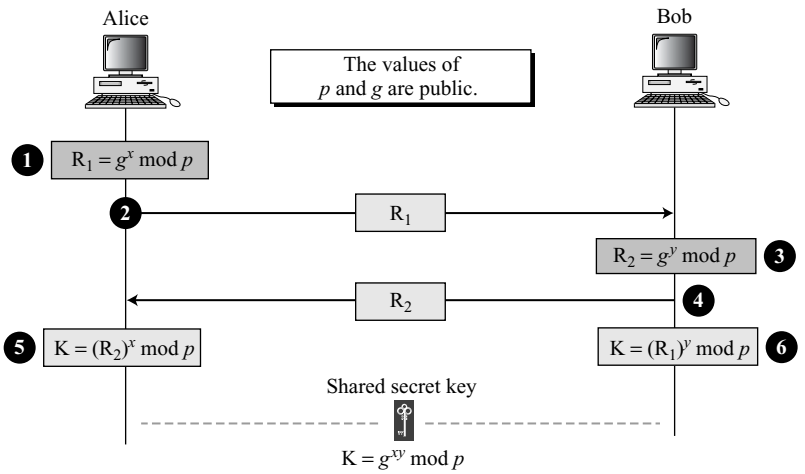
Diffie-Hellman Key Agreement

In the **Diffie-Hellman protocol** two parties create a symmetric session key without the need of a KDC. Before establishing a symmetric key, the two parties need to choose two numbers p and g. The first number, p, is a large prime number on the order of 300 decimal digits (1024 bits). The second number, g, is a generator of order $p - 1$ in the group $<\mathbf{Z}_{p^*}, \times>$. These two (group and generator) do not need to be confidential. They can be sent through the Internet; they can be public. Figure 15.9 shows the procedure.

The steps are as follows:

1. Alice chooses a large random number x such that $0 \le x \le p - 1$ and calculates $R_1 = g^x \bmod p$.
2. Bob chooses another large random number y such that $0 \le y \le p - 1$ and calculates $R_2 = g^y \bmod p$.
3. Alice sends R_1 to Bob. Note that Alice does not send the value of x; she sends only R_1.
4. Bob sends R_2 to Alice. Again, note that Bob does not send the value of y, he sends only R_2.
5. Alice calculates $K = (R_2)^x \bmod p$.
6. Bob also calculates $K = (R_1)^y \bmod p$.

Figure 15.9 *Diffie-Hellman method*



K is the symmetric key for the session.

$$K = (g^x \bmod p)^y \bmod p = (g^y \bmod p)^x \bmod p = g^{xy} \bmod p$$

Bob has calculated $K = (R_1)^y \bmod p = (g^x \bmod p)^y \bmod p = g^{xy} \bmod p$. Alice has calculated $K = (R_2)^x \bmod p = (g^y \bmod p)^x \bmod = g^{xy} \bmod p$. Both have reached the same value without Bob knowing the value of x and without Alice knowing the value of y.

The symmetric (shared) key in the Diffie-Hellman method is $K = g^{xy} \bmod p$.

Example 15.1

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

1. Alice chooses $x = 3$ and calculates $R_1 = 7^3 \bmod 23 = 21$.
2. Bob chooses $y = 6$ and calculates $R_2 = 7^6 \bmod 23 = 4$.
3. Alice sends the number 21 to Bob.
4. Bob sends the number 4 to Alice.
5. Alice calculates the symmetric key $K = 4^3 \bmod 23 = 18$.
6. Bob calculates the symmetric key $K = 21^6 \bmod 23 = 18$.

The value of K is the same for both Alice and Bob; $g^{xy} \bmod p = 7^{18} \bmod 35 = 18$.

Example 15.2

Let us give a more realistic example. We used a program to create a random integer of 512 bits (the ideal is 1024 bits). The integer p is a 159-digit number. We also choose g, x, and y as shown below:

p	764624298563493572182493765955030507476338096726949748923573772860925235666660755423637423309661180033338106194730130950414738700999178043654878580798758 1
g	2
x	557
y	273

The following shows the values of R_1, R_2, and K.

R_1	844920284205665505216172947491035094143433698520012660862863631067673619959280828586700802131859290945140217500319973312945836083821943065966020157955354
R_2	435262838709200379470747114895581627636389116262115557975123379218566310011435718208390040181876486841753831165342691630263421106721508589625520128859414 3
K	155638000664522290596225827523270765273218046944423678520320400146406500887936651204257426776608327911017153038674561252213151610976584200120408643361774 0

Analysis of Diffie-Hellman

The Diffie-Hellman concept, shown in Figure 15.10, is simple but elegant. We can think of the secret key between Alice and Bob as made of three parts: g, x, and y. The first part is public. Everyone knows 1/3 of the key; g is a public value. The other two parts must be added by Alice and Bob. Each of them add one part. Alice adds x as the second part for Bob; Bob adds y as the second part for Alice. When Alice receives the 2/3 completed key from Bob, she adds the last part, her x, to complete the key. When Bob receives the 2/3-completed key from Alice, he adds the last part, his y, to complete the key. Note that although the key in Alice's hand consists of g, y, and x and the key in Bob's hand consists of g, x, and y, these two keys are the same because $g^{xy} = g^{yx}$.

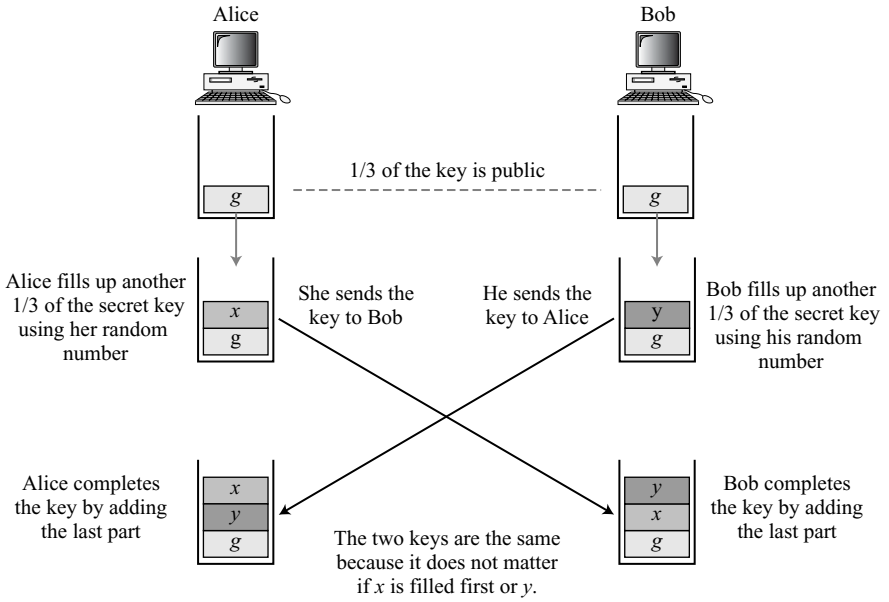
Note also that although the two keys are the same, Alice cannot find the value y used by Bob because the calculation is done in modulo p; Alice receives $g^y \bmod p$ from Bob, not g^y. To know the value of y, Alice must use the discrete logarithm that we discussed in a previous chapter.

Security of Diffie-Hellman

The Diffie-Hellman key exchange is susceptible to two attacks: the discrete logarithm attack and the man-in-the-middle attack.

Discrete Logarithm Attack The security of the key exchange is based on the difficulty of the discrete logarithm problem. Eve can intercept R_1 and R_2. If she can find x

Figure 15.10 *Diffie-Hellman idea*



from $R_1 = g^x \bmod p$ and y from $R_2 = g^y \bmod p$, then she can calculate the symmetric key $K = g^{xy} \bmod p$. The secret key is not secret anymore. To make Diffie-Hellman safe from the discrete logarithm attack, the following are recommended.

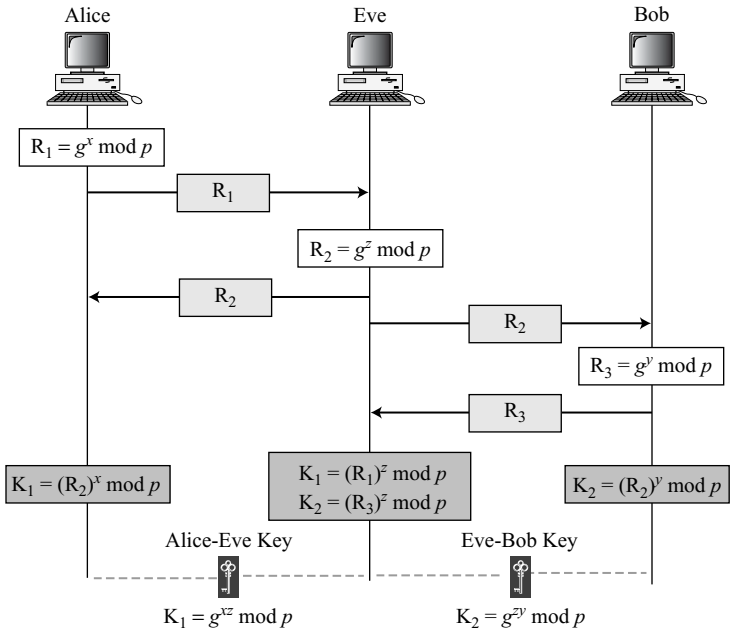
1. The prime p must be very large (more than 300 decimal digits).
2. The prime p must be chosen such that $p - 1$ has at least one large prime factor (more than 60 decimal digits).
3. The generator must be chosen from the group $<\mathbf{Z}_{p^*}, \times>$.
4. Bob and Alice must destroy x and y after they have calculated the symmetric key. The values of x and y must be used only once.

Man-in-the-Middle Attack The protocol has another weakness. Eve does not have to find the value 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 15.11 shows the situation.

The following can happen:

1. Alice chooses x, calculates $R_1 = g^x \bmod p$, and sends R_1 to Bob.
2. Eve, the intruder, intercepts R_1. She chooses z, calculates $R_2 = g^z \bmod p$, and sends R_2 to both Alice and Bob.
3. Bob chooses y, calculates $R_3 = g^y \bmod p$, and sends R_3 to Alice. R_3 is intercepted by Eve and never reaches Alice.
4. Alice and Eve calculate $K_1 = g^{xz} \bmod p$, which becomes a shared key between Alice and Eve. Alice, however, thinks that it is a key shared between Bob and herself.

Figure 15.11 *Man-in-the-middle attack*



5. Eve and Bob calculate $K_2 = g^{zy} \bmod p$, which becomes a shared key between Eve and Bob. Bob, however, thinks that it is a key shared between Alice and himself.

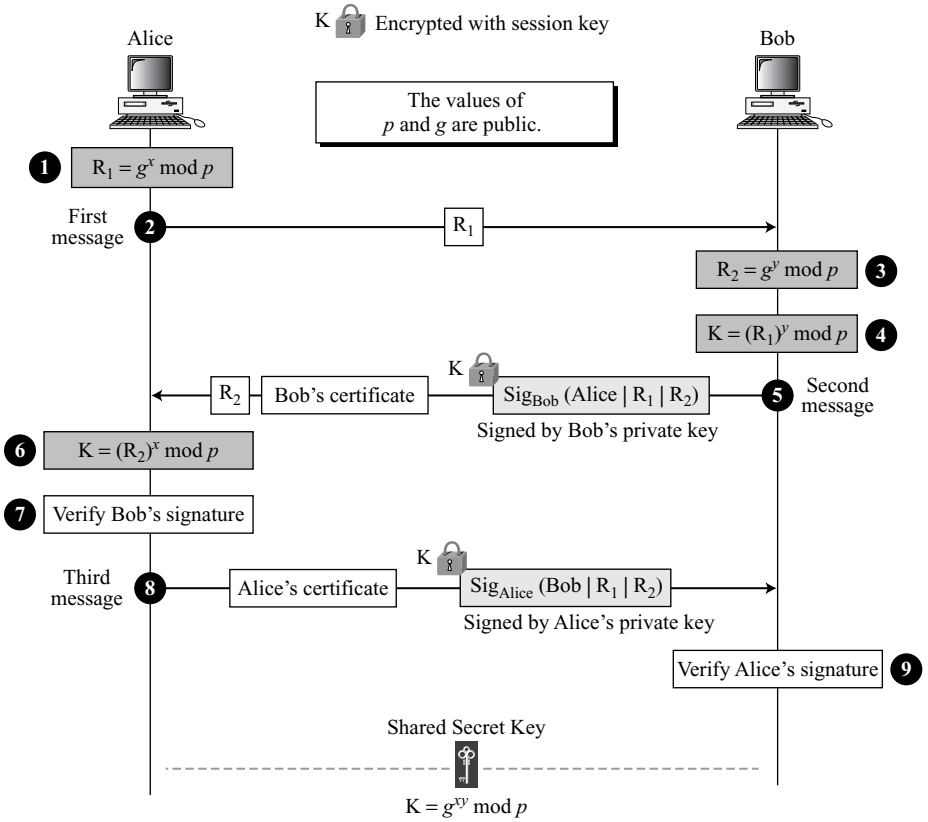
In other words, two keys, instead of one, are created: one between Alice and Eve, one between Eve and Bob. When Alice sends data to Bob encrypted with K_1 (shared by Alice and Eve), it can be deciphered and read by Eve. Eve can send the message to Bob encrypted by K_2 (shared key between Eve and Bob); 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. A similar 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 R_1, sent by Alice to Bob, and R_3, 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. The next method, based on the Diffie-Hellman uses authentication to thwart this attack.

Station-to-Station Key Agreement

The **station-to-station protocol** is a method based on Diffie-Hellman. It uses digital signatures with public-key certificates (see the next section) to establish a session key between Alice and Bob, as shown in Figure 15.12.

Figure 15.12 *Station-to-station key agreement method*



The following shows the steps:

- After calculating R_1, Alice sends R_1 to Bob (steps 1 and 2 in Figure 15.12).
- After calculating R_2 and the session key, Bob concatenates Alice's ID, R_1, and R_2. He then signs the result with his private key. Bob now sends R_2, the signature, and his own public-key certificate to Alice. The signature is encrypted with the session key (steps 3, 4, and 5 in Figure 15.12).
- After calculating the session key, if Bob's signature is verified, Alice concatenates Bob's ID, R_1, and R_2. She then signs the result with her own private key and sends it to Bob. The signature is encrypted with the session key (steps 6, 7, and 8 in Figure 15.12).
- If Alice's signature is verified, Bob keeps the session key (step 9 in Figure 15.12).

Security of Station-to-Station Protocol

The station-to-station protocol prevents man-in-the-middle attacks. After intercepting R_1, Eve cannot send her own R_2 to Alice and pretend it is coming from Bob because Eve cannot forge the private key of Bob to create the signature—the signature cannot be verified with Bob's public key defined in the certificate. In the same way, Eve cannot forge Alice's private key to sign the third message sent by Alice. The certificates, as we will see in the next section, are trusted because they are issued by trusted authorities.

15.4 PUBLIC-KEY DISTRIBUTION

In asymmetric-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.

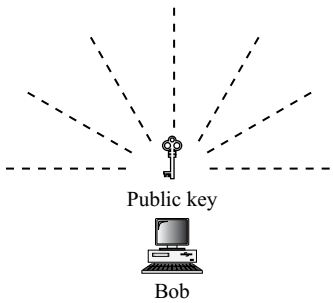
In public-key cryptography, everyone has access to everyone's public key; public keys are available to the public.

Public keys, like secret keys, need to be distributed to be useful. Let us briefly discuss the way public keys can be distributed.

Public Announcement

The naive approach is to announce public keys publicly. Bob can put his public key on his website or announce it in a local or national newspaper. When Alice needs to send a confidential message to Bob, she can obtain Bob's public key from his site or from the newspaper, or even send a message to ask for it. Figure 15.13 shows the situation.

Figure 15.13 *Announcing a public key*

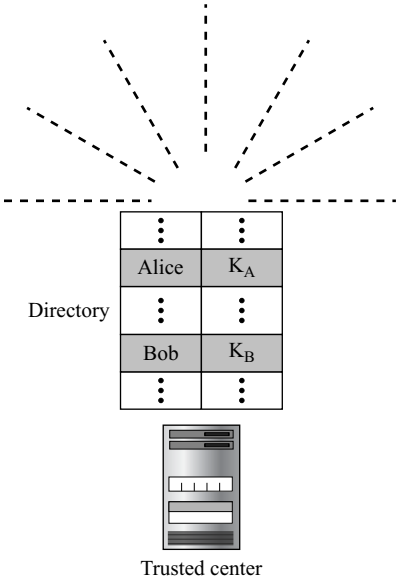


This approach, however, is not secure; it is subject to forgery. For example, Eve could make such a public announcement. Before Bob can react, damage could be done. Eve can fool Alice into sending her a message that is intended for Bob. Eve could also sign a document with a corresponding forged private key and make everyone believe it was signed by Bob. The approach is also vulnerable if Alice directly requests Bob's public key. Eve can intercept Bob's response and substitute her own forged public key for Bob's public key.

Trusted Center

A more secure approach is to have a trusted center retain a directory of public keys. The directory, like the one used in a telephone system, is dynamically updated. Each user can select a private and public key, keep the private key, and deliver the public key for insertion into the directory. The center requires that each user register in the center and prove his or her identity. The directory can be publicly advertised by the trusted center. The center can also respond to any inquiry about a public key. Figure 15.14 shows the concept.

Figure 15.14 *Trusted center*



Controlled Trusted Center

A higher level of security can be achieved if there are added controls on the distribution of the public key. The public-key announcements can include a timestamp and be signed by an authority to prevent interception and modification of the response. If Alice needs to know Bob's public key, she can send a request to the center including Bob's name and a timestamp. The center responds with Bob's public key, the original request, and the timestamp signed with the private key of the center. Alice uses the public key of the center, known by all, to verify the timestamp. If the timestamp is verified, she extracts Bob's public key. Figure 15.15 shows one scenario.

Certification Authority

The previous approach can create a heavy load on the center if the number of requests is large. The alternative is to create **public-key certificates.** Bob wants two things; he wants people to know his public key, and he wants no one to accept a forged public key as his. 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 being forged, the CA signs the certificate with its private key. Now Bob can upload the signed certificate. Anyone who wants Bob's public key downloads the signed certificate and uses the center's public key to extract Bob's public key. Figure 15.16 shows the concept.

Figure 15.15 *Controlled trusted center*

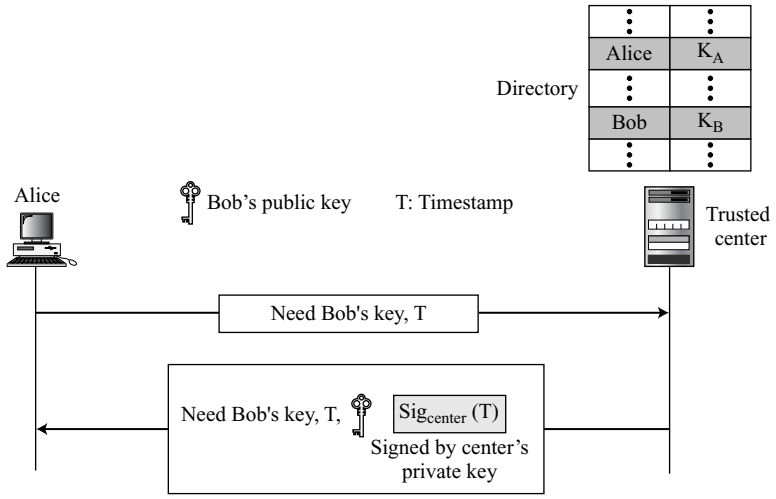
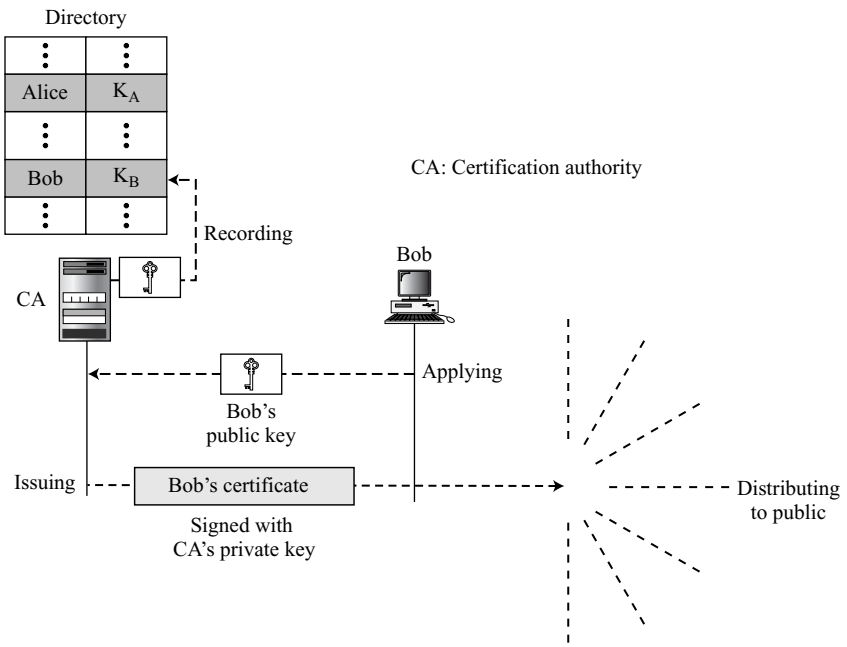


Figure 15.16 *Certification authority*



X.509

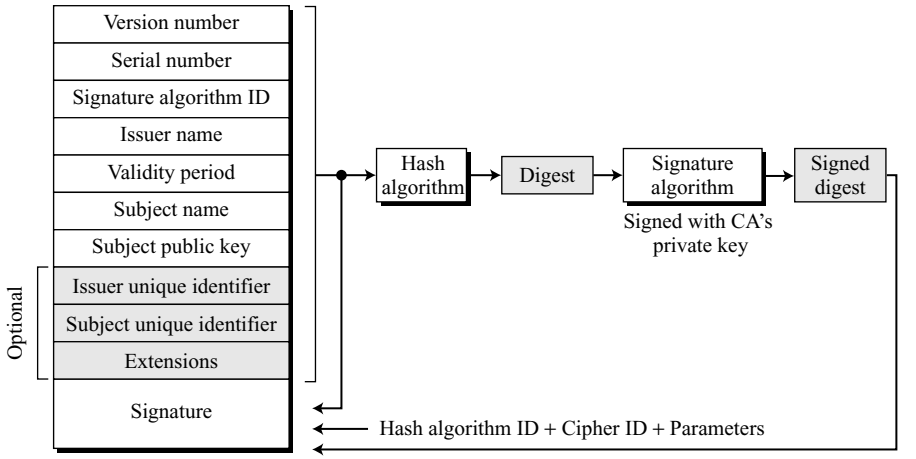
Although the use of a CA has solved the problem of public-key fraud, it has created a side-effect. Each certificate may have a different format. If Alice wants to use a program to automatically download different certificates and digests belonging to different people, the program may not be able to do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

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

Certificate

Figure 15.17 shows the format of a certificate.

Figure 15.17 *X.509 certificate format*



A certificate has the following fields:

- **Version number.** This field defines the version of X.509 of the certificate. The version number started at 0; the current version (third version) is 2.
- **Serial number.** This field defines a number assigned to each certificate. The value is unique for each certificate issuer.
- **Signature algorithm ID.** This field identifies the algorithm used to sign the certificate. Any parameter that is needed for the signature is also defined in this field.
- **Issuer name.** This field identifies the certification authority that issued the certificate. The name is normally a hierarchy of strings that defines a country, a state, organization, department, and so on.

- **Validity Period.** This field defines the earliest time (not before) and the latest time (not after) the certificate is valid.
- **Subject name.** This field defines the entity to which the public key belongs. It is also a hierarchy of strings. Part of the field defines what is called the *common name,* which is the actual name of the beholder of the key.
- **Subject public key.** This field defines the owner's public key, the heart of the certificate. The field also defines the corresponding public-key algorithm (RSA, for example) and its parameters.
- **Issuer unique identifier.** This optional field allows two issuers to have the same *issuer* field value, if the *issuer unique identifiers* are different.
- **Subject unique identifier.** This optional field allows two different subjects to have the same *subject* field value, if the *subject unique identifiers* are different.
- **Extensions.** This optional field allows issuers to add more private information to the certificate.
- **Signature.** This field is made of three sections. The first section contains all other fields in the certificate. The second section contains the digest of the first section encrypted with the CA's public key. The third section contains the algorithm identifier used to create the second section.

Certificate Renewal

Each certificate has a period of validity. If there is no problem with the certificate, the CA issues a new certificate before the old one expires. The process is like the renewal of credit cards by a credit card company; the credit card holder normally receives a renewed credit card before the one expires.

Certificate Revocation

In some cases a certificate must be revoked before its expiration. Here are some examples:

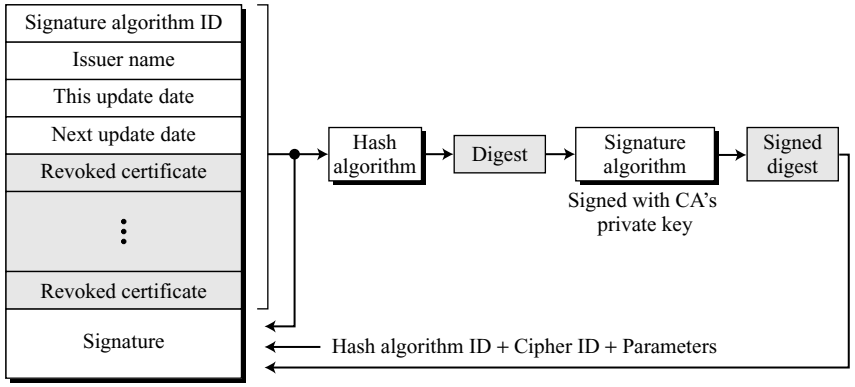
a. The user's (subject's) private key (corresponding to the public key listed in the certificate) might have been comprised.
b. The CA is no longer willing to certify the user. For example, the user's certificate relates to an organization that she no longer works for.
c. The CA's private key, which can verify certificates, may have been compromised. In this case, the CA needs to revoke all unexpired certificates.

The revocation is done by periodically issuing a certificate revocation list (CRL). The list contains all revoked certificates that are not expired on the date the CRL is issued. When a user wants to use a certificate, she first needs to check the directory of the corresponding CA for the last certificate revocation list. Figure 15.18 shows the certificate revocation list.

A certificate revocation list has the following fields:

- **Signature algorithm ID.** This field is the same as the one in the certificate.
- **Issuer name.** This field is the same as the one in the certificate.
- **This update date.** This field defines when the list is released.
- **Next update date.** This field defines the next date when the new list will be released.

Figure 15.18 *Certificate revocation format*



- **Revoked certificate.** This is a repeated list of all unexpired certificates that have been revoked. Each list contains two sections: user certificate serial number and revocation date.
- **Signature.** This field is the same as the one in the certificate list.

Delta Revocation

To make revocation more efficient, the delta certificate revocation list (delta CRL) has been introduced. A delta CRL is created and posted on the directory if there are changes after *this update date* and *next update date*. For example, if CRLs are issued every month, but there are revocations in between, the CA can create a delta CRL when there is a change during the month. However, a delta CRL contains only the changes made after the last CRL.

Public-Key Infrastructures (PKI)

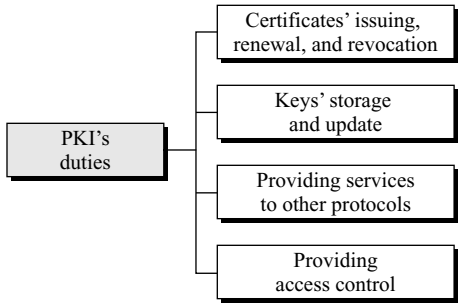
Public-Key Infrastructure (PKI) is a model for creating, distributing, and revoking certificates based on the X.509. The Internet Engineering Task Force (see Appendix B) has created the Public-Key Infrastructure X.509 (PKIX).

Duties

Several duties have been defined for a PKI. The most important ones are shown in Figure 15.19.

- **Certificates' issuing, renewal, and revocation.** These are duties defined in the X.509. Because the PKIX is based on X.509, it needs to handle all duties related to certificates.
- **Keys' storage and update.** A PKI should be a storage place for private keys of those members that need to hold their private keys somewhere safe. In addition, a PKI is responsible for updating these keys on members' demands.

Figure 15.19 *Some duties of a PKI*



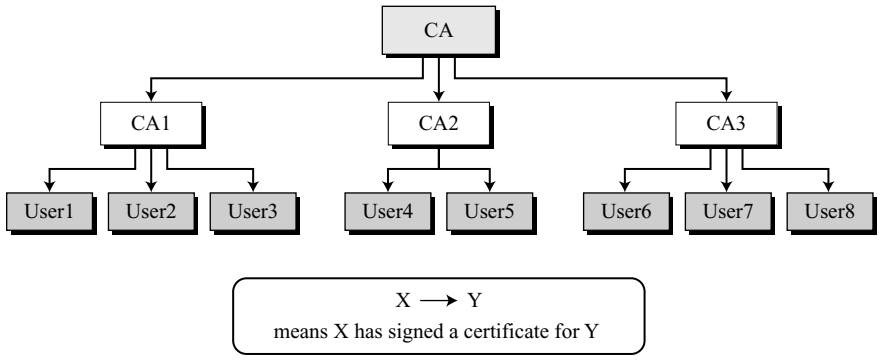
- **Providing services to other protocols.** As we see will in the next few chapters, some Internet security protocols, such as IPSec and TLS, are relying on the services by a PKI.
- **Providing access control.** A PKI can provide different levels of access to the information stored in its database. For example, an organization PKI may provide access to the whole database for the top management, but limited access for employees.

Trust Model

It is not possible to have just one CA issuing all certificates for all users in the world. There should be many CAs, each responsible for creating, storing, issuing, and revoking a limited number of certificates. The **trust model** defines rules that specify how a user can verify a certificate received from a CA.

Hierarchical Model In this model, there is a tree-type structure with a root CA. The root CA has a self-signed, self-issued certificate; it needs to be trusted by other CAs and users for the system to work. Figure 15.20 shows a trust model of this kind with three hierarchical levels. The number of levels can be more than three in a real situation.

Figure 15.20 *PKI hierarchical model*



The figure shows that the CA (the root) has signed certificates for CA1, CA2, and CA3; CA1 has signed certificates for User1, User2, and User3; and so on. PKI uses the following notation to mean the certificate issued by authority X for entity Y.

X<<Y>>

Example 15.3

Show how User1, knowing only the public key of the CA (the root), can obtain a verified copy of User3's public key.

Solution

User3 sends a chain of certificates, CA<<CA1>> and CA1<<User3>>, to User1.

a. User1 validates CA<<CA1>> using the public key of CA.
b. User1 extracts the public key of CA1 from CA<<CA1>>.
c. User1 validates CA1<<User3>> using the public key of CA1.
d. User1 extracts the public key of User 3 from CA1<<User3>>.

Example 15.4

Some Web browsers, such as Netscape and Internet Explorer, include a set of certificates from independent roots without a single, high-level, authority to certify each root. One can find the list of these roots in the Internet Explorer at *Tools/Internet Options/Contents/Certificate/ Trusted roots* (using pull-down menu). The user then can choose any of this root and view the certificate.

Mesh Model The hierarchical model may work for an organization or a small community. A larger community may need several hierarchical structures connected together. One method is to use a mesh model to connect the roots together. In this model, each root is connected to every other root, as shown in Figure 15.21.

Figure 15.21 shows that the mesh structure connects only roots together; each root has its own hierarchical structure, shown by a triangle. The certifications between the roots are cross-certificates; each root certifies all other roots, which means there are N ($N - 1$) certificates. In Figure 15.21, there are 4 nodes, so we need $4 \times 3 = 12$ certificates. Note that each double-arrow line represents two certificates.

Example 15.5

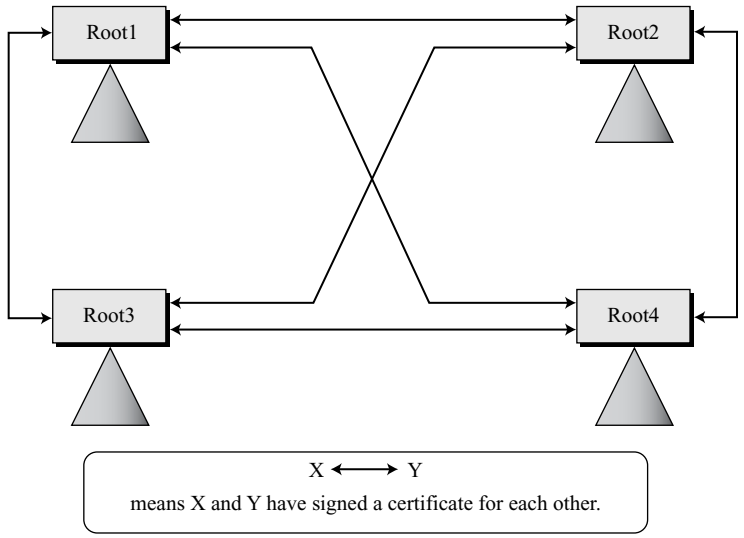
Alice is under the authority Root1; Bob is under the authority Root4. Show how Alice can obtain Bob's verified public key.

Solution

Bob sends a chain of certificates from Root4 to Bob. Alice looks at the directory of Root1 to find Root1<<Root1>> and Root1<< Root4>> certificates. Using the process shown in Figure 15.21, Alice can verify Bob's public key.

Web of Trust This model is used in Pretty Good Privacy, a security service for electronic mail discussed in Chapter 16.

Figure 15.21 *Mesh model*



15.5 RECOMMENDED READING

The following books and websites give more details about subjects discussed in this chapter. The items enclosed in brackets refer to the reference list at the end of the book.

Books

For further discussion of symmetric-key and asymmetric-key management, see [Sti06], [KPS02], [Sta06], [Rhe03], and [PHS03].

WebSites

The following websites give more information about topics discussed in this chapter.

http://en.wikipedia.org/wiki/Needham-Schroeder
http://en.wikipedia.org/wiki/Otway-Rees
http://en.wikipedia.org/wiki/Kerberos_%28protocol%29
en.wikipedia.org/wiki/Diffie-Hellman
www.ietf.org/rfc/rfc2631.txt

15.6 KEY TERMS AND CONCEPTS

authentication server (AS)
bucket brigade attack
certification authority (CA)
Diffie-Hellman protocol
Kerberos
key-distribution center (KDC)
man-in-the-middle attack
Needham-Schroeder protocol
Otway-Rees protocol
public-key certificate
public-key infrastructure (PKI)
session key
station-to-station protocol
ticket
ticket-granting server (TGS)
trust model
X.509

15.7 SUMMARY

- Symmetric-key cryptography needs a shared secret key between two parties. If N people need to communicate with each other, $N(N - 1)/2$ keys are needed. The number of keys is not the only problem; the distribution of keys is another.
- A practical solution is the use of a trusted third party, referred to as a key-distribution center (KDC). A KDC can create a session (temporary) key between Alice and Bob using their keys with the center. The keys of Alice and Bob are used to authenticate Alice and Bob to the center.
- Several different approaches have been proposed to create the session key using ideas discussed in Chapter 14 for entity authentication. Two of the most elegant ones are Needham-Schroeder protocol, which is a foundation for many other protocols, and Otway-Rees Protocol.
- Kerberos is both an authentication protocol and a KDC. Several systems, including Windows 2000, use Kerberos. Three servers are involved in the Kerberos protocol: an authentication server (AS), a ticket-granting server (TGS), and a real (data) server.
- Alice and Bob can create a session key between themselves without using a KDC. This method of session-key creation is referred to as the symmetric-key agreement. We discussed two methods: Diffie-Hellman and station-to-station. The first is susceptible to the man-in-the-middle attack; the second is not.
- Public keys, like secret keys, need to be distributed to be useful. Certificate authorities (CAs) provide certificates as proof of the ownership of public keys. X.509 is a recommendation that defines the structure of certificates issued by CAs.
- Public Key Infrastructure (PKI) is a model for creating, distributing, and revoking certificates based on the X.509. The Internet Engineering Task Force has created the Public Key Infrastructure X.509 (PKIX). The duties of a PKI include certificate issuing, private key storage, services to other protocols, and access control.

A PKI also defines trust models, the relationship between certificate authorities. The three trust models mentioned in this chapter are hierarchical, mesh, and web of trust.

15.8 PRACTICE SET

Review Questions

1. List the duties of a KDC.
2. Define a session key and show how a KDC can create a session key between Alice and Bob.
3. Define Kerberos and name its servers. Briefly explain the duties of each server.
4. Define the Diffie-Hellman protocol and its purpose.
5. Define the man-in-the-middle attack.
6. Define the station-to-station protocol and mention its purpose.
7. Define a certification authority (CA) and its relation to public-key cryptography.
8. Define the X.509 recommendation and state its purpose.
9. List the duties of a PKI.
10. Define a trust model and mention some variations of this model discussed in this chapter.

Exercises

11. In Figure 15.4, what happens if the ticket for Bob is not encrypted in step 2 with K_B, but is encrypted instead by K_{AB} in step 3?
12. Why is there a need for two nonces in the Needham-Schroeder protocol?
13. In the Needham-Schroeder protocol, how is Alice authenticated by the KDC? How is Bob authenticated by the KDC? How is the KDC authenticated to Alice? How is the KDC authenticated to Bob? How is Alice authenticated to Bob? How is Bob authenticated to Alice?
14. Can you explain why in the Needham-Schroeder protocol, Alice is the party that is in contact with the KDC, but in the Otway-Rees protocol, Bob is the party that is in contact with the KDC?
15. There are two nonces (R_A and R_B) in the Needham-Schroeder protocol, but three nonces (R_A, R_B, and R) in the Otway-Rees protocol. Can you explain why there is a need for one extra nonce, R_2, in the first protocol?
16. Why do you think we need only one timestamp in Kerberos instead of two nonces as in Needham-Schroeder or three nonces as in Otway-Rees?
17. In the Diffie-Hellman protocol, $g = 7$, $p = 23$, $x = 3$, and $y = 5$.
    a. What is the value of the symmetric key?
    b. What is the value of R_1 and R_2?

18. In the Diffie-Hellman protocol, what happens if x and y have the same value, that is, Alice and Bob have accidentally chosen the same number? Are R_1 and R_2 the same? Do the session keys calculated by Alice and Bob have the same value? Use an example to prove your claims.
19. In a trivial (not secure) Diffie-Hellman key exchange, $p = 53$. Find an appropriate value for g.
20. In station-to-station protocol, show that if the identity of the receiver is removed from the signature, the protocol becomes vulnerable to the man-in-the-middle attack.
21. Discuss the trustworthiness of root certificates provided by browsers.