

Chapter 15 Security Protocols

Up to this point, we have covered many basic cryptographic tools, ranging from encryption algorithms to hash algorithms to digital signatures. A natural question arises: Can we just apply these tools directly to make computers and communications secure?

At first glance, one might think that public key methods are the panacea for all of security. They allow two parties who have never met to exchange messages securely. They also provide an easy way to authenticate the origin of a message and, when combined with hash functions, these signature operations can be made efficient.

Unfortunately, the answer is definitely no and there are many problems that still remain. In discussing public key algorithms, we never really discussed how the public keys are distributed. We have casually said that Alice will announce her public key for Bob to use. Bob, however, should not be too naive in just believing what he hears. How does he know that it is actually Alice that he is communicating with? Perhaps Alice's evil friend, Mallory, is pretending to be Alice but is actually announcing Mallory's public key instead. Similarly, when you access a website to make a purchase, how do you know that your transaction is really with a legitimate merchant and that no one has set up a false organization? The real challenge in these problems is the issue of authentication, and Bob should always confirm that he is communicating with Alice before sending any important information.

Combining different cryptographic tools to provide security is much trickier than grabbing algorithms off of the shelf. Instead, security protocols involving the

exchange of messages between different entities must be carefully thought out in order to prevent clever attacks. This chapter focuses on such security protocols.

15.1 Intruders-in-the-Middle and Impostors

If you receive an email asking you to go to a website and update your account information, how can you be sure that the website is legitimate? An impostor can easily set up a web page that looks like the correct one but simply records sensitive information and forwards it to Eve. This is an important authentication problem that must be addressed in real-world implementations of cryptographic protocols. One standard solution uses certificates and a trusted authority and will be discussed in [Section 15.5](#). Authentication will also play an important role in the protocols in many other sections of this chapter.

Another major consideration that must be addressed in communications over public channels is the intruder-in-the-middle attack, which we'll discuss shortly. It is another cause for several of the steps in the protocols we discuss.

15.1.1 Intruder-in-the-Middle Attacks

Eve, who has recently learned the difference between a knight and a rook, claims that she can play two chess grandmasters simultaneously and either win one game or draw both games. The strategy is simple. She waits for the first grandmaster to move, then makes the identical move against the second grandmaster. When the second grandmaster responds, Eve makes that play against the first grandmaster. Continuing in this way, Eve cannot

lose both games (unless she runs into time trouble because of the slight delay in transferring the moves).

A similar strategy, called the **intruder-in-the-middle attack**, can be used against many cryptographic protocols. Many of the technicalities of the algorithms in this chapter are caused by efforts to thwart such an attack.

Let's see how this attack works against the Diffie-Hellman key exchange from Section 10.4.

Let's recall the protocol. Alice and Bob want to establish a key for communicating. The Diffie-Hellman scheme for accomplishing this is as follows:

1. Either Alice or Bob selects a large, secure prime number p and a primitive root $\alpha \pmod{p}$. Both p and α can be made public.
2. Alice chooses a secret random x with $1 \leq x \leq p - 2$, and Bob selects a secret random y with $1 \leq y \leq p - 2$.
3. Alice sends $\alpha^x \pmod{p}$ to Bob, and Bob sends $\alpha^y \pmod{p}$ to Alice.
4. Using the messages that they each have received, they can each calculate the session key K . Alice calculates K by $K \equiv (\alpha^y)^x \pmod{p}$, and Bob calculates K by $K \equiv (\alpha^x)^y \pmod{p}$.

Here is how the intruder-in-the-middle attack works.

1. Eve chooses an exponent z .
2. Eve intercepts α^x and α^y .
3. Eve sends α^z to Alice and to Bob (Alice believes she is receiving α^y and Bob believes he is receiving α^x).
4. Eve computes $K_{AE} \equiv (\alpha^x)^z \pmod{p}$ and $K_{EB} \equiv (\alpha^y)^z \pmod{p}$. Alice, not realizing that Eve is in the middle, also computes K_{AE} , and Bob computes K_{EB} .
5. When Alice sends a message to Bob, encrypted with K_{AE} , Eve intercepts it, deciphers it, encrypts it with K_{EB} , and sends it to Bob. Bob decrypts with K_{EB} and obtains the message. Bob has no reason to believe the communication was insecure. Meanwhile, Eve is reading the juicy gossip that she has obtained.

To avoid the intruder-in-the-middle attack, it is desirable to have a procedure that authenticates Alice's and Bob's identities to each other while the key is being formed. A protocol that can do this is known as an **authenticated key agreement protocol**.

A standard way to stop the intruder-in-the-middle attack is the **station-to-station (STS) protocol**, which uses digital signatures. Each user U has a digital signature function sig_U with verification algorithm ver_U . For example, sig_U could produce an RSA or ElGamal signature, and ver_U checks that it is a valid signature for U . The verification algorithms are compiled and made public by the trusted authority Trent, who certifies that ver_U is actually the verification algorithm for U and not for Eve.

Suppose now that Alice and Bob want to establish a key to use in an encryption function E_K . They proceed as in the Diffie-Hellman key exchange, but with the added feature of digital signatures:

1. They choose a large prime p and a primitive root α .
2. Alice chooses a random x and Bob chooses a random y .
3. Alice computes $\alpha^x \pmod{p}$, and Bob computes $\alpha^y \pmod{p}$.
4. Alice sends α^x to Bob.
5. Bob computes $K \equiv (\alpha^x)^y \pmod{p}$.
6. Bob sends α^y and $E_K(sig_B(\alpha^y, \alpha^x))$ to Alice.
7. Alice computes $K \equiv (\alpha^y)^x \pmod{p}$.
8. Alice decrypts $E_K(sig_B(\alpha^y, \alpha^x))$ to obtain $sig_B(\alpha^y, \alpha^x)$.
9. Alice asks Trent to verify that ver_B is Bob's verification algorithm.
10. Alice uses ver_B to verify Bob's signature.
11. Alice sends $E_K(sig_A(\alpha^x, \alpha^y))$ to Bob.
12. Bob decrypts, asks Trent to verify that ver_A is Alice's verification algorithm, and then uses ver_A to verify Alice's signature.

This protocol is due to Diffie, van Oorschot, and Wiener. Note that Alice and Bob are also certain that they are using the same key K , since it is very unlikely that an incorrect key would give a decryption that is a valid signature.

Note the role that trust plays in the protocol. Alice and Bob must trust Trent's verification if they are to have confidence that their communications are secure. Throughout this chapter, a trusted authority such as Trent will be an important participant in many protocols.

15.2 Key Distribution

So far in this book we have discussed various cryptographic concepts and focused on developing algorithms for secure communication. But a cryptographic algorithm is only as strong as the security of its keys. If Alice were to announce to the whole world her key before starting an AES session with Bob, then anyone could eavesdrop. Such a scenario is absurd, of course. But it represents an extreme version of a very important issue: If Alice and Bob are unable to meet in order to exchange their keys, can they still decide on a key without compromising future communication?

In particular, there is the fundamental problem of sharing secret information for the establishment of keys for symmetric cryptography. By symmetric cryptography, we mean a system such as AES where both the sender and the recipient use the same key. This is in contrast to public key methods such as RSA, where the sender has one key (the encryption exponent) and the receiver has another (the decryption exponent).

In key establishment protocols, there is a sequence of steps that take place between Alice and Bob so that they can share some secret information needed in the establishment of a key. Since public key cryptography methods employ public encryption keys that are stored in public databases, one might think that public key cryptography provides an easy solution to this problem. This is partially true. The main downside to public key cryptography is that even the best public key cryptosystems are computationally slow when compared with the best symmetric key methods. RSA, for example, requires exponentiation, which is not as fast as the mixing of bits that takes place in AES. Therefore,

sometimes RSA is used to transmit an AES key that will then be used for transmitting vast amounts of data. However, a central server that needs to communicate with many clients in short time intervals sometimes needs key establishment methods that are faster than current versions of public key algorithms. Therefore, in this and in various other situations, we need to consider other means for the exchange and establishment of keys for symmetric encryption algorithms.

There are two basic types of key establishment. In **key agreement** protocols, neither party knows the key in advance; it is determined as a result of their interaction. In **key distribution** protocols, one party has decided on a key and transmits it to the other party.

Diffie-Hellman key exchange (see [Sections 10.4](#) and [15.1](#)) is an example of key agreement. Using RSA to transmit an AES key is an example of key distribution.

In any key establishment protocol, authentication and intruder-in-the-middle attacks are security concerns. Pre-distribution, which will be discussed shortly, is one solution. Another solution involves employing a server that will handle the task of securely giving keys to two entities wishing to communicate. We will also look at some other basic protocols for key distribution using a third party. Solutions that are more practical for Internet communications are treated in later sections of this chapter.

15.2.1 Key Pre-distribution

In the simplest version of this protocol, if Alice wants to communicate with Bob, the keys or key schedules (lists describing which keys to use at which times) are decided upon in advance and somehow this information is sent securely from one to the other. For example, this method

was used by the German navy in World War II. However, the British were able to use codebooks from captured ships to find daily keys and thus read messages.

There are some obvious limitations and drawbacks to pre-distribution. First, it requires two parties, Alice and Bob, to have met or to have established a secure channel between them in the first place. Second, once Alice and Bob have met and exchanged information, there is nothing they can do, other than meeting again, to change the key information in case it gets compromised. The keys are predetermined and there is no easy method to change the key after a certain amount of time. When using the same key for long periods of time, one runs a risk that the key will become compromised. The more data that are transmitted, the more data there are with which to build statistical attacks.

Here is a general and slightly modified situation. First, we require a trusted authority whom we call Trent. For every pair of users, call them (A, B) , Trent produces a random key K_{AB} that will be used as a key for a symmetric encryption method (hence $K_{BA} = K_{AB}$). It is assumed that Trent is powerful and has established a secure channel to each of the users. He distributes all the keys that he has determined to his users. Thus, if Trent is responsible for n users, each user will be receiving $n - 1$ keys to store, and Trent must send $n(n - 1)/2$ keys securely. If n is large, this could be a problem. The storage that each user requires is also a problem.

One method for reducing the amount of information that must be sent from the trusted authority is the **Blom key pre-distribution scheme**. Start with a network of n users, and let p be a large prime, where $p \geq n$. Everyone has knowledge of the prime p . The protocol is now the following:

1. Each user U in the network is assigned a distinct public number $r_U \pmod{p}$.

2. Trent chooses three secret random numbers a , b , and $c \bmod p$.

3. For each user U , Trent calculates the numbers

$$a_U \equiv a + br_U \pmod{p} \quad b_U \equiv b + cr_U \pmod{p}$$

and sends them via his secure channel to U .

4. Each user U forms the linear polynomial

$$g_U(x) = a_U + b_U x.$$

5. If Alice (A) wants to communicate with Bob (B), then Alice computes $K_{AB} = g_A(r_B)$, while Bob computes $K_{BA} = g_B(r_A)$.

6. It can be shown that $K_{AB} = K_{BA}$ (Exercise 2). Alice and Bob communicate via a symmetric encryption system, for example, AES, using the key (or a key derived from) K_{AB} .

Example

Consider a network consisting of three users Alice, Bob, and Charlie. Let $p = 23$, and let

$$r_A = 11, \quad r_B = 3, \quad r_C = 2.$$

Suppose Trent chooses the numbers

$a = 8$, $b = 3$, $c = 1$. The corresponding linear polynomials are given by

$$g_A(x) = 18 + 14x, \quad g_B(x) = 17 + 6x, \quad g_C(x) = 14 + 5x.$$

It is now possible to calculate the keys that this scheme would generate:

$$K_{AB} = g_A(r_B) = 14, \quad K_{AC} = g_A(r_C) = 0, \quad K_{BC} = g_B(r_C) = 6.$$

It is easy to check that $K_{AB} = K_{BA}$, etc., in this example.

If the two users Eve and Oscar conspire, they can determine a , b , and c and therefore find all numbers a_A , b_A for all users. They proceed as follows. They know the numbers a_E , b_E , a_O , b_O . The defining equations for the last three of these numbers can be written in matrix form as

$$\begin{array}{cccccc} 0 & 1 & r_E & a & & b_E \\ 1 & r_O & 0 & b & \equiv & a_O \\ 0 & 1 & r_O & c & & b_O \end{array} \pmod{p}.$$

The determinant of the matrix is $r_E - r_O$. Since the numbers r_A were chosen to be distinct mod p , the determinant is nonzero mod p and therefore the system has a unique solution a, b, c .

Without Eve's help, Oscar has only a 2×3 matrix to work with and therefore cannot find a, b, c . In fact, suppose he wants to calculate the key K_{AB} being used by Alice and Bob. Since $K_{AB} \equiv a + b(r_A + r_B) + c(r_A r_B)$ (see [Exercise 2](#)), there is the matrix equation

$$\begin{array}{cccccc} 1 & r_A + r_B & r_A r_B & a & & K_{AB} \\ 1 & r_O & 0 & b & \equiv & a_O \\ 0 & 1 & r_O & c & & b_O \end{array} \pmod{p}.$$

The matrix has determinant $(r_O - r_A)(r_O - r_B) \not\equiv 0 \pmod{p}$. Therefore, there is a solution a, b, c for every possible value of K_{AB} . This means that Oscar obtains no information about K_{AB} .

For each $k \geq 1$, there are Blom schemes that are secure against coalitions of at most k users, but which succumb to conspiracies of $k + 1$ users. See [Blom].

15.2.2 Authenticated Key Distribution

Key pre-distribution schemes are often impractical because they require significant resources to initialize and do not allow for keys to be changed or replaced easily when keys are deemed no longer safe. One way around these problems is to introduce a **trusted authority**, whose task is to distribute new keys to

communicating parties as they are needed. This trusted third party may be a server on a computer network, or an organization that is trusted by both Alice and Bob to distribute keys securely.

Authentication is critical to key distribution. Alice and Bob will ask the trusted third party, Trent, to give them keys. They want to make certain that there are no malicious entities masquerading as Trent and sending them false key messages. Additionally, when Alice and Bob exchange messages with each other, they will each need to make certain that the person they are talking to is precisely the person they think they are talking to.

One of the main challenges facing key distribution is the issue of replay attacks. In a replay attack, an opponent may record a message and repeat it at a later time in hope of either pretending to be another party or eliciting a particular response from an entity in order to compromise a key. To provide authentication and protect against replay attacks, we need to make certain that vital information, such as keys and identification parameters, are kept confidential. Additionally, we need to guarantee that each message is fresh; that is, it isn't a repeat of a message from a long time ago.

The task of confidentiality can be easily accomplished using existing keys already shared between entities. These keys are used to encrypt messages used in the distribution of session keys and are therefore often called key encrypting keys. Unfortunately, no matter how we look at it, there is a chicken-and-egg problem: In order to distribute session keys securely, we must assume that entities have already securely shared key encrypting keys with the trusted authority.

To handle message freshness, however, we typically need to attach extra data fields in each message we exchange.

There are three main types of data fields that are often introduced in order to prevent replay attacks:

- Sequence numbers: Each message that is sent between two entities has a sequence number associated with it. If an entity ever sees the same sequence number again, then the entity concludes that the message is a replay. The challenge with sequence numbers is that it requires that each party keep track of the sequence numbers it has witnessed.
- Timestamps: Each message that is sent between two entities has a statement of the time period for when that message is valid. This requires that both entities have clocks that are set to the same time.
- Nonces: A nonce is a random message that is allowed to be used only once and is used as part of a challenge-response mechanism. In a challenge-response, Alice sends Bob a message involving a nonce and requires Bob to send back a correct response to her nonce.

We will now look at two examples of key distribution schemes and analyze attacks that may be used against each in order to bypass the intended security. These two examples should highlight how difficult it is to distribute keys securely.

We begin with a protocol known as the **wide-mouthed frog protocol**, which is one of the simplest symmetric key management protocols involving a trusted authority. In this protocol, Alice chooses a session key K_{AB} to communicate with Bob and has Trent transfer it to Bob securely:

1. Alice \rightarrow Trent : $E_{K_{AT}}[t_A \parallel ID_B \parallel K_{AB}]$.
2. Trent \rightarrow Bob : $E_{K_{BT}}[t_T \parallel ID_A \parallel K_{AB}]$.

Here, K_{AT} is a key shared between Alice and Trent, while K_{BT} is a key shared between Bob and Trent. Alice's and Bob's identifying information are given by ID_A and ID_B , respectively. The parameter t_A is a timestamp supplied by Alice, while t_T is a timestamp given by Trent. It is assumed that Alice, Trent, and Bob have synchronized clocks. Bob will accept K_{AB} as fresh

if it arrives within a window of time. The key K_{AB} will be valid for a certain period of time after t_T .

The purpose behind the two timestamps is to allow Bob to check to see that the message is fresh. In the first message, Alice sends a message with a timestamp t_A . If Trent gets the message and the time is not too far off from t_A , he will then agree to translate the message and deliver it to Bob.

The problem with the protocol comes from the second message. Here, Trent has updated the timestamp to a newer timestamp t_T . Unfortunately, this simple change allows for a clever attack in which the nefarious Mallory may cause Trent to extend the lifetime of an old key. Let us step through this attack.

1. After seeing one exchange of the protocol, Mallory pretends to be Bob wanting to share a key with Alice. Mallory sends Trent the replay $E_{K_{BT}}[t_T \parallel ID_A \parallel K_{AB}]$.
2. Trent sends $E_{K_{AT}}[t'_T \parallel ID_B \parallel K_{AB}]$ to Alice, with a new timestamp t'_T . Alice thinks this is a valid message since it came from Trent and was encrypted using Trent's key. The key K_{AB} will now be valid for a period of time after t'_T .
3. Mallory then pretends to be Alice and gets $E_{K_{BT}}[t''_T \parallel ID_A \parallel K_{AB}]$. The key K_{AB} will now be valid for a period of time after $t''_T > t'_T$.
4. Mallory continues alternately playing Trent against Bob and then Trent against Alice.

The net result is that the malicious Mallory can use Trent as an agent to force Alice and Bob to continue to use K_{AB} indefinitely. Of course, Alice and Bob should keep track of the fact that they have seen K_{AB} before and begin to suspect that something suspicious is going on when they repeatedly see K_{AB} . The protocol did not explicitly state that this was necessary, however, and security protocols should be very explicit on what it is that they assume and don't assume. The true problem,

though, is the fact that Trent replaces t_A with t_T . If Trent had not changed t_T and instead had left t_A as the timestamp, then the protocol would have been better off.

Another example of an authenticated key exchange protocol is due to Needham and Schroeder. In the Needham–Schroeder protocol, Alice and Bob wish to obtain a session key K_S from Trent so that they can talk with each other. The protocol involves the following steps:

1. Alice \rightarrow Trent : $ID_A \parallel ID_B \parallel r_1$
2. Trent \rightarrow Alice : $E_{K_{AT}}[K_S \parallel ID_B \parallel r_1 \parallel E_{K_{BT}}[K_S \parallel ID_A]]$
3. Alice \rightarrow Bob : $E_{K_{BT}}[K_S \parallel ID_A]$
4. Bob \rightarrow Alice : $E_{K_S}[r_2]$
5. Alice \rightarrow Bob : $E_{K_S}[r_2 - 1]$

Just as in the earlier protocol, K_{AT} is a key shared between Alice and Trent, while K_{BT} is a key shared between Bob and Trent. Unlike the wide-mouthed frog protocol, the Needham–Schroeder protocol does not employ timestamps but instead uses random numbers r_1 and r_2 as nonces. In the first step, Alice sends Trent her request, which is a statement of who she is and whom she wants to talk to, along with a random number r_1 . Trent gives Alice the session key K_S and gives Alice a package $E_{K_{BT}}[K_S \parallel ID_A]$ that she will deliver to Bob. In the next step, she delivers the package to Bob. Bob can decrypt this to get the session key and the identity of the person he is talking with. Next, Bob sends Alice his own challenge by sending the second nonce r_2 . In the final step, Alice proves her identity to Bob by answering his challenge. Using $r_2 - 1$ instead of r_2 prevents Mallory from replaying message 4.

The purpose of r_1 is to prevent the reuse of old messages. Suppose r_1 is omitted from Steps 1 and 2. If

Eve sees that Alice wants to communicate with Bob, she could intercept Trent's message in Step 2 and substitute a transmission from a previous Step 2. Then Alice and Bob would communicate with a previously used session key, something that should generally be avoided. When r_1 is omitted, Alice has no way of knowing that this happened unless she has kept a list of previous session keys K_S .

Observe that the key exchange portion of the protocol is completed at the end of the third step. The last two exchanges, however, seem a little out of place and deserve some more discussion. The purpose of the nonce in step 4 and step 5 is to prevent replay attacks in which Mallory sends an old $E_{K_{BT}}[K_S \parallel ID_A]$ to Bob. If we didn't have step 4 and step 5, Bob would automatically assume that K_S is the correct key to use. Mallory could use this strategy to force Bob to send out more messages to Alice involving K_S . Step 4 and step 5 allow Bob to issue a challenge to Alice where she can prove to Bob that she really knows the session key K_S . Only Alice should be able to use K_S to calculate $E_{K_S}[r_2 - 1]$.

In spite of the the apparent security that the challenge-response in step 4 and step 5 provides, there is a potential security problem that can arise if Mallory ever figures out the session key K_S . Let us step through this possible attack.

1. Mallory \rightarrow Bob: $E_{K_{BT}}[K_S \parallel ID_A]$
2. Bob \rightarrow Alice: $E_{K_S}[r_3]$
3. Mallory \rightarrow Bob: $E_{K_S}[r_3 - 1]$

Here, Mallory replays an old message from step 3 of Needham-Schroeder as if Mallory were Alice. When Bob gets this message, he issues a challenge to Alice in the form of a new nonce r_3 . Mallory can intercept this challenge and, since she knows the session key K_S , she can respond correctly to the challenge. The net result is

that Mallory will have passed Bob's authentication challenge as if she were Alice. From this point on, Bob will communicate using K_S and believe he is communicating with Alice. Mallory can use Alice's identity to complete her evil plans.

Building a solid key distribution protocol is very tough. There are many examples in the security literature of key distribution schemes that have failed because of a clever attack that was found years later. It might seem a lost cause since we have examined two protocols that both have weaknesses associated with them. However, in the rest of this chapter we shall look at protocols that have so far proven successful. We begin our discussion of successful protocols in the next section, where we will discuss Kerberos, which is an improved variation of the Needham–Schroeder key exchange protocol. Kerberos has withstood careful scrutiny by the community and has been adopted for use in many applications.

15.3 Kerberos

Kerberos (named for the three-headed dog that guarded the entrance to Hades) is a real-world implementation of a symmetric cryptography protocol whose purpose is to provide strong levels of authentication and security in key exchange between users in a network. Here we use the term *users* loosely, as a user might be an individual, or it might be a program requesting communication with another program. Kerberos grew out of a larger development project at MIT known as Project Athena. The purpose of Athena was to provide a huge network of computer workstations for the undergraduate student body at MIT, allowing students to access their files easily from anywhere on the network. As one might guess, such a development quickly raised questions about network security. In particular, communication across a public network such as Athena is very insecure. It is easily possible to observe data flowing across a network and look for interesting bits of information such as passwords and other types of information that one would wish to remain private. Kerberos was developed in order to address such security issues. In the following, we present the basic Kerberos model and describe what it is and what it attempts to do. For more thorough descriptions, see [Schneier].

Kerberos is based on a client/server architecture. A client is either a user or some software that has some task that it seeks to accomplish. For example, a client might wish to send email, print documents, or mount devices. Servers are larger entities whose function is to provide services to the clients. As an example, on the Internet and World Wide Web there is a concept of a domain name server (DNS), which provides names or addresses to clients such as email programs or Internet browsers.

The basic Kerberos model has the following participants:

- Cliff: a client
- Serge: a server
- Trent: a trusted authority
- Grant: a ticket-granting server

The trusted authority is also known as an authentication server. To begin, Cliff and Serge have no secret key information shared between them, and it is the purpose of Kerberos to give each of them information securely. A result of the Kerberos protocol is that Serge will have verified Cliff's identity (he wouldn't want to have a conversation with a fake Cliff, would he?), and a session key will be established.

The protocol, depicted in [Figure 15.1](#), begins with Cliff requesting a ticket for ticket-granting service from Trent. Since Trent is the powerful trusted authority, he has a database of password information for all the clients (for this reason, Trent is also sometimes referred to as the Kerberos server). Trent returns a ticket that is encrypted with the client's secret password information. Cliff would now like to use the service that Serge provides, but before he can do this, he must be allowed to talk to Serge. Cliff presents his ticket to Grant, the ticket-granting server. Grant takes this ticket, and if everything is OK (recall that the ticket has some information identifying Cliff), then Grant gives a new ticket to Cliff that will allow Cliff to make use of Serge's service (and only Serge's service; this ticket will not be valid with Sarah, a different server). Cliff now has a service ticket, which he can present to Serge. He sends Serge the service ticket as well as an authentication credential. Serge checks the ticket with the authentication credential to make sure it is valid. If this final exchange checks out, then Serge will provide the service to Cliff.

Figure 15.1 Kerberos.

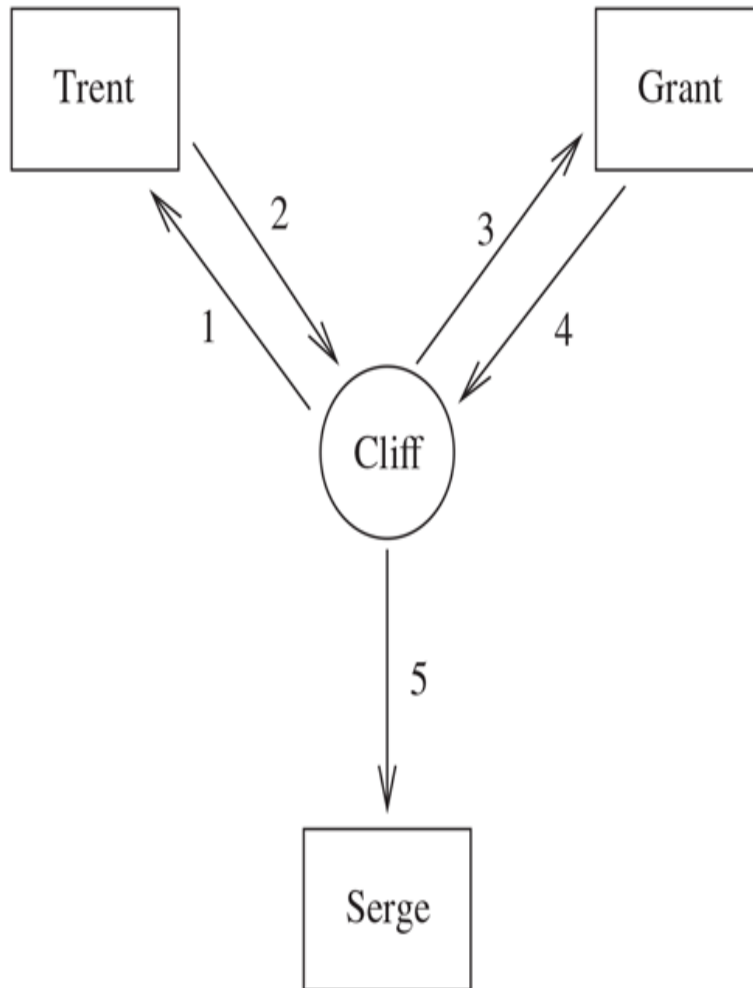


Figure 15.1 Full Alternative Text

The Kerberos protocol is a formal version of protocols we use in everyday life, where different entities are involved in authorizing different steps in a process; for example, using an ATM to get cash, then buying a ticket for a ride at a fair.

We now look at Kerberos in more detail. Kerberos makes use of a symmetric encryption algorithm. In the original release of Kerberos Version V, Kerberos used DES operating in CBC mode; however, Version V was later updated to allow for more general symmetric encryption algorithms.

1. Cliff to Trent: Cliff sends a message to Trent that contains his name and the name of the ticket-granting server that he will use (in this case Grant).
2. Trent to Cliff: Trent looks up Cliff's name in his database. If he finds it, he generates a session key K_{CG} that will be used between Cliff and Grant. Trent also has a secret key K_C with which he can communicate with Cliff, so he uses this to encrypt the Cliff–Grant session key:

$$T = e_{K_C}(K_{CG}).$$

In addition, Trent creates a Ticket Granting Ticket (TGT), which will allow Cliff to authenticate himself to Grant. This ticket is encrypted using Grant's personal key K_G (which Trent also has):

$$TGT = \text{Grant's name} \parallel e_{K_G}(\text{Cliff's name, Cliff's Address, Timestamp1, } K_{CG}).$$

Here \parallel is used to denote concatenation. The ticket that Cliff receives is the concatenation of these two subtickets:

$$\text{Ticket} = T \parallel TGT.$$

3. Cliff to Grant: Cliff can extract K_{CG} using the key K_C , which he shares with Trent. Using K_{CG} , Cliff can now communicate securely with Grant. Cliff now creates an authenticator, which will consist of his name, his address, and a timestamp. He encrypts this using K_{CG} to get

$$\text{Auth}_{CG} = e_{K_{CG}}(\text{Cliff's name, Cliff's address, Timestamp2}).$$

Cliff now sends Auth_{CG} as well as TGT to Grant so that Grant can administer a service ticket.

4. Grant to Cliff: Grant now has Auth_{CG} and TGT. Part of TGT was encrypted using Grant's secret key, so Grant can extract this portion and can decrypt it. Thus he can recover Cliff's name, Cliff's address, Timestamp1, as well as K_{CG} . Grant can now use K_{CG} to decrypt Auth_{CG} in order to verify the authenticity of Cliff's request. That is, $d_{K_{CG}}(\text{Auth}_{CG})$ will provide another copy of Cliff's name, Cliff's address, and a different timestamp. If the two versions of Cliff's name and address match, and if Timestamp1 and Timestamp2 are sufficiently close to each other, then Grant will declare Cliff valid. Now that Cliff is approved by Grant, Grant will generate a session key K_{CS} for Cliff to communicate with Serge and will also return a service ticket. Grant has a secret key K_S , which he shares with Serge. The service ticket is

$$\text{ServTicket} = e_{K_S}(\text{Cliff's name, Cliff's address, Timestamp3, ExpirationTime, } K_{CS}).$$

Here ExpirationTime is a quantity that describes the length of validity for this service ticket. The session key is encrypted using a session key between Cliff and Grant:

$$e_{K_{CG}}(K_{CS}).$$

Grant sends ServTicket and $e_{K_{CG}}(K_{CS})$ to Cliff.

5. Cliff to Serge: Cliff is now ready to start making use of Serge's services. He starts by decrypting $e_{K_{CG}}(K_{CS})$ in order to get the session key K_{CS} that he will use while communicating with Serge. He creates an authenticator to use with Serge:

$$\text{Auth}_{CS} = e_{K_{CS}}(\text{Cliff's name, Cliff's address, Timestamp4}).$$

Cliff now sends Serge Auth_{CS} as well as ServTicket. Serge can decrypt ServTicket and extract from this the session key K_{CS} that he is to use with Cliff. Using this session key, he can decrypt Auth_{CS} and verify that Cliff is who he says he is, and that Timestamp4 is within ExpirationTime of Timestamp3. If Timestamp4 is not within ExpirationTime of Timestamp3, then Cliff's ticket is stale and Serge rejects his request for service. Otherwise, Cliff and Serge may make use of K_{CS} to perform their exchange.

Some comments about the different versions of Kerberos is appropriate. There are two main versions of Kerberos that one will find discussed in the literature: Kerberos Version IV, which was developed originally as part of MIT's Project Athena; and Kerberos Version V, which was originally published in 1993 and was intended to address limitations and weaknesses in Version IV. Kerberos Version V was subsequently revised in 2005. We now describe some of the differences between the two versions.

Both Version IV and Version V follow the basic model that we have presented. Kerberos Version IV was designed to work with the computer networks associated with Project Athena, and consequently Version V was enhanced to support authentication and key exchange on larger networks. In particular, Kerberos Version IV was limited in its use to Internet Protocol version 4 (IPV4) addresses to specify clients and servers, while Kerberos Version V expanded the use of network protocols to support multiple IP addresses and addresses associated with other types of network protocols, such as the longer addresses of Internet Protocol version 6 (IPV6).

In terms of the cryptography that Kerberos uses, Version IV only allowed the use of the Data Encryption Standard (DES) with a nonstandard mode of operation known as Propagating Cipher Block Chaining (PCBC). Due to concerns about the security of PCBC mode, it was removed in Version V. Instead, Version V allows for the use of more general symmetric encryption algorithms, such as the use of AES, and incorporated integrity checking mechanisms directly into its use of the normal CBC mode of operation.

The tickets used in Kerberos Version IV had strict limits to their duration. In Kerberos Version V, the representation for the tickets was expanded to include start and stop times for the certificates. This allows for tickets in Version V to be specified with arbitrary durations. Additionally, the functionality of tickets in Version V was expanded. In Version V, it is possible for servers to forward, renew, and postdate tickets, while in Version IV authentication forwarding is not allowed. Overall, the changes made to Version V were intended to make Kerberos more secure and more flexible, allowing it to operate in more types of networks, support a broader array of cipher algorithms, and allow for it to address a broader set of application requirements.

15.4 Public Key Infrastructures (PKI)

Public key cryptography is a powerful tool that allows for authentication, key distribution, and non-repudiation. In these applications, the public key is published, but when you access public keys, what assurance do you have that Alice's public key actually belongs to Alice? Perhaps Eve has substituted her own public key in place of Alice's. Unless confidence exists in how the keys were generated, and in their authenticity and validity, the benefits of public key cryptography are minimal.

In order for public key cryptography to be useful in commercial applications, it is necessary to have an infrastructure that keeps track of public keys. A public key infrastructure, or PKI for short, is a framework consisting of policies defining the rules under which the cryptographic systems operate and procedures for generating and publishing keys and certificates.

All PKIs consist of certification and validation operations. Certification binds a public key to an entity, such as a user or a piece of information. Validation guarantees that certificates are valid.

A **certificate** is a quantity of information that has been signed by its publisher, who is commonly referred to as the **certification authority (CA)**. There are many types of certificates. Two popular ones are identity certificates and credential certificates. Identity certificates contain an entity's identity information, such as an email address, and a list of public keys for the entity. Credential certificates contain information describing access rights. In either case, the data are typically encrypted using the CA's private key.

Suppose we have a PKI, and the CA publishes identity certificates for Alice and Bob. If Alice knows the CA's public key, then she can take the encrypted identity certificate for Bob that has been published and extract Bob's identity information as well as a list of public keys needed to communicate securely with Bob. The difference between this scenario and the conventional public key scenario is that Bob doesn't publish his keys, but instead the trust relationship is placed between Alice and the publisher. Alice might not trust Bob as much as she might trust a CA such as the government or the phone company. The concept of trust is critical to PKIs and is perhaps one of the most important properties of a PKI.

It is unlikely that a single entity could ever keep track of and issue every Internet user's public keys. Instead, PKIs often consist of multiple CAs that are allowed to certify each other and the certificates they issue. Thus, Bob might be associated with a different CA than Alice, and when requesting Bob's identity certificate, Alice might only trust it if her CA trusts Bob's CA. On large networks like the Internet, there may be many CAs between Alice and Bob, and it becomes necessary for each of the CAs between her and Bob to trust each other.

In addition, most PKIs have varying levels of trust, allowing some CAs to certify other CAs with varying degrees of trust. It is possible that CAs may only trust other CAs to perform specific tasks. For example, Alice's CA may only trust Bob's CA to certify Bob and not certify other CAs, while Alice's CA may trust Dave's CA to certify other CAs. Trust relationships can become very elaborate, and, as these relationships become more complex, it becomes more difficult to determine to what degree Alice will trust a certificate that she receives.

In the following two sections, we discuss two examples of PKIs that are used in practice.

15.5 X.509 Certificates

Suppose you want to buy something on the Internet. You go to the website Gigafirm.com, select your items, and then proceed to the checkout page. You are asked to enter your credit card number and other information. The website assures you that it is using secure public key encryption, using Gigafirm's public key, to set up the communications. But how do you know that Eve hasn't substituted her public key? In other words, when you are using public keys, how can you be sure that they are correct? This is the purpose of Digital Certificates.

One of the most popular types of certificate is the X.509. In this system, every user has a certificate. The validity of the certificates depends on a chain of trust. At the top is a **certification authority** (CA). These are often commercial companies such as VeriSign, GTE, AT&T, and others. It is assumed that the CA is trustworthy. The CA produces its own certificate and signs it. This certificate is often posted on the CA's website. In order to ensure that their services are used frequently, various CAs arrange to have their certificates packaged into Internet browsers such as Chrome, Firefox, Safari, Internet Explorer, and Edge.

The CA then (for a fee) produces certificates for various clients, such as Gigafirm. Such a certificate contains Gigafirm's public key. It is signed by the CA using the CA's private key. Often, for efficiency, the CA authorizes various **registration authorities** (RA) to sign certificates. Each RA then has a certificate signed by the CA.

A certificate holder can sometimes then sign certificates for others. We therefore get a **certification hierarchy**

where the validity of each certificate is certified by the user above it, and this continues all the way up to the CA.

If Alice wants to verify that Gigafirm's public key is correct, she uses her copy of the CA's certificate (stored in her computer) to get the CA's public key. She then uses it to verify the signature on Gigafirm's certificate. If it is valid, she trusts the certificate and thus has a trusted public key for Gigafirm. Of course, she must trust the CA's public key. This means that she trusts the company that packaged the CA's certificate into her computer. The computer company of course has a financial incentive to maintain a good reputation, so this trust is reasonable. But if Alice has bought a used computer in which Eve has tampered with the certificates, there might be a problem (in other words, don't buy used computers from your enemies, except to extract unerased information).

Figure 15.2 A Certification Hierarchy.

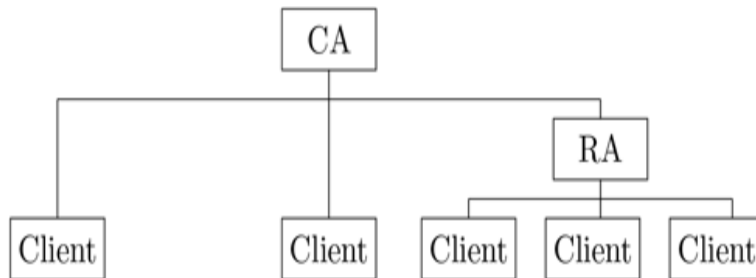


Figure 15.2 Full Alternative Text

Figures 15.3, 15.4, and 15.5 show examples of X.509 certificates. The ones in Figures 15.3 and 15.4 are for a CA, namely VeriSign. The part in Figure 15.3 gives the general information about the certificate, including its possible uses. Figure 15.4 gives the detailed information. The one in Figure 15.5 is an edited version of the Details part of a certificate for the bank Wells Fargo.

Figure 15.3 CA's Certificate; General.

This certificate has been verified for the following uses:	
Email Signer Certificate	
Email Recipient Certificate	
Status Responder Certificate	
Issued to:	
Organization (O): VeriSign, Inc.	
Organizational Unit (OU): Class 1 Public Primary Certification Authority - G2	
Serial Number: 39:CA:54:89:FE:50:22:32:FE:32:D9:DB:FB:1B:84:19	
Issued By:	
Organization (O): VeriSign, Inc.	
Organizational Unit (OU): Class 1 Public Primary Certification Authority - G2	
Validity:	
Issued On: 05/17/98	
Expires On: 05/18/18	
Fingerprints:	
SHA1 Fingerprint: 04:98:11:05:6A:FE:9F:D0:F5:BE:01:68:5A:AC:E6:A5:D1:C4:45:4C	
MD5 Fingerprint: F2:7D:E9:54:E4:A3:22:0D:76:9F:E7:0B:BB:B3:24:2B	

Figure 15.3 Full Alternative Text

Figure 15.4 CA's Certificate; Details.

Certificate Hierarchy
▸ Verisign Class 1 Public Primary Certification Authority - G2
Certificate Fields
Verisign Class 1 Public Primary Certification Authority - G2 Certificate Version: Version 1 Serial Number: 39:CA:54:89:FE:50:22:32:FE:32:D9:DB:FB:1B:84:19 Certificate Signature Algorithm: PKCS #1 SHA-1 With RSA Encryption Issuer: OU = VeriSign Trust Network OU = (c) 1998 VeriSign, Inc. - For authorized use only OU = Class 1 Public Primary Certification Authority - G2 O = VeriSign, Inc. C = US Validity Not Before: 05/17/98 20:00:00 (05/18/98 00:00:00 GMT) Not After: 05/18/18 19:59:59 (05/18/18 23:59:59 GMT) Subject: OU = VeriSign Trust Network OU = (c) 1998 VeriSign, Inc. - For authorized use only OU = Class 1 Public Primary Certification Authority - G2 O = VeriSign, Inc. C = US Subject Public Key Info: PKCS #1 RSA Encryption Subject's Public Key: 30 81 89 02 81 81 00 aa d0 ba be 16 2d b8 83 d4 ca d2 0f bc 76 31 ca 94 d8 1d 93 8c 56 02 bc d9 6f 1a 6f 52 36 6e 75 56 0a 55 d3 df 43 87 21 11 65 8a 7e 8f bd 21 de 6b 32 3f 1b 84 34 95 05 9d 41 35 eb 92 eb 96 dd aa 59 3f 01 53 6d 99 4f ed e5 e2 2a 5a 90 c1 b9 c4 a6 15 cf c8 45 eb a6 5d 8e 9c 3e f0 64 24 76 a5 cd ab 1a 6f b6 d8 7b 51 61 6e a6 7f 87 c8 e2 b7 e5 34 dc 41 88 ea 09 40 be 73 92 3d 6b e7 75 02 03 01 00 01 Certificate Signature Algorithm: PKCS #1 SHA-1 With RSA Encryption Certificate Signature Value: 8b f7 1a 10 ce 76 5c 07 ab 83 99 dc 17 80 6f 34 39 5d 98 3e 6b 72 2c e1 c7 a2 7b 40 29 b9 78 88 ba 4c c5 a3 6a 5e 9e 6e 7b e3 f2 02 41 0c 66 be ad fb ae a2 14 ce 92 f3 a2 34 8b b4 b2 b6 24 f2 e5 d5 e0 c8 e5 62 6d 84 7b cb be bb 03 8b 7c 57 ca f0 37 a9 a0 af 8a aa 03 ba 1d 28 ac d9 26 76

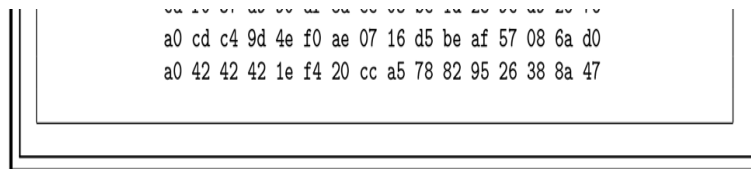


Figure 15.4 Full Alternative Text

Figure 15.5 A Client's Certificate.

Certificate Hierarchy

- Verisign Class 3 Public Primary CA
 - www.verisign.com/CPS Incorp. by Ref. LIABILITY LTD.(c)97VeriSign
 - online.wellsfargo.com

Certificate Fields

Verisign Class 3 Public Primary Certification Authority
Certificate
Version: Version 3
Serial Number: 03:D7:98:CA:98:59:30:B1:B2:D3:BD:28:B8:E7:2B:8F
Certificate Signature Algorithm: md5RSA
Issuer: OU = www.verisign.com/CPS Incorp. ...

- OU = VeriSign International Server CA - Class 3
- OU = VeriSign, Inc.
- O = VeriSign Trust Network
- C = US

Validity

Not Before: Sunday, September 21, 2003 7:00:00 PM
Not After: Wednesday, September 21, 2005 6:59:59 PM

Subject: CN = online.wellsfargo.com

- OU = Terms of use at www.verisign.com.rpa (c)00
- OU = Class 1 Public Primary Certification Authority - G2
- OU = ISG
- O = Wells Fargo and Company
- L = San Francisco
- S = California
- C = US

Subject Public Key Info: PKCS #1 RSA Encryption

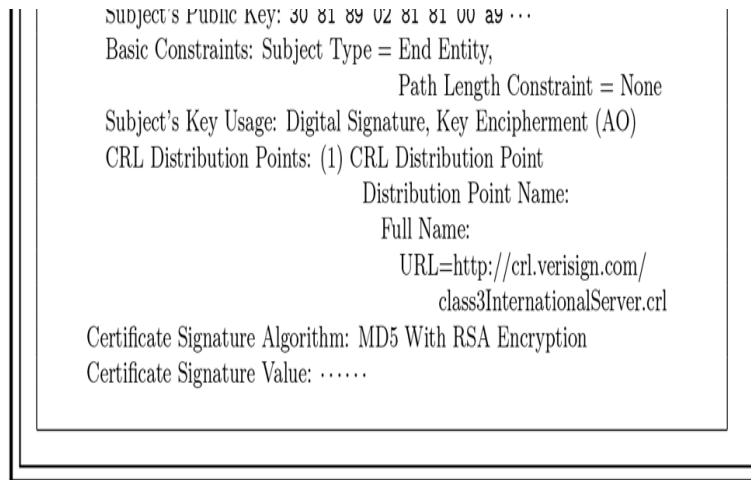


Figure 15.5 Full Alternative Text

Some of the fields in Figure 15.4 are as follows:

1. Version: There are three versions, the first being *Version 1* (from 1988) and the most recent being *Version 3* (from 1997).
2. Serial number: There is a unique serial number for each certificate issued by the CA.
3. Signature algorithm: Various signature algorithms can be used. This one uses *RSA* to sign the output of the hash function *SHA-1*.
4. Issuer: The name of the CA that created and signed this certificate. *OU* is the organizational unit, *O* is the organization, *C* is the country.
5. Subject: The name of the holder of this certificate.
6. Public key: Several options are possible. This one uses *RSA* with a 1024-bit modulus. The key is given in hexadecimal notation. In hexadecimal, the letters *a, b, c, d, e, f* represent the numbers 10, 11, 12, 13, 14, 15. Each pair of symbols is a byte, which is eight bits. For example, *b6* represents 11, 6, which is 10110110 in binary.

The last three bytes of the public key are 01 00 01, which is $65537 = 2^{16} + 1$. This is a very common encryption exponent e for *RSA*, since raising something to this power by successive squaring (see Section 3.5) is fast. The preceding bytes 02 03 and the bytes 30 81 89 02 81 81 00 at the beginning of the key are control symbols. The remaining 128 bytes aa d0 ba ... 6b e7 75 are the 1024-bit *RSA* modulus n .

7. Signature: The preceding information on the certificate is hashed using the hash algorithm specified – in this case, *SHA-1* – and then signed by raising to the CA's private *RSA* decryption exponent.

The certificate in [Figure 15.5](#) has a few extra lines. One notable entry is under the heading *Certificate Hierarchy*. The certificate of Wells Fargo has been signed by the Registration Authority (RA) on the preceding line. In turn, the RA's certificate has been signed by the root CA. Another entry worth noting is *CRL Distribution Points*. This is the **certificate revocation list**. It contains lists of certificates that have been revoked. There are two common methods of distributing the information from these lists to the users. Neither is perfect. One way is to send out announcements whenever a certificate is revoked. This has the disadvantage of sending a lot of irrelevant information to most users (most people don't need to know if the Point Barrow Sunbathing Club loses its certificate). The second method is to maintain a list (such as the one at the listed URL) that can be accessed whenever needed. The disadvantage here is the delay caused by checking each certificate. Also, such a website could get overcrowded if many people try to access it at once. For example, if everyone tries to trade stocks during their lunch hour, and the computers check each certificate for revocation during each transaction, then a site could be overwhelmed.

When Alice (or, usually, her computer) wants to check the validity of the certificate in [Figure 15.5](#), she sees from the certificate hierarchy that VeriSign's RA signed Wells Fargo's certificate and the RA's certificate was signed by the root CA. She verifies the signature on Wells Fargo's certificate by using the public key (that is, the RSA pair (n, e)) from the RA's certificate; namely, she raises the encrypted hash value to the e th power mod n . If this equals the hash of Wells Fargo's certificate, then she trusts Wells Fargo's certificate, as long as she trusts the RA's certificate. Similarly, she can check the RA's certificate using the public key on the root CA's certificate. Since she received the root CA's certificate from a reliable source (for example, it was packaged in her Internet browser, and the company doing this has a

financial incentive to keep a good reputation), she trusts it. In this way, Alice has established the validity of Wells Fargo's certificate. Therefore, she can confidently do online transactions with Wells Fargo.

There are two levels of certificates. The **high assurance** certificates are issued by the CA under fairly strict controls. High assurance certificates are typically issued to commercial firms. The **low assurance** certificates are issued more freely and certify that the communications are from a particular source. Therefore, if Bob obtains such a certificate for his computer, the certificate verifies that it is Bob's computer but does not tell whether it is Bob or Eve using the computer. The certificates on many personal computers contain the following line:

Subject: Verisign Class 1 CA Individual Subscriber -
Persona Not Validated.

This indicates that the certificate is a low assurance certificate. It does not make any claim as to the identity of the user.

If your computer has Edge, for example, click on *Tools*, then *Internet Options*, then *Content*, then *Certificates*. This will allow you to find the CA's whose certificates have been packaged with the browser. Usually, the validity of most of them has not been checked. But for the accepted ones, it is possible to look at the **certification path**, which gives the path (often one step) from the user's computer's certificate back to the CA.

15.6 Pretty Good Privacy

Pretty Good Privacy, more commonly known as *PGP*, was developed by Phil Zimmerman in the late 1980s and early 1990s. In contrast to X.509 certificates, PGP is a very decentralized system with no CA. Each user has a certificate, but the trust in this certificate is certified to various degrees by other users. This creates a **web of trust**.

For example, if Alice knows Bob and can verify directly that his certificate is valid, then she signs his certificate with her public key. Charles trusts Alice and has her public key, and therefore can check that Alice's signature on Bob's certificate is valid. Charles then trusts Bob's certificate. However, this does not mean that Charles trusts certificates that Bob signs – he trusts Bob's public key. Bob could be gullible and sign every certificate that he encounters. His signature would be valid, but that does not mean that the certificate is.

Alice maintains a file with a **keyring** containing the trust levels Alice has in various people's signatures. There are varying levels of trust that she can assign: no information, no trust, partial trust, and complete trust. When a certificate's validity is being judged, the PGP program accepts certificates that are signed by someone Alice trusts, or a sufficient combination of partial trusts. Otherwise it alerts Alice and she needs to make a choice on whether to proceed.

The primary use of PGP is for authenticating and encrypting email. Suppose Alice receives an email asking for her bank account number so that Charles can transfer millions of dollars into her account. Alice wants to be sure that this email comes from Charles and not from

Eve, who wants to use the account number to empty Alice's account. In the unlikely case that this email actually comes from her trusted friend Charles, Alice sends her account information, but she should encrypt it so that Eve cannot intercept it and empty Alice's account. Therefore, the first email needs authentication that proves that it comes from Charles, while the second needs encryption. There are also cases where both authentication and encryption are desirable. We'll show how PGP handles these situations.

To keep the discussion consistent, we'll always assume that Alice is sending a message to Bob. Alice's RSA public key is (n, e) and her private key is d .

Authentication.

1. Alice uses a hash function and computes the hash of the message.
2. Alice signs the hash by raising it to her secret decryption exponent $d \bmod n$. The resulting hash code is put at the beginning of the message, which is sent to Bob.
3. Bob raises the hash code to Alice's public RSA exponent e . The result is compared to the hash of the rest of the message.
4. If the result agrees with the hash, and if Bob trusts Alice's public key, the message is accepted as coming from Alice.

This authentication is the RSA signature method from [Section 13.1](#). Note the role that trust plays. If Bob does not trust Alice's public key as belonging to Alice, then he cannot be sure that the message did not come from Eve, with Eve's signature in place of Alice's.

Encryption.

1. Alice's computer generates a random number, usually 128 bits, to be used as the session key for a symmetric private key encryption algorithm such as *3DES*, *IDEA*, or *CAST-128* (these last two are block ciphers using 128-bit keys).
2. Alice uses the symmetric algorithm with this session key to encrypt her message.
3. Alice encrypts the session key using Bob's public key.
4. The encrypted key and the encrypted message are sent to Bob.

5. Bob uses his private RSA key to decrypt the session key. He then uses the session key to decrypt Alice's message.

The combination of a public key algorithm and a symmetric algorithm is used because encryption is generally faster with symmetric algorithms than with public key algorithms. Therefore, the public key algorithm RSA is used for the small encryption of the session key, and then the symmetric algorithm is used to encrypt the potentially much larger message.

Authentication and Encryption

1. Alice hashes her message and signs the hash to obtain the hash code, as in step (2) of the authentication procedure described previously. This hash code is put at the beginning of the message.
2. Alice produces a random 128-bit session key and uses a symmetric algorithm with this session key to encrypt the hash code together with the message, as in the encryption procedure described previously.
3. Alice uses Bob's public key to encrypt the session key.
4. The encrypted session key and the encryption of the hash code and message are sent to Bob.
5. Bob uses his private key to decrypt the session key.
6. Bob uses the session key to obtain the hash code and message.
7. Bob verifies the signature by using Alice's public key, as in the authentication procedure described previously.

Of course, this procedure requires that Bob trusts Alice's public key certificate. Also, the reason the signature is done before the encryption is so that Bob can discard the session key after decrypting and therefore store the plaintext message with its signature.

To set up a PGP certificate, Alice's computer uses random input obtained from keystrokes, timing, mouse movements, etc. to find primes p , q and then produce an RSA modulus $n = pq$ and encryption and decryption exponents e and d . The numbers n and e are then Alice's public key. Alice also chooses a secret passphrase. The secret key d is stored securely in her computer. When the computer needs to use her private key, the computer asks her for her passphrase to be sure that Alice is the correct person. This prevents Eve from using Alice's

computer and pretending to be Alice. The advantage of the passphrase is that Alice is not required to memorize or type in the decryption exponent d , which is probably more than one hundred digits long.

In the preceding, we have used RSA for signatures and for encryption of the session keys. Other possibilities are allowed. For example, Diffie-Hellman can be used to establish the session key, and DSA can be used to sign the message.

15.7 SSL and TLS

If you have ever paid for anything over the Internet, your transactions were probably kept secret by SSL or its close relative TLS. Secure Sockets Layer (SSL) was developed by Netscape in order to perform http communications securely. The first version was released in 1994. Version 3 was released in 1995. Transport Layer Security (TLS) is a slight modification of SSL version 3 and was released by the Internet Engineering Task Force in 1999. These protocols are designed for communications between computers with no previous knowledge of each other's capabilities.

In the following, we'll describe SSL version 3. TLS differs in a few minor details such as how the pseudorandom numbers are calculated. SSL consists of two main components. The first component is known as the record protocol and is responsible for compressing and encrypting the bulk of the data sent between two entities. The second component is a collection of management protocols that are responsible for setting up and maintaining the parameters used by the record protocol. The main part of this component is called the handshake protocol.

We will begin by looking at the handshake protocol, which is the most complicated part of SSL. Let us suppose that Alice has bought something online from Gigafirm and wants to pay for her purchase. The handshake protocol performs authentication between Alice's computer and the server at Gigafirm and is used to allow Alice and Gigafirm to agree upon various cryptographic algorithms. Alice's computer starts by sending Gigafirm's computer a message containing the following:

1. The highest version of SSL that Alice's computer can support
2. A random number consisting of a 4-byte timestamp and a 28-byte random number
3. A Cipher Suite containing, in decreasing order of preference, the algorithms that Alice's computer wants to use for public key (for example, RSA, Diffie-Hellman, ...), block cipher encryption (3DES, DES, AES, ...), hashing (SHA-1, MD5, ...), and compression (PKZip, ...)

Gigafirm's computer responds with a random 32-byte number (chosen similarly) and its choices of which algorithms to use; for example, RSA, DES, SHA-1, PKZip.

Gigafirm's computer then sends its X.509 certificate (and the certificates in its certification chain). Gigafirm can ask for Alice's certificate, but this is rarely done for two reasons. First, it would impede the transaction, especially if Alice does not have a valid certificate. This would not help Gigafirm accomplish its goal of making sales. Secondly, Alice is going to send her credit card number later in the transaction, and this serves to verify that Alice (or the thief who picked her pocket) has Alice's card.

We'll assume from now on that RSA was chosen for the public key method. The protocol differs only slightly for other public key methods.

Alice now generates a 48-byte *pre-master secret*, encrypts it with Gigafirm's public key (from its certificate), and sends the result to Gigafirm, who decrypts it. Both Alice and Gigafirm now have the following secret random numbers:

1. The 32-byte random number r_A that Alice sent Gigafirm.
2. The 32-byte random number r_G that Gigafirm sent Alice.
3. The 48-byte pre-master secret s_{pm} .

Note that the two 32-byte numbers were not sent securely. The pre-master secret is secure, however.

Since they both have the same numbers, both Alice and Gigafirm can calculate the *master secret* as the concatenation of

$$\begin{aligned} & \text{MD5}(s_{pm} || \text{SHA} - 1(A || s_{pm} || r_A || r_G)) \\ & \text{MD5}(s_{pm} || \text{SHA} - 1(BB || s_{pm} || r_A || r_G)) \\ & \text{MD5}(s_{pm} || \text{SHA} - 1(CCC || s_{pm} || r_A || r_G)). \end{aligned}$$

The A , BB , and CCC are strings added for padding. Note that timestamps are built into r_A and r_G . This prevents Eve from doing replay attacks, where she tries to use information intercepted from one session to perform similar transactions later.

Since MD5 produces a 128-bit (= 16-byte) output, the master secret has 48 bytes. The master secret is used to produce a *key block*, by the same process that the master secret was produced from the pre-master secret. Enough hashes are concatenated to produce a sufficiently long key block. The key block is then cut into six secret keys, three for communications from Alice to Gigafirm and three for communications from Gigafirm to Alice. For Alice to Gigafirm, one key serves as the secret key in the block cipher (3DES, AES, ...) chosen at the beginning of the communications. The second is a message authentication key. The third is the initial value for the CBC mode of the block cipher. The three other keys are for the corresponding purposes for Gigafirm to Alice.

Now Alice and Gigafirm are ready to communicate using the record protocol. When Alice sends a message to Gigafirm, she does the following:

1. Compresses the message using the agreed-upon compression method.
2. Hashes the compressed message together with the message authentication key (the second key obtained from the key block). This yields the hashed message authentication code.
3. Uses the block cipher in CBC mode to encrypt the compressed message together with the hashed message authentication code, and sends the result to Gigafirm.

Gigafirm now does the following:

1. Uses the block cipher to decrypt the message received. Gigafirm now has the compressed message and the hashed message authentication code.
2. Uses the compressed message and the Alice-to-Gigafirm message authentication key to recompute the hashed message authentication code. If it agrees with the hashed message authentication code that was in the message, the message is authenticated.
3. Decompresses the compressed message to obtain Alice's message.

Communications from Gigafirm are encrypted and decrypted similarly, using the other three keys deduced from the key block. Therefore, Alice and Gigafirm can exchange information securely.

15.8 Secure Electronic Transaction

Every time someone places an order in an electronic transaction over the Internet, large quantities of information are transmitted. These data must be protected from unwanted eavesdroppers in order to ensure the customer's privacy and prevent credit fraud. Requirements for a good electronic commerce system include the following:

1. Authenticity: Participants in a transaction cannot be impersonated and signatures cannot be forged.
2. Integrity: Documents such as purchase orders and payment instructions cannot be altered.
3. Privacy: The details of a transaction should be kept secure.
4. Security: Sensitive account information such as credit card numbers must be protected.

All of these requirements should be satisfied, even over public communication channels such as the Internet.

In 1996, the credit card companies MasterCard and Visa called for the establishment of standards for electronic commerce. The result, whose development involved several companies, is called the SET, or Secure Electronic TransactionTM protocol. It starts with the existing credit card system and allows people to use it securely over open channels.

The SET protocol is fairly complex, involving, for example, the SSL protocol in order to certify that the cardholder and merchant are legitimate and also specifying how payment requests are to be made. In the

following we'll discuss one aspect of the whole protocol, namely the use of dual signatures.

There are several possible variations on the following. For example, in order to improve speed, a fast symmetric key system can be used in conjunction with the public key system. If there is a lot of information to be transmitted, a randomly chosen symmetric key plus the hash of the long message can be sent via the public key system, while the long message itself is sent via the faster symmetric system. However, we'll restrict our attention to the simplest case where only public key methods are used.

Suppose Alice wants to buy a book entitled How to Use Other People's Credit Card Numbers to Defraud Banks, which she has seen advertised on the Internet. For obvious reasons, she feels uneasy about sending the publisher her credit card information, and she certainly does not want the bank that issued her card to know what she is buying. A similar situation applies to many transactions. The bank does not need to know what the customer is ordering, and for security reasons the merchant should not know the card number. However, these two pieces of information need to be linked in some way. Otherwise the merchant could attach the payment information to another order. **Dual signatures** solve this problem.

The three participants in the following will be the Cardholder (namely, the purchaser), the Merchant, and the Bank (which authorizes the use of the credit card).

The Cardholder has two pieces of information:

- GSO = Goods and Service Order, which consists of the cardholder's and merchant's names, the quantities of each item ordered, the prices, etc.
- PI = Payment Instructions, including the merchant's name, the credit card number, the total price, etc.

The system uses a public hash function; let's call it H . Also, a public key cryptosystem such as RSA is used, and the Cardholder and the Bank have their own public and private keys. Let E_C , E_M , and E_B denote the (public) encryption functions for the Cardholder, the Merchant, and the Bank, and let D_C , D_M , and D_B be the (private) decryption functions.

The Cardholder performs the following procedures:

1. Calculates $GSOMD = H(E_M(GSO))$, which is the message digest, or hash, of an encryption of GSO .
2. Calculates $PIMD = H(E_B(PI))$, which is the message digest of an encryption of PI .
3. Concatenates $GSOMD$ and $PIMD$ to obtain $PIMD||GSOMD$, then computes the hash of the result to obtain the payment-order message digest $POMD = H(PIMD||GSOMD)$.
4. Signs $POMD$ by computing $DS = D_C(POMD)$. This is the dual signature.
5. Sends $E_M(GSO)$, DS , $PIMD$, and $E_B(PI)$ to the Merchant.

The Merchant then does the following:

1. Calculates $H(E_M(GSO))$ (which should equal $GSOMD$).
2. Calculates $H(PIMD||H(E_M(GSO)))$ and $E_C(DS)$. If they are equal, then the Merchant has verified the Cardholder's signature and is therefore convinced that the order is from the Cardholder.
3. Computes $D_M(E_M(GSO))$ to obtain GSO .
4. Sends $GSOMD$, $E_B(PI)$, and DS to the Bank.

The Bank now performs the following:

1. Computes $H(E_B(PI))$ (which should equal $PIMD$).
2. Concatenates $H(E_B(PI))$ and $GSOMD$.
3. Computes $H(H(E_B(PI))||GSOMD)$ and $E_C(DS)$. If they are equal, the Bank has verified the Cardholder's signature.
4. Computes $D_B(E_B(PI))$, obtaining the payment instructions PI .

5. Returns an encrypted (with E_M) digitally signed authorization to the Merchant, guaranteeing payment.

The Merchant completes the procedure as follows:

1. Returns an encrypted (with E_C) digitally signed receipt to the Cardholder, indicating that the transaction has been completed.

The Merchant only sees the encrypted form $E_B(PI)$ of the payment instructions, and so does not see the credit card number. It would be infeasible for the Merchant or the Bank to modify any of the information regarding the order because the hash function is used to compute DS .

The Bank only sees the message digest of the Goods and Services Order, and so has no idea what is being ordered.

The requirements of integrity, privacy, and security are met by this procedure. In actual implementations, several more steps are required in order to protect authenticity. For example, it must be guaranteed that the public keys being used actually belong to the participants as claimed, not to impostors. Certificates from a trusted authority are used for this purpose.

15.9 Exercises

1. In a network of three users, A, B, and C, we would like to use the Blom scheme to establish session keys between pairs of users. Let $p = 31$ and let

$$r_A = 11 \quad r_B = 3 \quad r_C = 2.$$

Suppose Trent chooses the numbers

$$a = 8 \quad b = 3 \quad c = 1.$$

Calculate the session keys.

- 2.
1. Show that in the Blom scheme,
$$K_{AB} \equiv a + b(r_A + r_B) + cr_Ar_B \pmod{p}.$$
 2. Show that $K_{AB} = K_{BA}$.
 3. Another way to view the Blom scheme is by using a polynomial in two variables. Define the polynomial $f(x, y) = a + b(x + y) + cxy \pmod{p}$. Express the key K_{AB} in terms of f .
3. You (U) and I (I) are evil users on a network that uses the Blom scheme for key establishment with $k = 1$. We have decided to get together to figure out the other session keys on the network. In particular, suppose $p = 31$ and $r_U = 9$, $r_I = 2$. We have received $a_U = 18$, $b_U = 29$, $a_I = 24$, $b_I = 23$ from Trent, the trusted authority. Calculate a , b , and c .
4. Here is another version of the intruder-in-the-middle attack on the Diffie-Hellman key exchange in [Section 10.1](#). It has the “advantage” that Eve does not have to intercept and retransmit all the messages between Bob and Alice. Suppose Eve discovers that $p = Mq + 1$, where q is an integer and M is small. Eve intercepts α^x and α^y as before. She sends Bob $(\alpha^x)^q \pmod{p}$ and sends Alice $(\alpha^y)^q \pmod{p}$.
1. Show that Alice and Bob each calculate the same key K .
 2. Show that there are only M possible values for K , so Eve may find K by exhaustive search.
5. Bob, Ted, Carol, and Alice want to agree on a common key (cryptographic key, that is). They publicly choose a large prime p and a primitive root α . They privately choose random numbers b, t, c, a , respectively. Describe a protocol that allows them to

compute $K \equiv \alpha^{bta} \pmod{p}$ securely (ignore intruder-in-the-middle attacks).

6. Suppose naive Nelson tries to implement an analog of the three-pass protocol of [Section 3.6](#) to send a key K to Heidi. He chooses a one-time pad key K_N and XORs it with K . He sends $M_1 = K_N \oplus K$ to Heidi. She XORs what she receives with her one-time pad key K_H to get $M_2 = M_1 \oplus K_H$. Heidi sends M_2 to Nelson, who computes $M_3 = M_2 \oplus K_N$. Nelson sends M_3 to Heidi, who recovers K as $M_3 \oplus K_H$.

1. Show that $K = M_3 \oplus K_H$.
2. Suppose Eve intercepts M_1 , M_2 , M_3 . How can she recover K ?