Computer Networking: A Top-Down Approach Featuring the Internet
Order the preliminary edition for spring 2000!

Computer Networking

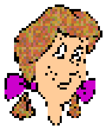
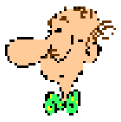
A Top-Down Approach Featuring the Internet

James F. Kurose and Keith W. Ross

Preface
Link to the Addison-Wesley WWW site for this book
Link to overheads for this book

Online Forum Discussion About This Book - with Voice!

7.1 What is Network Security?

Let us introduce Alice  and Bob  , two people who want to communicate "securely." This being a networking text, we should remark that Alice and Bob may be two routers that want to securely exchange routing tables, two hosts that want to establish a secure transport connection, or two email applications that want to exchange secure e-mail - all case studies that we will consider later in this chapter. Alice and Bob are well-known fixtures in the security community, perhaps because their names are more fun than a generic entity named "A" that wants to securely communicate with a generic entity named "B." Illicit love affairs, wartime communication, and business transactions are the commonly cited human needs for secure communications; preferring the first to the latter two, we're happy to use Alice and Bob as our sender and receiver, and imagine them in this first scenario.

7.1.1 Secure Communication

We said that Alice and Bob want to communicate "securely," but what precisely does this mean? Certainly, Alice wants *only* Bob to be able to understand a message that she has sent, even though they are communicating over an "insecure" medium where an intruder (Trudy, the intruder) may intercept, read, and perform computations on whatever is transmitted from Alice to Bob. Bob also wants to be sure that the message that he receives from Alice was indeed sent by Alice, and Alice wants to make sure that the person with whom she is communicating is indeed Bob. Alice and Bob also want to make sure that the contents of Alice's message have not been altered in transit. Given these considerations, we can identify the following desirable properties of **secure communication**:

- **Secrecy.** Only the sender and intended receiver should be able to understand the contents of the transmitted message. Because eavesdroppers may intercept the message, this necessarily requires that the message be somehow encrypted (disguise data) so that an intercepted message can not be decrypted (understood) by an interceptor. This aspect of secrecy is probably the most commonly perceived meaning of the term "secure communication." Note, however, that this is not only a restricted definition of secure communication (we list additional aspects of secure communication below), but a rather restricted definition of secrecy as well. For example, Alice might also want the mere fact that she is communicating with Bob (or the timing or frequency of her communications) to be a secret! We will study cryptographic techniques for encrypting and decrypting data in section 7.2.

- **Authentication.** Both the sender and receiver need to confirm the identity of other party involved in the communication - to confirm that the other party is indeed who or what they claim to be. Face-to-face human communication solves this problem easily by visual recognition. When communicating entities exchange messages over a medium where they can not "see" the other party, authentication is not so simple. Why, for instance, should you believe that a received email containing a text string saying that the email came from a friend of yours indeed came from that friend? If someone calls on the phone claiming to be your bank and asking for your account number, secret PIN, and account balances for verification purposes, would you give that information out over the phone? Hopefully not. We will examine authentication techniques in section 7.3, including several that, perhaps surprisingly, also rely on the cryptographic techniques we study in section 7.2

- **Message Integrity.** Even if the sender and receiver are able to authenticate each other, they also want to insure that the content of their communication is not altered, either malicously or by accident, in transmission.

Extensions to the checksumming techniques that we encountered in reliable transport and data link protocols will also be studied in section 7.3; these techniques also rely on cryptographic concepts in section 7.2

Having established what we mean by secure communication, let us next consider exactly what is meant by an "insecure channel." What information does an intruder have access to, and what actions can be taken on the transmitted data? Figure 7.1-1 illustrates the scenario.

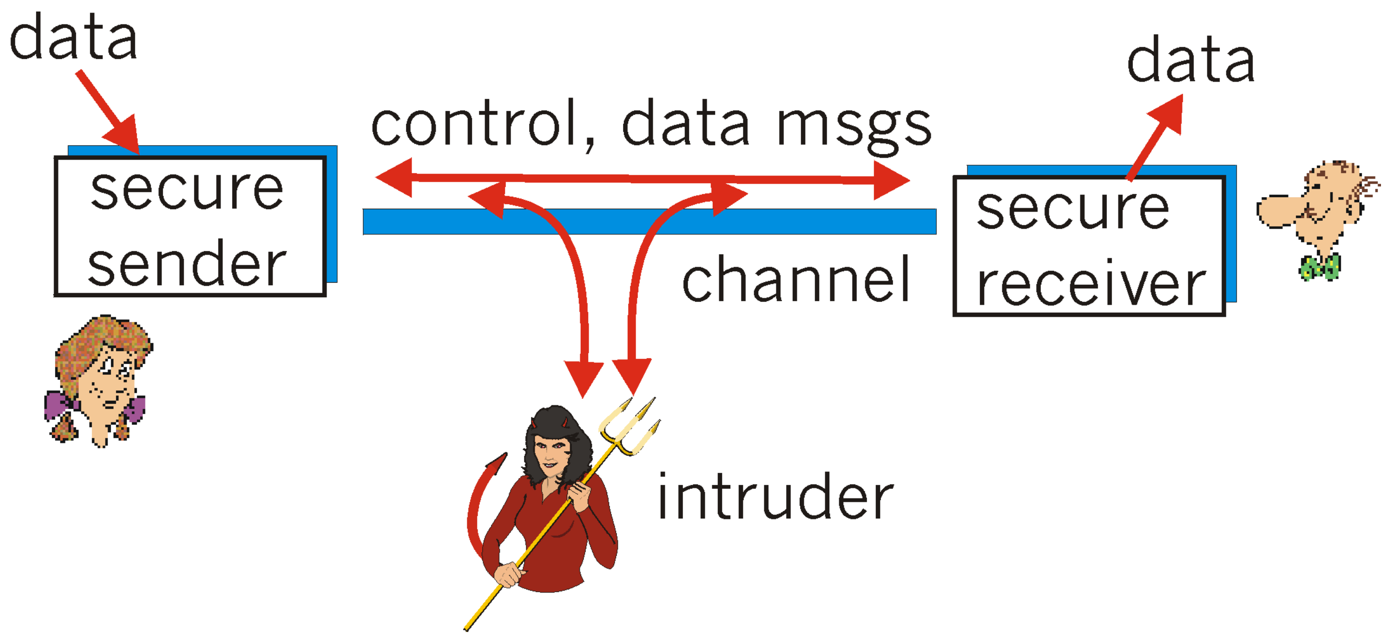


Figure 7.1-1: Sender, receiver and intruder (Alice, Bob, and Trudy)

Alice, the sender, wants to send data to Bob, the receiver. In order to securely exchange data, while meeting the requirements of secrecy, authentication, and message integrity, Alice and Bob will exchange both control message and data messages (in much the same way that TCP senders and receivers exchange both control segments and data segments). All, or some of these message will typically be encrypted. A **passive intruder** can listen to and record the control and data messages on the channel; an **active intruder** can remove messages from the channel and/or itself add messages into the channel.

7.1.2 Network Security Considerations in the Internet

Before delving into the technical aspects of network security in the following sections, let's conclude our introduction by relating our fictitious characters - Alice, Bob, and Trudy - to "real world" scenarios in today's Internet.

Let's begin with Trudy, the network intruder. Can a "real world" network intruder really listen to and record network messages? Is it easy to do so? Can an intruder actively inject or remove messages from the network? The answer to all of these questions is an emphatic "YES." A **packet sniffe**r is a program running in a network attached device that passively receives all data-link-layer frames passing by the device's network interface. In a broadcast environment such as an Ethernet LAN, this means that the packet sniffer receives all frames being transmitted from or to all hosts on the local area network. Any host with an Ethernet card can easily serve as a packet sniffer, as the Ethernet interface card needs only be set to "promiscuous mode" to receive all passing Ethernet frames. These frames can then be passed on to application programs that extract application-level data. For example, in the telnet scenario shown in Figure 7.1-2, the login password prompt sent from A to B, as well as the password entered at B are "sniffed" at host C. Packet sniffing is a double-edged sword - it can be invaluable to a network administrator for network monitoring and management (see Chapter 8) but also used by the unethical hacker. Packet-sniffing software is freely available at various WWW sites, and as commercial products. Professors teaching a networking course have been known to assign lab exercises that involve writing a packet-sniffing and application-level-data-reconstruction program.

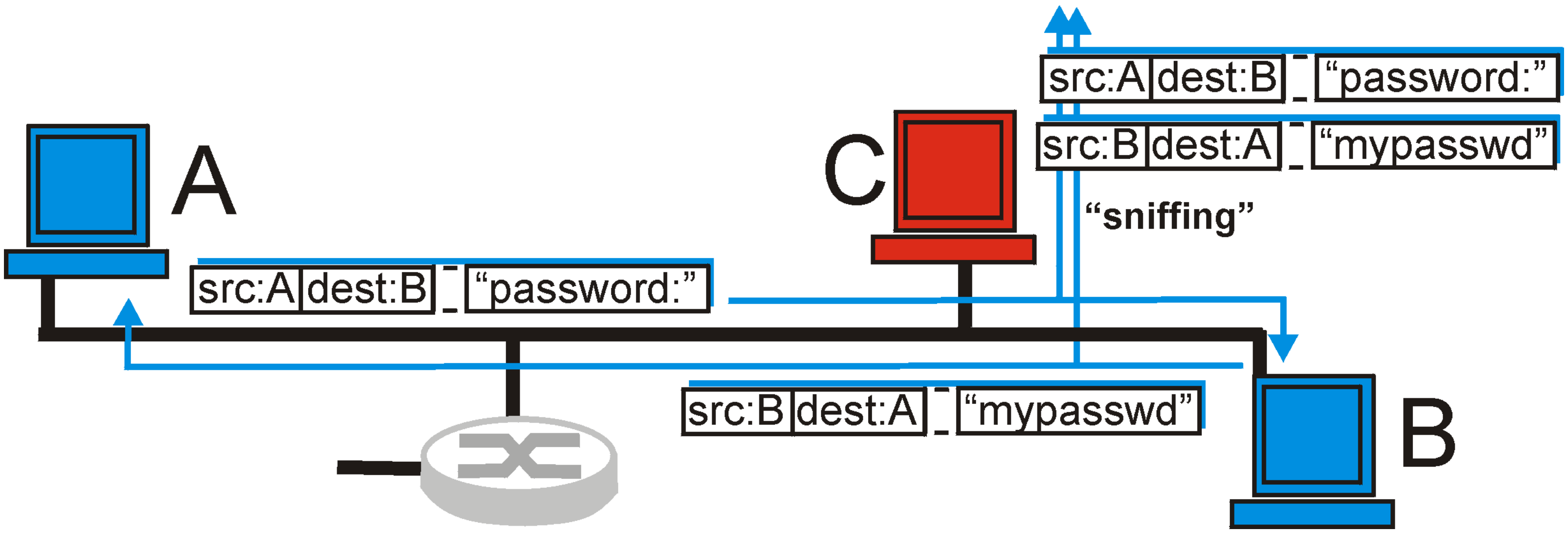


Figure 7.1-2: packet sniffing

Any Internet-connected device (e.g., a host) necessarily sends IP datagrams into the network. Recall from Chapter 4 that these datagrams carry the sender's IP address, as well as application-layer data. A user with complete control over that device's software (in particular its operating system) can easily modify the device's protocols to place an arbitrary IP address into a datagram's source address field. This is known as **IP spoofing.** A user can thus craft an IP packet containing any payload (application-level) data it desires and make it appear as if that data was sent from an arbitrary IP host. Packet sniffing and IP spoofing are just two of the more common forms of security "attacks" on the Internet. These and other network attacks are discussed in the collection of essays [Denning 1997]. A summary of reported attacks is maintained at the CERT Coordination Center [CERT 1999].

Having established that there are indeed real bogeymen (a.k.a. "Trudy") loose in the Internet, what are the Internet equivalents of Alice and Bob, our two friends who need to communicate securely? Certainly, "Bob" and "Alice" might be human user at two end systems, e.g., a real Alice and a real Bob who really *do* want to exchange secure email. (e.g., a user wanting to enter a credit card in a WWW form for an electronic purchase). They might also be participants in an electronic commerce transaction, e.g., a real Alice might want to securely transfer her credit card number to a WWW server to purchase an item on-line. Similarly, a real Alice might want to interact with her back on-line. As noted in [RFC 1636], however, the parties needing secure communication might also themselves be part of the network infrastructure. Recall that the domain name system (DNS, see section 2.5), or routing daemons that exchange routing tables (see section 4.5) require secure communication between two parties. The same is true for network management applications, a topic we examine in the following chapter. An intruder that could actively interfere with, control, or corrupt DNS lookups and updates, routing computations, or network management functions could wreak havoc in the Internet.

Having now established the framework, a few of the most important definitions, and the need for network security, let us next delve into cryptography, a topic of central importance to many aspects of network security..

References

[Cert 1999] CERT, "CERT Summaries," http://www.cert.org/summaries/

[Denning 1997] D. Denning (Editor), P. Denning (Preface), Internet Besieged : Countering Cyberspace Scofflaws, Addison-Wesley Pub Co, (Reading MA, 1997).

[Kessler 1998] G.C. Kessler, An Overview of Cryptography, May 1998, Hill Associates, http://www.hill.com/TechLibrary/index.htm

[NetscapePK 1998] Introduction to Public-Key Cryptography, Netscape Communications Corporation, 1998, http://developer.netscape.com/docs/manuals/security/pkin/contents.htm

[GutmannLinks 1999] P. Gutman, Security Resource Link Farm, http://www.cs.auckland.ac.nz/~pgut001/links.html

[GutmannTutorial 1999] P.Gutmann, Godzilla Crypto Tutorial, http://www.cs.auckland.ac.nz/~pgut001/tutorial/index.html

[RFC 1636] R. Braden, D. Clark, S. Crocker, C. Huitema, "Report of IAB Workshop on Security in the Internet Architecture," RFC 1636, Nov. 1994.

[RSA 1999] RSA's Cryptography FAQ, http://www.rsa.com/rsalabs/faq/

[Punks 1999] Cypherpunks Web Page, ftp://ftp.csua.berkeley.edu/pub/cypherpunks/Home.html

7.2 Principles of Cryptography

Although cryptography has a long history dating back to Julius Caesar (we will look at the so-called Caesar cipher shortly), modern cryptographic techniques, including many of those used in today's Internet, are based on advances made in past twenty years. The books [Kahn 1967, Singh 1999] provide a fascinating look at this long history. A detailed (but entertaining and readable) technical discussion of cryptography, particularly from a network standpoint, is [Kaufman 1995]. [Diffie 1998] provides a compelling and up-to-date examination of the political and social (e.g., privacy) issues that are now inextricably intertwined with cryptography. A complete discussion of cryptography itself requires a complete book [Kaufman 1995, Schneier 1996] and so below we only touch on the essential aspects of cryptography, particularly as they are practiced in today's Internet. Two excellent on-line sites are [Kessler 99] and the RSA Labs FAQ page [RSA 1999c].

Cryptographic techniques allow a sender to disguise data so that an intruder can gain no information from the intercepted data. The receiver, of course must be able to recover the original data from the disguised data. Figure 7.2-1 illustrates some of the important terminology:

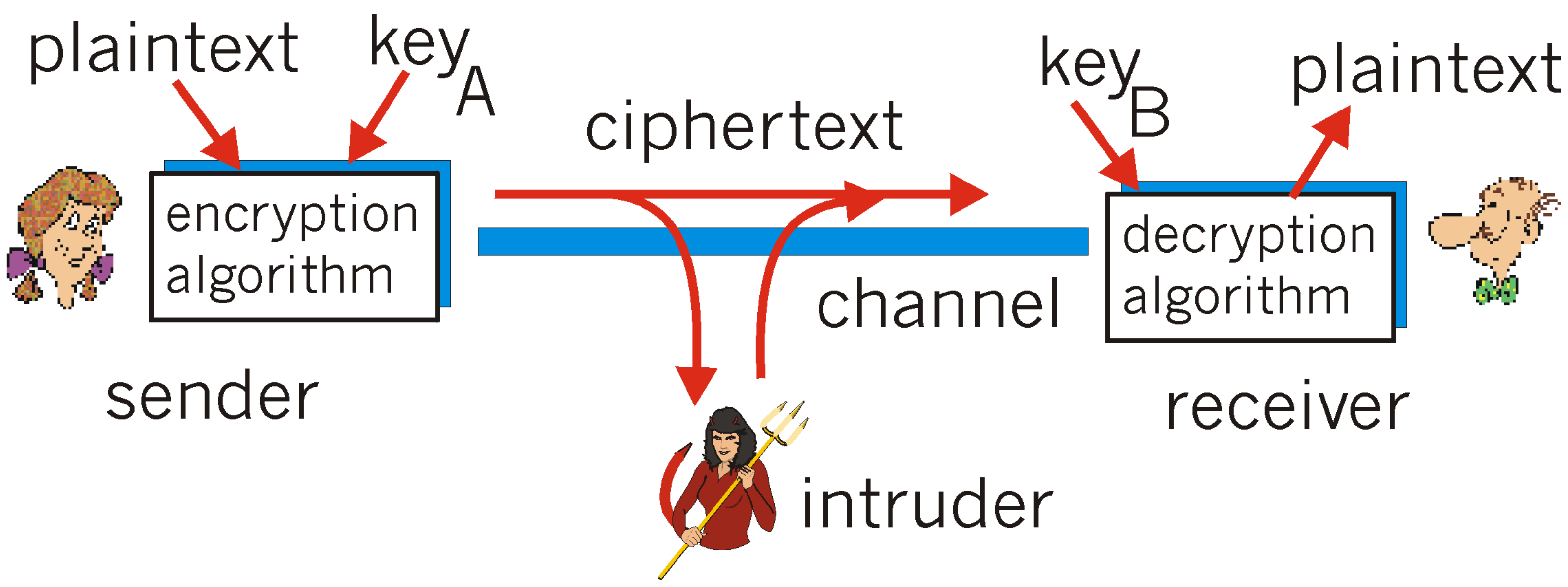


Figure 7.2-1: Cryptographic components

Suppose now that Alice wants to send a message to Bob. Alice's message in its original form (e.g., "Bob, I love you. Alice") is known as **plaintext**, or **cleartext.** Alice encrypts her plaintext message using an **encryption algorithm** so that the encrypted message, known as **ciphertext**, looks unintelligible to any intruder. Interestingly, in many modern cryptographic systems, including those used in the Internet, the encryption technique itself is *known* - published, standardized, and available to everyone (e.g., [RFC 1321, RFC 2437,RFC 2420), even a potential intruder! Clearly, if everyone knows the method for encoding data, then there must be some bit of secret information that prevents an intruder from decrypting the transmitted data. This is where keys come in.

In Figure 7.2-1 Alice provides a **key**, K_A, - a string of numbers or characters, as input to the encryption algorithm. The encryption algorithm takes the key and the plaintext as input and produces ciphertext as output. Similarly, Bob will provide a key K_B, to the **decryption algorithm**, that takes the ciphertext and Bob's key as input and produces the original plaintext as output. In so-called **symmetric key systems**, Alice and Bob's keys are identical and are secret. In **public key systems**, the key that Alice uses is known to all (!), while Bob's key is secret. In the following two subsections, we consider symmetric key and public key systems in more detail.

7.2.1 Symmetric Key Cryptography

All cryptographic algorithms involve substituting one thing for another, e.g., taking a piece of plaintext and computing the appropriate ciphertext that forms the encrypted message. Before studying a modern key-based cryptographic system, let us first "get our feet wet" by studying a very old simple symmetric key algorithm attributed to Julius Caesar, known as the Caesar cipher (a "cipher" is a method for encrypting data).

For English text, the **Caesar cipher** would work by taking each letter in the plaintext message and substituting the letter that is *k* letters later (allowing wraparound, i.e., having the letter "a" follow the letter "z") in the alphabet. For example if *k*=4, then the letter "a" in plaintext becomes "d" in ciphertext; "b" in plaintext becomes "e" in ciphertext, and so on. Here, the value of *k* serves as the key. As an example, the plaintext message "bob, I love you. alice." becomes "yly, f ilsb vlr. xifzb." in ciphertext. While the ciphertext does indeed look like gibberish, it wouldn't take long to break the code if you knew that the Caesar cipher was being used, as there are only 25 possible key values.

An improvement to the Caesar cipher is the so-called **monoalphabetic cipher** that also substitutes one letter in the alphabet with another letter in the alphabet. However, rather than substituting according to a regular pattern (e.g., substitution with an offset of *k* for all letters), any letter can be substituted for any other letter, as long as each letter has a unique substitute letter and vice versa. Many newspaers in the US carry cryptographic puzzles based on this cipher. The substitution rule in Figure 7.2-2 shows one possible rule for encoding plaintext.

plaintext letter:	a	b	c	d	e	f	g	h	i	f	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
ciphertext letter:	m	n	b	v	c	x	z	a	s	d	f	g	h	j	k	l	p	o	i	u	y	t	r	e	w	q

Figure 7.2-2: a monoalphabetic cipher

The plaintext message "bob, I love you. alice." becomes "nkn, s gktc wky. mgsbc" Thus, as in the case of the Caesar cipher, this looks like gibberish. A monoalphabetic cipher would also appear to be better than the Caesar cipher in that there are 26! (on the order of 10^{26}) possible pairings of letters rather than 25 possible pairings. A brute force approach of trying all 10^{26} possible pairings would require far too much work to be a feasible way of breaking the encryption algorithm and decoding the message. However, by statistical analysis of the plaintext language, e.g., knowing that the letters "e" and "t" are the most frequently occurring letters in typical text (accounting for 13% and 9% of letter occurrences), and knowing that particular two- and three-letter occurrences of letters appear quite often together (e.g., "in", "it", "the" "ion", "ing", etc.) make it relatively easy to break this code. If the intruder has some knowledge about the possible contents of the message, then it is even easier to break the code. For example, if Trudy the intruder is Bob's wife and suspects Bob of having an affair with Alice, then she might suspect that the names "bob" and "alice" appear in the text." If Trudy knew for certain that those two names appeared in the ciphertext and had a copy of the example ciphertext message above, then she could immediately determine 7 of the 26 letter pairings, since "alice" is the only five-letter word in the message, and "bob" is the only three-letter word that has an identical first and last letter. Thus, Trudy requires 10^9 fewer possibilities to be checked a by brute force method. Indeed, if Trudy suspected Bob of having an affair, she might well expect to find some other choice words in the message as well.

When considering how easy it might be for Trudy to break Bob and Alice's encryption scheme, one can distinguish three different scenarios, depending on what information the intruder has:

- **Ciphertext only attack.** In some cases, the intruder may only have access to the intercepted ciphertext, with no certain information about the contents of the plaintext message. We have seen how statistical analysis can help in a ciphertext only attack on an encryption scheme.

- **Known plaintext attack.** We saw above that if Trudy somehow knew for sure that "bob" and "alice" appeared in the ciphertext message then she could have determined the (plaintext, ciphertext) pairings for the letters a, l, i, c, e, b, and o. Trudy might also have been fortunate enough to have recorded all of the ciphertext transmissions and then found Bob's own decrypted version of one of transmissions scribbled on a piece of paper. When an intruder knows some of the (plaintext, ciphertext) pairings, we refer to this as a known plaintext attack on the encryption scheme.

- **Chosen plaintext attack.** In a chosen plaintext attack, the intruder is able to choose the plaintext message and obtain its corresponding ciphertext form. For the simple encryption algorithms we've seen so far, if Trudy could get Alice to send the message, "The quick fox jumps over the lazy brown dog," she can completely break the encryption scheme. We'll see shortly that for more sophisticated encryption techniques, a chosen plaintext attack does not necessarily mean that the encryption technique can be broken.

Five hundred years ago, techniques improving on monoalphabetic encryption, known as **polyalphabetic encryption** were invented. These techniques, incorrectly attributed to Blaise de Vigenere [Kahn 1967] have come to be known as **Vigenere ciphers**. The idea behind Vigenere ciphers is to use multiple monoalphabetic ciphers, with a specific monoalphabetic cipher to encode a letter in a specific position in the plaintext message. Thus, the same letter, appearing in different positions in the plaintext message might be encoded differently. The Vigenere cipher shown in Figure 7.2-3 has two different Caesar ciphers (with *k*=6 and *k*=20), shown as rows in Figure 7-2-3. One might choose to use these two Caesar ciphers, C_1 and C_2, in the repeating pattern C_1, C_2, C_2, $C_{1,}$ C_2. That is, the first letter of plaintext is to encoded using C_1, the second and third using C_2, the fourth using C_1, and the fifth using C_2. The pattern then repeats, with the sixth letter being encoded using C_1, the seventh with C_2, and so on. The plaintext message "bob, I love you. alice." is thus encrypted "ghu, n etox dhz." Note that the first "b" in the plaintext message is encrypted using C_1, while the second "b" is encrypted using C_2. In this example, the encryption and decryption "key" is the knowledge of the two Caesar keys (*k*=4, *k*=20) and the pattern C_1, C_2, C_1, C_2, C_2.

plaintext letter:	a	b	c	d	e	f	g	h	i	f	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
C_1(*k*=6):	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e
C_2(*k*=20):	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s

Figure 7.2-3: A Vigenere cipher using two Caesar ciphers

Data Encryption Standard (DES)

Let us now fast forward to modern time and examine the **Data Encryption Standard (DES)** [NIST 1993] , a symmetric key encryption standard published in 1977 and updated most recently in 1993 by the US National Bureau of Standards for commercial and non-classified US government use. DES encodes plaintext in 64 bit chunks using a 64-bit key. Actually, 8 of these 64 bits are odd parity bits (one bit in each of the 8 bytes is an odd partity bit for that byte), so the DES key is effectively 56 bits long. The National Institute of Standards (the successor to the National Bureau of Standards) states the goal of DES as follows: " The goal is to completely scramble the data and key so that every bit of the ciphertext depends on every bit of the data and every bit of the key .... with a good algorithm, there should be no correlation between the ciphertext and either the original data or key." [NIST 1999].

The basic operation of DES is illustrated in Figure 7.2-4. In our discussion we will overview DES operation, leaving the nitty-gritty bit-level details (there are *many*!) to those wishing to consult [Kaufman 1995, Schneier 1995] (with [Schneier 1995] including a C implementation as well). The DES consists of two permutation steps (the first and last steps of the algorithm) in which all 64 bits are permuted, and 16 identical "rounds" of operation in between. The operation of each round is identical, taking the output of the previous round as input. During each round, the rightmost 32 bits of the input are moved to the left 32 bits of the output. The entire 64-bit input to the *ith* round and the 48 bit key for the *ith* round (derived from the larger DES 56-bit) are taken as input to a function that involves expansion of four-bit input chunks into six-bit chunks, exclusive OR-ing with the expanded six bit chunks of the 48-bit key *Ki*, a substitution operation and further exclusive OR-ing with the leftmost 32 bits of the input; see [Kaufman 1995, Schneier 1995] for details. The resulting 32-bit output of the function is then used as the rightmost 32 bits of the rounds 64-bit output, as shown in Figure 7.2-4. Decryption works by reversing the algorithm's operations.

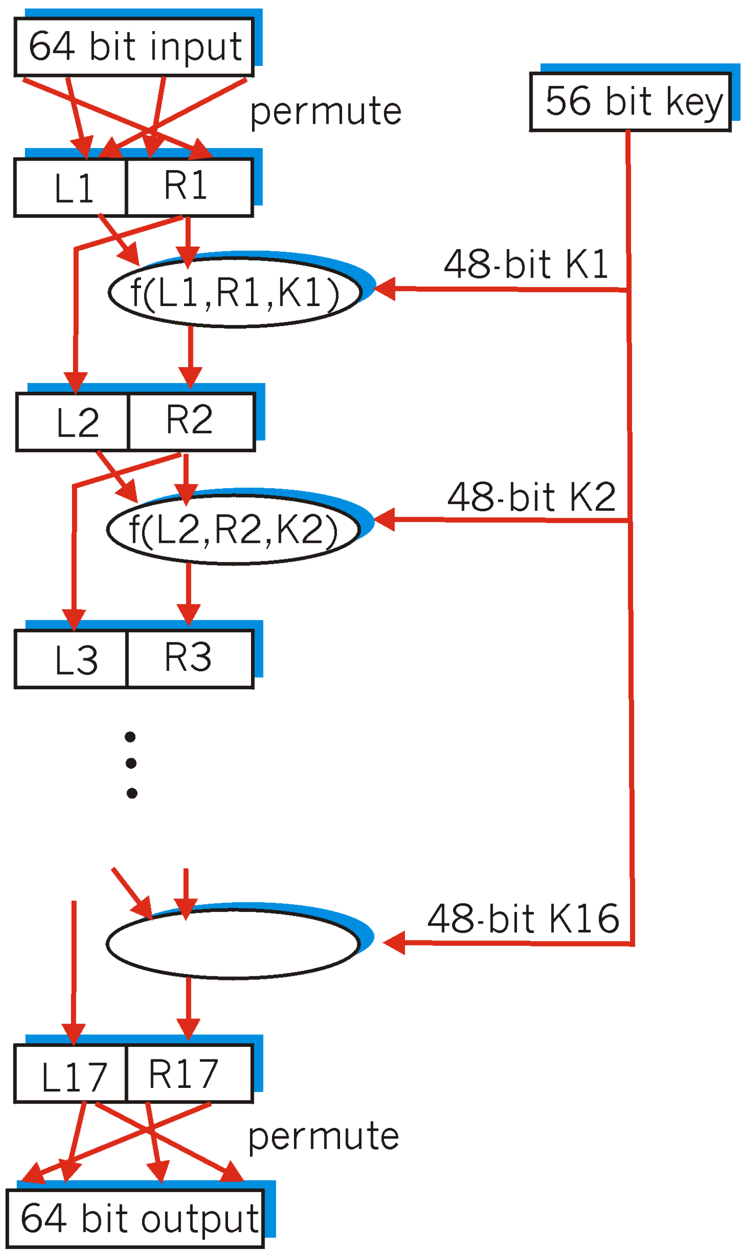


Figure 7.2-4: Basic operation of the DES

How well does DES work? How secure is it? No one can tell for sure, although recent speculation is that one could build a special purpose machine that exhaustively searched through the 56-bit key space for under a million dollars [Kaufman 1995]. In 1997, a network security company, RSA Data Security Inc, launched a DES Challenge contest to "crack" (decode) a short phrase it had encrypted using 56-bit DES. The unencoded phrase (“Strong cryptography makes the world a safer place.”) was determined only 140 days later by a team that used volunteers throughout the Internet to systematically explore the key space. The team claimed the $10,000 prize after testing only a quarter of the key space - about 18 quadrillion keys [RSA 1997]. The most recent 1999 DES Challenge III was won in a record time of a little over 22 hours, with a network of volunteers and a special purpose computer that was built for less that $250,000 (nick-named "DES Cracker") and is documented on-line [EFF 1999].

If 56-bit DES is considered too insecure, one can simply run the 56-bit algorithm multiple times, taking the 64-bit output from one iteration of DES as the input to the next DES iteration, using a different encryption key each time. For example, so-called **triple-DES** (3DES), is a proposed US government standard [NIST 1999b] and has been proposed as the encryption standard for the Point-to-Point protocol [RFC 2420], PPP, for the data link layer (see section 5.7). A detailed discussion of key lengths and the estimated time and budget needed to crack DES can be found in [Blaze 1996].

We should also note that our description above has only considered the encryption of a 64-bit quantity. When longer messages are encrypted, which is typically the case, DES is often used with a technique known as **cipher-block chaining**, in which the encrypted version of the *jth* 64-bit quantity of data is XOR'ed with the *(j+1)st* unit of data before the *(j+1)st* unit of data is encrypted.

7.2.2 Public Key Encryption

For more than 2000 years (since the time of the Caesar cipher and up to the 1970's), encrypted communication required that the two communicating parties share a common secret - the symmetric key used for encryption and decryption. One difficulty with this approach is that the two parties must somehow agree on the shared key; but to do so requires (presumably secure) communication! Perhaps the parties could first meet and agree on the key in person (e.g., two of Caesar's centurions might meet at the Roman baths) and thereafter communicate with encryption. In a networked world, however, communicating parties may never meet and may never converse except over the network. Is it possible for two parties to communicate with encryption without having a shared secret key that is known in advance? In 1976, Diffie and Hellman [Diffie 1976] demonstrated an algorithm (known now as Diffie-Hellman Key Exchange) to do just that - a radically different and marvelously elegant approach towards secure communication that has led to the development of today's public key cryptography systems. We will see shortly that public key cryptography systems also have several wonderful properties that make them useful not only for encryption, but for authentication and digital signatures as well. The ideas begun with [Diffie 1976] have evolved, with a significant milestone being [RSA 1978], into the public key systems in use today.

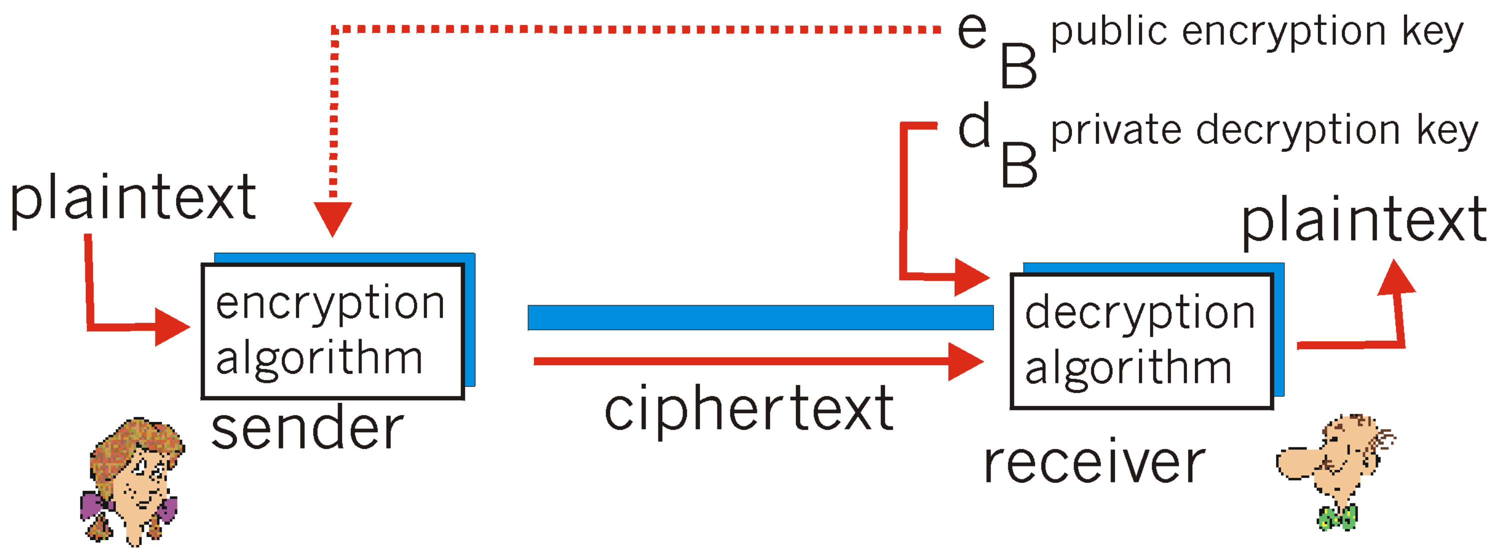


Figure 7.2-5: Public key cryptography

The use of public key cryptography is quite simple. Suppose Alice wants to communicate with Bob. As shown in Figure 7.2-5, rather than Bob and Alice sharing a single secret key (as in the case of symmetric key systems), Bob (the recipient of Alice's messages) instead has two keys - a **public key** that is available to *everyone* in the world (including Trudy the intruder!) and a **private key** that is known only to Bob. In order to communicate with Bob, Alice first fetches Bob's public key. Alice then encrypts her message to Bob using Bob's public key and a known (e.g., standardized) encryption algorithm. Bob receives Alice's encrypted message and uses his private key and a known (e.g., standardized) decryption algorithm to decrypt Alice's message. In this manner, Alice can send a secret message to Bob without either of them having to have to distribute any secret keys!

Using the notation of Figure 7.2-5, for any message *m*, $d_B(e_B(m)) = m$, i.e., applying Bob's public key then Bob's private key to the message *m* gives back *m*. We will see shortly that we can interchange the public key and private key encryption and get the same result, that is, $e_B(d_B(m)) = d_B(e_B(m)) = m$.

The use of public key cryptography is thus conceptually simple. But two immediate worries may spring to mind. A first concern is that although an intruder intercepting Alice's encrypted message will only see gibberish, the intruder knows both the key (Bob's public key, which is available for all the world to see) and the algorithm that Alice used for encryption. Trudy can thus mount a chosen plaintext attack, using the known standardized encryption algorithm and Bob's publicly available encryption key to encode any message she chooses! Trudy might well try, for example, to encode messages, or parts of messages, that she suspects that Alice might send. Clearly, if public key cryptography is to work, key selection and encryption/decryption must be done in such a way that it is impossible (or at least so hard to be impossible for all practical purposes) for an intruder to either determine Bob's private key or somehow otherwise decrypt or guess Alice's message to Bob. A second concern is that since Bob's encryption key is public, anyone can send an encrypted message to Bob, including Alice or someone *claiming* to be Alice. In the case of a single shared secret key, the fact that the sender knows the secret key implicitly identifies the sender to the receiver. In the case of public key cryptography, however, this is no longer the case since anyone can send an encrypted message to Bob using Bob's publicly available key. Certificates, which we will study in section 7.5, are needed to bind an entity (such as Bob) to a specific public key.

While there may be many algorithms and keys that have this property, the **RSA algorithm** (named after its founders, Ron Rivest, Adi Shamir, and Leonard Adleman) has become almost synonymous with public key cryptography. Let's first see how RSA works and then examine why it works. Suppose that Bob wants to receive encrypted messages, as shown in Figure 7.2-5. The are two inter-related components of RSA:

- choice of the public key and the private key
- the encryption and decryption algorithm

In order to choose the public and private keys, Bob must do the following:

- Choose two large prime numbers, *p* and *q*. How large should *p* and *q* be? The larger the values, the more difficult it is to break RSA but the longer it takes to perform the encoding and decoding. RSA

Laboratories recommends that the product of *p* and *q* be on the order of 768 bits for personal use and 1024 bits for corporate use [RSA 1999]. (Which leads one to wonder why corporate use is deemed so much more important than personal use!).

- Compute $n = pq$ and $z = (p-1)(q-1)$.
- Choose a number, *e*, less than *n*, which has no common factors (other than 1) with z. (In this case, *e* and *z* are said to be relatively prime). The letter '*e*' is used since this value will be used in encryption.
- Find a number, *d*, such that $ed - 1$ is exactly divisible (i.e., with no remainder) by z. The letter *'d'* is used because this value will be used in decryption. Put another way, given *e*, we choose *d* such that the integer remainder when *ed* is divided by *z* is 1. (The integer remainder when an integer *x* is divided by the integer *n*, is denoted $x \bmod n$).
- The public key that Bob makes available to the world is the pair of numbers *(n,e);* his private key is the pair of numbers *(n,d)*.

The encryption by Alice, and the decryption by Bob is done as follows:

- Suppose Alice wants to send Bob a bit pattern, or number, *m*, such that $m < n$. To encode, Alice performs the exponentiation, m^e, and then computes the integer remainder when $m^e$is divided by *n.* Thus, the encrypted value, *c*, of the plaintext message, *m*, that Alice sends is:

$$c = m^e \bmod n$$

- To decrypt the received ciphertext message, *c*, Bob computes

$$m = c^d \bmod n$$

which requires the use of his secret key, *(n,d)*.

As a simple example of RSA, suppose Bob chooses $p=5$ and $q=7$ (admittedly, these values are far too small to be secure). Then $n=35$ and $z=24$. Bob chooses $e=5$, since 5 and 24 have no common factors. Finally, Bob chooses $d=29$, since 5*29 - 1 (i.e., $ed - 1$) is exactly divisible by 24. Bob makes the two values, $n=35$ and $e=5$, public and keeps the value $d=29$ secret. Observing these two public values, suppose Alice now wants to send the letters 'l' 'o' 'v' and 'e' to Bob. Interpreting each letter as a number between 1 and 26 (with 'a' being 1, and 'z' being 26), Alice and Bob perform the encryption and decryption shown in Figures 7.2-6 and 7.2-7, respectively:

plaintext letter	*m:* numeric representation	m^e	ciphertext $c = m^e \bmod n$
l	12	248832	17
o	15	759375	15
v	22	5153632	22
e	5	3125	10

Figure 7.2-6: Alice's RSA encryption, e=5, n = 35

ciphertext c	c^d	$m = c^d \bmod n$	plaintext letter
17	481968572106750915091411825223072000	12	l
15	12783403948858939111232757568359400	15	o
22	8.5164331908653770195619449972111e+38	22	v
10	100000000000000000000000000000	5	e

Figure 7.2-7: Bob's RSA decryption, d=29, n=35

Given that "toy" example in Figures 7-7 and 7-8 has already produced some extremely large numbers, and given that we know that we saw earlier that p and q should each be several hundred bits long, several practical issues regarding RSA come to mind. How does one choose large prime numbers? How does one then choose *e* and *d*? How does one perform exponentiation with large numbers? A discussion of these important issues is beyond the scope of this book; see [Kaufman 1995] and the references therein for details.

We do note here that the exponentiation required by RSA is a rather time consuming process.   RSA Data Security [RSA 1999b] says its software toolkit can encrypt/decrypt at a throughput of 21.6 Kbits per second with a 512-bit value for *n* and 7.4 Kbits per second with a 1024-bit value. DES is at least one hundred times fast in software and between 1000 and 10000 times faster in hardware. As a result, RSA is often used in practice in combination with DES. For example, if Alice wants to send Bob a large amount of encrypted data at high speed, she could do the following. First Alice chooses a DES key that will be used to encode the data itself; this key is sometimes referred to as a session key, K_S. Alice must inform Bob of the session key, since this is the shared secret key they will use for DES. Alice thus encrypts the session key value using Bob's public RSA key, i.e., computes $c = (K_S)^e \bmod n$. Bob receives the RSA-encrypted session key, *c*, and decrypts to obtain the session key, K_S.. Bob now knows the session key that Alice will use for her DES-encrypted data transfer.

Why does RSA work?

The RSA encryption/decryption above appears rather magical. Why should it be that by applying the encryption algorithm and then the decryption algorithm, one recovers the original message?   In order to understand why RSA works, we'll need to perform arithmetic operations using so-called modulo-n arithmetic. In modular arithmetic, one performs the usual operations of addition, multiplication and exponentiation. However, the result of each operation is replaced by the integer remainder that is left when the result is divided by n. We will take $n = pq$, where p and q are the large prime numbers used in the RSA algorithm.

Recall that under RSA encryption, a message (represented by an integer), m, is first exponentiated to the power e using modulo-n arithmetic to encrypt. Decryption is performed by raising this value to the power d, again using modulo n arithmetic. The result of an encryption step, followed by a decryption step is thus $(m^e)^d$. Let's now see what we can say about this quantity. We have:

$$(m^e)^d \bmod n = m^{ed} \bmod n$$

Although we're trying to remove some of the "magic" about why RSA works, we'll need to use a rather magical result from number theory here. Specifically, we'll need the result that says if p and q are prime, and $n = pq$, then $x^{y \bmod n}$ is the same as $x^{(y \bmod (p-1)(q-1))} \bmod n$ [Kaufman 1995]. Applying this result, we have

$$(m^e)^d \bmod n = m^{(ed \bmod (p-1)(q-1))} \bmod n$$

But remember that we chose e and d such that $ed -1$ is exactly divisible (i.e., with no remainder) by $(p-1)(q-1)$, or equivalently that ed is divisible by $(p-1)(q-1)$ with a reminder of 1, and thus $ed \bmod (p-1)(q-1) = 1$. This gives us

$$(m^e)^d \bmod n = m^1 \bmod n = m$$

i.e., that

$$(m^e)^d \bmod n = m.$$

This is the result we were hoping for! By first exponentiating to the power of e (i.e., encrypting) and then exponentiating to the power of d (i.e., decrypting), we obtain the original value, m. Even *more* remarkable is the fact that if we first exponentiate to the power of d and then exponentiate to the power of e, i.e., we reverse the order of encryption and decryption, performing the decryption operation first and then applying the encryption operation, we also obtain the original value, *m!* (The proof for this result follows the exact same reasoning as above). We will see shortly that this wonderful property of the RSA algorithm,

$$(m^e)^d \bmod n = m = (m^d)^e \bmod n$$

will be of great use.

The security of RSA relies on the fact that there are no known algorithms for quickly factoring a number, in this case the public value n, into the primes p and q. If one knew p and q, then given the public value e, one could then easily compute the secret key, d. On the other hand, it is not know whether or not there exist fast algorithms for factoring a number, and in this sense the security of RSA is not "guaranteed."

References

[Blaze 1996] M. Blaze, W. Diffie, R. Rivest, B. Schneier, T. Shimomura, E. Thompson, and M. Weiner, "Minimal Key Lengths for Symmetric Ciphers to Provide Adequate Commercial Security," http://www.counterpane.com/keylength.html

[Diffie 1998] W. Diffie and S. Landau, Privacy on the Line, The Politics of Wiretapping and Encryption, MIT Press, 1998.

[EFF 1999] Electronic Frontier Foundation, "Cracking DES," http://www.eff.org/DEScracker/

[FIPS-46-1] US National Bureau of Standards, "Data Encryption Standard", Federal Information Processing Standard (FIPS) Publication 46-1, January 1988.

[Kahn 1967] D. Kahn, The Codebreakers, the Story of Secret Writing, The Macmillan Company, 1967
[Kaufman 1995] C. Kaufman, R. Perlman, M. Speciner, Network Security, Private Communication in a Public World, Prentice Hall, 1995.
[Kessler 1999] G. Kessler, "An Overview of Cryptography," Hill Associates Inc, http://www.hill.com/library/crypto.html
[NIST 1993] National Institute of Standards and Technology, Federal Information. Data Encryption Standard, Processing Standards Publication 46-2, 1993.
[NIST 1999] National Institute of Standards and Technology, "Dat Encryption Standard Fact Sheet," http://csrc.nist.gov/cryptval/des/des.txt
[NIST 1999b] National Institute of Standards and Technology, "Draft Federal Information Processing Standard (FIPS) 46-3, Data Encryption Standard (DES), and Request for Comments," http://csrc.nist.gov/cryptval/des/fr990115.htm
[RFC 1321] R. Rivest, "The MD5 Message-Digest Algorithm," RFC 1321, April 1992.
[RFC 2437] B. Kaliski, J. Staddon, "PKCS #1: RSA Cryptography Specifications, Version 2," RFC 2437, October 1998.
[RFC 2420] H. Kummert, "The PPP Triple-DES Encryption Protocol (3DESE)," RFC 2420, Sept. 1998.
[RSA 1978] R.L. Rivest, A. Shamir, and L.M. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM,* 21(2): 120-126, February 1978.
[RSA 1997] RSA Data Security Inc, "DES RSA Challege Cracked: Government encryption standard DES takes a fall," http://www.rsa.com/des/
[RSA 1999] RSA Laboratories, "How large a key should be used in RSA?" http://www.rsa.com/rsalabs/faq/html/3-1-5.html
[RSA 1999b] RSA Laboratories, "How fast is RSA," http://www.rsa.com/rsalabs/faq/html/3-1-2.html
[RSA 1999c] RSA Laboratories, "RSA Labs FAQ," http://www.rsa.com/rsalabs/faq/index.html
[Schneier 1995] B. Schneier, Applied Cryptography: Protocols, Algorithms, and Source Code in C, John Wiley and Sons, 1995.
[Singh 1999] S. Singh, "The Code Book : The Evolution of Secrecy from Mary, Queen of Scots to Quantum Cryptography," Doubleday Press, 1999.

Return to Table Of Contents

7.3 Authentication: Who are You?

Authentication is the process of proving one's identity to someone else. As humans, we authenticate each other in many ways: we recognize each others' faces when we meet; we recognize each others' voices on the telephone; we are authenticated by the customs official who checks us against the picture on our passport.

In this section we consider how one party can authenticate another party when the two are communicating over a network. We focus here on authenticating a "live" party, at the point in time when communication is actually occurring. We will see that this is a subtly different problem from proving that a message received at some point in the past (e.g., that may have been archived) did indeed come from that claimed sender. This latter problem is referred to as the **digital signature** problem, which we explore in section 7.4.

When performing authentication over the network, the communicating parties can not rely on biometric information, such as a visual appearance or a voiceprint. Indeed, we will see in our later case studies that it is often network elements such as routers and client/server processes that must authenticate each other. Here, authentication must be done solely on the basis of messages and data exchanged as part of an **authentication protocol.** Typically, an authentication protocol would run *before* the two communicating parties run some other protocol (e.g., a reliable data transfer protocol, a routing table exchange protocol, or an email protocol). The authentication protocol first establishes the identities of the parties to each others' satisfaction; only after authentication do the parties get down to the work at hand.

As in the case of our development of a reliable data transfer protocol, *rdt*, in Chapter 3, we will find it instructive here to develop various versions of an authentication protocol, which we will call *ap* ("authentication protocol"), and poke holes (i.e., find security flaws) in each version as we proceed. Let's begin by assuming that Alice needs to authenticate herself to Bob.

Authentication protocol *ap1.0*

Perhaps the simplest authentication protocol we can imagine is one where Alice simply sends a message to Bob saying she is Alice. This protocol is shown in Figure 7.3-1. The flaw here is obvious - there is no way for Bob to actually know that the person sending the message, "I am Alice" is indeed Alice. For example, Trudy (the intruder) could just as well send such a message.

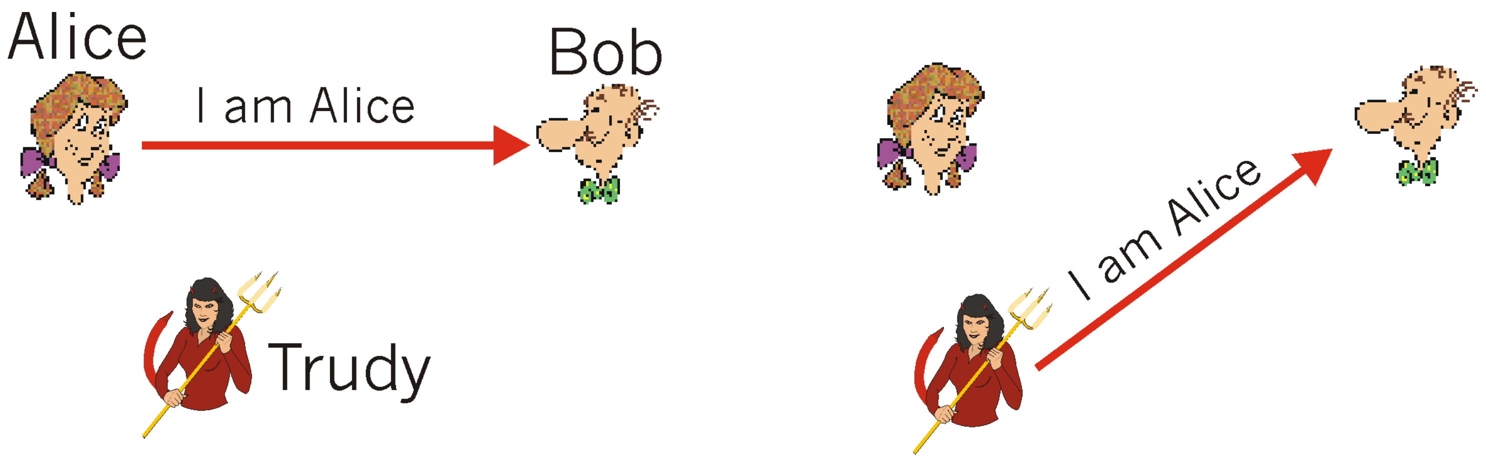


Figure 7.3-1: Protocol *ap1.0* and a failure scenario.

Authentication protocol *ap2.0*

In the case that Alice has a well-known network address (e.g., IP address) from which she always communicates, Bob could attempt to authenticate Alice by verifying that the source address on the IP datagram carrying the authentication message matches Alice's well-known address. If so, then Alice would be authenticated. This might stop a very network-naive intruder from impersonating Alice. But it wouldn't stop the determined student studying this book, or many others!

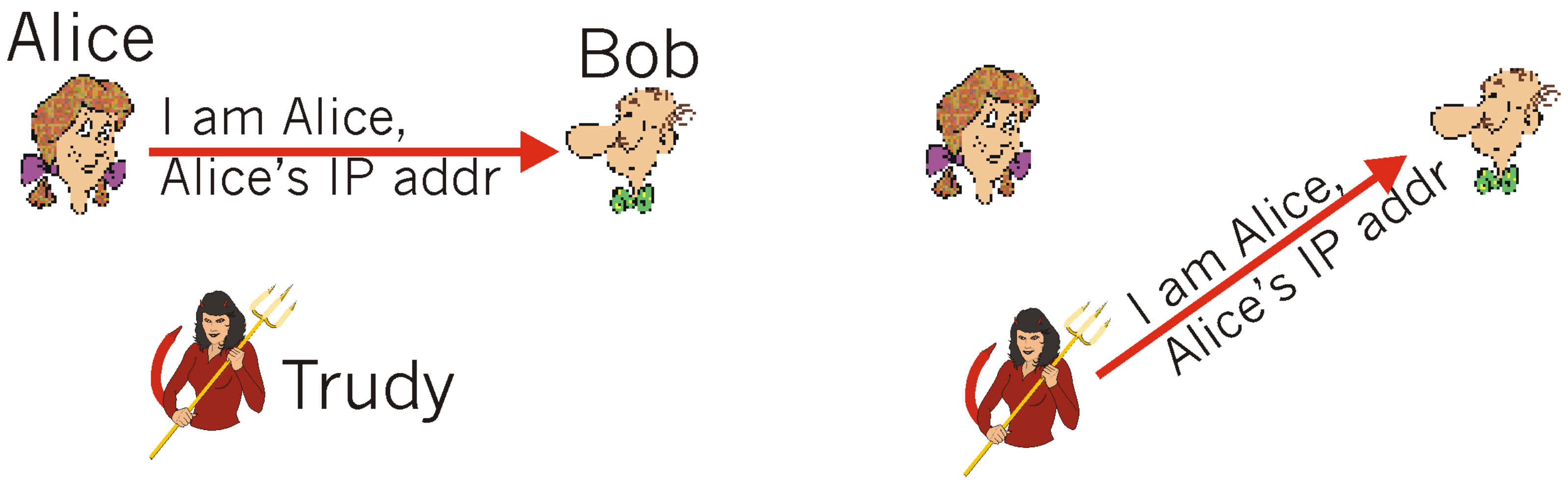


Figure 7.3-2: Protocol *ap2.0* and a failure scenario.

Given that we have now studied both the network and data link layers, we know that it is not that hard (e.g., if one had access to the operating system code and could build one's own operating system kernel, as is the case with Linux and several other freely available operating systems) to create an IP datagram, put whatever IP source address we want (e.g., including Alice's well-known IP address) into the IP datagram and send the datagram over the link layer protocol to the first hop router. From then on, the incorrectly-source-addressed datagram would be dutifully forwarded to Bob. This approach is a form of **IP spoofing**, a well-known security attack technique [Cert 96]. IP spoofing can be avoided if a router is configured to refuse IP datagrams that do not have a given source address. For example, Trudy's first hop router could be configured to only forward datagrams containing Trudy's IP source address. However, this capability is not universally deployed or enforced. Bob would thus be foolish to assume that Trudy's network manager (who might be Trudy herself!) had configured Trudy's first hop router to only forward appropriately-addressed datagrams.

Authentication protocol *ap3.0*

One classical approach to authentication is to use a secret password. We have PIN numbers to identify ourselves to automatic teller machines and login passwords for operating systems. The password is a shared secret between the authenticator and the person being authenticated. We saw in section 2.2.5 that HTTP uses a password-based authentication scheme. Telnet and FTP use password authentication as well. In protocol *ap3.0*, Alice thus sends her secret password to Bob, as shown in Figure 7.3-3.

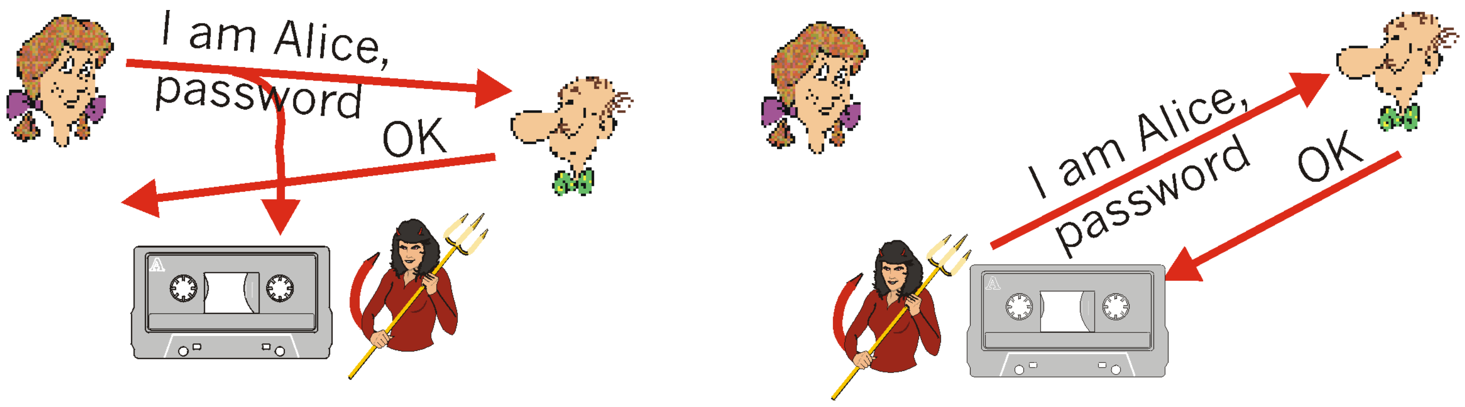


Figure 7.3-3: Protocol *ap3.0* and a failure scenario.

The security flaw here is clear. If Trudy eavesdrops on Alice's communication, then she can learn Alice's password. Lest you think this is unlikely, consider the fact that when one Telnet's to another machine and logs in, the login password is sent unencrypted to the Telnet server . Someone connected to the Telnet client or server's LAN can possibly "sniff" (read and store) all packets transmitted on the LAN and thus steal the login password. In fact, this is a well-known approach for stealing passwords (see, e.g., [Jimenez 1997]. Such a threat is obviously very real, so *ap3.0* clearly won't do.

Authentication protocol *ap3.1*

Having just studied the previous section on cryptography, our next idea for fixing *ap3.0* is naturally to use encryption. By encrypting the password, Trudy will not be able to learn Alice's password! If we assume that Alice and Bob share a symmetric secret key, $K_{A\text{-}B}$, then Alice can encrypt the password, send her identification message, "I am Alice," and her encrypted password to Bob. Bob then decrypts the password and, assuming the password is correct, authenticates Alice. Bob feels comfortable in authenticating Alice since not only does Alice know the password, but she also knows the shared secret key value needed to encrypt the password. Let's call this protocol *ap3.1*.

While it is true that *ap3.1* prevents Trudy from learning Alice's password, the use of cryptography here does not solve the authentication problem! Bob is again subject to a so-called **playback attack:** Trudy needs only eavesdrop on Alice's communication, record the encrypted version of the password, and then later play back the encrypted version of the password to Bob to pretend that she is Alice. The use of an encrypted password doesn't make the situation manifestly different from that in Figure 7.3-3.

Authentication protocol *ap4.0*

The problem with *ap3.1* is that the same password is used over and over again. One way to solve this problem would be to use a different password each time. Alice and Bob could agree on a sequence of passwords (or on an algorithm for generating passwords) and use each password only once, in sequence. This idea is used in the S/KEY system [RFC 1760], adopting an approach due to Lamport [Lamport 81] for generating a sequence of passwords.

Rather than just stop here with this solution, however, let us consider a more general approach for combating the playback attack. The failure scenario in Figure 7.3-3 resulted from the fact that Bob could not distinguish between the original authentication of Alice and the later playback of Alice's original authentication. That is, Bob could not tell if Alice was "live" (i.e., was currently really on the other end of the connection) or whether the messages he was receiving were a recorded playback of a previous authentication of Alice. The very (very!) observant reader will recall that the 3-way TCP handshake protocol needed to address the same problem - the server side of a TCP connection did not want to accept a connection if the received SYN segment was an old copy (retransmission) of a SYN segment from an earlier connection. How did the TCP server side solve the problem of determining if the client was really "live"? It chose an initial sequence number (which had not been used in a very long time), sent that number to the client, and then waited for the client to respond back with an ACK segment containing that number. We can adopt the same idea here for authentication purposes.

A **nonce** is a number that a protocol will only ever use once-in-a-lifetime. That is, once a protocol uses a nonce, it will never use that number again. Our *ap4.0* protocol uses a nonce as follows:

ap4.0:

- Alice sends the message, "I am Alice," to Bob
- Bob chooses a nonce, *R*, and sends it to Alice
- Alice encrypts the nonce using Alice and Bob's symmetric secret key, $K_{A-B.}$, and sends the encrypted nonce, $K_{A-B}(R)$ back to Bob. As in protocol *ap3.1*, it is the fact that Alice knows K_{A-B} and uses it to encrypt a value that lets Bob know that the message he receives was generated by Alice. The nonce is used to insure that Alice is "live."
- Bob decrypts the received message. If the decrypted nonce equals the nonce he sent Alice, then Alice is authenticated.

Protocol *ap4.0* is illustrated in Figure 7.3-4. By using the once-in-a-lifetime value, R, and then checking the returned value, $K_{A-B}(R)$, Bob can be sure that both Alice is who she says she is (since she knows the secret key value needed to encrypt *R*) and is "live" (since she has encrypted the nonce, *R*, that Bob just created).

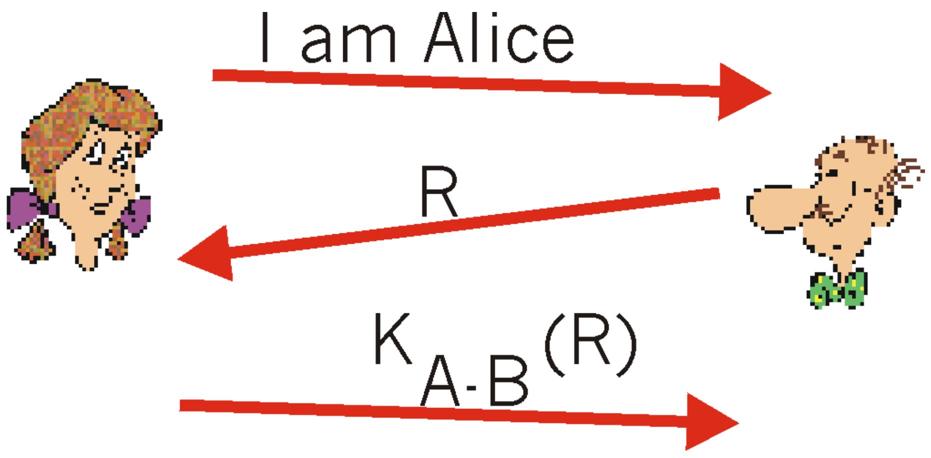


Figure 7.3-4: Protocol *ap 4.0:* no failure scenario.

Authentication protocol *ap5.0*

The use of a nonce and symmetric key cryptography formed the basis of our successful authentication protocol, *ap4.0*. A natural question is whether we can use a nonce and public key cryptography (rather than symmetric key cryptography) to solve the authentication problem. The use of a public key approach would obviate a difficulty in any shared key system - worrying about how the two parties learn the secret shared key value in the first place. A protocol that uses public key cryptography in a manner analogous to the use of symmetric key cryptography in protocol *ap4.0* is protocol *ap5.0:*

ap5.0:

- Alice sends the message, "I am Alice," to Bob
- Bob chooses a nonce, *R*, and sends it to Alice. Once again, the nonce will be used to insure that Alice is "live."
- Alice uses her decryption algorithm with her private key, d_A, to the nonce and sends the resulting value $d_A(R)$ to Bob. Since only Alice knows her decryption key, no one except Alice can generate $d_A(R)$.
- Bob applies Alice's public encryption algorithm, e_A to the received message, i.e., Bob computes $e_A(d_A(R))$. Recall from our discussion of RSA public key cryptography in section 7.2 that $e_A(d_A(R)) = R = d_A(e_A(R))$. Thus Bob computes *R* and authenticates Alice.

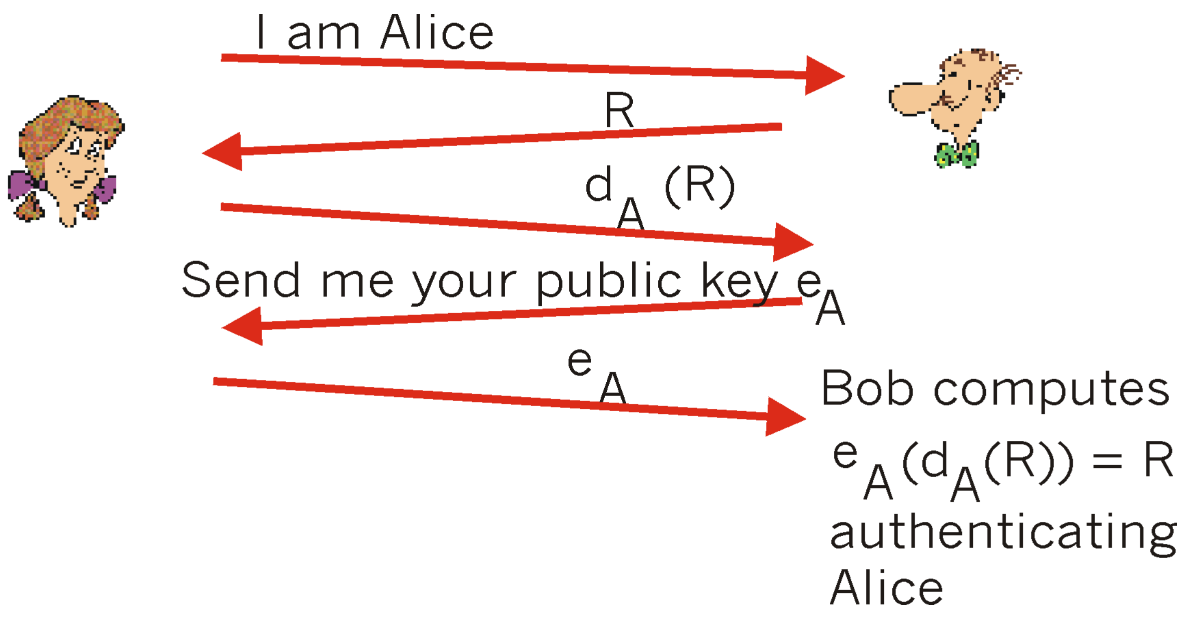


Figure 7.3-5: Protocol *ap 5.0* working correctly.

The operation of protocol *ap5.0* is illustrated in Figure 7.3-5.

Is protocol *ap5.0* as secure as protocol *ap4.0*? Both use nonces. Since *ap5.0* uses public key techniques, it requires that Bob retrieve Alice's public key. This leads to an interesting scenario, shown in Figure 7.3-6, in which Trudy may be able to impersonate Alice to Bob:

- Trudy sends the message, "I am Alice" to Bob
- Bob chooses a nonce, *R*, and sends it to Alice, but the message is intercepted by Trudy.
- Trudy applies her decryption algorithm with her private key, d_T, to the nonce and sends the resulting value, $d_T(R)$, to Bob. To Bob, $d_T(R)$ is just a bunch of bits and he doesn't know whether the bits represent $d_T(R)$ or $d_A(R)$.
- Bob must now get Alice's public key in order to apply e_A to the value he just received. He sends a message to Alice asking her for e_A. Trudy intercepts this message as well, and replies back to Bob with e_T, that is Trudy's public key. Bob computes $e_T(d_T(R)) = R$, and thus authenticates Trudy as Alice!

From the above scenario, it is clear that protocol *ap5.0* is only as "secure" as is the distribution of public keys. There *are* secure ways of distributing public keys, a topic we will examine soon in section 7.5.

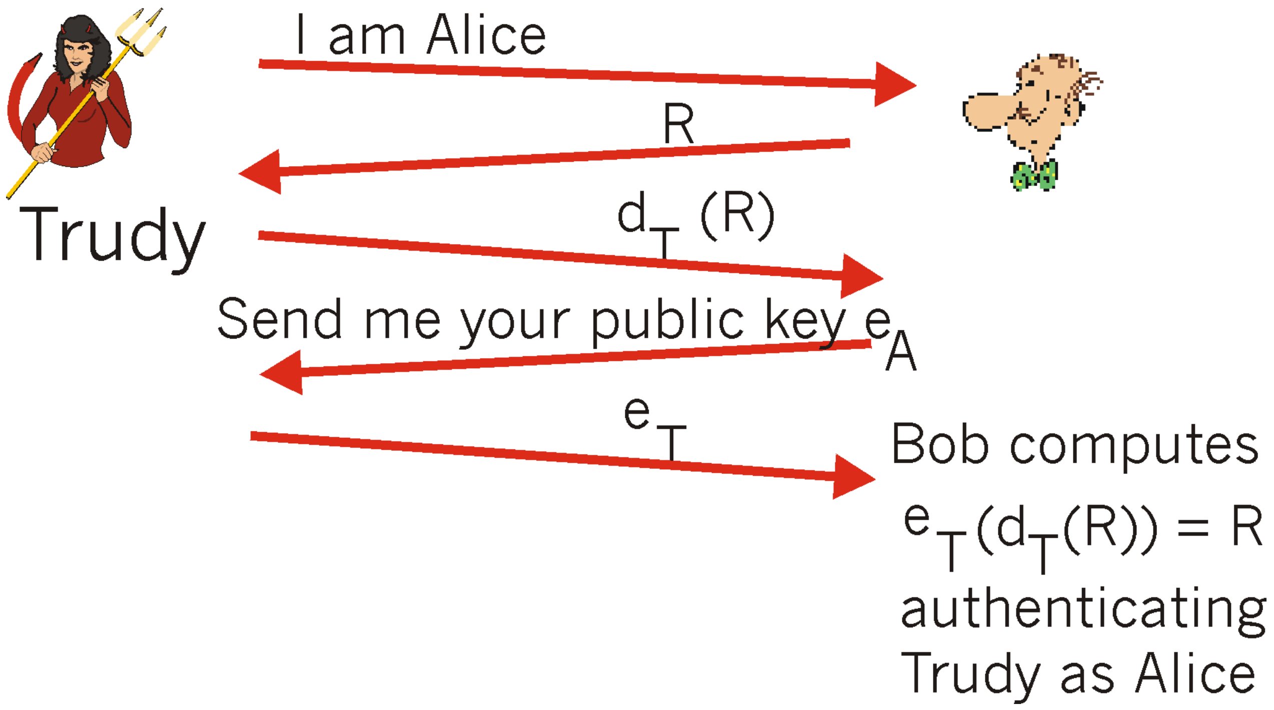


Figure 7.3-6: A security hole in protocol *ap5.0*

In the scenario in Figure 7.3-6, Bob and Alice might together eventually discover that something is amiss, as Bob will claim to have interacted with Alice, but Alice knows that she has never interacted with Bob. There is an even more insidious attack that would avoid this detection. In the scenario in Figure 7.3-7, both Alice and Bob are talking to each other, but by exploiting the same hole in the authentication protocol, Trudy is able to *transparently* interpose herself between Alice and Bob. In particular, if Bob begins sending encrypted data to Alice using the encryption key he receives from Trudy, Trudy can recover the plaintext of the communication from Bob to Alice. At the same time, Trudy can forward Bob's data to Alice (after re-encrypting data using Alice's real public key).

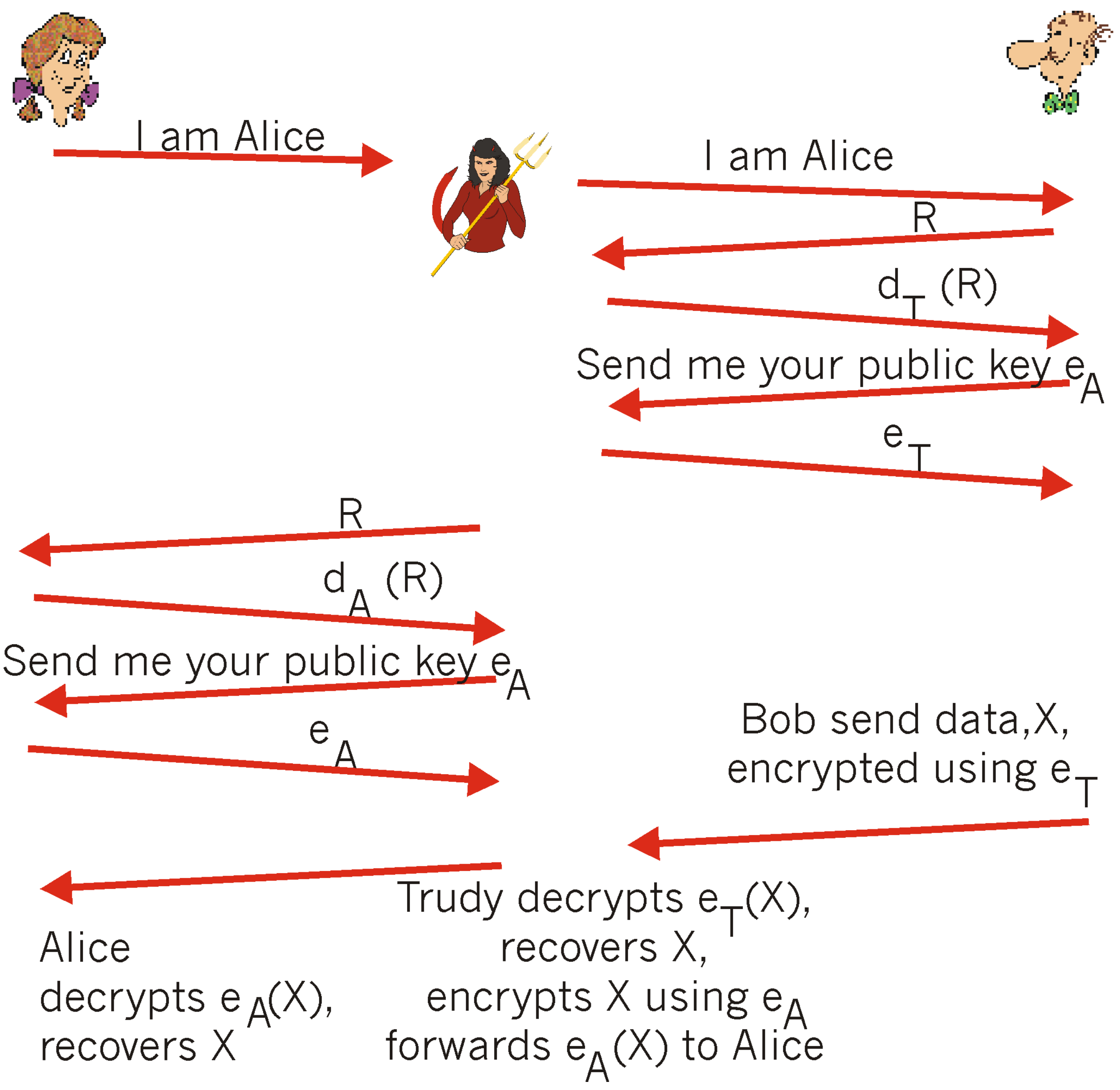

I am Alice
I am Alice
R
d_T (R)
Send me your public key e_A
e_T
R
d_A (R)
Send me your public key e_A
e_A
Bob send data,X, encrypted using e_T
Trudy decrypts e_T(X), recovers X, encrypts X using e_A forwards e_A(X) to Alice
Alice decrypts e_A(X), recovers X

Figure 7.3-7: A "man-in-the-middle" attack

Bob is happy to be sending encrypted data, and Alice is happy to be receiving data encrypted using her own public key; both are unaware of Trudy's presence. Should Bob and Alice meet later and discuss their interaction, Alice will have received exactly what Bob sent, so nothing will be detected as being amiss. This is one example of the so-called **man-in-the-middle attack** (more appropriately here, a "woman-in-the-middle" attack). It is also sometimes known as a **bucket-brigade attack**, since Trudy's passing of data between Alice and Bob resembles the passing of buckets of water along a chain of people (a so-called "bucket brigade") who are putting out a fire using a remote source of water.

References

[Cert 96] CERT, "Advisory CA-96.21: TCP SYN Flooding and IP Spoofing Attacks," http://www.cert.org/advisories/index.html

[Jimenez 1997] D. Jimenez, "Outside Hackers Infiltrate MIT Network, Compromise Security, " The Tech, Volume 117, Number 49 (Oct. 1997) , pp 1.
[Lamport 1981] Lamport, L., "Password Authentication with Insecure Communication", *Communications of the ACM*, Vol. 24, No. 11, November 1981, 770-772.
[RFC 1760] N. Haller, "The S/KEY One-Time Password System," RFC 1760, Feb. 1995.

Return to Table Of Contents

7.4 Integrity

Think of the number of the times you've signed your name to a piece of paper during the last week. You sign checks, credit card statements, legal documents, and letters. Your signature attests to the fact that you (as opposed to someone else) have acknowledged and/or agreed with the document's contents. In a digital world, one often want to indicate the owner or creator of a document, or to signify one's agreement with a document's content. A **digital signature** is a cryptographic technique for achieving these goals in a digital world.

Just as with human signatures, digital signing should be done in such a way that a digital signatures are verifiable, non-forgible, and non-repudiable. That is, it must be possible to "prove" that a document signed by an individual was indeed signed by that individual (the signature must be verifiable) and that *only* that individual could have signed the document (the signature can not be forged, and a signer can not later repudiate or deny having signed the document). This is easily accomplished with public key cryptography.

7.4.1 Generating Digital Signatures

Suppose that Bob wants to digitally sign a "document," *m*. We can think of the document as a file or a message that Bob is going to sign and send. As shown in Figure 7.4-1, to sign this document, Bob simply uses his private decryption key, d_B, to compute $d_B(m)$. At first, it might seem odd that Bob is running a decryption algorithm over a document that hasn't been encrypted. But recall that "decryption" is nothing more than a mathematical operation (exponentiation to the power of *d* in RSA; see section 7.2) and recall that Bob's goal is not to scramble or obscure the contents of the document, but rather to sign the document in a manner that is verifiable, non-forgible, and non-repudiable. Bob has the document, *m*, and his digital signature of the document, $d_B(m)$.

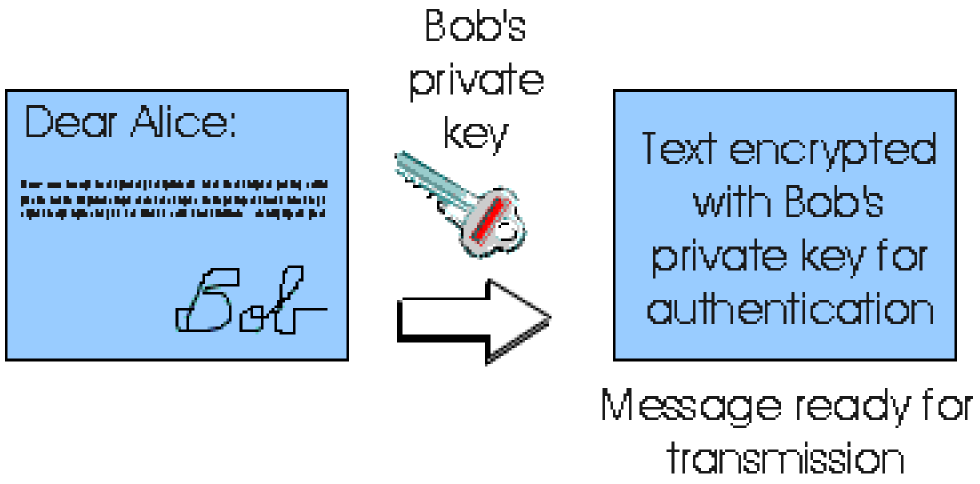

Dear Alice:
Bob's private key
Text encrypted with Bob's private key for authentication
Message ready for transmission

Figure 7.4-1: Creating a digital signature for a document.

Does the digital signature, $d_B(m)$, meet our requirements of being verifiable, non-forgible, and non-repudiable? Suppose Alice has m and $d_B(m)$. She wants to prove in court (being litigious) that Bob had indeed signed the document and was the only person who could have possibly signed the document. Alice takes Bob's public key, e_B, and applies it to the digital signature, $d_B(m)$, associated with the document, m. That is, she computes $e_B(d_B(m))$, and voila, with a dramatic flurry, she produces m, which exactly matches the original document! Alice then argues that only Bob could have signed the document because:

- Whoever signed the message must have used the private encryption key, d_B in computing the signature $d_B(m)$, such that $e_B(d_B(m)) = m$.
- The only person who could known the private key, d_B , is Bob. Recall from our discussion of RSA in section 7.2 that knowing the public key e_B is of no help in learning the private key d_B. Therefore, the only person who could know d_B is the person who generated the pair of keys, (e_B,d_B) in the firstplace, Bob.

It is also important to note that if the original document, m, is ever modified to some alternate form, m', the signature that Bob created for m will not be valid for m', since $e_B(d_B(m))$ does not equal m'.

Thus we see that public key cryptography techniques provide a simple and elegant way to digitally sign documents that is verifiable, non-forgible, and non-repudiable, and that protects against later modification of the document.

7.4.2 Message Digests

We have seen above that public key encryption technology can be used to create a digital signature. One concern with signing data by encryption, however, is that encryption and decryption are computationally expensive. When digitally signing a really important document, say a merger between two large multinational corporations or an agreement with a child to have him/her clean her room weekly, computational cost may not may be important. However, many network devices and processes (e.g., routers exchanging routing table information and email user agents exchanging email) routinely exchange data that may not need to be encrypted. Nonetheless, they do want to ensure that:

- the sender of the data is as claimed, i.e., that the sender has signed the data and this signature can be checked
- the transmitted data has not been changed since the sender created and signed the data.

Given the overheads of encryption and decryption, signing data via complete encryption/decryption can be overkill. A more efficient approach using so-called message digests can accomplish these two goals without full message encryption.

A **message digest** is in many ways like a checksum. Message digest algorithms take a message, *m*, of arbitrary length and compute a fixed length "fingerprint" of the data known as a message digest, *H(m)*. The message digest protects the data in the sense that if *m* is changed to *m'* (either maliciously or by accident) then *H(m)*, computed for the original data (and transmitted with that data), will not match the *H(m)* computed over the changed data. While the message digest provides for data integrity, how does it help with signing the message *m*? The goal here is that rather than having Bob digitally sign (encrypt) the entire message by computing $d_B(m)$, he should be able to sign just the message digest by compting $d_B(H(m))$. That is, having *m* and $d_B(H(m))$ together (note that *m* is not typically encrypted) should be "just as good as" having a signed complete message, $d_B(m)$; this means that *m* and $d_B(H(m))$ together should be non-forgible, verifiable, and non-repudiable. Nonforgible will require that the message digest algorithm that computes the message digest have some special properties, as we will see below.

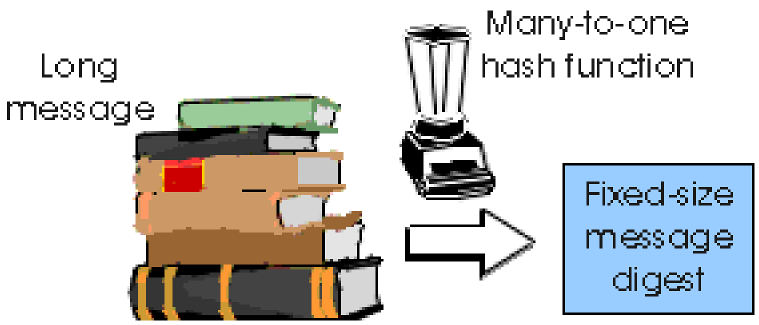


Figure 7.4-2: Hash functions are used to create message digests.

Our definition of a message digest may seem quite similar to the definition of a checksum (e.g., the Internet checksum, see section 4.4) or a more powerful error detection code such as a cyclic redundancy check (see section 5.1). Is it really any different? Checksums, cyclic redundancy checks, and message digests are all examples of so-called **hash functions.** As shown in Figure 7.4-2, a hash function takes an input, *m*, and computes a fixed-size string known as a hash. The Internet checksum, CRC's and message digests all meet this definition. If signing a message digest is going to be "just as good as" signing the entire message, in particular if it is going to satisfy the non-forgibility requirement, then a message digest algorithm must have the following additional properties:

1. Given a message digest value, *x*, it is computationally infeasible to find a message, *y*, such that $H(y) = x$;
2. It is computationally infeasible to find any two messages *x* and *y* such that $H(x) = H(y)$.

Informally, these two properties mean that it is computationally infeasible for an intruder to substitute one message for another message that is protected by a message digest. That is, if *(m,H(m))* are the message and message digest pair created by the sender, then an intruder can not forge the contents of another message, *y*, that has the same message digest value as the original message. When Bob signs *m* by computing $d_B(H(m))$, we know that no other message can be substituted for *m*. Furthermore, Bob's digital signature of *H*(*m*) uniquely identifies Bob as the verifiable, non-repudiable signer of *H(m)* (and as a consequence, *m* as well) as discussed above in section 7.4.1.

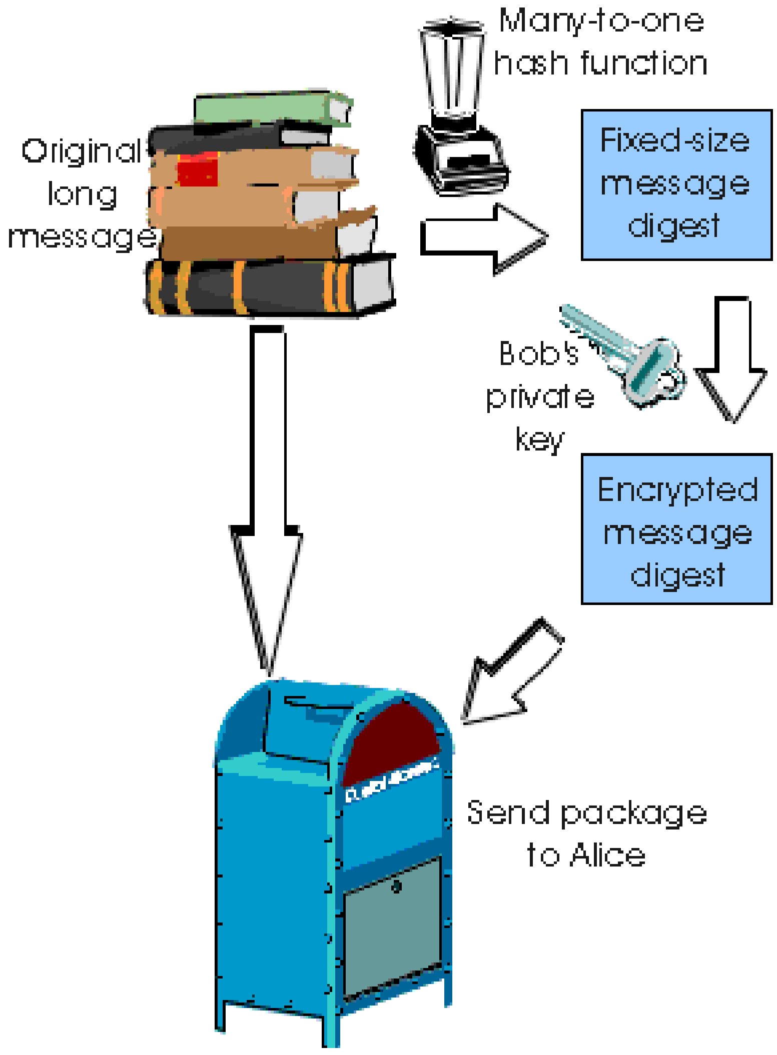


Figure 7.4-3: Sending a digitally signed message.

In the context of Bob sending a message to Alice, Figure 7.4-3 provides a summary of the operational procedure of creating a digital signature. Bob puts his original long message through a hash function to create a messge digest. He then encrypts the message digest with his own private key. The original message (in clear text) along with the digitally signed message digest (henceforth referred to as the digital signature) is then sent to Alice. Figure 7.4-4 provides a summary of the operational procedure of verifying message integrity. Alice applies the Bob's public key to the message to recover the message digest. Alice also applies the hash function to the clear text message to obtain a second message digest. If the two message digests match, then the recipientAlice can be sure about the integrity of the message, and sure that Bob sent the message.

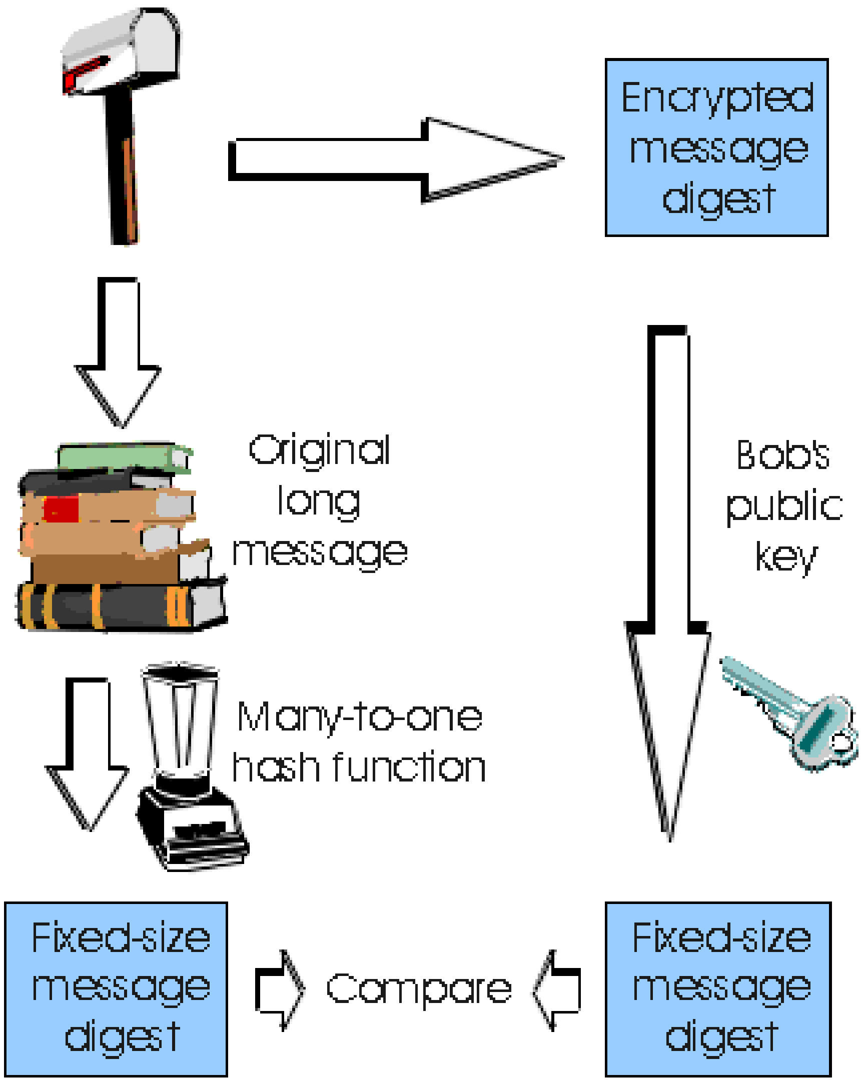

Encrypted message digest
Original long message
Many-to-one hash function
Bob's public key
Fixed-size message digest
Compare
Fixed-size message digest

Figure 7.4-4: Verifying the integrity of a signed message.

7.4.3 Hash Function Algorithms

Let's convince ourselves that a simple checksum, such as the Internet checksum, would make a poor message digest algorithm. Rather than performing 1 complement's arithmetic (as in the Internet checksum), let us compute a checksum by treating each character as a byte and adding the bytes together using 4-byte chunks at a time. Suppose Bob owes Alice $100.99" and sends an IOU to Alice consisting of the text string "IOU100.99BOB". The ASCII representation (in hexadecimal notation) for these letters is 49, 4F, 55, 31, 30, 30, 2E, 39, 39, 42, 4F, 42.

Figure 7.4-5 (top) shows that the 4-byte checksum for this message is B2 C1 D2 AC. A slightly different message (and a much more costly one for Bob) is shown in the bottom half of Figure 7.5-1. The message "IOU100.99BOB" and "IOU900.19BOB" have the same checksum! Thus, this simple checksum algorithm violates the two required requirements above. Given the original data, it is simple to find another set of data with the same checksum. Clearly, for security purposes. we are going to need a more powerful hash function than a checksum.

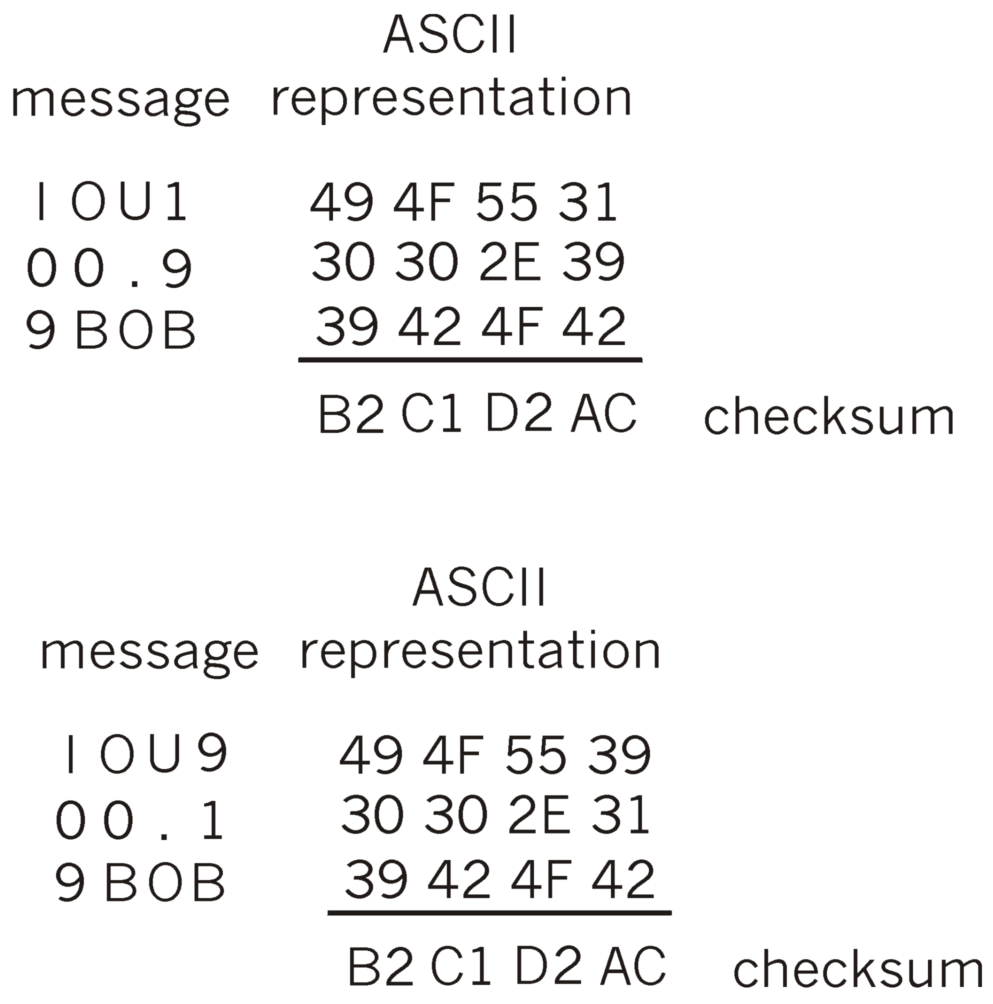


Figure 7.4-5: Initial message and fraudulent message have the same checksum!

The MD5 message digest algorithm by Ron Rivest [RFC 1321] is in wide use today. It computes a 128-bit message digest in a four-step process consisting of a padding step (adding a 1 followed by enough zero's so that the length of the message satisfies certain conditions), an append step (appending a 64-bit representation of the message length before padding), an initialization of an accumulator, and a final looping step in which the message's 16-word blocks are processed (mangled) in four rounds of processing. It is not known whether MD5 actually satisfies the requirements listed above. The author of MD5 claims "It is conjectured that the difficulty of coming up with two messages having the same message digest is on the order of 2^{64} operations, and that the difficulty of coming up with any message having a given message digest is on the order of 2^{128} operations. "[RFC 1321]. No one has argued with this claim. For a description of MD5 (including a C source code implementation) see [RFC 1321]. Computational aspects of MD5 are discussed in [RFC 1810].

The second major message digest algorithm in use today is SHA-1, the Secure Hash Algorithm [FIPS 1995]. This algorithm is based on principles similar to those used in the design of MD4 [RFC 1320], the predecessor to MD5. The Secure Hash Algorithm (SHA-1), a US federal standard, is required for use whenever a secure message digest algorithm is required for federal applications. It produces a 160-bit message digest.

References

[FIPS 95] Federal Information Processing Standard, "Secure Hash Standard", FIPS Publication 180-1.
[RFC 1320] R. Rivest, The MD4 Message-Digest Algorithm, RFC 1320, April 1992.
[RFC 1321] R.L. Rivest, The MD5 Message-Digest Algorithm, RFC 1321, April 1992.
[RFC 1810] J. Touch, "Report on MD5 Performance," RFC 1810, June 1995.

7.5 Key Distribution and Certification

In section 7.2 we saw that a drawback of symmetric key cryptography was the need for the two communicating parties to have agreed upon their secret key ahead of time. With public key cryptography, this *a priori* agreement on a secret value is not needed. However, as we saw in our discussion of authentication protocol *ap5.0* in Section 7.3, public key encryption has its own difficulties, in particular the problem of obtaining someone's true public key. Both of these problems - determining a shared key for symmetric key cryptography, and securely obtaining the public key for public key cryptography - can be solved using a **trusted intermediary.** For symmetric key cryptograghy , the trusted intermediary is called a **Key Distribution Center (KDC**), which is a single, trusted network entity with whom one has established a shared secret key. We will see that one can use the KDC to obtain the shared keys needed to communicate securely with *all* other network entities. For public key cryptography, the trusted intermediary is called a **Certification Authority (CA)**. A certification authority certifies that a public key belongs to a particular entity (a person or a network entity). For a certified public key, if one can safely trust the CA that the certified the key, then one can be sure about to whom the public key belongs. Once a public key is certified, then it can be distributed from just about anywhere, including a public key server, a personal Web page or a diskette.

7.5.1 The Key Distribution Center

Suppose once again that Bob and Alice want to communicate using symmetric key cryptography. They have never met (perhaps they just met in an on-line chat room) and thus have not established a shared secret key in advance. How can they now agree on a secret key, given that they can only communicate with each other over the network? A solution often adopted in practice is to use a trusted Key Distribution Center (KDC).

The KDC is a server that shares a different secret symmetric key with each registered user. This key might be manually installed at the server when a user first registers. The KDC knows the secret key of each user and each user can communicate securely with the KDC using this key. Let's see how knowledge of this one key allows a user to securely obtain a key for communicating with any other registered user. Suppose that Alice and Bob are users of the KDC; they only know their individual key, $K_{A\text{-}KDC}$ and $K_{B\text{-}KDC}$, respectively, for communicating securely with the KDC. Alice takes the first step, and they proceed as illustrated in Figure 7.5-1.

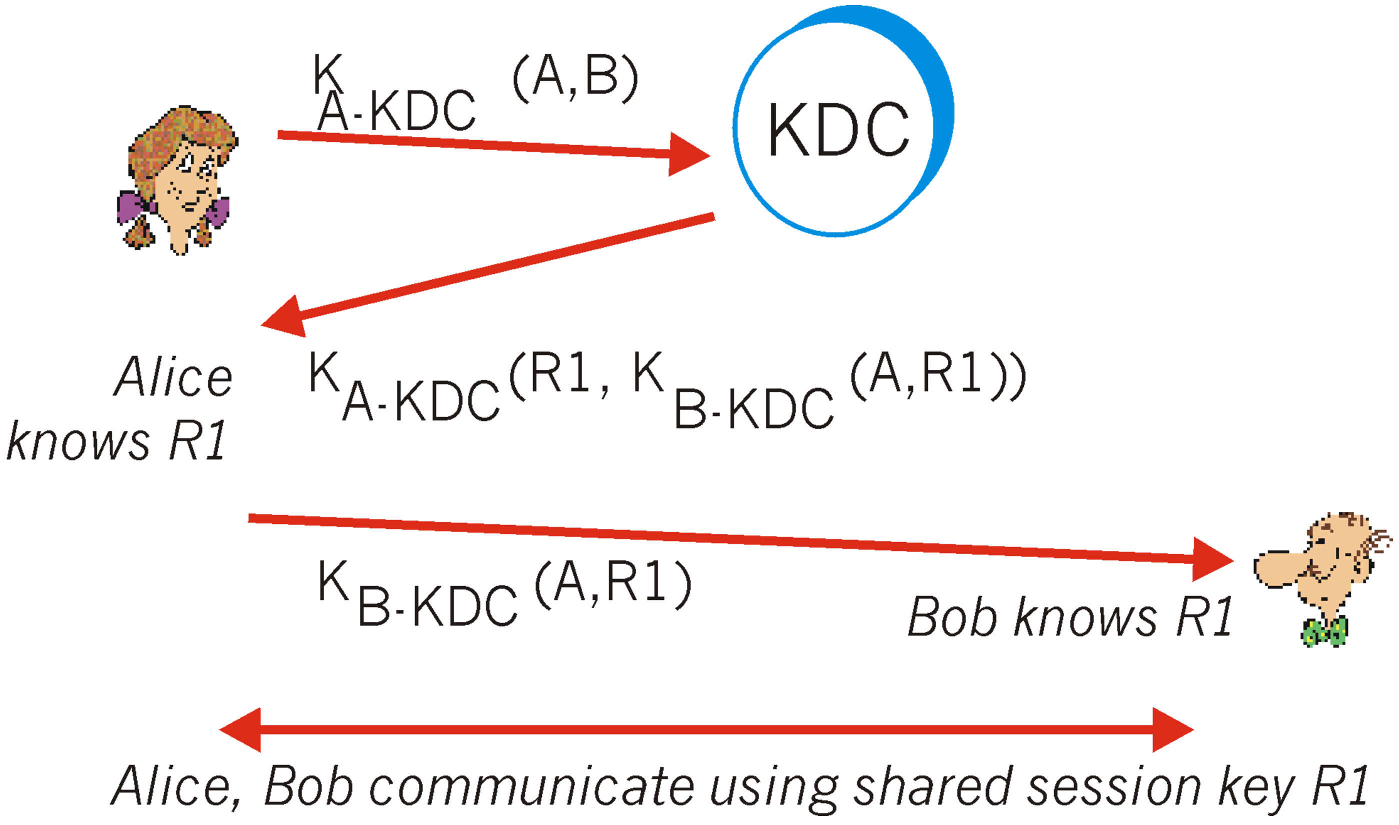

$K_{A\text{-}KDC}$ (A,B)
KDC
Alice knows R1
$K_{A\text{-}KDC}(R1, K_{B\text{-}KDC}(A,R1))$
$K_{B\text{-}KDC}(A,R1)$
Bob knows R1
Alice, Bob communicate using shared session key R1

Figure 7.5-1: Setting up a one-time session key using a Key Distribution Center

- Using $K_{A\text{-}KDC}$ to encrypt her communication with the KDC, Alice sends a message to the KDC saying she (*A*) wants to communicate with Bob (*B*). We denote this message, $K_{A\text{-}KDC}$ *(A,B).* As part of this exchange, Alice should authenticate the KDC (see homework problems), e.g., using an authentication protocol (e.g., our protocol *ap4.0)* and the shared key $K_{A\text{-}KDC}$.
- The KDC, knowing $K_{A\text{-}KDC}$, decrypts $K_{A\text{-}KDC}$ *(A,B).* The KDC then authenticates Alice. The KDC then generates a random number, *R1*. This is the shared key value that Alice and Bob will use to perform symmetric encryption when they communicate with each other. This key is referred to as a **one-time session key** (see section 7.5.3 below), as Alice and Bob will use this key for only this one session that they are currently setting up. The KDC now needs to inform Alice and Bob of the value of *R1*. The KDC thus sends back an encrypted message to Alice containing the following:
  - *R1*, the one-time session key that Alice and Bob will use to communicate;
  - a pair of values: A, and *R1*, encrypted by the KDC using Bob's key, $K_{B\text{-}KDC}$. We denote this $K_{B\text{-}KDC}$*(A,R1)*. It is important to note that KDC is sending Alice not only the value of *R1* for her own use, but also an encrypted version of *R1* and Alice's name encrypted using Bob's key. Alice can't decrypt this pair of values in the message (she doesn't know Bob's encryption key), but then she doesn't really need to. We'll see shortly that Alice will simply forward this encrypted pair of values to Bob (who can decrypt them).

  These items are put into a message and encrypted using Alice's shared key. The message from the

KDC to Alice is thus $K_{A\text{-}KDC}(R1,K_{B\text{-}KDC}(R1))$.

- Alice receives the message from the KDC, verifies the nonce, extracts *R1* from the message and saves it. Alice now knows the one-time session key, *R1*. Alice also extracts $K_{B\text{-}KDC}(A,R1)$ and forwards this to Bob.
- Bob decrypts the received message, $K_{B\text{-}KDC}(A,R1)$, using $K_{B\text{-}KDC}$ and extracts A and *R1*. Bob now knows the one-time session key, *R1,* and the person with whom he is sharing this key, *A*. Of course, he takes care to authenticate Alice using *R1* before proceeding any further.

7.5.2 Kerberos

Kerberos [RFC 1510, Neuman 1994] is an authentication service developed at MIT that uses symmetric key encryption techniques and a Key Distribution Center. Although it is conceptually the same as the generic KDC we described in section 7.5.1, its vocabulary is slightly different. Kerberos also contains several nice variations and extensions of the basic KDC mechanisms. Kerberos was designed to authenticate users accessing network servers and was initially targeted for use within a single administrative domain such as a campus or company. Thus, Kerberos is framed in the language of users who want to access network services (servers) using application-level network programs such as Telnet (for remote login) and NFS (for access to remote files), rather than than human-to-human conversants who want to authenticate themselves to each other, as in our examples above. Nonetheless, the key (pun intended) underlying techniques remains the same.

The **Kerberos Authentication Server** (AS) plays the role of the KDC. The AS is the repository of not only the secret keys of all users (so that each user can communicate securely with the AS) but also information about which users have access privileges to which services on which network servers. When Alice wants to access a service on Bob (who we now think of as a server), the protocol closely follows our example in Figure 7.5-1:

- Alice contacts the Kerberos AS, indicating that she wants to use Bob. All communication between Alice and the AS is encrypted using a secret key that is shared between Alice and the AS. In Kerberos, Alice first provides her name and password to her local host. Alice's local host and the AS then determine the one-time secret session key for encrypting communication between Alice and the AS.
- The AS authenticates Alice, checks that she has access privileges to Bob, and generates a one-time symmetric session key, R1, for communication between Alice and Bob. The Authentication Server (in Kerberos parlance, now referred to as the Ticket Granting Server) sends Alice the value of R1, and also a **ticket** to Bob's services. The ticket contains Alice's name, the one-time session key, R1, and an expiration time, all encrypted using Bob's secret key (known only by Bob and the AS), as in Figure 7.5-1. Alice's ticket is valid only until its expiration time, and will be rejected by Bob is presented after that time. For Kerberos V4, the maximum lifetime of a ticket is about 21 hours. In Kerberos V5, the lifetime must expire before the end of year 9999 - a definite Y10K problem!
- Alice then sends her ticket to Bob. She also sends along an R1-encrypted timestamp that is used as a nonce. Bob decrypts the ticket using his secret key, obtains the session key, decrypts the

timestamp using the just-learned session key. Bob sends back the timestamp value plus one (in Kerberos V5) or simply the timestamp itself (in Kerberos V5).

The most recent version of Kerberos (V5) provides support for multiple Authentication Servers, delegation of access rights, and renewable tickets. [Kaufman 95] [RFC 1510] provide ample details.

7.5.3 Public Key Certification

One of the principle features of public key encryption is that it is possible for two entities to exchange secret messages without having to exchange secret keys. For example, when Alice wants to send a secret message to Bob, she simply encrypts the message with Bob's public key and sends the encrypted message to Bob; she doesn't need to know Bob's secret (i.e., private) key, nor does Bob need to know her secrect key. Thus, public key cryptography obviates the need for KDC infrastructure, such as Kerberos.

Of course, with public key encryption, the communicating entities still have to exchange public keys. A user can make its public key pubicly available in many ways, e.g., by posting the key on the user's personal Web page, placing the key in a public key server, or by sending the key to a correspondent by e-mail. A Web commerce site can place its public key on its server in a manner that browsers automatically download the public key when connecting to the site. Routers can place their public keys on public key servers, thereby allowing other browsers and network entities to retrieve them.

There is, however, a subtle, yet critical, problem with public key cryptography. To gain insight to this problem, let's consider an Internet commerce example. Suppose that Alice is in the pizza delivery business and she accepts orders over the Internet. Bob, a pizza lover, sends Alice a plaintext message which includes his home address and the type of pizza he wants. In this message, Bob also includes a digital signature (e.g.,, an encrypted message digest for the original plaintext message). As discussed in Section 7.4, Alice can obtain Bob's public key (from his personal Web page, a public key server, or from an e-mail message) and verify the digital signature. In this manner Alice makes sure that Bob (rather than some adolescent prankster) indeed made the order.

This all sounds fine until clever Trudy comes along. As shown in Figure 7.5-2, Trudy decides to play a prank. Trudy sends a message to Alice in which she says she is Bob, gives Bob's home address, and orders a pizza. She also attaches a digital signature, but she attaches the signature by signing the message digest with her (i.e., Trudy's) private key. Trudy also masquerades as Bob by sending Alice Trudy's public key but saying that it belongs to Bob. In this example, also will apply Trudy's public key (thinking that it is Bob's) to the digital signature and conclude that the plaintext message was indeed created by Bob. Bob will be very surprised when the delivery person brings to his home a pizza with everything on it! Here, as in the flawed authentication scenario in Figure 7.3-7, the man-in-the-middle attack is the room cause of our difficulties.

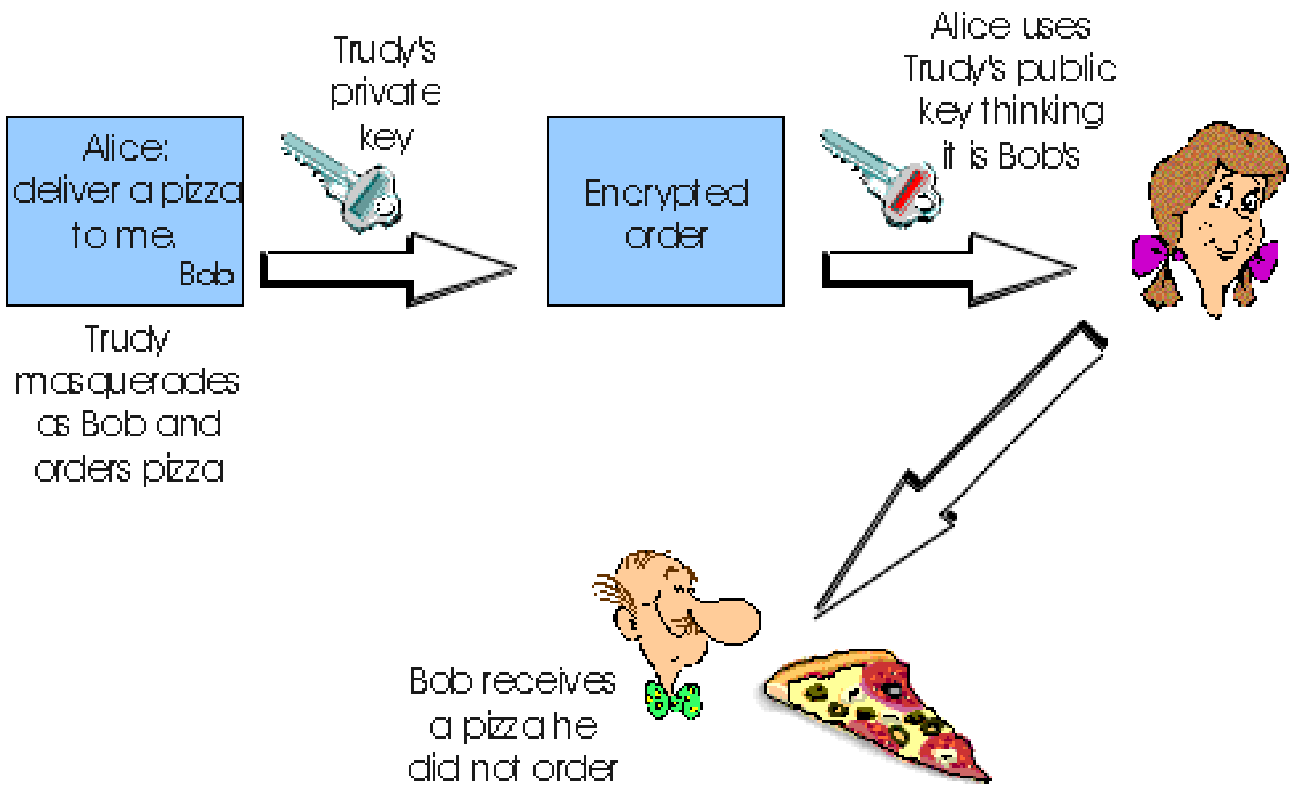

Alice: deliver a pizza to me. Bob
Trudy masquerades as Bob and orders pizza
Trudy's private key
Encrypted order
Alice uses Trudy's public key thinking it is Bob's
Bob receives a pizza he did not order

Figure 7.5-2: Trudy masquerades as Bob using public key cryptography.

We see from this example that in order for public key cryptography to be useful, entities (users, browsers, routers, etc.) need to know *for sure* that they have the public key of the entity with which they are communicating. For example, when Alice is communicating with Bob using public key cryptography, she needs to know for sure that the public key that is supposed to be Bob's is indeed Bob's.

Binding a public key to a particular entity is typically done by a **certification authority** (CA), which validates identities and issue certificates. A CA has the following roles:

- First to verify that entity (a person, a router, etc) is who it says it is. There are no mandated procedures for how certification is done. When dealing with a CA, one must trust the CA to have performed a suitably rigorous identity verification. For example, if Trudy were able to walk into Fly-by-Night Certificate Authority and simply announce "I am Alice" and receive keys associated with the identity of "Alice," then one shouldn't put much faith in public keys offered by the Fly-by-Night Certificate Authority. On the other hand, one might (or might not!) be more willing to trust a CA that is part of a federal- or state-sponsored program (e.g., [Utah 1999]). One can trust the "identify" associated with a public key only to the extent that one can trust a CA and its identity verification techniques. What a tangled web of trust we spin!
- Once the CA verifies the entity of the entity, the CA creates a **certificate** that binds the public key of the identiy to the identity. The certificate contains the public key and identifying information about the owner of the public key (for example a human name or an IP address). The certificate is digitally signed by the CA. These steps are shown in Figure 7.5-3.

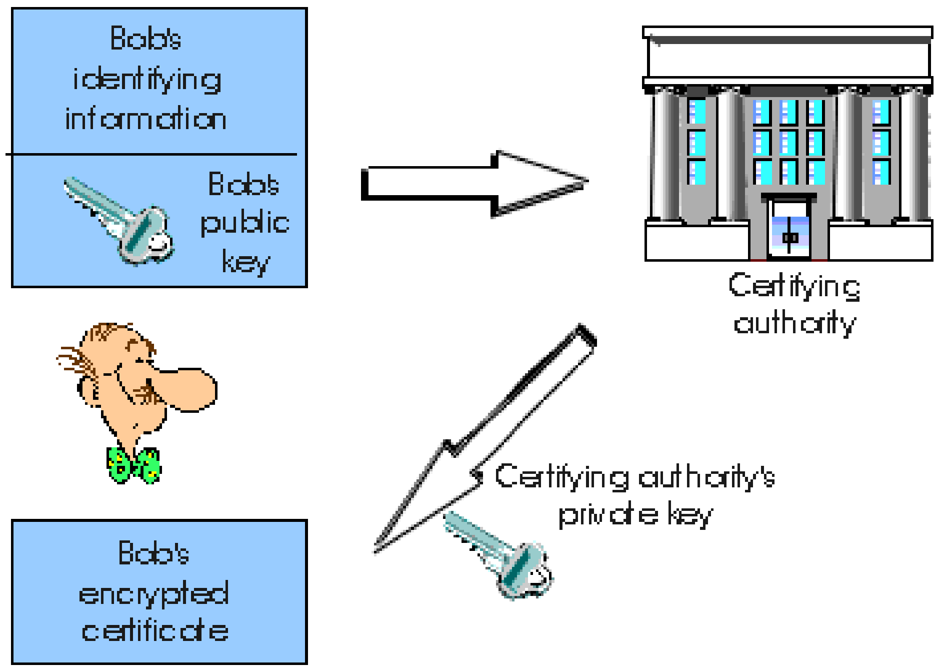

Bob's identifying information
Bob's public key
Certifying authority
Certifying authority's private key
Bob's encrypted certificate

Figure 7.5-3: Bob obtains a certificate from the certification authority.

Let us now see how certificates can be used to combat pizza-ordering pranksters, like Trudy, and other undesirables. When Alice recieves Bob's order, she gets Bob's certificate, which may be on his Web page, in an e-mail message or in a certificate server. Alice uses the CA's public key to verify that the public key in the certificate is indeed Bob's. If we assume that the public key of the CA itself is known to all (for example, it could published in a trusted, public, and well-known place, such as *The New York Times*, so that it is known to all and can not be spoofed), then Alice can be sure that she is indeed dealing with Bob.

Both the International Telecommunication Union and the IETF have developed standards for Certification Authorities. ITU X.509 [ITU 1993] specifies an authentication service as well as a specific syntax for certificates. RFC 1422 [RFC 1422] describes CA-based key management for use with secure Internet e-mail. It is compatible with X.509 but goes beyond X.509 by establishing procedures and conventions for a key management architecture. Figure 7.5-4 describes some of the important field in a certificate.

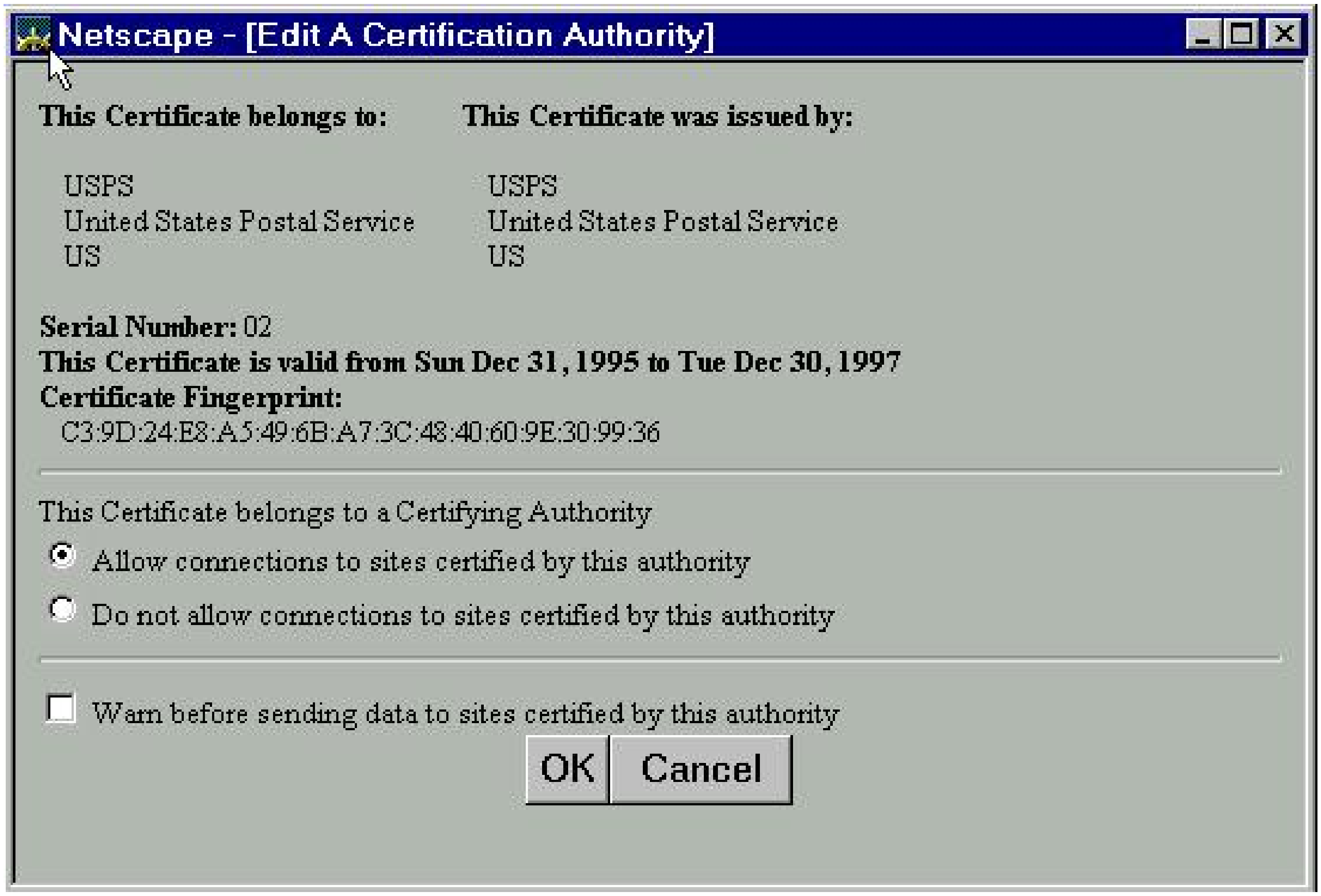
Field name	**Description**
version	version number of X.509 specification
serial number	CA-issued unique identifier for a certificate
signature	specifies the algorithm used by Ca to "sign" this certificate
issuer name	identity of CA issuing this certificate, in so-called Distinguished Name(DN) [RFC 1779] format

validity period	start and end of period of validity for certificate
subject name	identity of entity whose public key is associated with this certificate, in DN format
subject public key	the subject's public key as well as an indication of the public key algorithm (and algorithm parameters) to be used with this key

Figure 7.5-4: Selected fields in a X.509 and RFC 1422 public key certificate

With the recent boom in electronic commerce and the consequent widespread need for secure transactions, there has been increased interest in Certification Authorities. Among the companies providing CA services are Cybertrust [Cybertrust 1990] Verisign [Verisign 1999] and Netscape [Netscape 1999].

A certificate issued by the US Postal Service, as viewed through a Netscape browser, is shown in Figure 7.5-5.



This Certificate belongs to:

USPS
United States Postal Service
US

This Certificate was issued by:

USPS
United States Postal Service
US

Serial Number: 02
This Certificate is valid from Sun Dec 31, 1995 to Tue Dec 30, 1997
Certificate Fingerprint:
C3:9D:24:E8:A5:49:6B:A7:3C:48:40:60:9E:30:99:36

This Certificate belongs to a Certifying Authority

[✓ Allow connections to sites certified by this authority]
Do not allow connections to sites certified by this authority

☐ Warn before sending data to sites certified by this authority

Figure 7.5-5: A US Postal Service issued certificate

7.5.4 One-Time Session Keys

We have seen above that a **one-time session key** is generated by a KDC for use in symmetric key encryption of a single session between two parties. By using the one-time session keys from the KDC, a user is freed from having to establish *a priori* its own shared key for each and every network entity with whom it wishes to communicate. Instead, a user need only have one shared secret key for communicating with the KDC, and will receive one-time session keys from the KDC for all of its communication with other network entities.

One time session keys are also used in public key cryptography. Recall from our discussion in section 7.2.2, that a public key encryption technique such as RSA is orders of magnitude more computationally expensive that a symmetric key system such as DES. Thus, public key systems are often used for authentication purposes. Once two parties have authenticated each other, they then use public-key-encrypted communication to agree on a shared one-time symmetric session key. This symmetric session key is then used to encrypt the remainder of the communication using a more efficient symmetric encryption technique, such as DES.

References

[Cybertrust 1999] Cybertrust Solutions homepage, http://www.cybertrust.com
[ITU 1993] International Telecommunication Union, Recommendation X.509 (11/93) The Directory: Authentication framework
[Kaufman 1995] C. Kaufman, R. Perlman, M. Speciner, Network Security, Private Communication in a Public World, Prentice Hall, 1995.
[Netscape 1999] Netscape Certificate Server FAQ, http://sitesearch.netscape.com/certificate/v1.0/faq/index.html
[Neuman 1994] B. Neuman and T. Tso, "Kerberos: An Authentication Service for Computer Networks," IEEE Communication Magazine, Vol. 32, No. 9, (Sept. 1994), pp. 33-38.
[RFC 1422] S. Kent, "Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management", RFC 1422, Feb., 1993.
[RFC 1510] J. Kohl, C. Neuman, "The Kerberos Network Authentication Service (V5)," RFC 1510, Sept. 1993.
[**RFC 1779]** S. Kille, "A String Representation of Distinguished Names," RFC 1779, March 1995.
[Utah 1999] State of Utah Department of Commerce, Utah Digital Signature Program, http://www.commerce.state.ut.us/web/commerce/digsig/dsmain.htm
[Verisign 1999] Verisign home page, http://www.verisign.com/

7.6 Secure E-Mail

In previous sections of this chapter, we have examined fundamental issues in network security, including symmetric key and public key encryption, authentication, key distribution, message integrity and digital signatures. In this section and the following two sections, we'll next examine how these techniques are being used to provide security in the Internet. Being consistent with the general structure of this book, we begin at the top of the protocol stack and discuss application-layer security. Our approach here is use a specific application, namely, e-mail, as a case study for application-layer security. We then move down the protocol stack. In Section 7.7 we examine the SSL protocol, which provides security at the transport layer for TCP. And in Section 7.8, we'll consider IPsec, which provides security at the network layer.

Interestingly, it is possible to provide security services in any of the top four layers of the Internet protocol stack [Molva 1999]. When security is provided for a specific application-layer protocol, then the application using the protocol will enjoy one or more security services, such as secrecy, authentication or integrity. When security is provided for a transport-layer protocol, then all applications that use that protocol enjoy the security services of the transport protocol. When security is provided at the network layer on a host-to-host basis, then all transport layer segments (and hence all application-layer data) enjoy the security services of the network layer. When security is provided on a link basis, then all IP datagrams traveling over the link receive security services of the link.

One might wonder why security functionality is being provided at multiple layers in the Internet? Wouldn't it suffice to simply provide the security functionality at the network layer, and be done with it? There are two answers to this question. First, although security at the network layer can offer "blanket coverage" by encrypting all the data in the datagrams (i.e., all the transport-layer segments) and by authenticating all source IP addresses, it can't provide user-level security. For example, a commerce site can not rely on IP-layer security to authenticate a customer who is purchasing goods at the commerce site. Thus, there is a need for security functionality at higher layers as well as blanket coverage at lower layers. Second, in the Internet it is generally easier to deploy new services, including security services, at the higher-layers of the protocol stack. While waiting for security to be broadly deployed at the network layer (which is arguably still many years in the future) many application developers "just do it" and introduce security functionality into to their favorite applications. A classic example is PGP, which provides for encryption of e-mail (and will be discussed later in this section). Requiring only client and server application code, PGP was one the first security technologies to be broadly used in the Internet. Similarly, transport-layer security with SSL was broadly introduced into the Internet, as it too only required new code in the end systems. However, IP-layer security -- so-called IPsec -- is taking much longer to broadly deploy, as it requires significant changes in the routers in the network core.

7.6.1 Principle of Secure E-Mail

In this section we use many of the tools introduced in the previous section to create a high-level design of a secure e-mail system. We create this high-level design in an incremental manner, at each step introducing new security services. When designing a secure e-mail system, let us keep in mind the racy example introduced in Section 7.1 -- the illicit love affair between Alice and Bob. In the context of e-mail, Alice wants to send an e-mail message to Bob, and Trudy wants to intrude.

Before plowing ahead and designing a secure e-mail system for Alice and Bob, we should first consider which security features would be most desirable for them. First and foremost is *secrecy*. As discussed in Section 7.1, neither Alice nor Bob wants Trudy to read Alice's e-mail message. The second feature that Alice and Bob would most likely want to see in the secure e-mail system is *sender authentication*. In particular, when Bob receives the message from Alice, "*I don't* *love you anymore. I never want to see you again. Formerly yours, Alice*" , Bob would naturally want to be sure that the message came from Alice and not from Trudy. Another feature that the two lovers would appreciate is *message integrity*, i.e., assurance that the message Alice sends is not modified while enroute to Bob. Finally, the e-mail system should provide *receiver authentication*, i.e., Alice wants to make sure that she is indeed sending the letter to Bob and not to someone else (e.g., Trudy) who is impersonating as Bob.

So let's begin by addressing the foremost concern of Alice and Bob, namely, secrecy. The most straightforward way to provide secrecy is for Alice to encrypt the message with symmetric key technology (such as DES) and for Bob to decrypt the message upon message receipt. As discussed in Section 7.2, if the symmetric key is long enough, and if only Alice and Bob have the key, then it is extremely difficult for anyone else (including Trudy) to read the message. Although this approach is straightforward, it has a fundamental problem as we discussed in Section 7.2 -- it is difficult to distribute a symmetric key so that only Alice and Bob have copies of the key. So we naturally consider an alternative approach, namely, public key cryptography (using, for example, RSA). In the public-key approach, Bob makes his public key publicly available (for example, in a public-key server or on his personal Web page), Alice encrypts her message with Bob's public key, and sends the encrypted message to Bob's e-mail address. (The encrypted message is encapsulated with MIME headers and sent over ordinary SMTP, as discussed in Section 2.4.) When Bob receives the message, he simply decrypts it with his private key. Assuming that Alice knows for sure that the public key is Bob's public key (and that the key is long enough), then this approach is an excellent means to provide the desired secrecy. One problem, however, is that public-key encryption is relatively inefficient, particularly for long messages. (Long e-mail messages are now commonplace in the Internet, due to increasing use of attachments, images, audio and video.) To overcome the efficiency problem, let's make use of a session key (discussed in Section 7.4). In particular, Alice (1) selects a symmetric key, K_S, at random, (2) encrypts her message, m, with the symmetric key, K_S,(3) encrypts the symmetric key with Bob's public key, e_B, (4) concatenates the encrypted message and the encrypted symmetric key to form a "package", and (5) sends the package to Bob's e-mail address. The steps are illustrated in Figure 7.6-1. (In this and the subsequent figures, the "+" represents concatenation and the "-" represents de-concatenation.) When Bob receives the package, he (1) uses his private key d_B to obtain the symmetric key, S, and (2) uses the symmetric key S to decrypt the message m.

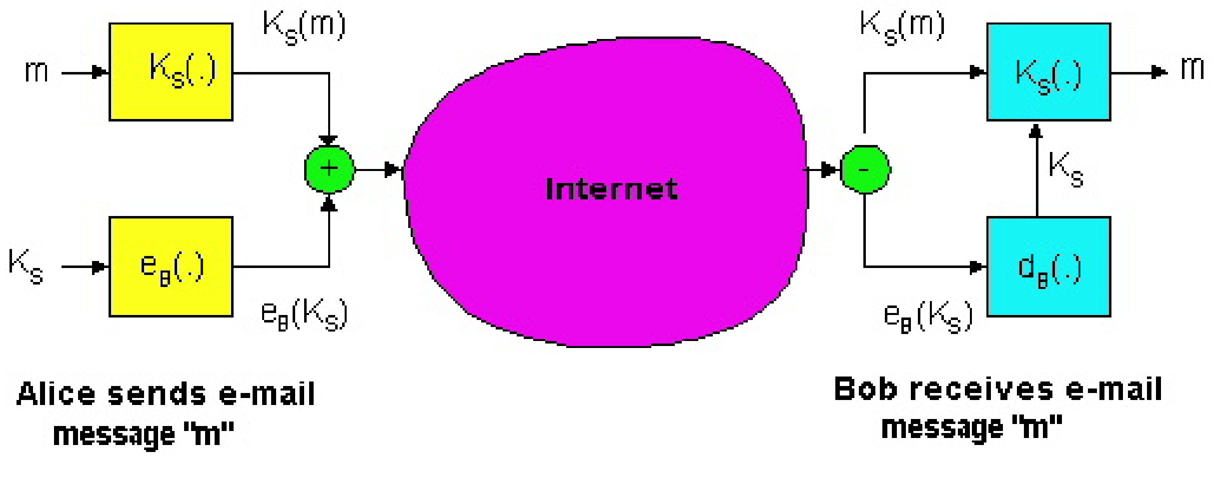


Figure 7.6-1: Alice uses a symmetric session key, K_S, to send a secret e-mail to Bob.

Having designed a secure e-mail system that provides secrecy, let's now design another system that provides both sender authentication and integrity. We'll suppose, for the moment, that Alice and Bob are no longer concerned with secrecy (they what to share their feelings with everyone!), and are only concerned about sender authentication and message integrity. To accomplish this task, we use digital signatures and message digests, as described in Section 7.4.

Specifically, Alice (1) applies a hash function, H (e.g., MD5), to her message m to obtain a message digest, (2) encrypts the result of the hash function with her private key, d_A, to create a digital signature, (3) concatenates the original (unencrypted message) with the signature to create a package, (4) and sends the package to Bob's e-mail address. When Bob receives the package, he (1) he applies Alice's public key, e_A, to the electronic signature and (2) compares the result of this operation to his own hash, H, of the message. The steps are illustrated in Figure 7.6-2. As discussed in Section 7.4, if the two results are the same, Bob can be pretty confident that message came from Alice and is unaltered.

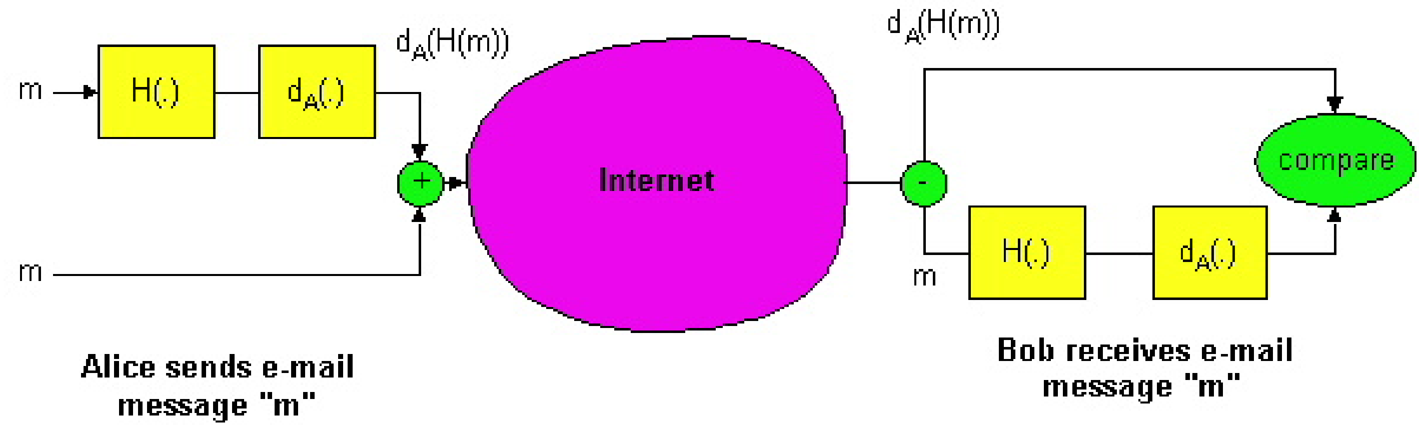

m
H(.)
d_A(.)
d_A(H(m))
+
m
Internet
d_A(H(m))
compare
-
m
H(.)
d_A(.)
Alice sends e-mail message "m"
Bob receives e-mail message "m"

Figure 7.6-2: Using hash functions and digital signatures to provide sender authentication and message integrity.

Now lets consider designing an e-mail system that provides secrecy, sender authentication *and* message integrity. This can be done by combining the procedures in Figure 7.6-1 and 7.6-2. Alice first creates a preliminary package, exactly as in Figure 7.6-2, which consists of her original message along with a digitally-signed hash of the message. She then treats this preliminary package as a message in itself, and sends this new message through the sender steps in Figure 7.6-1, creating a new package that is sent to Bob. The steps applied by Alice are shown in Figure 7.6-3. When Bob receives the package, he first applies his side of Figure 7.6-1 and then his side of Figure 7.6-2. It should be clear that this design achieves the goal of providing secrecy, sender authentication and message integrity. Note in this scheme that Alice applies public key encryption twice: once with her own private key and once with Bob's public key. Similarly, Bob applies public key encryption twice - once with his private key and once with Alice's public key.

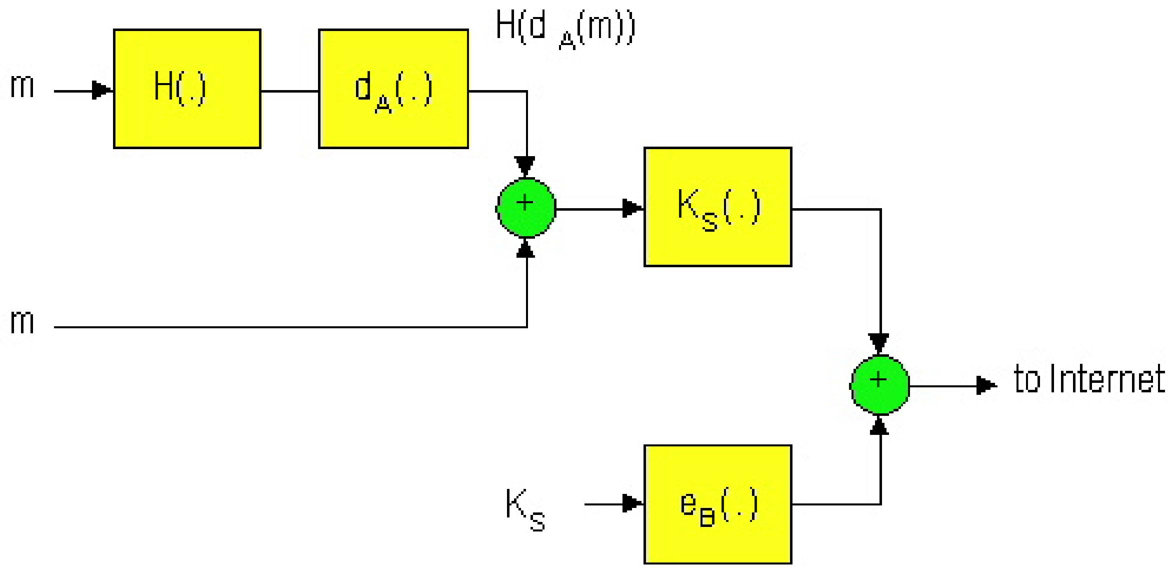


Figure 7.6-3: Alice uses symmetric-key cryptography, public-key cryptography, a hash function and a digital signature to provide secrecy, sender authentication and message integrity.

The secure e-mail design outlined in Figure 7.6-3 probably provides satisfactory security for most e-mail users for most occasions. But there is still one important issue that remains to be addressed. The design in Figure 7.6-3 requires Alice to obtain Bob's public key, and requires Bob to obtain Alice's public key. The distribution of these public keys is a non-trivial problem. For example, Trudy might masquerade as Bob and give Alice her own public key while saying that it is Bob's public key. As we learned in Section 7.5, a popular approach for securely distributing public keys is to *certify* the public keys.

7.6.2 PGP

Originally written by Phil Zimmerman in 1991, pretty good privacy (PGP) is e-mail an encryption scheme that has become a de-facto standard, with thousands of users all over the globe. Versions of PGP are available in the public domain; for example, you can find the PGP software for your favorite platform as well as lots of interesting reading at the International PGP Home Page [PGPI 1999]. (A particularly interesting essay by the author of PGP is [Zimmerman 1999]). PGP is also commercially available [Network Associates 1999], and is also available as a plug-in for many e-mail user agents, including Microsoft's Exchange and Outlook, and Qualcomm's Eudora.

The PGP design is, in essence, the same as the design shown in Figure 7.6-3. Depending on the version, the PGP software uses MD5 or SHA for calculating the message digest; CAST, Triple-DES or IDEA for symmetric key encryption; and RSA for the public key encryption. In addition, PGP provides data compression.

When PGP is installed, the software creates a public key pair for the user. The public key can be posted on the user's Web site or placed in a public key server. The private key is protected by the use of a password. The password has to be entered every time the user accesses the private key. PGP gives the user the option of digitally signing the message, encrypting the message, or both digitally signing and encrypting. Figure 7.6-4 shows a PGP signed message. This message appears after the MIME header. The encoded data in the message is $d_A(H(m))$, i.e., the digitally signed message digest. As we discussed above, in order for Bob to verify the integrity of the message, he needs to have access to Alice's public key.

```
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA1

Bob:

My husband is out of town tonight.

Passionately yours, Alice

-----BEGIN PGP SIGNATURE-----
Version: PGP for Personal Privacy 5.0
Charset: noconv

yhHJRHhGJGhgg/12EpJ+lo8gE4vB3mqJhFEvZP9t6n7G6m5Gw2
```

```
-----END PGP SIGNATURE-----
```

Figure 7.6-4: A PGP signed message.

Figure 7.6-5: shows a PGP secret message. This message also appears after the MIME header. Of course, the plaintext message is not included within the secret e-mail message. When a sender (such as Alice) wants both secrecy and integrity, the PGP would contain a message like that of Figure 7.6-5 contained within the message of Figure 7.6-4.

```
-----BEGIN PGP MESSAGE-----
Version: PGP for Personal Privacy 5.0

u2R4d+/jKmn8Bc5+hgDsqAewsDfrGdszX68liKm5F6Gc4sDfcXyt
RfdSlOjuHgbcfDssWe7/K=lKhnMikLo0+l/BvcX4t==Ujk9PbcD4
Thdf2awQfgHbnmKlok8iy6gThlp
-----END PGP MESSAGE
```

Figure 7.6-5:A secrect PGP message.

PGP also provides a mechanism for public key certification, but the mechanism is quite different from the conventional certification authority tath we examined in section 7.5. PGP public keys are certified by a web of trust. Alice can certify any pair of key and user name for which she believes the pair really belongs together. In addition, PGP permits Alice to say that she trusts another user to vouch for the authenticiy of more keys. Some PGP users sign each other's keys is by holding *key signing parties*. Users physically gather, exchange floppy disks containing public keys, and certify each other's keys by signing them with their private keys. PGP public keys are also distributed by *PGP public key servers* on the Internet. When a user submits a public key to such a server, the server stores a copy of the key, sends a copy of the key to all the other public-key servers, and serves the key to anyone who requests it. Although key signing parties and PGP public key servers actually exist, by far the most common way for users to distribute their public keys is posting them on their personal Web pages. Of course, keys on personal Web pages are not certified by anyone, but they are easy to access.

References

[Molva 1999] R. Molva, Internet Security Architecture, Computer Networks, 1999
[PGPI 1999] The International PGP Home Page, http://www.pgpi.com .
[Network Associates 1999] Network Associates, http://www.nai.com/default_pgp.asp .
[Zimmerman 1999] P. Zimmerman, "Why do you need PGP?" http://www.pgpi.org/doc/whypgp/en/

7.7 Internet Commerce

In the previous section, we considered the application-layer use (in secure e-mail) of the various security technologies that we studied earlier in this chapter: encryption, authentication, key distribution, message integrity and digital signatures. In this section we'll continue our case study of various security mechanisms by dropping down a layer in the protocol stack an covering secure sockets and a secure transport layer. We'll take Internet commerce as a motivating application, since business and financial transactions are an important driver for Internet security.

We consider **Internet commerce** to be the purchasing of "goods" over the Internet. Here we'll use the term "goods" in a very broad sense to include books, CDs, hardware, software, airline tickets, stocks and bonds, consulting services, etc. In the 1990s many schemes were designed for Internet commerce, some providing minimal levels of security and others providing a high-level of security along with customer anonymity (similar to the anonymity provided by ordinary person-to-person cash transactions [Loshin 1997].) In the late 1990s, however, there was a major shake out, as only a few of these schemes were widely implemented in Web browsers and servers. As of this writing, two schemes have taken hold: SSL, which is currently used by the vast majority of Internet transactions; and SET, which is to expected to fiercely compete with SSL in the upcoming years.

There are three major players in this infrastructure: the customer who is purchasing a good, the merchant who is selling the good, and the merchant's bank, which authorizes the purchase. We shall see in our discussion below that Internet commerce with SSL provides security for communication between the first two of these three players (i.e., the customer and the merchant), whereas SET provides security for communication among all three players.

7.7.1 Internet Commerce Using SSL

Let's walk through a typical Internet commerce scenario. Bob is surfing the Web and arrives at the Alice Incorporated site which is selling some durable good. The Alice Incorporated site displays a form in which Bob is supposed to enter the quantity desired, his address and his payment card number. Bob enters this information, clicks on "submit", and then expects to receive (say, from, ordinary mail) the good; he also expects to receive a charge for the good in his next payment card statement. This all sounds good, but if no security measures are taken -- such as encryption or authentication -- Bob could be in for a few surprises:

- An intruder could intercept the order and obtain Bob's payment card information. The intruder could then make purchases at Bob's expense.
- The site could display Alice Incorporated famous logo, but actually be a site maintained by Trudy, who is masquerading as Alice Incorporated. Trudy could take Bob's money and run. Or Trudy could make her own purchases and have them billed to Bob's account.

Many other surprises are possible, and we will discuss a few of these in the next subsection. But the two problems listed above are among the most serious. Internet commerce using the SSL protocol can address both these problems.

Secure sockets layer (SSL), originally developed by Netscape, is a protocol designed to provide data encryption and authentication between a Web client and a Web server. The protocol begins with a handshake phase that negotiates an encryption algorithm (e.g., DES or RSA) and keys, and authenticates the server to the client. Optionally, the client can also be authenticated to the server. Once the handshake is complete and the transmission of application data begins, and all data is encrypted using session keys negotiated during the handshake phase. SSL is widely used in Internet commerce, being implemented in almost all popular browsers and Web servers. Furthermore, it is also the basis of the Transport Layer Security (TLS) protocol [RFC 2246].

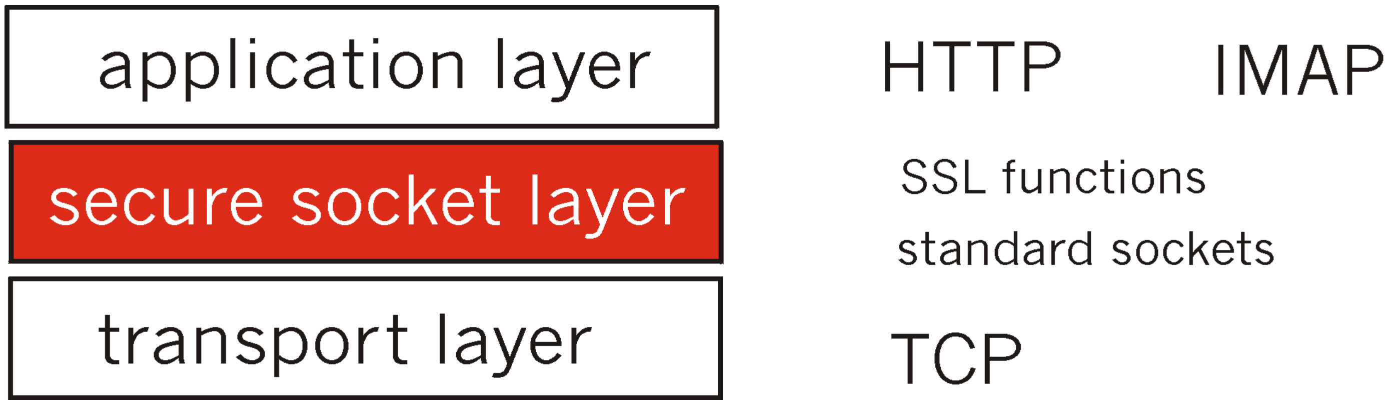


Figure 7.7-1: Secure socket layer

SSL and TLS are not limited to the Web application; for example, they are also used for authentication and data encryption for IMAP mail access. SSL can be viewed as a layer that sits between the application layer and the transport layer, as shown in Figure 7.7-1. On the sending side, SSL receives from the application raw application data (such as an HTTP or IMAP message), encrypts the data and directs the encrypted data to a TCP socket. On the receiving side, SSL reads from the TCP socket, decrypts the data, and directs the data to the application. Although SSL can be used with many Internet applications, we shall discuss it in the context of the Web, where it is principally being used today for Internet commerce.

SSL provides the following features:

- *SSL server authentication*, allowing a user to confirm a server's identity. An SSL-enabled browser maintains a list of trusted certifying authorities (CAs) along with the public keys of the CAs. When the browser wants to do business with an SSL-enabled Web server, the browser obtains from the server a certificate containing the server's public key. The certificate is issued (i.e., digitally signed) by a certificate authority (CA) listed in the client's list of trusted CAs. This feature allows the browser to authenticate the server before the user submits a payment card number. In the context of the earlier example, this server authentication enables Bob to verify that he is indeed sending his payment card number to Alice Incorporated, and not to someone else who might be masquerading as Alice Incorporated.

- *An encrypted SSL session*, in which all information sent between browser and server is encrypted by sending software (browser or Web server) and decrypted by the receiving software (browser or Web server). This confidentially may be important to both the customer and the merchant. Also, SSL provides a mechanism for detecting tampering of the information by an intruder.
- *SSL client authentication*, allowing a server to confirm a user's identity. Analogous to server authentication, client authentication makes use of client certificates, which have also been issued by CAs. This authentication is important if the server, for example, is a bank sending confidential financial information to a customer and wants to check the recipient's identity. Client authentication, although supported by SSL, is optional. To keep our discussion focused, we will henceforth ignore it.

How SSL Works

A user, say Bob, surfs the Web and clicks on a link that takes him to a secure page housed by Alice's SSL-enabled server. The protocol part of the URL for this page is "https" rather than the ordinary "http". The browser and server then run the SSL handshake protocol, which (1) authenticates the server and (2) generates a shared symmetric key. Both of these tasks make use of the RSA public-key technology. The main flow of events in the handshake phase is shown in Figure 7.7-2. During this phase, Alice sends Bob her certificate, from which Bob obtains Alice's public key. Bob then creates a random symmetric key, encrypts it with Alice's public key, and sends the encrypted key to Alice. Bob and Alice now share a symmetric session key. Once this handshake protocol is complete, all data sent between the browser and server (over TCP connections) is encrypted using the symmetric session key.

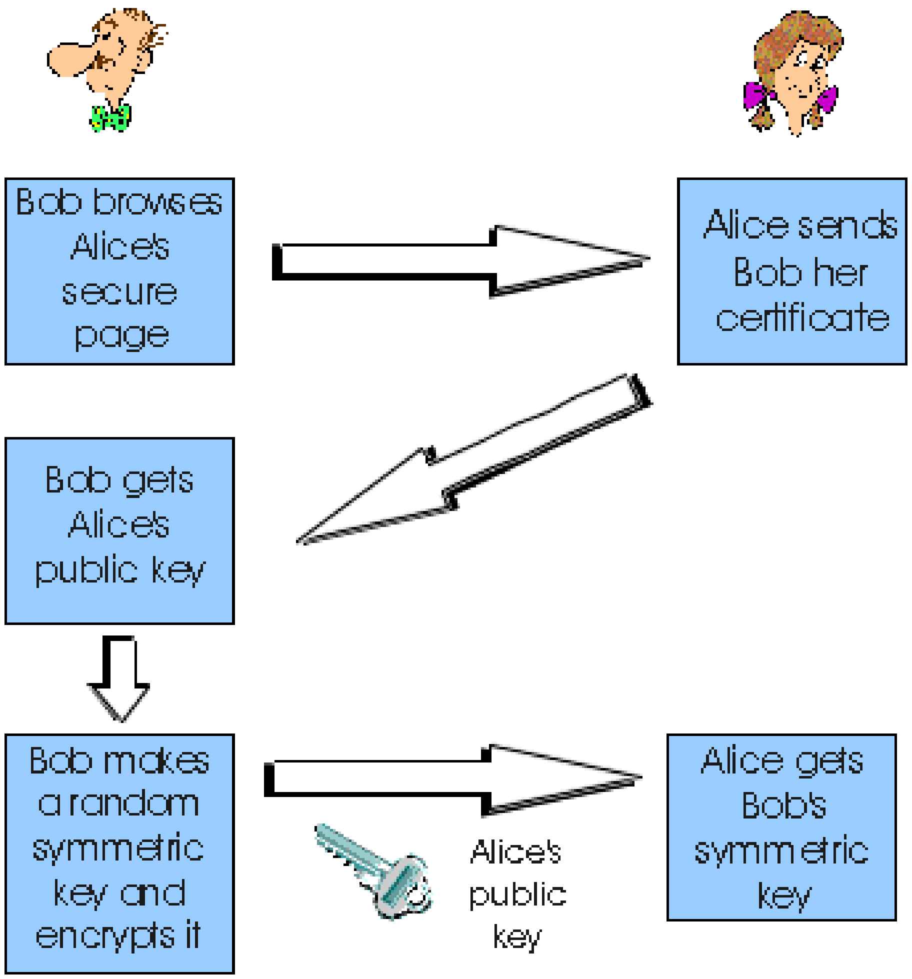

Bob browses Alice's secure page
Alice sends Bob her certificate
Bob gets Alice's public key
Bob makes a random symmetric key and encrypts it
Alice's public key
Alice gets Bob's symmetric key

Figure 7.7-2: High-level overview of the handshake phase of SSL.

Having given a high-level overview of SSL, let's take a closer look at some of more important details. The SSL handshake performs the following steps:

1. The browser sends the server the browser's SSL version number and cryptography preferences. The browser sends its cryptography preferences because the browser and server negotiate which symmetric key algorithm they are going to use.
2. The server sends the browser the server's SSL version number, cryptography preferences and its certificate. Recall that the certificate includes the server's RSA public key and is certified by some CA, that is, the certificate has been encrypted by a CA's private key.
3. The browser has an entrusted list of CAs and a public key for each CA on the list. When the browser receives the certificate from the server, it checks to see if the CA is on the list. If no, the user is warned of the problem and informed that an encrypted and authenticated connection cannot be established. If yes, the browser uses the CA's public key to decrypt the certificate and obtain the server's public key.
4. The browser generates a symmetric session key, encrypts it with the server's public key, and sends the encrypted session key to the server.
5. The browser sends a message to the server informing it that future messages from the client will be

encrypted with the session key. It then sends a separate (encrypted) message indicating that the browser portion of the handshake is finished.

6. The server sends a message to the browser informing it that future messages from the server will be encrypted with the session key. It then sends a separate (encrypted) message indicating that the server portion of the handshake is finished.
7. The SSL handshake is now complete, and the SSL session has begun. The browser and the server use the session keys to encrypt and decrypt the data they send to each other and to validate its integrity.

SSL handshake actually has many more steps than listed above. You can find more information about SSL at Netscape's Security Developer Central [NetscapeSecurity 1999]. In addition to payment card purchases, we point out here that SSL can (and is) used for other financial transactions including online banking and stock trading.

SSL in Action

We recommend that you visit a secure Web site, such as a Quebec maple syrup site [Quebec 1999]. When you enter a secure section of such a site, SSL will perform the handshake protocol. Assuming that the server's certificate checks out, the browser will notify you, for example by displaying a special icon.. All information sent between you and the server will now be encrypted. Your browser should let you actually see the certificate for the merchant. (For example, with Internet Explorer, go to File, Properties, Certificates.) In April 1999, the maple syrup site's certificate included the following information:

Company: Netfarmers Enterprises Inc.
*Certification Authority:*Thawte Certification
Public Key (in hexadecimal): 88:79:85:D5:D0:7D:60:39:10:51:31:EC:17:DE:E7:80

If your browser lets you do secure transactions with the merchant, then you should also be able to see the certificate for CA, i.e., Thawte Certification. (For example, with Internet Explorer, go to View, Internet Options, Content, Certificate Authorities.)

The Limitations of SSL in Internet Commerce

Due to its simplicity and early development, SSL is widely implemented in browsers, servers and Internet commerce products. These SSL-enabled servers and browsers provide a popular platform for payment card transactions. Nevertheless, we should keep in mind that SSL was not specifically tailored for payment card transactions, but instead for generic secure communication between a client and server. Because of this generic design, SSL lacks many features that payment-card industry would like to see in an Internet commerce protocol.

Consider once again what happens when Bob makes a purchase from Alice Incorporated over SSL. The signed certificate that Bob receives from Alice assures Bob that he is really dealing with Alice

Incorporated, and that Alice Incorporated is a bona fide company. However, the generic certificate does not indicate whether Alice Incorporated is authorized to accept payment-card purchases nor if the company is a reliable merchant. This opens the door for merchant fraud. And there is a similar problem for client authorization. Even if SSL client authentication is used, the client certificate does not tie Bob to a specific authorized payment card; thus, Alice Incorporated has no assurance about whether Bob is authorized to make a payment-card purchase. This opens the door to all kinds of fraud, including purchases with stolen credit cards and customer repudiation of purchased goods [Abbott 1999].

Of course, this kind of fraud is already rampant in mail order and telephone order (MOTO) purchases. With MOTO transactions, the law dictates that the merchant accepts liability for fraudulent transactions. Thus, if a customer makes a MOTO purchase with a payment card and claims to have never made the purchase, then the merchant is liable, that is, the merchant is legally bound to return the money to the customer (unless the merchant can prove that the customer actually ordered and received the goods). Similarly, if a MOTO purchase is made with a stolen payment card, the merchant is again liable. On the other hand, with physically-present transactions, the merchant's bank accepts the liability; as you might expect, it is more difficult for a customer to repudiate a physcially-present purchase which involves a hand-written signature or a PIN (personal identification number).

SSL purchases are similar to MOTO purchases, and naturally the merchant is liable for a fraudulent SSL purchase. It would be preferable, of course, to use a protocol that provides superior authentication of the customer and of the merchant, something that is as good or better than a physically-present transaction. Authentication involving payment-card authorization would reduce fraud and merchant liability.

7.7.2 Internet Commerce Using SET

SET (Secure Electronic Transactions) is a protocol specifically designed to secure payment-card transactions over the Internet. It was originally developed by Visa International and MasterCard International in February 1996 with participation from leading technology companies around the world. SET Secure Electronic Transaction LLC (commonly referred to as SETCo) was established in December 1997 as a legal entity to manage and promote the global adoption of SET [SETCo 1999]. Some of the principle characteristics of SET include:

- SET is designed to encrypt specific kinds of payment-related messages; it cannot be used to encrypt arbitrary data (such as text and images) as can SSL.
- The SET protocol involves all three players mentioned at the beginning of this section, namely, the customer, the merchant and *the merchant's bank*. All sensitive information sent between the three parties is encrypted.
- SET requires all three players to have certificates. The customer's and merchant's certificates are issued by their banks, thereby assuring that these players are permitted to make and receive payment-card purchases. The customer certificate provides merchants with assurance that transactions will not be fraudulently charged back. It is an electronic representation of the customer's payment card. It basically contains information about the account, the issuing financial

institution, and other cryptographic information. The merchant certificate assures the consumer that merchant is authorized to accept payment-card purchases. It contains information about the merchant, the merchant's bank, and the financial institution issuing the certificate.

- SET specifies the legal meaning of the certificates held by each party and the apportionment of liabilities connected with a transaction [Abbott 1999].
- In a SET transaction, the customer's payment-card number is passed to the merchant's bank without the merchant ever seeing the number in plain text. This feature prevents fraudulent or careless merchants from stealing or accidentally leaking the payment-card number.

A SET transaction uses three software components:

- **Browser wallet:** The browser wallet application is integrated with the browser and provides the customer with storage and management of payment cards and certificates while shopping. It responds to SET messages from the merchant, prompting the customer to select a payment card for payment.
- **Merchant server:** The merchant server is the merchandizing and fulfillment engine for merchants selling on the Web. For payments, it processes cardholder transactions and communicates with the merchant's bank or approval and subsequent payment capture.
- **Acquirer gateway:** The acquirer gateway is the software component at the merchant's bank. It processes the merchant's payment card transaction for authorization and payment.

In what follows, we give a highly simplified overview of the SET protocol. In reality, the protocol is substantially more complex.

Steps in Making a Purchase

Suppose Bob wants to purchase a good over the Internet from Alice Incorporated.

1. Bob indicates to Alice that he is interested in making a credit card purchase.
2. Alice sends the customer an invoice and a unique transaction identifier.
3. Alice sends Bob the merchant's certificate which includes the merchant's public key. Alice also sends the certificate for her bank, which includes the bank's public key. Both of these certificates are encrypted with the private key of a certifying authority.
4. Bob uses the certifying authority's public key to decrypt the two certificates. Bob now has Alice's public key and the bank's public key.
5. Bob generates two packages of information: the *order information* (**OI**) package and the *purchase instructions* (**PI**) package. The **OI**, destined for Alice, contains the transaction identifier and brand of card being used; it does not include Bob's card number. The **PI**, destined for Alice's bank, contains the transaction identifier, the card number and the purchase amount agreed to Bob. The OI and PI are *dual encrypted*: the **OI** is encrypted with Alice's public key; the **PI** is encrypted with Alice's bank's public key. (We are bending the truth here in order to see the big picture. In reality, the **OI** and **PI** are encrypted with a customer-merchant session key and a customer-bank session

key.) Bob sends the **OI** and the **PI** to Alice.

6. Alice generates an authorization request for the card payment request, which includes the transaction identifier.
7. Alice sends to her bank a message encrypted with the bank's public key. (Actually, a session key is used.) This message includes the authorization request, the **PI** package received from Bob, and Alice's certificate.
8. Alice's bank receives the message and unravels it. The bank checks for tampering. It also makes sure that the transaction identifier in the authorization request matches the one in Bob's **PI** package.
9. Alice's bank then sends a request for payment authorization to Bob's payment-card bank through traditional bank-card channels -- just as Alice's bank would request authorization for any normal payment-card transaction.
10. Once Bob's bank authorizes the payment, Alice's bank sends a response to the Alice, which is (of course) encrypted. The response includes the transaction identifier.
11. If the transaction is approved, Alice sends its own response message to Bob. This message serves as a receipt and informs Bob that the payment was accepted and that the goods will be delivered.

One of the key features of SET is the non-exposure of the credit number to the merchant. This feature is provided in Step 5, in which the customer encrypts the credit card number with the bank's key. Encrypting the number with the bank's key prevents the merchant from seeing the credit card. Note that the SET protocol closely parallels the steps taken in a standard payment-card transaction. To handle all the SET tasks, the customer will have a so-called digital wallet that runs the client-side of the SET protocol and stores customer payment-card information (card number, expiration date, etc.). Readers interested in learning more about SET are encouraged to see SETCo page [SETCo 1999] or the SET documentation at the MasterCard site [Master 1999]. There are also several good books on SET [Merkow 1998] [Loeb 1998].

References

[Abbott 1999] S. Abbott, "The Debate for Secure E-Commerce," Performance Computing, February 1999, http://www.performancecomputing.com/features/9902f1.shtml

[Loeb 1998] L. Loeb, Secure Electronic Transactions : Introduction and Technical Reference," Artech House, New York, 1998.

[Loshin 1997] P. Loshin, P. Murphy, "Electronic Commerce : On-Line Ordering and Digital Money", Charles River Media, August 1997

[Merkow 1998] M. Merkow, K. Wheeler, and J. Breithaupt, "Building SET Applications for Secure Transactions," John Wiley and Sons, New York, 1998.

[Master 1999] SET Secure Electronic Transaction, MasterCard Web site, http://www.mastercard.com/shoponline/set/

[NetscapeSecurity 1999] Security Developer Central, Netscape Site, http://developer.netscape.com/tech/security/

[Quebec 1999] Quebec Maple Syrup homepage, http://www.jam.ca/syrup/

[RFC 2246] T. Dierks and C. Allen, The TLS Protocol, [RFC 22246], January 1999.
[Setco1999] SETCo LLC Website, http://www.setco.org/

Return to Table Of Contents

7.8 Network Layer Security: IPsec

Having examined case studies of the use of various security mechanisms at the application, socket, and transport layers, our final case study naturally takes us down to the network layer. Here, we'll examine the the IP Security protocol, more commonly known as **IPsec** - a suite of protocols that provides security at the network layer. IPsec is a rather complex animal, and different parts of it are described in more than a dozen RFCs. In this section, we'll discuss IPsec in a specific context, namely, in the context that *all* hosts in the Internet support IPsec. Although this context is many years away, the context will simplify the discussion and help us understand the key features of IPsec. Two key RFCs are [RFC 2401], which describes the overall IP security architecture and [RFC 2411], which provides an overview of the IPsec protocol suite and the documents describing it. A nice introduction to IPsec is given in [Kessler].

Before getting into the specifics of IPsec, let's step back and consider what it means to provide security at the network layer. Consider first what it means to provide **network layer secrecy**. The network layer would provide secrecy if all data carried by all IP datagrams were encrypted. This means that whenever a host wants to send a datagram, it encrypts the data field of the datagram before shipping it out into the network. In principle, the encryption could be done with symmetric key encryption, public key encryption or with session keys that have are negotiated using public key encryption. The data field could be a TCP segment, a UDP segment, an ICMP message, etc. If such a network layer service were in place, all data sent by hosts -- including e-mail, Web pages, control and management messages (such as ICMP and SNMP) -- would be hidden from any third party that is "wire tapping" the network. (However, the unencrypted data could be snooped at points in the source or destination hosts.) Thus, such a service would provide a certain "blanket coverage" for all Internet traffic, thereby giving all of us a certain sense of security.

In addition to secrecy, one might want the network layer to also provide **source authentication**. When a destination host receives an IP datagram with a particular IP source address, it might authenticate the source by making sure that the IP datagram was indeed generated by the host with that IP source address. Such a service prevents attackers from spoofing IP addresses.

In the IPsec protocol suite there are two principal protocols: the **Authentication Header (AH) protocol** and the **Encapsulation Security Payload (ESP) protocol**. When a source host sends secure datagrams to a destination host, it does so with either the AH protocol or with the ESP protocol.The AH protocol provides source authentication and data integrity but does not provide secrecy. The ESP protocol provides data integrity and secrecy. Providing more services, the ESP protocol is naturally more complicated and requires more processing than the AH protocol. We'll discuss both of these protocols below.

For both the AH and the ESP protocols, before sending secured datagrams from a source host to a destination host, the source and network hosts handshake and create a network layer logical connection. This logical channel is called a **security agreement (SA)**. Thus, IPsec transforms the traditional connectionless network layer of the Internet to a layer with logical connections! The logical connection defined by a SA is a simplex connection, that is, it is unidirectional. If both hosts want to send secure datagrams to each other, then two SAs (i.e., logical connections) need to be established, one in each direction. A SA is uniquely identified by a 3-tuple consisting of:

- a security protocol (AH or ESP) identifier;
- the source IP address for the simplex connection;
- a 32-bit connection identifier called the Security Paramter Index (SPI).

For a given SA (that is, a given logical connection from source host to destination host), each IPsec datagram will have a special field for the SPI. All of the datagrams in the SA will use the same SPI value in this field.

Authentication Header (AH) Protocol

As mentioned above, the AH protocol provides source host identification and data integrity but not secrecy. When a particular source host wants to send one or more datagrams to a particular destination, it first establishes an SA with the destination. After having established the SA, the source can send secured datagrams to the destination host. The secured datagrams include the AH header, which is inserted between the original IP datagram data (e.g., a TCP or UDP segment) and the IP header, as shown in Figure 7.8-1. Thus the AH header augments the original data field, and this augmented data field is encapsulated as a standard IP datagram. For the protocol field in the IP header, the value 51 is used to indicate that the datagram includes an AH header. When the destination host recieves the IP datagram, it takes note of the 51 in the protocol field, and processes the datagram using the AH protocol. (Recall that the protocol field in the IP datagram is traditionally used to distinguish between UDP, TCP, ICMP, etc.) Intermediate routers process the datagrams just as they always have -- they examine the destination IP address and route the datagrams accordingly.

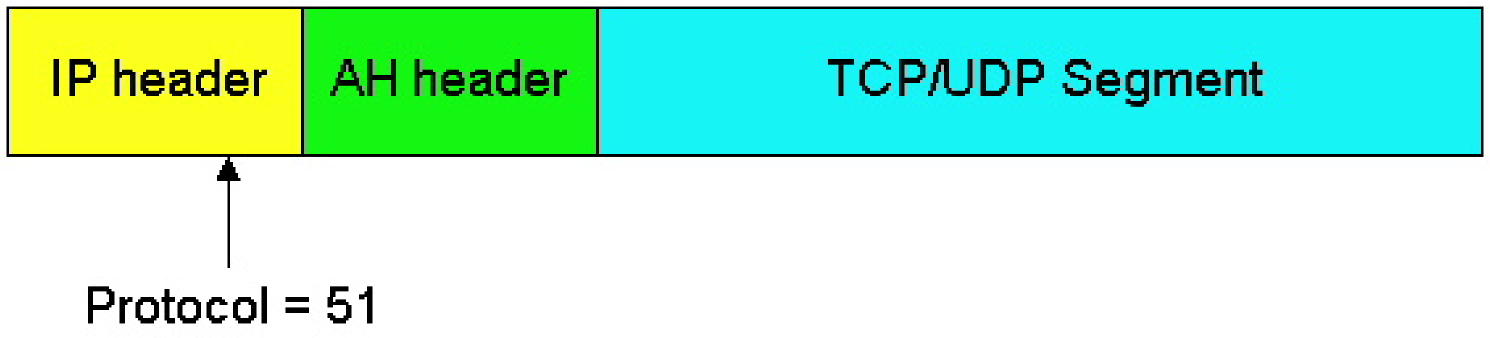


Figure 7.8-1: Position of the AH header in the IP datagram.

The AH header includes several fields, including:

- **Next Header** field, which has the role that the protocol field has for an ordinary datagram. It indicates if the data following the AH header is a TCP segment, UDP segment, ICMP segment, etc. (Recall that protocol field in the datagram is now being used to indicate the AH protocol, so it can no longer be used to indicate the transport-layer protocol.)

- **Security Parameter Index (SPI)** field, an arbitrary 32-bit value that, in combination with the destination IP address and the security protocol, uniquely identifies the SA for the datagram.
- **Sequence Number** field, a 32-bit field containing a sequence number for each datagram. It is initally set to 0 at the establishment of an SA. The AH protocol uses the sequence numbers to prevent playback and man-in-the-middle attacks (see Section 7.3).
- **Authentication Data** field, a variable-length field containing signed message digest (i.e., a digital signature) for this packet. The message digist is calculated over the original IP datagram, thereby providing source host authentication and IP datagram integrity. The digital signature is computed using the authentication algorithm specified by the SA, such as DES, MD5 or SHA.

When the destination host receives an IP datagram with an AH header, it determines the SA for the packet and then authenticates the integrity of the datagram by processing the authentication data field. The IPsec authentication scheme (for both the AH and ESP protocols) uses a scheme called HMAC, which is an encrypted message digest described in [RFC 2104]. HMAC uses a shared secret key between two parties rather than public key methods for message authentication. Further details about the AH protocol can be found in [RFC 2402].

The ESP Protocol

The ESP protocol provides network layer secrecy as well as source host authentication. Once again, it all begins with a source host establishing a SA with a destination host. Then the source host can send secured datagrams to the destination host. As shown in Figure 7.8-2, a secured datagram is created by surrounding the original IP datagram data with header and trailer fields, and then inserting this encapsulated data into the data field of an IP datagram. For the protocol field in the header of the IP datagram, the value 50 is used to indicate that the datagram includes an ESP header and trailer. When the destination host recieves the IP datagram, it takes note of the 50 in the protocol field, and processes the datagram using the ESP protocol. As shown in Figure 7.8-2, the original IP datagram data along with the ESP Trailer field are encrypted. Secrecy is provided with DES-CBC encryption [RFC 2405]. The ESP header consists of a 32-bit field for the SPI and 32-bit field for the sequence number, which have exactly the same role as in the AH protocol. The trailer includes the Next Header field, which also has exactly the same role. Note that because the Next Header field is encrypted along with the original data, an intruder will not be able to determine the transport protocol that is being used. Following the trailer there is the Authentication Data field, which again serves the same role as in the AH protocol. Further details about the AH protocol can be found in [RFC 2406].

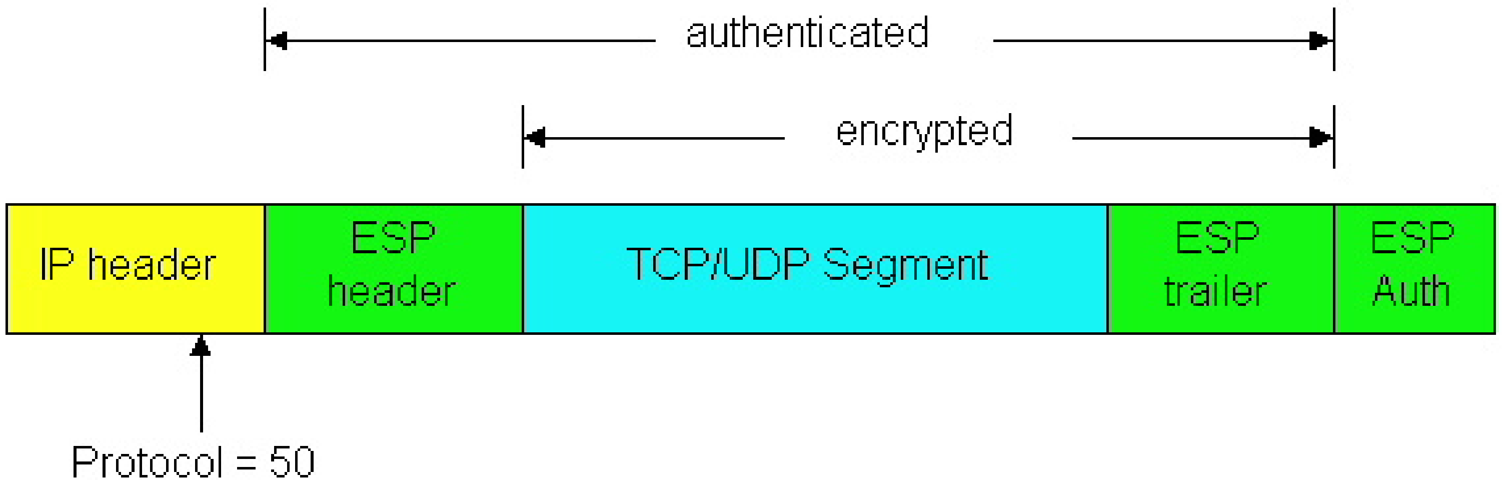


Figure 7.8-2: The ESP fields in the IP datagram.

SA and Key Management

For sucessful deployment of IPsec, a scalable and automated SA and key management scheme is necessary. Several protocols have been defined for these tasks, including:

- The Internet Key Exchange (IKE) algorithm [RFC 2409] is the default key management protocol for IPsec.
- The Internet Security Assoication and Key Management Protocol (ISKMP) defines procedures for establishing and tearing down SAs [RFC 2407] [RFC 2408]. ISKMP's security association is completely separate from IKE key exchange.

This wraps up our summary of IPsec. We have discussed IPsec in the context of IPv4 and the "transport mode". IPsec also defines a "tunnel mode," in which routers introduce the security functionality rather than the hosts. Finally, IPsec describes encryption procedures for IPv6 as well as IPv4.

References

[Kessler] G.C. Kessler, An Overview of Cryptography, May 1998, Hill Associates, http://www.hill.com/TechLibrary/index.htm

[RFC 2104] H. Krawczyk, M.Bellare, R. Canetti, HMAC: Keyed-Hashing for Message Authentication, [RFC 2104], February 1997.

[RFC 2401] S. Kent and R. Atkinson, Security Architecture for the Internet Protocol, [RFC 2401], November 1998.

[RFC 2402] S. Kent and R. Atkinson, IP Authentication Header, [RFC 2402], November 1998.

[RFC 2405] C. Madson and N.Doraswamy, The ESP DES-CBC Cipher Algorithm with Explicit IV, [RFC 2405], November 1998.

[RFC 2406] S. Kent and R. Atkinson, IP Authentication Header, [RFC 2406], November 1998.

[RFC 2407] D. Piper, The Internet IP Security Domain of Interpretation for ISAKMP, [RFC 2407], November 1998

[RFC 2408] D. Maughan, M. Schertler, M. Schneider and J. Turner, Internet Security Association and Key Management Protocol (ISAKMP), [RFC 2408], November 1998.

[RFC 2409] D. Harkins and D. Carrel, The Internet Key Exchange (IKE), [RFC 2409], November 1998

[RFC 2411] R. Thayer, N. Doraswamy and R. Glenn, "IP Security Document Road Map," [RFC 2411], November 1998

7.9 Summary

In this chapter, we've examined the various mechanisms that our secret lovers, Bob and Alice, can use to communicate "securely." We've seen that Bob and Alice are interested in secrecy (so that they alone are able to understand the contents of a transmitted message), authentication (so that they are sure that they are talking with each other), and message integrity (so that they are sure that their messages are not altered in transit). Of course, the need for secure communication is not confined to secret lovers. Indeed, we saw in section 7.1 that security is needed at various layers in a network architecture to protect against "bad guys" who may sniff packets, remove packets from the network, or inject falsely addressed packets into the network.

The first part of this chapter presented various principles underlying secure communication. We covered cryptographic techniques for coding and decoding data in Section 7.2, including both symmetric key cryptography and public key cryptography. DES and RSA were examined as specific case studies of these two major classes of cryptographic techniques in use in today's networks. In section 7.3 we turned our attention to authentication, and developed a series of increasingly sophisticated authentication protocols to ensure that a conversant is indeed who he/she claims to be, and is "live." We saw that both symmetric key cryptography and public key cryptography can play an important role not only in disguising data (encryption/decryption), but also in performing authentication. Techniques for "signing" a digital document in a manner that is verifiable, non-forgible, and non-repudiable were covered in Section 7.4. Once again, the application of cryptographic techniques proved essential. We examined both digital signatures and message digests - a shorthand way of signing a digital document. In section 7.5 we examined key distribution protocols. We saw that for symmetric key encryption, a key distribution center - a single trusted network entity - can be used to distribute a shared symmetric key among communicating parties. For public key encryption, a certification authority distributes certificates to validate public keys.

Armed with the techniques covered in sections 7.2 through 7.5, Bob and Alice can communicate securely (one can only hope that they are networking students who have learned this material and can thus avoid having their tryst uncovered by Trudy!). In the second part of this chapter we thus turned our attention to the use of various security techniques in networks. In section 7.6, we used e-mail as a case study for application-layer security, designing an e-mail system that provided secrecy, sender authentication and message integrity. We also examined the use of pgp as a public-key e-mail encryption scheme. Our cases studies continued as we headed down the protocol stack and examined
the secure sockets layer (SSL) and secure electronic transactions, the two primary protocols in use today for secure electronic commerce. Both are based on public key techniques. Finally, in section 7.8 we examined a suite of security protocols for the IP layer of the Internet - the so-called IPsec protocols. These can be used to provide secrecy, authentication and message integrity between two communication IP devices.

Homework Problems and Discussion Questions

Review Questions

1.) What are the differences between message secrecy and message integrity? Can you have one without the other? Justify your answer.

2.) What is the difference between an active and a passive intruder?

3.) What is an important difference between a symmetric key system and a public key system?

4.) Suppose that an intruder has an encrypted message as well as the decrypted version of that message. Can the intruder mount a cipher-text only attack, a known-plaintext or a chosen-plaintext attack?

5.) Suppose *N* people want to communicate with each of the *N-1* other people using symmetric key encryption. All communication between any to people, *i* and *j*, is visible to all other people, and no other person should be able to decode their communication. How many keys are required in the system as a whole? Now suppose that public key encryption is used. How many keys are required in this case?

6.) What is the purpose of a nonce in an authentication protocol?

7.) What does it mean to say that a nonce is a once-in-a-lifetime value? In whose lifetime?

8.) What is the man-in-the-middle attack? Can this attack occur when symmetric keys are used?

9.) What does it mean for a signed document to be verifiable, non-forgible, and non-repudiable?

10.) In what way does a message digest provide a better message integrity check than a checksum such as the Internet checksum?

11.) In what way does a message digest provide a "better" digital signature than using a public key digital signature?

12.) Is the message associated with a message digested encrypted? Since either "yes" or "no" are acceptable answers here, you should explain your answer.

13.) What is a key distribution center? What is a certification authority?

14.) Summarize the key differences in the services provided by the Authentication Header protocol and the Encapsulation Security Payload (ESP) protocol in IPsec.

Problems

1.) Using the monoalphabetic cipher in Figure 7-3. Encode the message "This is an easy problem." Decode the message "rmij'u uamu xyj."

2.) Show that Eve's known plaintext attack in which she knows the (ciphertext, plaintext) translation pairs for seven letters reduces the number of possible substitutions to be checked by approximately 10^9.

3.) Consider the Vigenere system shown in Figure 7-4. Will a chosen plaintext attack that is able to get the plaintext encoding of the message, "The quick fox jumps over the lazy brown dog" be sufficient to decode all messages? Why?

4.) Using RSA, choose p = 3 and q = 11, and encode the phrase "hello". Apply the decryption algorithm, to the encrypted version to recover the original plaintext message.

5.) In the man-in-the-middle attack in Figure 7.3-7, Alice has not authenticated Bob. If Alice were to require Bob to authenticate himself using *ap5.0,* would the man-in-the-middle attack be avoided? Explain your reasoning.

6.) The Internet BGP routing protocol uses the MD5 message digest rather than public key encryption to sign BGP messages. Why do you think MD5 was chosen over public key encryption?

7.) Compute a third message, different than the two messages in Figure 7.4-5, that has the same checksum as the messages in Figure 7.4-5.

8.) Augment the KDC protocol shown in Figure 7.5-1 to include the necessary authentication messages. Be sure to show the use of nonces and indicate which key values are used to encrypt which messages

9.) In the protocol and discussion of Figure 7.5-1, why doesn't Alice have to explicitly authenticate Bob?

10.) In the protocol in Figure 7.5-2, Alice did not include her own identity in the message to the CA. Anyone could thus spoof a message from Alice to the CA. Does this compromise the integrity of the CA's public key distribution? Justify your answer.

11.) Why is there no explicit authentication in the protocol in Figure 7.5-2 ? Is authentication needed? Why?

12.) Consider the KDC and the CA servers. Suppose a KDC goes down? What is the impact on the ability of parties to communicate securely, i.e., who can, and can not, communicate? Justify your answer. Suppose now that a CA goes down. What is the impact of this failure?

Discussion Questions

1.) Suppose that an intruder could both insert and remove DNS messages into the network. Give three scenarios showing the problems that such an intruder could cause.

2.) No one has formally "proven" that 3-DES or RSA are "secure." Given this, what evidence do we have they are indeed secure?

3.) If IPsec provides security at the network layer, why is it that security mechanisms are still needed at layers above IP?

4.) Go to the International PGP homepage (http://www.pgpi.org/). What version of pgp are you legally allowed to download, given the country you are in?