Chapter 10

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 to ask at this point is: Can we just apply these tools directly to secure computers and communications?

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 securely exchange messages. 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 twin, Mallory, is pretending to be Alice but is actually announcing Mallory's public key instead. Similarly, when you access a web site 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 really confirm that he is communicating with Alice before sending any important information.

Combining different cryptographic tools together 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.

10.1 Intruders-in-the-Middle and Impostors

If you receive an email asking you to go to a web site and update your account information, how can you be sure that the web site is legitimate? An impostor can easily set up a web page that looks like the correct one, but which 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 10.7. Authentication will also play an important role in the protocols in many other sections of this chapters.

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.

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 7.4.

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^x and Bob believes he is receiving α^y).

4. Eve computes $K_{AO} \equiv (\alpha^x)^z \pmod{p}$ and $K_{OB} \equiv (\alpha^x)^z \pmod{p}$. Alice, not realizing that Eve is in the middle, also computes K_{AO}, and Bob computes K_{OB}.

5. When Alice sends a message to Bob, encrypted with K_{AO}, Eve intercepts it, deciphers it, encrypts it with K_{OB}, and sends it to Bob. Bob decrypts with K_{OB} 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.

10.2 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 a DES 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 DES 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 on 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 DES. Therefore, sometimes RSA is used to transmit a DES 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 7.4 and 10.1) is an example of key agreement. Using RSA to transmit a DES 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 basic protocols for key distribution using a third party. Solutions that are more practical for Internet communcations are treated in later sections of this chapter.

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 mod 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, DES, 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 $a = 8, \quad b = 3, \quad c = 1$. 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

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

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), Oscar has the matrix equation

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

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].

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 easily changed or replaced 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 securely distribute keys.

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 who they think they are talking to.

One of the key 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 hopes to pretend to be another party, or to elicit 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 securely distribute session keys, 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 only be used once, and is used as part of a challenge-response mechanism. In a challenge-response, Alice sends Bob a nonce, and waits for 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 securely distribute keys.

We begin with a protocol known as the Wide-Mouthed Frog Protocol. The Wide-Mouthed Frog Protocol is one of the simplest symmetric key management protocols involving a trusted authority. In the Wide-Mouthed Frog Protocol, Alice chooses a session key K_{AB} to communicate with Bob and has Trent transfer it to Bob securely.

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

Here, K_{AT} is a key shared between Alice and Trent, while K_{BT} is a key shared between Bob and Trent. 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 in 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 could pretend to be Bob wanting to share a key with Alice. Mallory would send Trent the replay $E_{K_{BT}}[t_T \| ID_A \| K_{AB}]$.
2. Trent would send back $E_{K_{AT}}\left[t_T' \| ID_B \| K_{AB}\right]$, where t_T' is a new timestamp. Alice would think 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 could then pretend to be Alice and get $E_{K_{BT}}\left[t''_T \| ID_A \| K_{AB}\right]$. The key K_{AB} will now be valid for a period of time after $t''_T > t'_T$.
4. Mallory would continue to alternately play Trent against Bob, and then Trent against Alice.

The net result is that the Malicious Mallory can use Trent as an agent to indefinitely force Alice and Bob to continue to use K_{AB}. Of course, Alice and Bob should keep track of the fact that they have seen K_{AB} before and begin to suspect that something fishy 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 culprit, 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 → Trent : $ID_A \| ID_B \| r_1$
2. Trent → Alice : $E_{K_{AT}}\left[K_S \| ID_B \| r_1 \| E_{K_{BT}}[K_S \| ID_A]\right]$
3. Alice → Bob : $E_{K_{BT}}\left[K_S \| ID_A\right]$
4. Bob → Alice : $E_{K_S}[r_2]$
5. Alice → 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 nonces r_1 and r_2. In the first step, Alice sends Trent her request, which is a statement of who she is and who 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 $E_{K_{BT}}[K_S \| ID_A]$ 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 who 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.

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 replays an old $E_{K_{BT}}[K_S \| ID_A]$. 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 involving K_S. Step 4 and step 5 allows 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. Alice → Bob : $E_{K_{BT}}[K_S \| ID_A]$
2. Bob → Alice : $E_{K_S}[r_3]$
3. Mallory → Bob : $E_{K_S}[r_3 - 1]$.

Here, Mallory replays an old message from step 3 of Needham-Schroeder as if she 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 communicating with Alice. Mallory can use Alice's identity to complete her evil plans.

Building a solid key distribution protocol is very tough. The security literature is littered with many examples 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.

10.3 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 M.I.T. known as Project Athena. The purpose of Athena was to provide a huge network of computer workstations for the undergraduate student body at M.I.T., 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 and 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 e-mail, 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 e-mail 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 10.1, 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

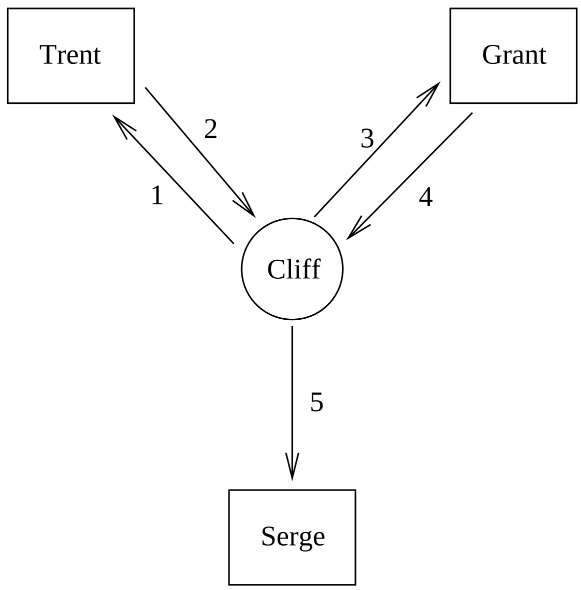

Figure 10.1: Kerberos.

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.

The Kerberos protocol is a formal version of protocols we use in everyday life (for example cashing a check at a bank, or getting on a ride at a fair).

We now look at Kerberos in more detail. Kerberos makes use of a symmetric encryption algorithm. In Version V, Kerberos makes use of DES operating in CBC mode; however, any symmetric encryption algorithm would suffice.

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):

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

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

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

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:

$$e_{K_{CG}}\left(K_{CS}\right).$$

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

5. Cliff to Serge: Cliff is now ready to start making use of Serge's services. He starts by decrypting $e_{K_{CG}}\left(K_{CS}\right)$ 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:

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

   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.

10.4 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 e-mail 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.

10.5 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 Certificate Authority (CA). These are often commercial companies such as VeriSign, GTE, ATT, 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 Netscape and Microsoft Internet Explorer.

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 authozizes 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.

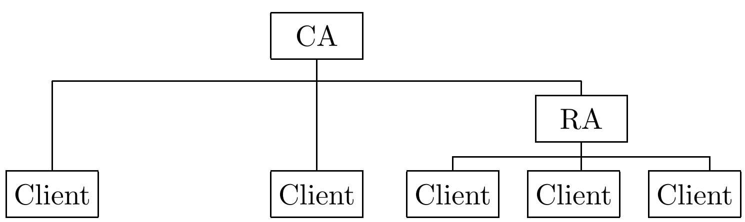

Figure 10.2: A Certification Hierarchy

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 verifies 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).

Figures 10.3, 10.4, and 10.5 show examples of X.509 certificates. The ones in Figures 10.3 and 10.4 are for a CA, namely VeriSign. The part in Figure 10.3 gives the general information about the certificate, including its possible uses. Figure 10.4 gives the detailed information. The one in Figure 10.5 is an edited version of the Details part of a certificate for the bank Wells Fargo.

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 10.3: CA's Certificate; General

Some of the fields in Figure 10.4 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.

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 90 af 8a ee 03 be 1d 28 9c d9 26 76
            a0 cd c4 9d 4e f0 ae 07 16 d5 be af 57 08 6a d0
            a0 42 42 42 1e f4 20 cc a5 78 82 95 26 38 8a 47
```

Figure 10.4: CA's Certificate; Details

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
    Subject's Public Key: 30 81 89 02 81 81 00 a9 ···
    Basic Constraints: Subject Type = End Entity,
                       Path Length Constraint = None
    Subject's Key Usage: Digital Signature, Key Encipherment (AO)
    CRL Distribution Points: (1) CRL Distribution Point
                               Distribution Point Name:
                                 Full Name:
                                   URL=http://crl.verisign.com/
                                     class3InternationalServer.crl
  Certificate Signature Algorithm: MD5 With RSA Encryption
  Certificate Signature Value: ······
```

Figure 10.5: A Client's 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 8 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` $\cdots$ `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 10.5 has a few extra lines. One notable entry is under the heading *Certificate Hierarchy.* The certificate of the 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 web site 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 10.5, 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 eth 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 on-line 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 Internet Explorer, click on *Tools*, then *Internet Options*, then *Content*. 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** that gives the path (often one step) from the user's computer's certificate back to the CA.

10.6 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.

Each user, for example 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 someone can assign: no information, no trust, partial trust, and complete trust. When a certificate's validity is being judged, the