UNIT – III

Cryptographic Hash Functions: Message Authentication, Secure Hash Algorithm (SHA-512), Message authentication codes: Authentication requirements, HMAC, CMAC, Digital signatures, Elgamal Digital Signature Scheme.

Key Management and Distribution: Symmetric Key Distribution Using Symmetric & Asymmetric Encryption, Distribution of Public Keys, Kerberos, X.509 Authentication Service, Public – Key Infrastructure

Message Authentication:

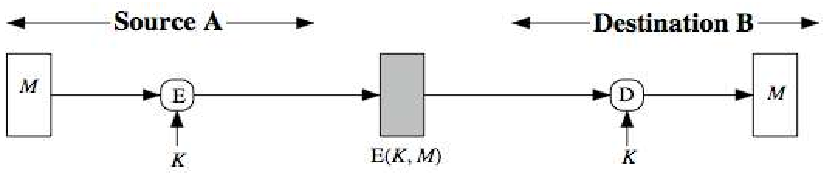
Message authentication is a procedure to verify that received messages come from the authorized source and have not been altered. It is intended against the attacks like content modification, sequence modification, timing modification and repudiation There are three classes functions that may be used to produce an authenticator. They are:

- *Message encryption*–the ciphertext serves as authenticator
- *Message authentication code (MAC)–*a public function of the message and a secret key producing a fixed-length value to serve as authenticator. This does not provide a digital signature because A and B share the same key.
- *Hash function*–a public function mapping an arbitrary length message into a fixed-length hash value to serve as authenticator. This does not provide a digital signature because there is no key.

(a) Message Encryption:

Message encryption by itself can provide a measure of authentication. The analysis differs for symmetric and public-key encryption schemes.

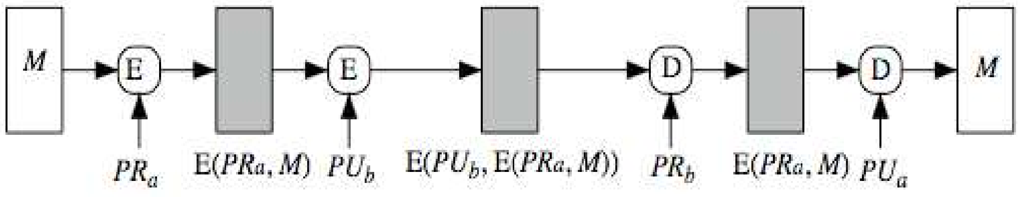
If use symmetric encryption, No other party knows the key, then confidentiality is provided. Here in the symmetric encryption the ciphertext of the entire message serves as its authenticator, see the following diagram,…. it shows that the cipher text acts as an authenticator



(a) Symmetric encryption: confidentiality and authentication

If use public-key techniques (Asymmetric techniques) , can use a digital signature which can only have been created by key owner to validate the integrity of the message contents. To provide both confidentiality and authentication,

A can encrypt M first using its private key, which provides the digital signature, and then using B's public key, which provides confidentiality (Stallings Figure 12.1d).

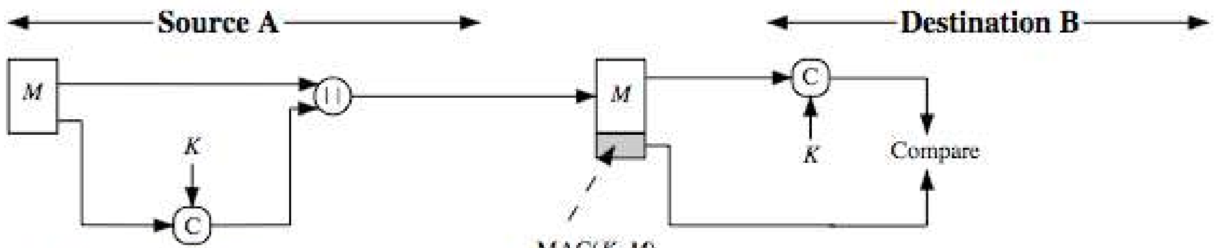


(d) Public-key encryption: confidentiality, authentication, and signature

(b) Message Authentication Code

An alternative authentication technique involves the use of a secret key to generate a small fixed-size block of data, known as cryptographic checksum or MAC, which is appended to the message. This technique assumes that both the communicating parties say A and B share a common secret key K.

When A has a message to send to B, it calculates MAC as a function C of key and message given as: **MAC=C_K(M)**



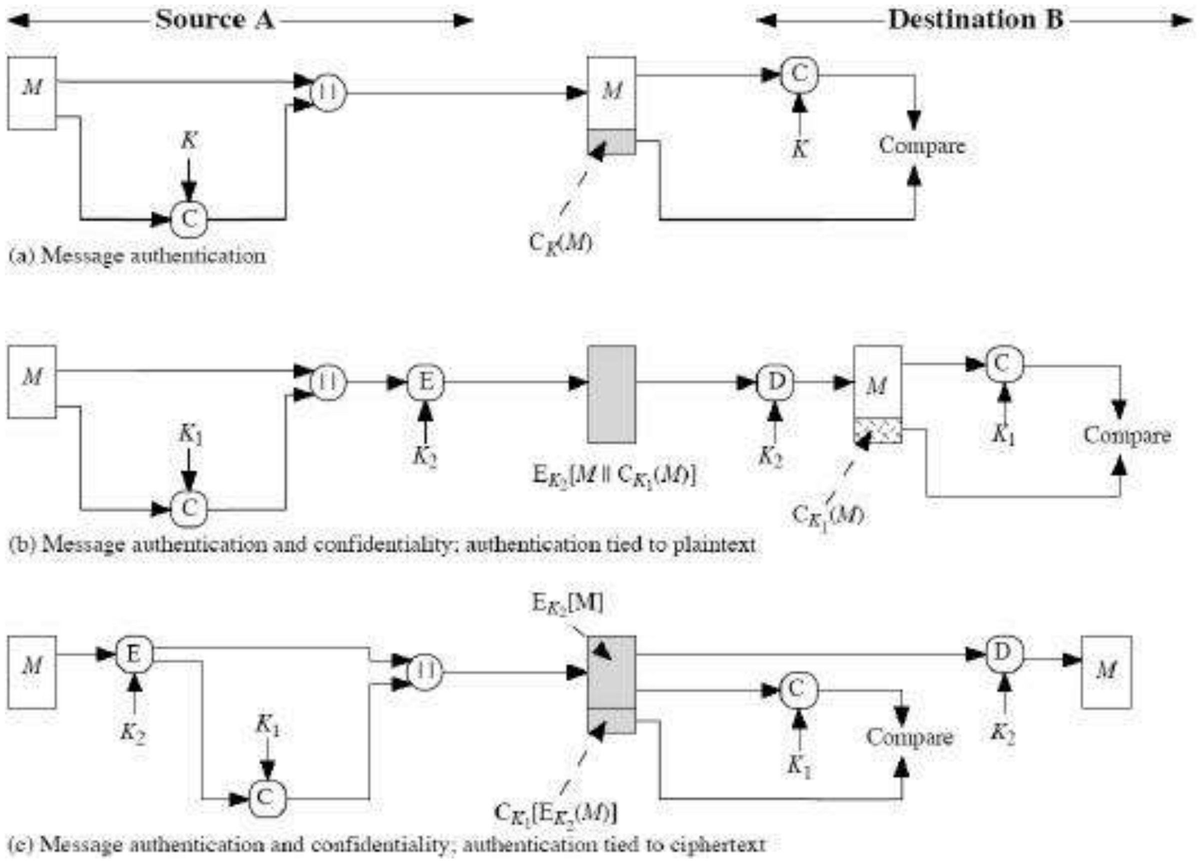
(a) Message authentication

The message and the MAC are transmitted to the intended recipient, Here the recipient uses same shared key and generate MAC value and compare the received value from sender.If they both are same The sender is authenticated .The receiver has to check the following conditions also:

1. The receiver is assured that the message has not been altered: Any alternations been done the MAC’s do not match.

2. The receiver is assured that the message is from the alleged sender: No one except the sender has the secret key and could prepare a message with a proper MAC.
3. If the message includes a sequence number, then receiver is assured of proper sequence as an attacker cannot successfully alter the sequence number.

Basic uses of Message Authentication Code (MAC) are shown in the figure:


Source A
Destination B
M
||
K
C
Compare
$C_K(M)$
(a) Message authentication
E
D
K_1
K_2
$E_{K_2}[M \| C_{K_1}(M)]$
$C_{K_1}(M)$
(b) Message authentication and confidentiality; authentication tied to plaintext
$E_{K_2}[M]$
$C_{K_1}[E_{K_2}(M)]$
(c) Message authentication and confidentiality; authentication tied to ciphertext

There are three different situations where use of a MAC is desirable:

- If a message is broadcast to several destinations in a network (such as a military control center), then it is cheaper and more reliable to have just one node responsible to evaluate the authenticity – message will be sent in plain with an attached authenticator.
- If one side has a heavy load, it cannot afford to decrypt all messages –it will just check the authenticity of some randomly selected messages.
- Authentication of computer programs in plaintext is very attractive service as they need not be decrypted every time wasting of processor resources. Integrity of the program can always be checked by MAC.

© Hash Function:

A **hash function** H accepts a variable-length block of data *M* as input and produces a fixed-size hash value *h* = H(*M*). The kind of hash function needed for security applications is referred to as a **cryptographic hash function**. A cryptographic hash function is an algorithm for which it is computationally infeasible (because no attack is significantly more efficient than brute force) to find either (a) a data object that maps to a pre-specified hash result (the one-way property) Because of these characteristics, hash functions are often used to determine whether data has changed or not.

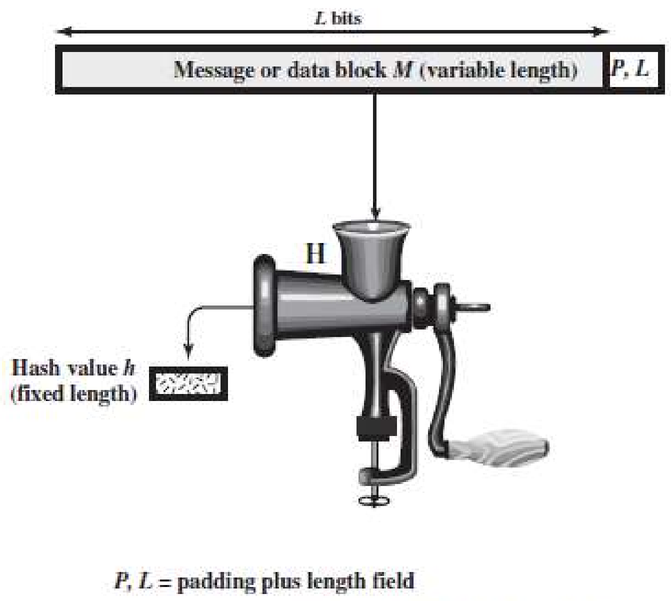

L bits
Message or data block *M* (variable length)
P, *L*
H
Hash value *h* (fixed length)
P, *L* = padding plus length field

Figure 11.1 Cryptographic Hash Function; $h = H(M)$

Hash functions are commonly used to create a one-way password file. Here explains a scheme in which a hash of a password is stored by an operating system rather than the password itself. Thus, the actual password is not retrievable by a hacker who gains access to the password file. In simple terms, when a user enters a password, the hash of that password is compared to the stored hash value for verification. This approach to password protection is used by most operating systems.

Hash functions can be used for intrusion detection and virus detection. Store H(F) for each file on a system and secure the hash values (e.g., on a CD-R that is kept secure). One can later determine if a file has been modified by recomputing H(F). An intruder would need to change F without changing H(F).

A cryptographic hash function can be used to construct a pseudorandom function (PRF) or a pseudorandom number generator (PRNG).

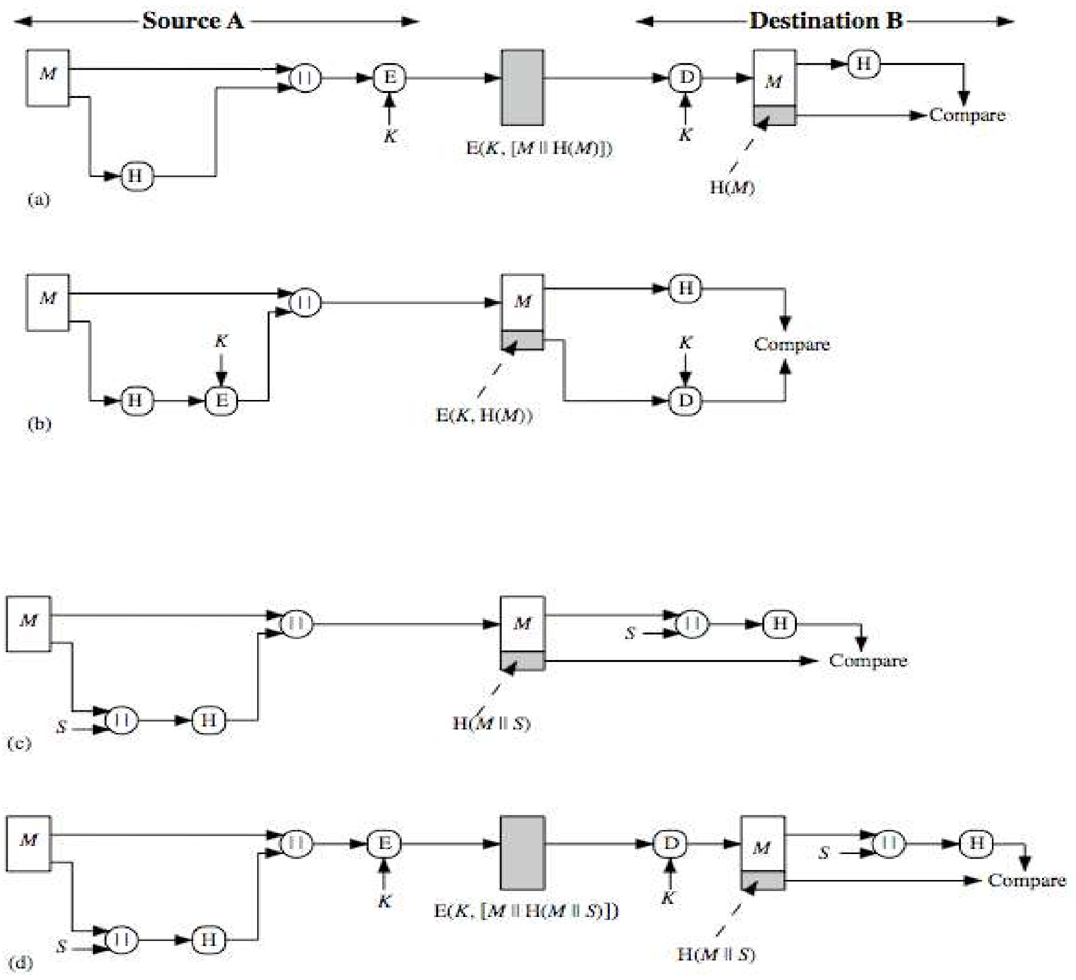


Figure 11.2 illustrates a variety of ways in which a hash code can be used to provide message authentication, as follows:

a. The message plus concatenated hash code is encrypted using symmetric encryption. Since only A and B share the secret key, the message must have come from A and has not been altered. The hash code provides the structure or redundancy required to achieve authentication.
b. Only the hash code is encrypted, using symmetric encryption. This reduces the processing burden for those applications not requiring confidentiality.
c. Shows the use of a hash function but no encryption for message authentication. The technique assumes that the two communicating parties share a common secret value S. A computes the hash value over the concatenation of M and S and appends the resulting hash value to M. Because B possesses S, it can recompute the hash value to verify. Because the secret value itself is not sent, an opponent cannot modify an intercepted message and cannot generate a false message.
d. Confidentiality can be added to the approach of (c) by encrypting the entire message plus the hash code.

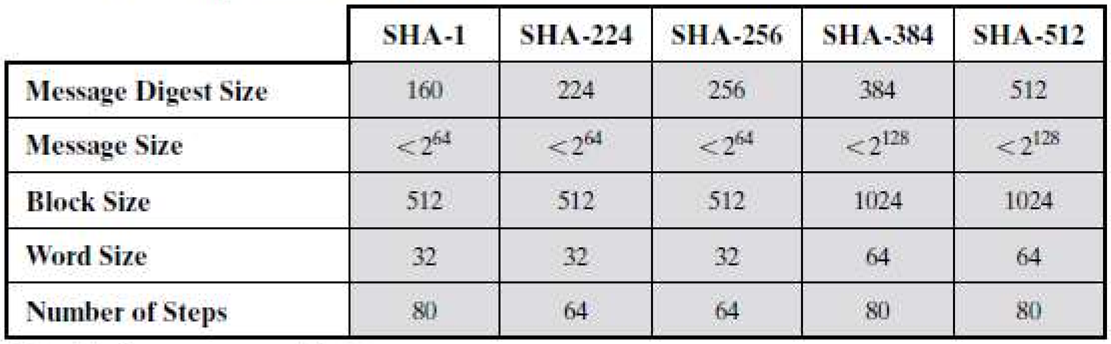
When confidentiality is not required, method (b) has an advantage over methods (a) and (d), which encrypts the entire message, in that less computation is required.

<u>SECURE HASH ALGORITHM:</u>

In recent years, the most widely used hash function has been the Secure Hash Algorithm (SHA). Indeed, because virtually every other widely used hash function had been found to have substantial cryptanalytic weaknesses, SHA was more or less the last remaining standardized hash algorithm by 2005.

- SHA was developed by the National Institute of Standards and Technology (NIST) and published as a federal information processing standard (FIPS 180) in 1993.
- When weaknesses were discovered in SHA, now known as **SHA-0**, a revised version was issued as FIPS
- 180-1 in 1995 and is referred to as **SHA-1**.
- The actual standards document is entitled “Secure Hash Standard.” SHA is based on the hash function MD4, and its design closely models MD4.

Table 11.3 Comparison of SHA Parameters



	SHA-1	SHA-224	SHA-256	SHA-384	SHA-512
Message Digest Size	160	224	256	384	512
Message Size	$<2^{64}$	$<2^{64}$	$<2^{64}$	$<2^{128}$	$<2^{128}$
Block Size	512	512	512	1024	1024
Word Size	32	32	32	64	64
Number of Steps	80	64	64	80	80

Note: All sizes are measured in bits.

SHA-1 produces a hash value of 160 bits. In 2002, NIST produced a revised version of the standard, FIPS 180-2, that defined three new versions of SHA, with hash value lengths of 256, 384, and 512 bits, known as SHA-256, SHA-384, and SHA-512, respectively. Collectively, these hash algorithms are known as **SHA-2**. These new versions have the same underlying structure and use the same types of modular arithmetic and logical binary operations as SHA-1. A revised document was issued as FIP PUB 180-3 in 2008, which added a 224-bit version (Table 11.3).

<u>SHA-512 Logic</u>

The algorithm takes as input a message with a maximum length of $< 2^{128}$ bits and produces as output a 512-bit message digest. The input is processed in 1024-bit blocks .This follows the general structure depicted in Figure 11.9. The processing consists of the following steps.

Step 1 Append padding bits. The message is padded so that its length is congruent to 896 modulo 1024 (or) length should be $<2^{128}$,then the Padding is added, even if the message is already of the desired length. Thus, the number of padding bits is in the range of 1 to 1024. The padding consists of a single 1 bit followed by the necessary number of 0 bits.

Step 2 Append length. A block of 128 bits is appended to the message. This block is treated as an unsigned 128-bit integer (most significant byte first) and contains the length of the original message (before the padding). The outcome of the first two steps yields a message that is an integer multiple of 1024 bits in length.

In Figure 11.9, the expanded message is represented as the sequence of 1024-bit blocks *M*1, *M*2, c, *MN*, so that the total length of the expanded message is *N* * 1024 bits.

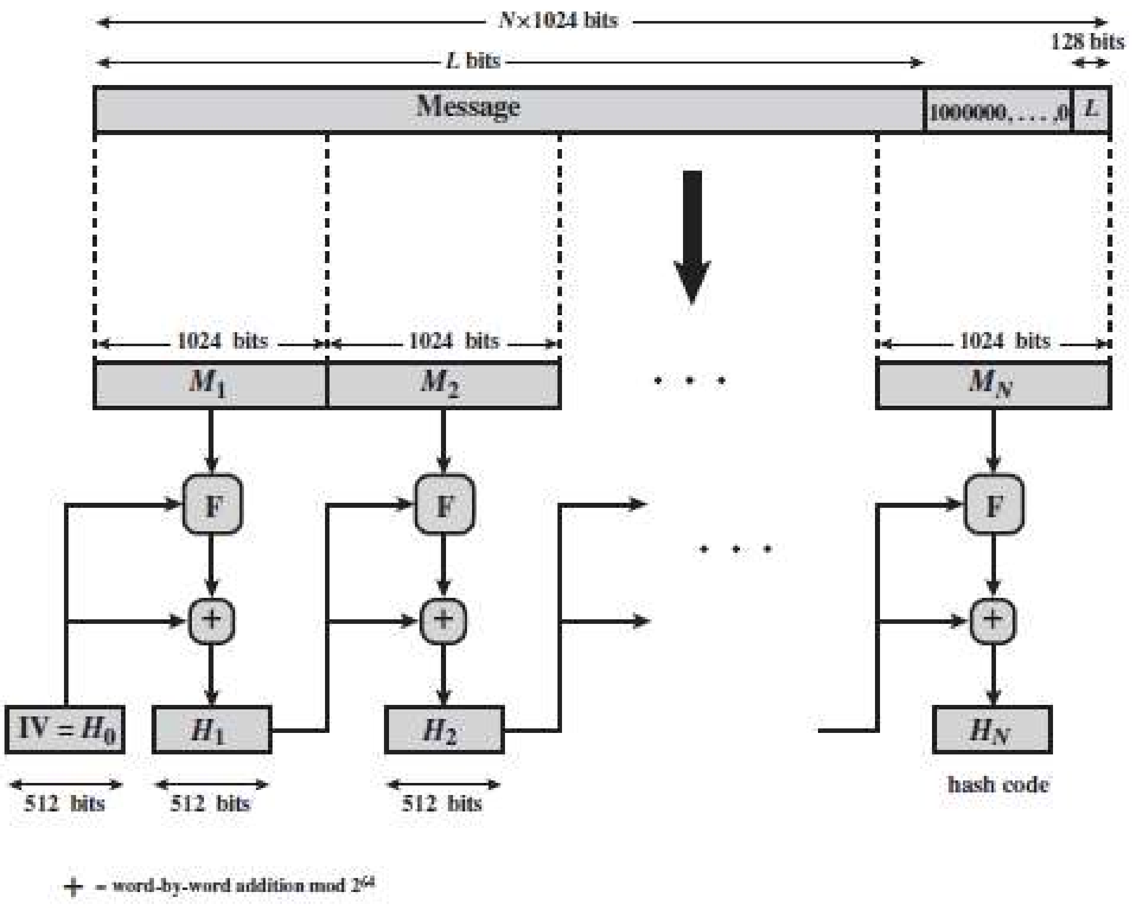


Figure 11.9 Message Digest Generation Using SHA-512

Step 3 Initialize hash buffer. A 512-bit buffer is used to hold intermediate and final results of the hash function. The buffer can be represented as eight 64-bit registers (a, b, c, d, e, f, g, h). These registers are initialized to the following

64-bit integers (hexadecimal values):

a = 6A09E667F3BCC908 e = 510E527FADE682D1

b = BB67AE8584CAA73B f = 9B05688C2B3E6C1F

c = 3C6EF372FE94F82B g = 1F83D9ABFB41BD6B

d = A54FF53A5F1D36F1 h = 5BE0CD19137E2179

Step 4 Process message in 1024-bit (128-word) blocks. The heart of the algorithm is a module that consists of 80 rounds; this module is labeled F in Figure 11.9. The logic is illustrated in Figure 11.10.

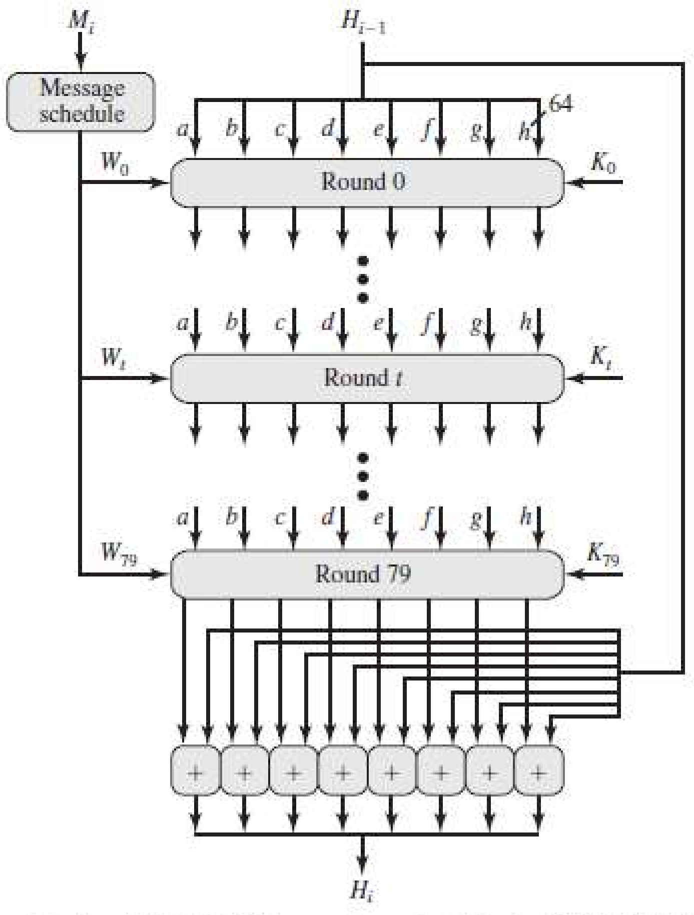


Figure 11.10 SHA-512 Processing of a Single 1024-Bit Block

(a) Each round takes as input the 512-bit buffer value, abcdefgh, and updates the contents of the buffer.

(b) At input to the first round, the buffer has the value of the intermediate hash value, *Hi*-1. Each round *t* makes use of a 64-bit value *Wt*, derived from the current 1024-bit block being processed (*Mi*).

(c) These values are derived using a message schedule described subsequently. Each round also makes use of an additive constant *Kt*, where 0 … *t* … 79 indicates one of the 80 rounds.

(d) The constants provide a "randomized" set of 64-bit patterns, which should eliminate any regularities in the input data. Table 11.4 shows these constants in hexadecimal format (from left to right).

The output of the eightieth round is added to the input to the first round (*Hi*-1) to produce *Hi*. The addition is done independently for each of the eight words in the buffer with each of the corresponding words in *Hi*-1, using addition modulo 264.

Step 5 Output. After all *N* 1024-bit blocks have been processed, the output from the *N*th stage is the 512-bit message digest.

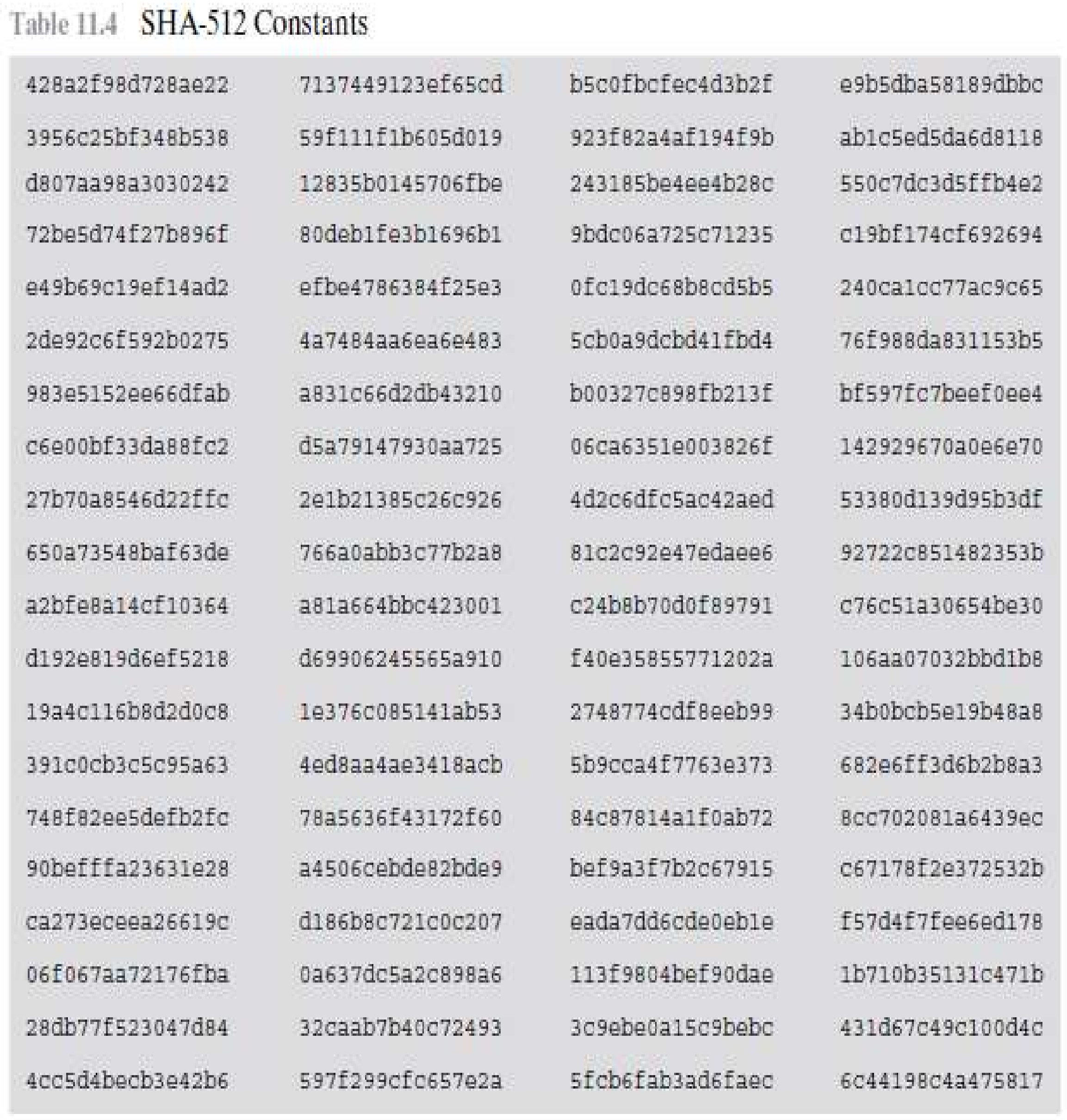


Table 11.4 SHA-512 Constants

428a2f98d728ae22	7137449123ef65cd	b5c0fbcfec4d3b2f	e9b5dba58189dbbc
3956c25bf348b538	59f111f1b605d019	923f82a4af194f9b	ab1c5ed5da6d8118
d807aa98a3030242	12835b0145706fbe	243185be4ee4b28c	550c7dc3d5ffb4e2
72be5d74f27b896f	80deb1fe3b1696b1	9bdc06a725c71235	c19bf174cf692694
e49b69c19ef14ad2	efbe4786384f25e3	0fc19dc68b8cd5b5	240ca1cc77ac9c65
2de92c6f592b0275	4a7484aa6ea6e483	5cb0a9dcbd41fbd4	76f988da831153b5
983e5152ee66dfab	a831c66d2db43210	b00327c898fb213f	bf597fc7beef0ee4
c6e00bf33da88fc2	d5a79147930aa725	06ca6351e003826f	142929670a0e6e70
27b70a8546d22ffc	2e1b21385c26c926	4d2c6dfc5ac42aed	53380d139d95b3df
650a73548baf63de	766a0abb3c77b2a8	81c2c92e47edaee6	92722c851482353b
a2bfe8a14cf10364	a81a664bbc423001	c24b8b70d0f89791	c76c51a30654be30
d192e819d6ef5218	d69906245565a910	f40e35855771202a	106aa07032bbd1b8
19a4c116b8d2d0c8	1e376c085141ab53	2748774cdf8eeb99	34b0bcb5e19b48a8
391c0cb3c5c95a63	4ed8aa4ae3418acb	5b9cca4f7763e373	682e6ff3d6b2b8a3
748f82ee5defb2fc	78a5636f43172f60	84c87814a1f0ab72	8cc702081a6439ec
90befffa23631e28	a4506cebde82bde9	bef9a3f7b2c67915	c67178f2e372532b
ca273eceea26619c	d186b8c721c0c207	eada7dd6cde0eb1e	f57d4f7fee6ed178
06f067aa72176fba	0a637dc5a2c898a6	113f9804bef90dae	1b710b35131c471b
28db77f523047d84	32caab7b40c72493	3c9ebe0a15c9bebc	431d67c49c100d4c
4cc5d4becb3e42b6	597f299cfc657e2a	5fcb6fab3ad6faec	6c44198c4a475817

SHA-512 Round Function

Let us look in more detail at the logic in each of the 80 steps of the processing of one 512-bit block

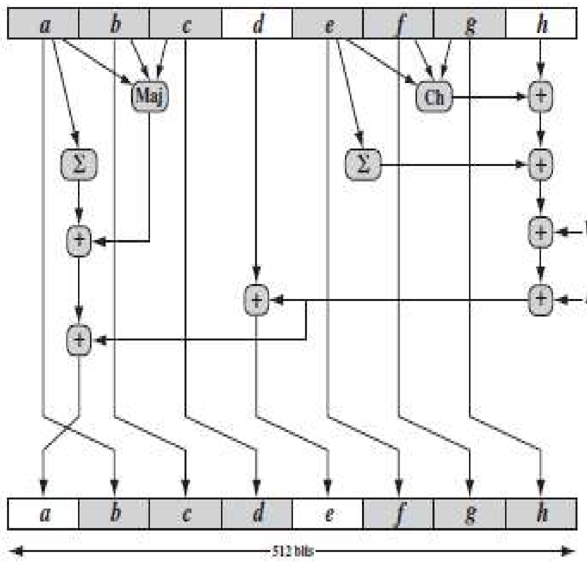


Figure 11.11 Elementary SHA-512 Operation (single round)

Each round is defined by the following set of equations:

$$T_1 = h + \mathrm{Ch}(e, f, g) + \left(\textstyle\sum_1^{512} e\right) + W_t + K_t$$
$$T_2 = \left(\textstyle\sum_0^{512} a\right) + \mathrm{Maj}(a, b, c)$$
$$h = g$$
$$g = f$$
$$f = e$$
$$e = d + T_1$$
$$d = c$$
$$c = b$$
$$b = a$$
$$a = T_1 + T_2$$

where

t	= step number; $0 \le t \le 79$
$\mathrm{Ch}(e, f, g)$	= (e AND f) $\oplus$ (NOT e AND g) *the conditional function: If e then f else g*
$\mathrm{Maj}(a, b, c)$	= (a AND b) $\oplus$ (a AND c) $\oplus$ (b AND c) *the function is true only of the majority (two or three) of the arguments are true*
$\left(\sum_0^{512} a\right)$	= $\mathrm{ROTR}^{28}(a) \oplus \mathrm{ROTR}^{34}(a) \oplus \mathrm{ROTR}^{39}(a)$
$\left(\sum_1^{512} e\right)$	= $\mathrm{ROTR}^{14}(e) \oplus \mathrm{ROTR}^{18}(e) \oplus \mathrm{ROTR}^{41}(e)$
$\mathrm{ROTR}^n(x)$	= circular right shift (rotation) of the 64-bit argument x by n bits

W_t = a 64-bit word derived from the current 1024-bit input block

K_t = a 64-bit additive constant, + = addition modulo 2^{64}

SHA-512 message schedule Expansion

Two observations can be made about the round function.

1. Six of the eight words of the output of the round function involve simply permutation (*b*, *c*, *d*, *f* , *g*, *h*) by means of rotation. This is indicated by shading in Figure 11.11.

2. Only two of the output words (*a*, *e*) are generated by substitution. Word *e* is a function of input variables (*d*, *e*, *f* , *g*, *h*), as well as the round word *Wt* and the constant *Kt*. Word *a* is a function of all of the input variables except *d*, as well as the round word *Wt* and the constant *Kt*. It remains to indicate how the 64-bit word values *Wt* are derived from the 1024-bit message.

Figure 11.12 illustrates the mapping.

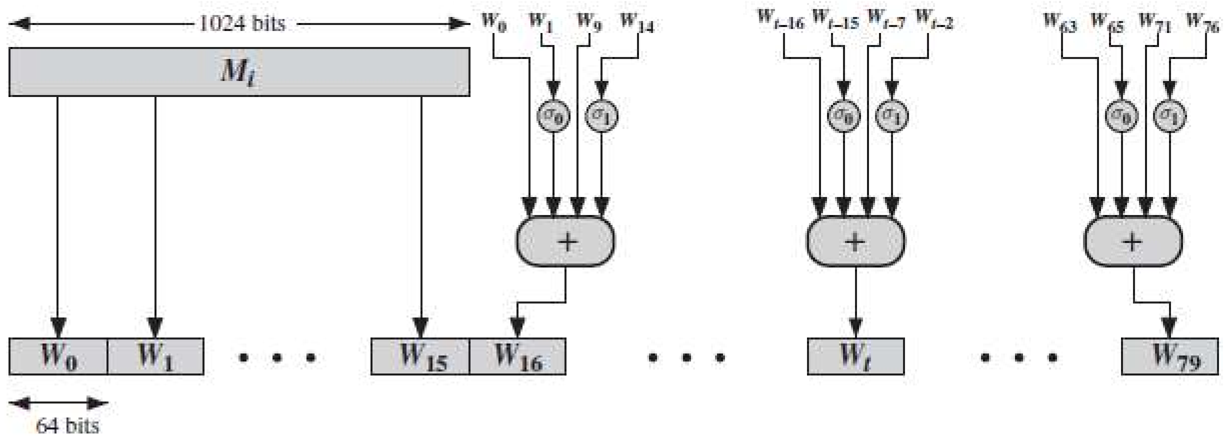

1024 bits
M_i
W_0 W_1 W_9 W_{14}
W_{t-16} W_{t-15} W_{t-7} W_{t-2}
W_{63} W_{65} W_{71} W_{76}
σ_0
σ_1
+
W_0 W_1 W_{15} W_{16} W_t W_{79}
64 bits

Figure 11.12 Creation of 80-word Input Sequence for SHA-512 Processing of Single Block

The first 16 values of *Wt* are taken directly from the 16 words of the current block. The remaining values are defined as

Wt = s1 512(*Wt*-2) + *Wt*-7 + s0 512(*Wt*-15) + *Wt*-16

where

s0 512(*x*) = ROTR1(*x*) ⊕ ROTR8(*x*) ⊕ SHR7(*x*)

s1 512(*x*) = ROTR19(*x*) ⊕ ROTR61(*x*) ⊕ SHR6(*x*)

ROTR*n*(*x*) = circular right shift (rotation) of the 64-bit argument *x* by *n* bits

SHR*n*(*x*) = left shift of the 64-bit argument *x* by *n* bits with padding by zeros on the right

+ = addition modulo 264

Thus, in the first 16 steps of processing, the value of *Wt* is equal to the corresponding word in the message block. For the remaining 64 steps, the value of *Wt* consists of the circular left shift by one bit of the XOR of four of the preceding values of *Wt*, with two of those values subjected to shift and rotate operations.

Message Authentication Requirements:

In the context of communications across a network, the following attacks can be identified.

1. **Disclosure:** Release of message contents to any person or process not possessing the appropriate cryptographic key.
2. **Traffic analysis:** Discovery of the pattern of traffic between parties. In a connection- oriented application, the frequency and duration of connections could be determined. In either a connection-oriented or connectionless environment, the number and length of messages between parties could be determined.
3. **Masquerade:** Insertion of messages into the network from a fraudulent source. This includes the creation of messages by an opponent that are purported to come from an authorized entity. Also included are fraudulent acknowledgments of message receipt or nonreceipt by someone other than the message recipient.
4. **Content modification:** Changes to the contents of a message, including insertion, deletion, transposition, and modification.
5. **Sequence modification:** Any modification to a sequence of messages between parties, including insertion, deletion, and reordering.
6. **Timing modification:** Delay or replay of messages. In a connection-oriented application, an entire session or sequence of messages could be a replay of some previous valid session, or individual messages in the sequence could be delayed or replayed. In a connectionless application, an individual message (e.g., datagram) could be delayed or replayed.
7. **Source repudiation:** Denial of transmission of message by source.
8. **Destination repudiation:** Denial of receipt of message by destination.

Measures to deal with the first two attacks are in the realm of message confidentiality and are dealt with in Part One. Measures to deal with items (3) through (6) in the foregoing list are generally regarded as message authentication. Mechanisms for dealing specifically with item (7) come under the heading of digital signatures. Generally, a digital signature technique will also counter some or all of the attacks listed under items (3) through (6). Dealing with item (8) may require a combination of the use of digital signatures and a protocol designed to counter this attack. In summary, message authentication is a procedure to verify that received messages come from the alleged source and have not been altered. Message authentication may also verify sequencing and timeliness.

HMAC Algorithm

HMAC has been issued as RFC 2104, has been chosen as the mandatory-to-implement MAC for IP security, and is used in other Internet protocols, such as SSL. HMAC has also been issued as a NIST standard (FIPS 198).

HMAC Design Objectives:

RFC 2104 lists the following design objectives for HMAC.

• To use, without modifications, available hash functions. In particular, to use hash functions that perform well in software and for which code is freely and widely available.

• To allow for easy replaceability of the embedded hash function in case faster or more secure hash functions are found or required.

• To preserve the original performance of the hash function without incurring a significant degradation.

• To use and handle keys in a simple way.

• To have a well understood cryptographic analysis of the strength of the authentication mechanism based on reasonable assumptions about the embedded hash function.

The first two objectives are important to the acceptability of HMAC. HMAC treats the hash function as a “black box.” This has two benefits. First, an existing implementation of a hash function can be used as a module in implementing HMAC.

HMAC Algorithm

Figure 12.5 illustrates the overall operation of HMAC. Define the following terms.

H = embedded hash function (e.g., MD5, SHA-1, RIPEMD-160)

IV = initial value input to hash function

M = message input to HMAC (including the padding specified in the embedded
hash function)

Yi = *i* th block of M, 0 … *i* … (*L* - 1)

L = number of blocks in *M*

b = number of bits in a block

n = length of hash code produced by embedded hash function

K = secret key; recommended length is Ú *n*; if key length is greater than *b*, the
key is input to the hash function to produce an *n*-bit key

K+ = *K* padded with zeros on the left so that the result is *b* bits in length

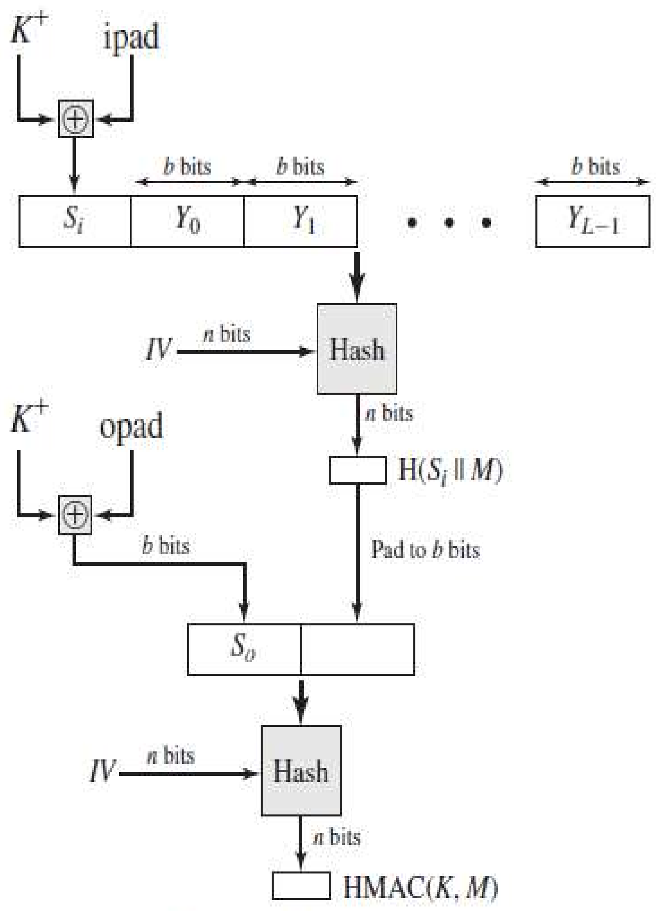


Figure 12.5 HMAC Structure

ipad = 00110110 (36 in hexadecimal) repeated *b*/8 times

opad = 01011100 (5C in hexadecimal) repeated *b*/8 times

Then HMAC can be expressed as

HMAC(*K*, *M*) = H[(*K*+ _opad) } H[(*K*+ _ipad) } *M*]]

We can describe the algorithm as follows.

1. Append zeros to the left end of *K* to create a *b*-bit string *K*+ (e.g., if *K* is of length 160 bits and *b* = 512, then *K* will be appended with 44 zeroes).

2. XOR (bitwise exclusive-OR) *K*+ with ipad to produce the *b*-bit block *Si*.

3. Append *M* to *Si*.

4. Apply H to the stream generated in step 3.

5. XOR *K*+ with opad to produce the b-bit block *So*.

6. Append the hash result from step 4 to *So*.

7. Apply H to the stream generated in step 6 and output the result.

Note that the XOR with ipad results in flipping one-half of the bits of *K*. Similarly, the XOR with opad results in flipping one-half of the bits of *K*, using a different set of bits. In effect, by passing *Si* and *So* through the compression function of the hash algorithm, we have pseudorandomly generated two keys from *K*.

Security of HMAC

The security of a MAC function is generally expressed in terms of the probability of successful forgery with a given amount of time spent by the forger and a given number of message-tag pairs created with the same key. In essence, it is proved in [BELL96a] that for a given level of effort (time, message–tag pairs) on messages generated by a legitimate user and seen by the attacker, the probability of successful attack on HMAC is equivalent to one of the following attacks on the embedded hash function.

1. The attacker is able to compute an output of the compression function even with an *IV* that is random, secret, and unknown to the attacker.
2. The attacker finds collisions in the hash function even when the *IV* is random and secret.

In the first attack, we can view the compression function as equivalent to the hash function applied to a message consisting of a single *b*-bit block. For this attack, the *IV* of the hash function is replaced by a secret, random value of *n* bits. An attack on this hash function requires either a brute-force attack on the key, which is a level of effort on the order of 2*n*, or a birthday attack, which is a special case of the second attack, discussed next.

In the second attack, the attacker is looking for two messages *M* and *M*_ that produce the same hash: H(*M*) = H(*M*_). the attacker can choose any set of messages and work on these off line on a dedicated computing facility to find a collision. Because the attacker knows the hash algorithm and the default *IV*, the attacker can generate the hash code for each of the messages that the attacker generates. However, when attacking HMAC, the attacker cannot generate message/ code pairs off line because the attacker does not know *K*. Therefore, the attacker must observe a sequence of messages generated by HMAC under the same key and perform the attack on these known messages. For a hash code length of 128 bits, this requires 264 observed blocks (272 bits) generated using the same key. On a 1-Gbps link, one would need to observe a continuous stream of messages with no change in key for about 150,000 years in order to succeed. Thus, if speed is a concern, it is fully acceptable to use MD5 rather than SHA-1 as the embedded hash function for HMAC.

CMAC

The Data Authentication Algorithm (DAA), which is now obsolete. Then we examine CMAC, which is designed to overcome the deficiencies of DAA.

Data Authentication Algorithm

The **Data Authentication Algorithm** (DAA), based on DES, has been one of the most widely used MACs for a number of years. The algorithm is both a FIPS publication (FIPS PUB 113) and an ANSI standard (X9.17). However, as we discuss subsequently, security weaknesses in this algorithm have been discovered, and it is being replaced by newer and stronger algorithms.

The algorithm can be defined as using the cipher block chaining (CBC) mode of operation of DES (Figure 6.4) with an initialization vector of zero. The data (e.g., message, record, file, or program) to be authenticated are grouped into contiguous 64-bit blocks: *D*1, *D*2,c, *DN*. If necessary, the final block is padded on the right with zeroes to form a full 64-bit block. Using the DES encryption algorithm E and a secret key *K*, a data authentication code (DAC) is calculated as follows (Figure 12.7).

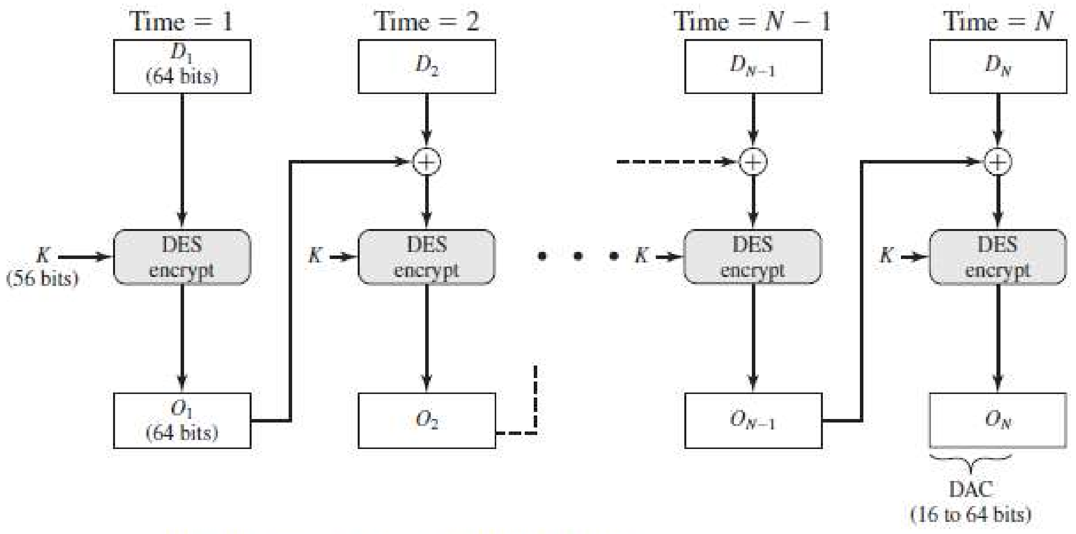


Figure 12.7 Data Authentication Algorithm (FIPS PUB 113)

*O*1 = E(*K*, *D*)

*O*2 = E(*K*, [*D*2 _*O*1])

*O*3 = E(*K*, [*D*3 _*O*2])

.

.

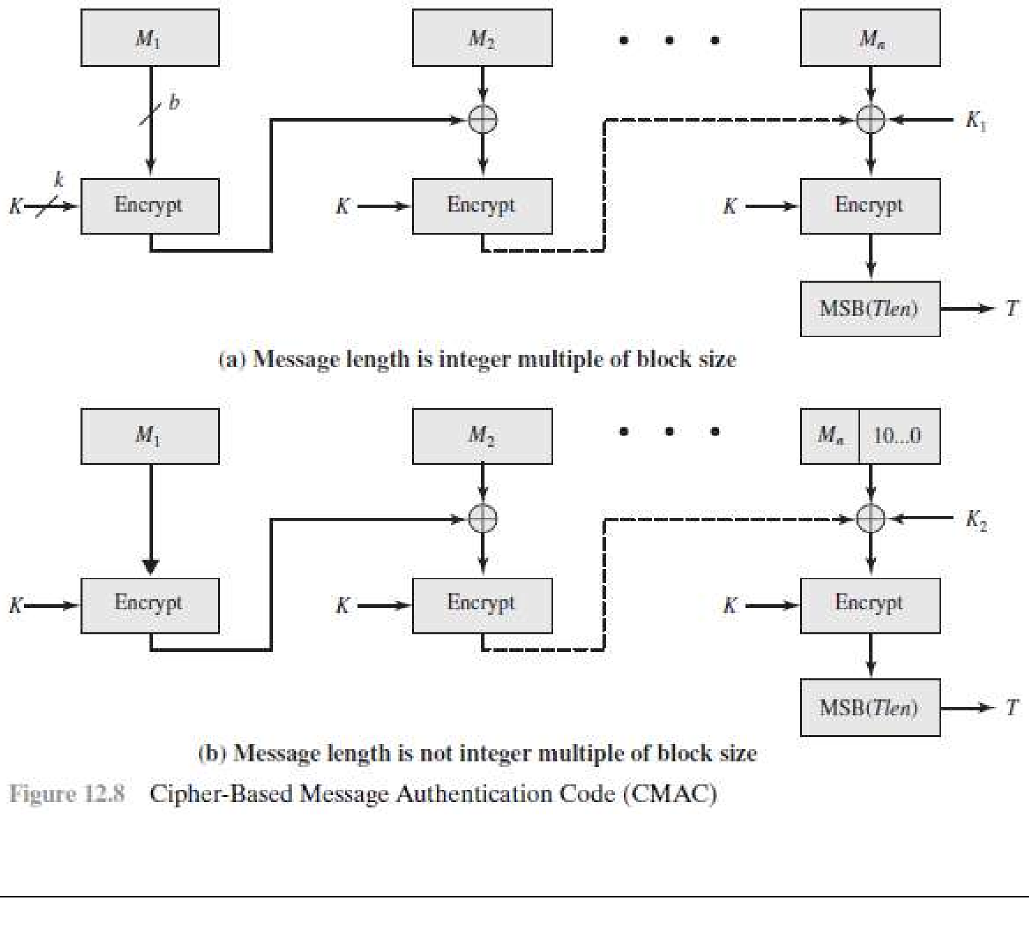
.*ON* = E(*K*, [*DN*_*ON*-1])

The DAC consists of either the entire block *ON* or the leftmost *M* bits of the block, with 16 … *M* … 64.

Cipher-Based Message Authentication Code (CMAC)

Black and Rogaway [BLAC00] demonstrated that this limitation could be overcome using three keys: one key *K* of length *k* to be used at each step of the cipher block chaining and two keys of length *b*, where *b* is the cipher block length. This proposed construction was refined by Iwata and Kurosawa so that the two *n*-bit keys could be derived from the encryption key, rather than being provided separately [IWAT03]. This refinement, adopted by NIST, is the **Cipher-based Message Authentication Code** (CMAC) mode of operation for use with AES and triple DES. It is specified in NIST Special Publication 800-38B.

First, let us define the operation of CMAC when the message is an integer multiple *n* of the cipher block length *b*. For AES, *b* = 128, and for triple DES, *b* = 64. The message is divided into *n* blocks (*M*1, *M*2,c, *Mn*). The algorithm makes use of a *k*-bit encryption key *K* and a *b*-bit constant, *K*1. For AES, the key size *k* is 128, 192, or 256 bits; for triple DES, the key size is 112 or 168 bits. CMAC is calculated as follows (Figure 12.8).



M_1
M_2
M_n
b
k
K
Encrypt
K_1
MSB(*Tlen*)
T
(a) Message length is integer multiple of block size
M_n
10...0
K_2
(b) Message length is not integer multiple of block size

Figure 12.8 Cipher-Based Message Authentication Code (CMAC)

$$C1 = E(K, M1)$$

$$C2 = E(K, [M2 \_C1])$$

$$C3 = E(K, [M3 \_C2])$$

.

.

.

$$Cn = E(K, [Mn \_Cn\text{-}1 \_K1])$$

$$T = \text{MSB}Tlen(Cn)$$

where

T = message authentication code, also referred to as the tag

$Tlen$ = bit length of T

MSBs(X) = the s leftmost bits of the bit string X

If the message is not an integer multiple of the cipher block length, then the final block is padded to the right (least significant bits) with a 1 and as many 0s as necessary so that the final block is also of length b. The CMAC operation then proceeds as before, except that a different b-bit key K2 is used instead of K1.

The two b-bit keys are derived from the k-bit encryption key as follows.

$$L = E(K, 0b\ )$$

$$K1 = L \# x$$

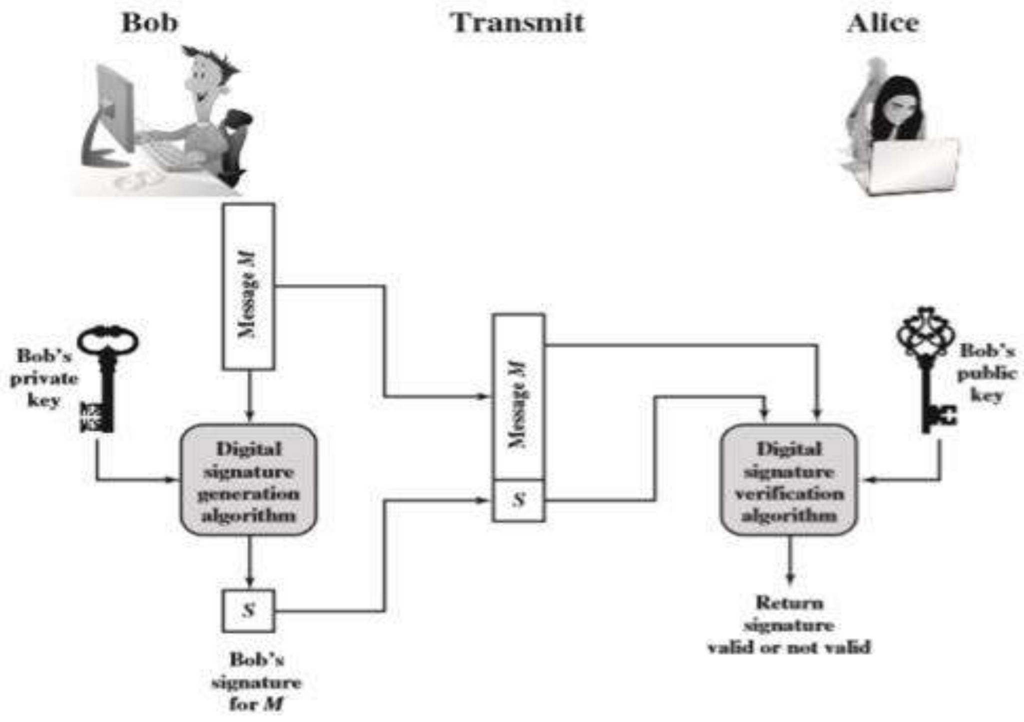
$$K2 = L \# x2 = (L \# x) \# x$$

where multiplication (#) is done in the finite field GF(2b) and x and x2 are first and second-order polynomials that are elements of GF(2b). Thus, the binary representation of x consists of b - 2 zeros followed by 10; the binary representation of x2 consists of b - 3 zeros followed by 100. The finite field is defined with respect to an irreducible polynomial that is lexicographically first among all such polynomials with the minimum possible number of nonzero terms. For the two approved block sizes, the polynomials are $x64 + x4 + x3 + x + 1$ and $x128 + x7 + x2 + x + 1$.

To generate K1 and K2, the block cipher is applied to the block that consists entirely of 0 bits. The first subkey is derived from the resulting ciphertext by a left shift of one bit and, conditionally, by XORing a constant that depends on the block size. The second subkey is derived in the same manner from the first subkey.

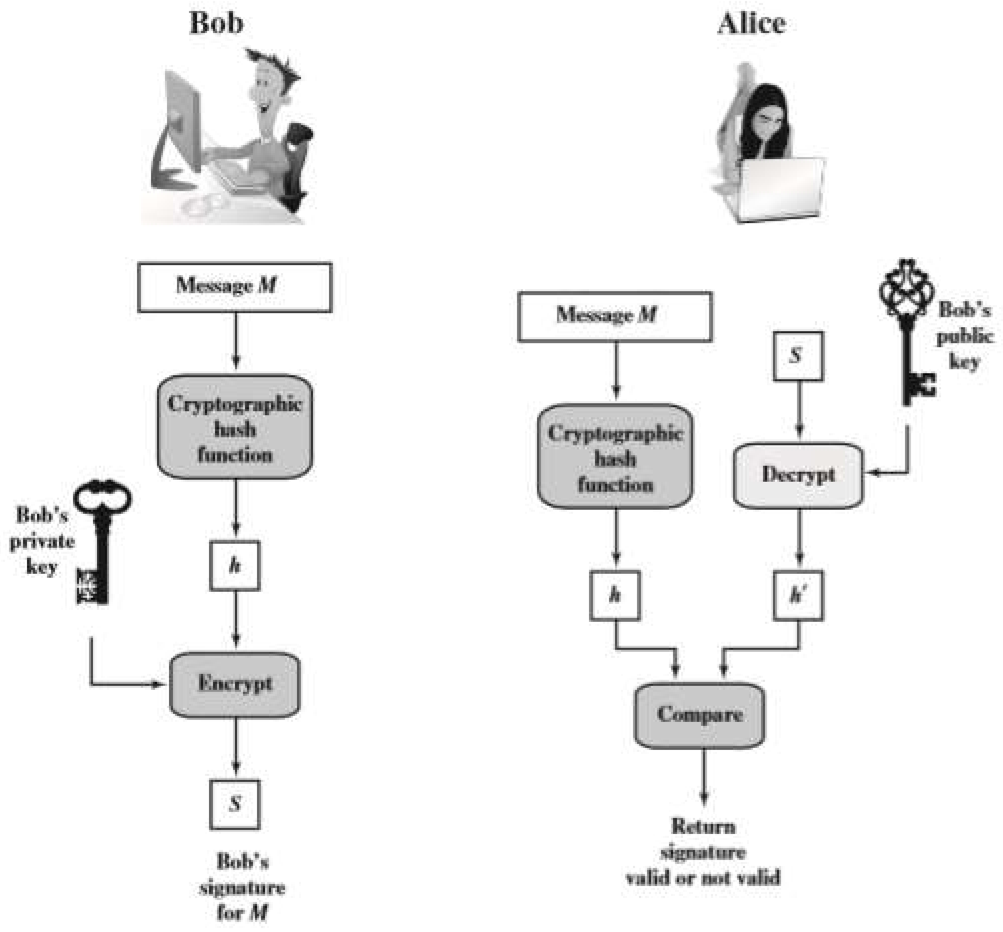
DIGITAL SIGNATURE

A digital signature is a digital code (generated and authenticated by public key encryption) which is attached to an electronically transmitted document to verify its contents and the sender's identity.


Bob
Transmit
Alice
Bob's private key
Message M
Digital signature generation algorithm
S
Bob's signature for M
Message M
S
Digital signature verification algorithm
Bob's public key
Return signature valid or not valid

- In the above figure, represented the generic model of Digital signature process.
- Bob can sign a message using a digital signature generation algorithm.
- The inputs to the algorithm are the message and Bob's private key.
- Any other user, say Alice, can verify the signature using a verification algorithm, whose inputs are the message, the signature, and Bob's public key.

In simplified terms, the essence of the digital signature mechanism is shown in Figure,



Properties

Message authentication protects two parties who exchange messages from any third party. However, it does not protect the two parties against each other.

The digital signature must have the following properties:

- It must verify the author and the date and time of the signature.
- It must authenticate the contents at the time of the signature.
- It must be verifiable by third parties, to resolve disputes.

Thus, the digital signature function includes the authentication function.

Attacks and Forgeries

Following are the types of attacks, here A denotes the user whose signature method is being attacked, and C denotes the attacker.

- **Key-only attack**: C only knows A’s public key.
- **Known message attack**: C is given access to a set of messages and their signatures.
- **Generic chosen message attack**: C chooses a list of messages before attempting to breaks A’s signature scheme, independent of A’s public key. C then obtains from A valid signatures for the chosen messages. The attack is generic, because it does not depend on A’s public key; the same attack is used against everyone.

- **Directed chosen message attack**: Similar to the generic attack, except that the list of messages to be signed is chosen after C knows A's public key but before any signatures are seen.
- **Adaptive chosen message attack:** C is allowed to use A as an "oracle." This means that C may request from A signatures of messages that depend on previously obtained message-signature pairs.] then defines success at breaking a signature scheme as an outcome in which C can do any of the following with a non-negligible probability:
- **Total break**: C determines A's private key.
- **Universal forgery**: C finds an efficient signing algorithm that provides an equivalent way of constructing signatures on arbitrary messages.
- **Selective forgery**: C forges a signature for a particular message chosen by C.
- **Existential forgery:** C forges a signature for at least one message. C has no control over the message. Consequently, this forgery may only be a minor nuisance to A.

Digital Signature Requirements

Following are the requirements for a digital signature.

- The signature must use some information unique to the sender to prevent both forgery and denial.

- It must be relatively easy to produce the digital signature.
- It must be relatively easy to recognize and verify the digital signature.
- It must be computationally infeasible to forge a digital signature, either by constructing a new message for an existing digital signature or by constructing a fraudulent digital signature for a given message.
- It must be practical to retain a copy.
- Be practical to save digital signature in storage
- A secure hash function, embedded in a scheme, provides a basis for satisfying these requirements. However, care must be taken in the design of the details of the scheme.

Direct Digital Signature

- involves only sender and receiver
- assumed receiver has sender's public-key
- digital signature made by sender signing entire message or hash with private-key
- can encrypt using receivers public-key
- important that sign first then encrypt message and signature
- security depends on sender's private-key

Digital Signature Characteristics

- A public key scheme …

Two key pairs: a (long-time, permanent) durable private/public key pair

a (nonce-like, one-time, per-message) disposable private/public key pair

- Both key pairs generated by SENDER
- Signature is two numbers, depending on message hash and secret info
- A verification calculation succeeds iff the two numbers correctly depend on the secret info
- Disposable private/public key pair makes a collection of signatures of the sender uncorrelated, so hard to break, analytically or statistically.

ELGAMAL DIGITAL SIGNATURE SCHEME

The **ElGamal signature scheme** is a digital signature scheme which is based on the difficulty of computing discrete logarithms. It was described by Taher Elgamal in 1984.

- use private key for encryption (signing)
- uses public key for decryption (verification)
- each user generates their keys, for example: user A generates their key
  - chooses a secret key (number): $1 < x_A < q-1$
  - compute their public key: $y_A = a\ x_A \bmod q$
- Alice signs a message M to Bob by computing , the hash $m = H(M)$, $0 <= m <= (q-1)$
  - chose random integer K with $1 <= K <= (q-1)$ and $\gcd(K,q-1)=1$
  - compute temporary key: $S_1 = a^k \bmod q$
  - compute K^{-1} the inverse of K mod (q-1)
  - compute the value: $S_2 = K^{-1}(m-x_AS_1) \bmod (q-1)$
  - signature is:(S_1,S_2)
- any user B can verify the signature by computing
  - $V_1 = a^m \bmod q$
  - $V_2 = y_A^{S1}\ S_1^{S2} \bmod q$
  - signature is valid if $V_1 = V_2$
- **Example problem:**

For example, let us start with the prime field GF (19); that is, q = 19.

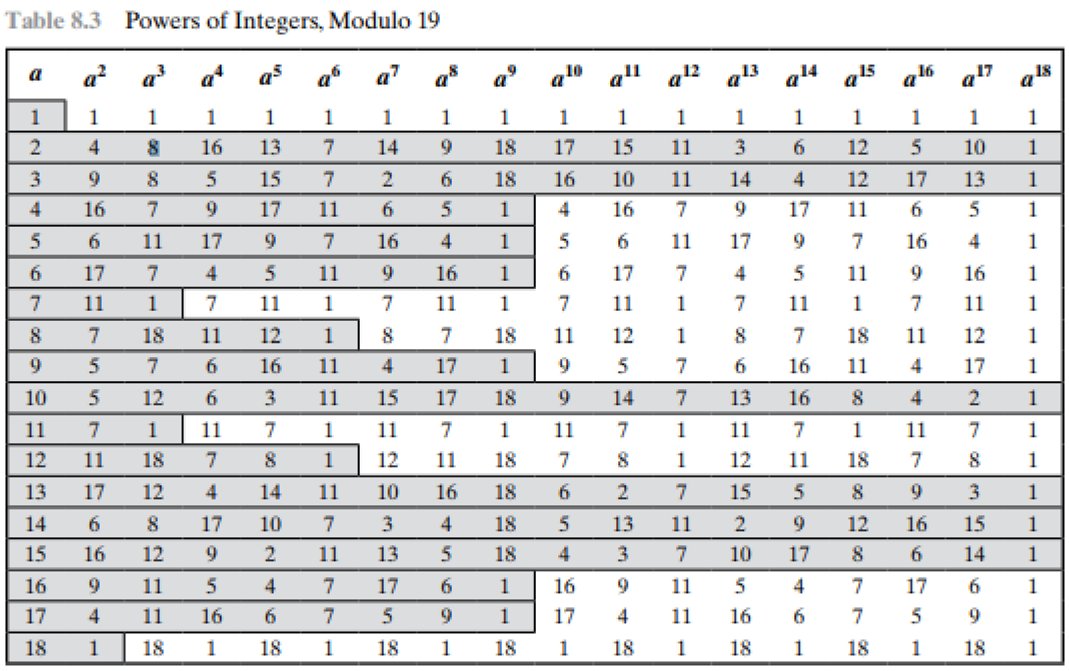


Table 8.3 Powers of Integers, Modulo 19

a	a^2	a^3	a^4	a^5	a^6	a^7	a^8	a^9	a^{10}	a^{11}	a^{12}	a^{13}	a^{14}	a^{15}	a^{16}	a^{17}	a^{18}
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	4	8	16	13	7	14	9	18	17	15	11	3	6	12	5	10	1
3	9	8	5	15	7	2	6	18	16	10	11	14	4	12	17	13	1
4	16	7	9	17	11	6	5	1	4	16	7	9	17	11	6	5	1
5	6	11	17	9	7	16	4	1	5	6	11	17	9	7	16	4	1
6	17	7	4	5	11	9	16	1	6	17	7	4	5	11	9	16	1
7	11	1	7	11	1	7	11	1	7	11	1	7	11	1	7	11	1
8	7	18	11	12	1	8	7	18	11	12	1	8	7	18	11	12	1
9	5	7	6	16	11	4	17	1	9	5	7	6	16	11	4	17	1
10	5	12	6	3	11	15	17	18	9	14	7	13	16	8	4	2	1
11	7	1	11	7	1	11	7	1	11	7	1	11	7	1	11	7	1
12	11	18	7	8	1	12	11	18	7	8	1	12	11	18	7	8	1
13	17	12	4	14	11	10	16	18	6	2	7	15	5	8	9	3	1
14	6	8	17	10	7	3	4	18	5	13	11	2	9	12	16	15	1
15	16	12	9	2	11	13	5	18	4	3	7	10	17	8	6	14	1
16	9	11	5	4	7	17	6	1	16	9	11	5	4	7	17	6	1
17	4	11	16	6	7	5	9	1	17	4	11	16	6	7	5	9	1
18	1	18	1	18	1	18	1	18	1	18	1	18	1	18	1	18	1

It has primitive roots {2, 3, 10, 13, 14, 15}, as shown in the table,

We choose a = 10.

Alice generates a key pair as follows:

1. Alice chooses X_A = 16.

2. Then $Y_A = a^{XA}$ mod q = a^{16} mod 19 = 4.

3. Alice's private key is 16; Alice's pubic key is {q, a, Y_A} = {19, 10, 4}.

Suppose Alice wants to sign a message with hash value m = 14.

1. Alice chooses K = 5, which is relatively prime to q - 1 = 18.

2. S1 = a^K mod q = 105 mod 19 = 3 (see Table).

3. K-1 mod (q - 1) = 5-1 mod 18 = 11.

4. S2 = K-1 (m - X_A S1) mod (q - 1) = 11 (14 - (16) (3))mod 18 = -374 mod 18 = 4.

Bob can verify the signature as follows.

1. V1 = a^m mod q = 1014 mod 19 = 16.

2. V2 = $(Y^A)^{S1}$ $(S1)^{S2}$ mod q = (4^3) (3^4) mod 19 = 5184 mod 19 = 16.

As, V1=V2, thus, the signature is valid.

Key Management and Distribution

Symmetric Key Distribution Using Symmetric Encryption:

For symmetric encryption to work, the two parties to an exchange must share the same key, and that key must be protected from access by others. Furthermore, frequent key changes are usually desirable to limit the amount of data compromised if an attacker learns the key. Therefore, the strength of any cryptographic system rests with the *key distribution technique*, a term that refers to the means of delivering a key to two parties who wish to exchange data without allowing others to see the key. For two parties A and B, key distribution can be achieved in a number of ways, as follows:

1. A can select a key and physically deliver it to B.
2. A third party can select the key and physically deliver it to A and B.
3. If A and B have previously and recently used a key, one party can transmit the new key to the other, encrypted using the old key.
4. If A and B each has an encrypted connection to a third party C, C can deliver a key on the encrypted links to A and B.

Options 1 and 2 call for manual delivery of a key. For link encryption, this is a reasonable requirement, because each link encryption device is going to be exchanging data only with its partner on the other end of the link. However, for **end- to-end encryption** over a network, manual delivery is awkward. In a distributed system, any given host or terminal may need to engage in exchanges with many other hosts and terminals over time. Thus, each device needs a number of keys supplied dynamically. The problem is especially difficult in a wide-area distributed system.

The scale of the problem depends on the number of communicating pairs that must be supported. If end-to-end encryption is done at a network or IP level, then a key is needed for each pair of hosts on the network that wish to communicate. Thus, if there are N hosts, the number of required keys is $[N(N - 1)]/2$.

Returning to our list, option 3 is a possibility for either link encryption or end-to-end encryption, but if an attacker ever succeeds in gaining access to one key, then all subsequent keys will be revealed. Furthermore, the initial distribution of potentially millions of keys still must be made. For end-to-end encryption, some variation on option 4 has been widely adopted.

In this scheme, a key distribution center is responsible for distributing keys to pairs of users (hosts, processes, applications) as needed. Each user must share a unique key with the key distribution center for purposes of key distribution. The use of a key distribution center is based on the use of a hierarchy of keys. At a minimum, two levels of keys are used (Figure 14.2).

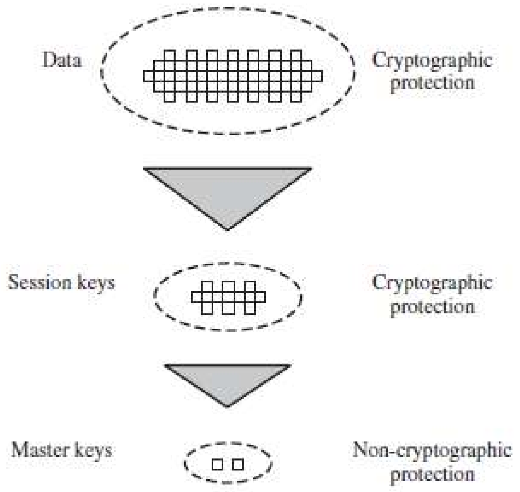


Figure 14.2 The Use of a Key Hierarchy

Communication between end systems is encrypted using a temporary key, often referred to as a **session key**. Typically, the session key is used for the duration of a logical connection, such as a frame relay connection or transport connection, and then discarded. Each session key is obtained from the key distribution center over the same networking facilities used for end-user communication. Accordingly, session keys are transmitted in encrypted form, using a **master key** that is shared by the key distribution center and an end system or user. For each end system or user, there is a unique master key that it shares with the key distribution center. Of course, these master keys must be distributed in some fashion.

A Key Distribution Scenario

The key distribution concept can be deployed in a number of ways. A typical scenario is illustrated in Figure 14.3, which is based on a figure in [POPE79]. The scenario assumes that each user shares a unique master key with the key distribution center (KDC).

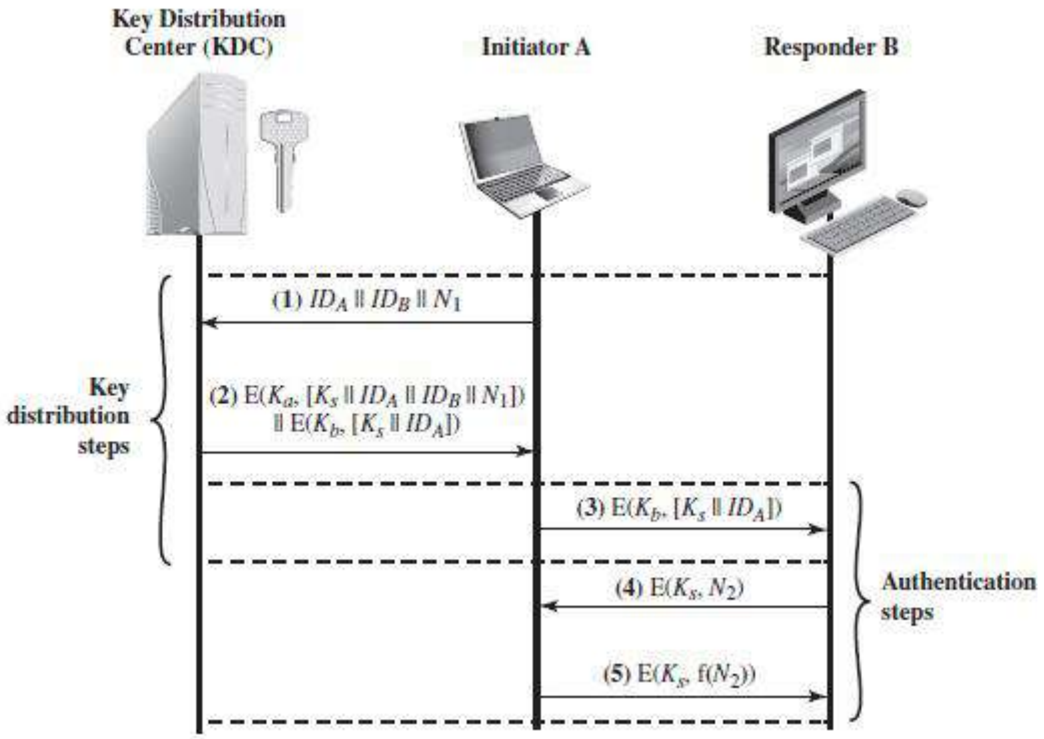

Key Distribution Center (KDC)
Initiator A
Responder B
Key distribution steps
(1) $ID_A \| ID_B \| N_1$
(2) $E(K_a, [K_s \| ID_A \| ID_B \| N_1]) \| E(K_b, [K_s \| ID_A])$
(3) $E(K_b, [K_s \| ID_A])$
(4) $E(K_s, N_2)$
(5) $E(K_s, f(N_2))$
Authentication steps

Figure 14.3 Key Distribution Scenario

Let us assume that user A wishes to establish a logical connection with B and requires a one-time session key to protect the data transmitted over the connection. A has a master key, *Ka*, known only to itself and the KDC; similarly, B shares the master key *Kb* with the KDC. The following steps occur.

1. A issues a request to the KDC for a session key to protect a logical connection to B. The message includes the identity of A and B and a unique identifier, *N*1, for this transaction, which we refer to as a **nonce**. The nonce may be a counter, or a random number; the minimum requirement is that it differs with each request. Also, to prevent masquerade, it should be difficult for an opponent to guess the nonce. Thus, a random number is a good choice for a nonce.

2. The KDC responds with a message encrypted using *Ka*. Thus, A is the only one who can successfully read the message, and A knows that it originated at the KDC. The message includes two items intended for A:

- The one-time session key, *Ks*, to be used for the session
- The original request message, including the nonce, to enable A to match this response with the appropriate request Thus, A can verify that its original request was not altered before reception by the KDC and, because of the nonce, that this is not a replay of some previous request.

In addition, the message includes two items intended for B:

- The one-time session key, *Ks*, to be used for the session
- An identifier of A (e.g., its network address), *IDA*.

These last two items are encrypted with *Kb* (the master key that the KDC shares with B). They are to be sent to B to establish the connection and prove A's identity.

3. A stores the session key for use in the upcoming session and forwards to B the information that originated at the KDC for B, namely, *E*(*Kb*,[*Ks* } *IDA*]). Because this information is encrypted with *Kb*, it is protected from eavesdropping. B now knows the session key (*Ks*), knows that the other party is A (from *IDA*), and knows that the information originated at the KDC (because it is encrypted using *Kb*).
At this point, a session key has been securely delivered to A and B, and they may begin their protected exchange. However, two additional steps are desirable:

4. Using the newly minted session key for encryption, B sends a nonce, *N*2, to A.

5. Also, using *Ks*, A responds with f(*N*2), where f is a function that performs some transformation on *N*2 (e.g., adding one).

These steps assure B that the original message it received (step 3) was not a replay. Note that the actual key distribution involves only steps 1 through 3, but that steps 4 and 5, as well as step 3, perform an authentication function.

<u>Different types of Key Distribution Centers:</u>

Hierarchical Key Control

It is not necessary to limit the key distribution function to a single KDC. Indeed, for very large networks, it may not be practical to do so. As an alternative, a hierarchy of KDCs can be established. For example, there can be local KDCs, each responsible for a small domain of the overall internetwork, such as a single LAN or a single building. For communication among entities within the same local domain, the local KDC is responsible for key distribution. If two entities in different domains desire a shared key, then the corresponding local KDCs can communicate through a global KDC. In this case, any one of the three KDCs involved can actually select the key.

The hierarchical concept can be extended to three or even more layers, depending on the size of the user population and the geographic scope of the internetwork. A hierarchical scheme minimizes the effort involved in master key distribution, because most master keys are those shared by a local KDC with its local entities.

Session Key Lifetime

The more frequently session keys are exchanged, the more secure they are, because the opponent has less ciphertext to work with for any given session key. On the other hand, the distribution of session keys delays the start of any exchange and places a burden on network capacity. A security manager must try to balance these competing considerations in determining the lifetime of a particular session key.

For connection-oriented protocols, one obvious choice is to use the same session key for the length of time that the connection is open, using a new session key for each new session. If a logical connection has a very long lifetime, then it would be prudent to change the session key periodically, perhaps every time the PDU (protocol data unit) sequence number cycles.

For a connectionless protocol, such as a transaction-oriented protocol, there is no explicit connection initiation or termination. Thus, it is not obvious how often one needs to change the session key. The most secure approach is to use a new session key for each exchange. However, this negates one of the principal benefits of connectionless protocols, which is minimum overhead and delay for each transaction. A better strategy is to use a given session key for a certain fixed period only or for a certain number of transactions.

A Transparent Key Control Scheme

The steps involved in establishing a connection are shown in Figure 14.4. When one host wishes to set up a connection to another host, it transmits a connection-request packet (step 1). The SSM saves that packet and applies to the KDC for permission to establish the connection (step 2). The communication between the SSM and the KDC is encrypted using a master key shared only by this SSM and the KDC. If the KDC approves the connection request, it generates the session key and delivers it to the two appropriate SSMs, using a unique permanent key for each SSM (step 3). The requesting SSM can now release the connection request packet, and a connection is set up between the two end systems (step 4). All user data exchanged between the two end systems are encrypted by their respective SSMs using the one-time session key.

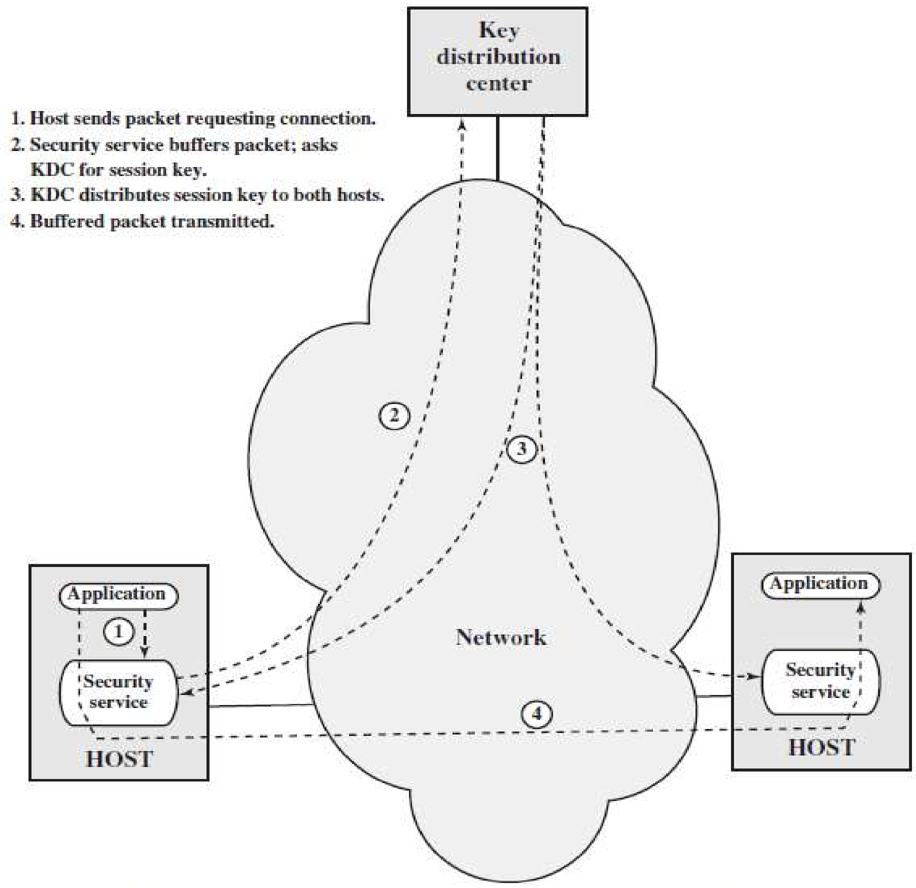

Key distribution center
1. Host sends packet requesting connection.
2. Security service buffers packet; asks KDC for session key.
3. KDC distributes session key to both hosts.
4. Buffered packet transmitted.
Application
Security service
HOST
Network
Application
Security service
HOST
1
2
3
4

Figure 14.4 Automatic Key Distribution for Connection-Oriented Protocol

The automated key distribution approach provides the flexibility and dynamic characteristics needed to allow a number of terminal users to access a number of hosts and for the hosts to exchange data with each other.

Decentralized Key Control

The use of a key distribution center imposes the requirement that the KDC be trusted and be protected from subversion. This requirement can be avoided if key distribution is fully decentralized. Although full decentralization is not practical for larger networks using symmetric encryption only, it may be useful within a local context. A decentralized approach requires that each end system be able to communicate in a secure manner with all potential partner end systems for purposes of session key distribution. Thus, there may need to be as many as $[n(n - 1)]/2$ master keys for a configuration with n end systems.

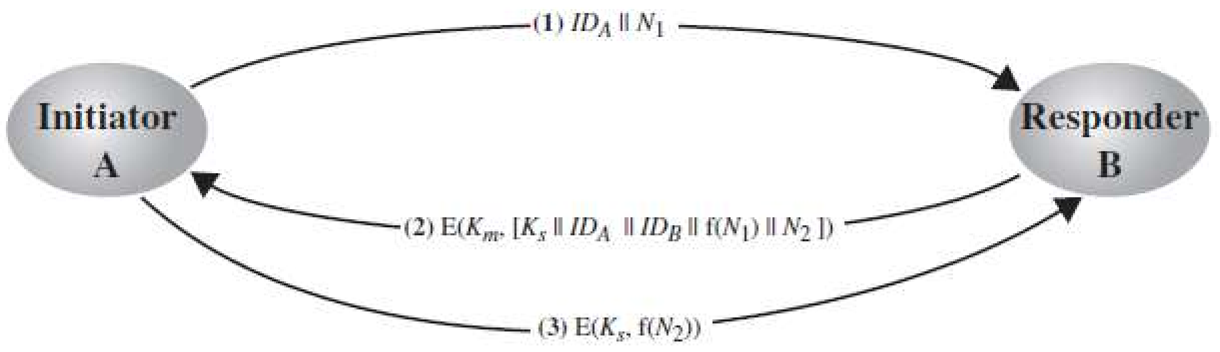


Figure 14.5 Decentralized Key Distribution

A session key may be established with the following sequence of steps (Figure 14.5).

1. A issues a request to B for a session key and includes a nonce, *N*1.

2. B responds with a message that is encrypted using the shared master key. The response includes the session key selected by B, an identifier of B, the value f(*N*1), and another nonce, *N*2.

3. Using the new session key, A returns f(*N*2) to B.

Thus, although each node must maintain at most (*n* - 1) master keys, as many session keys as required may be generated and used. Because the messages transferred using the master key are short, cryptanalysis is difficult. As before, session keys are used for only a limited time to protect them.

Symmetric Key Distribution Using Asymmetric Encryption

Simple Secret Key Distribution

An extremely simple scheme was put forward by Merkle [MERK79], as illustrated in Figure 14.7. If A wishes to communicate with B, the following procedure is employed:

1. A generates a public/private key pair {*PUa*, *PRa*} and transmits a message to B consisting of *PUa* and an identifier of *A*, *IDA*.

2. B generates a secret key, *Ks*, and transmits it to A, which is encrypted with A’s public key.

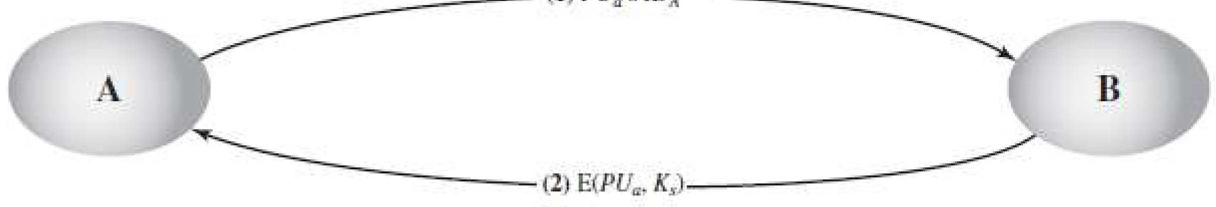


Figure 14.7 Simple Use of Public-Key Encryption to Establish a Session Key

3. A computes D(*PRa*, E(*PUa*, *Ks*)) to recover the secret key. Because only A can decrypt the message, only A and B will know the identity of *Ks*.

4. A discards *PUa* and *PRa* and B discards *PUa*. A and B can now securely communicate using conventional encryption and the session key *Ks*. At the completion of the exchange, both A and B discard *Ks*.

The protocol depicted in Figure 14.7 is insecure against an adversary who can intercept messages and then either relay the intercepted message or substitute another message (see Figure 1.3c). Such an attack is known as a **man-in-the-middle attack** [RIVE84]. We saw this type of attack in the present case, if an adversary, D, has control of the intervening communication channel, then D can compromise the communication in the following fashion without being detected (Figure 14.8).

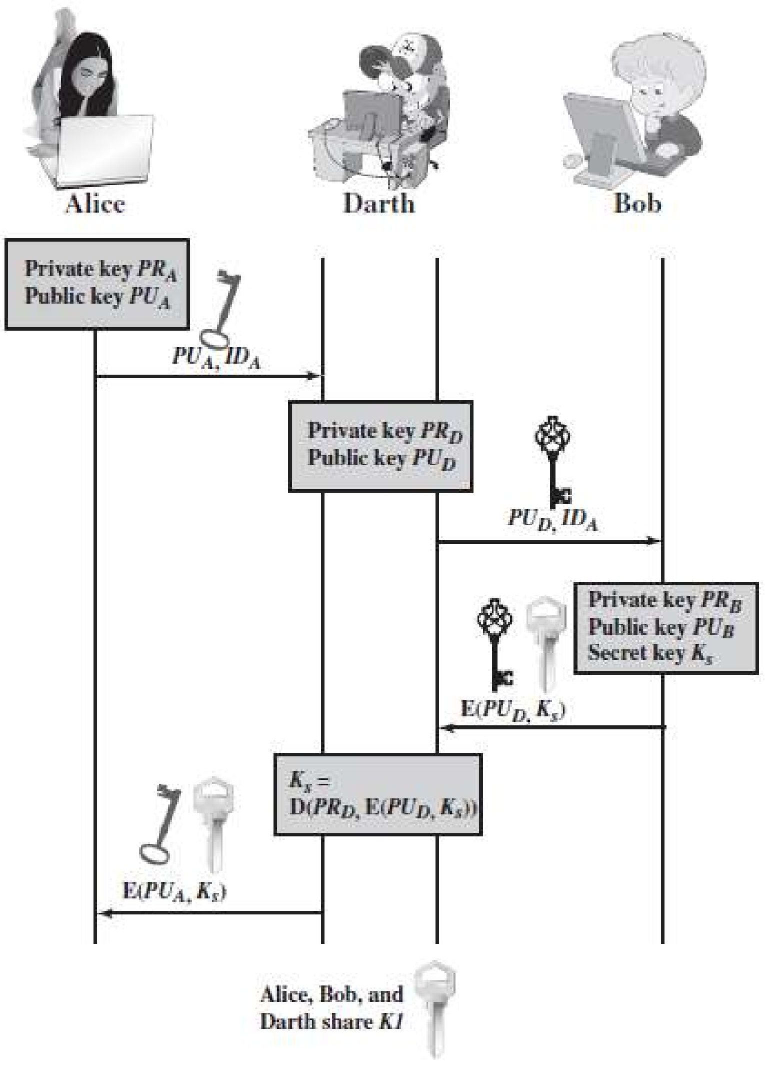


Figure 14.8 Another Man-in-the-Middle Attack

1. A generates a public/private key pair {*PUa*, *PRa*} and transmits a message intended for B consisting of *PUa* and an identifier of A, *IDA*.
2. D intercepts the message, creates its own public/private key pair {*PUd*, *PRd*} and transmits *PUs* 0 0 *IDA* to B.
3. B generates a secret key, *Ks*, and transmits E(*PUs*, *Ks*).
4. D intercepts the message and learns *Ks* by computing D(*PRd*, E(*PUd*, *Ks*)).
5. D transmits E(*PUa*, *Ks*) to A.

The result is that both A and B know *Ks* and are unaware that *Ks* has also been revealed to D. A and B can now exchange messages using *Ks*. D no longer actively interferes with the communications channel but simply eavesdrops. Knowing *Ks*, S can decrypt all messages, and both A and B are unaware of the problem. Thus, this simple protocol is only useful in an environment where the only threat is eavesdropping.

Secret Key Distribution with Confidentiality and Authentication

Figure 14.9, based on an approach suggested in [NEED78], provides protection against both active and passive attacks. We begin at a point when it is assumed that A and B have exchanged public keys by one of the schemes described subsequently in this chapter. Then the following steps occur.

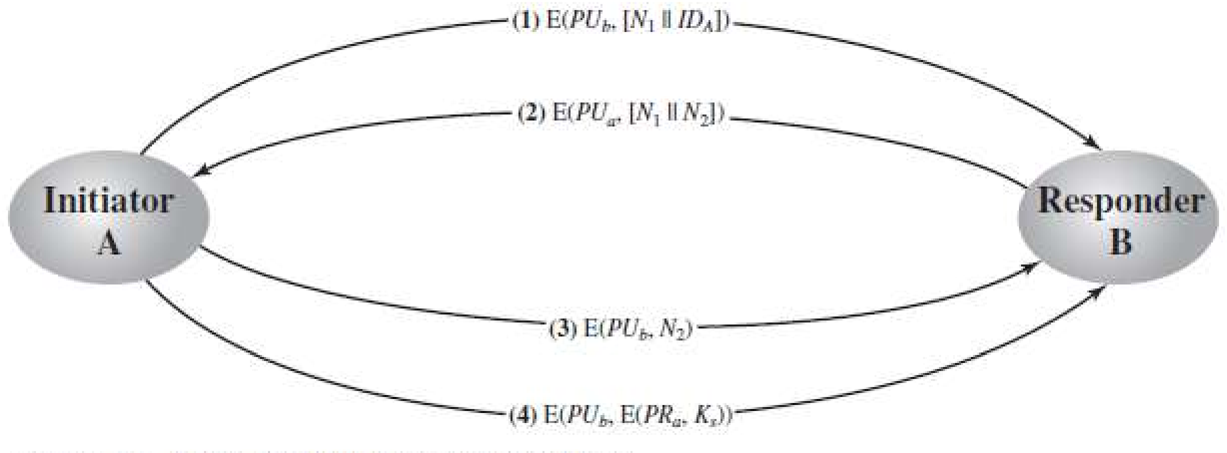

(1) $E(PU_b, [N_1 \| ID_A])$
(2) $E(PU_a, [N_1 \| N_2])$
Initiator A
Responder B
(3) $E(PU_b, N_2)$
(4) $E(PU_b, E(PR_a, K_s))$

Figure 14.9 Public-Key Distribution of Secret Keys

1.

1.A uses B's public key to encrypt a message to B containing an identifier of A(*IDA*) and a nonce (*N*1), which is used to identify this transaction uniquely.

2. B sends a message to A encrypted with *PUa* and containing A's nonce (*N*1) as well as a new nonce generated by B (*N*2). Because only B could have decrypted message (1), the presence of *N*1 in message (2) assures A that the correspondent is B.

3. A returns *N*2, encrypted using B's public key, to assure B that its correspondent is A.

4. A selects a secret key *Ks* and sends *M* = E(*PUb*, E(*PRa*, *Ks*)) to B. Encryption of this message with B's public key ensures that only B can read it; encryption with A's private key ensures that only A could have sent it.

5. B computes D(*PUa*, D(*PRb*, *M*)) to recover the secret key.

The result is that this scheme ensures both confidentiality and authentication in the exchange of a secret key.

Distribution of Public Keys:

Several techniques have been proposed for the distribution of public keys. Virtually all these proposals can be grouped into the following general schemes:

• Public announcement
• Publicly available directory
• Public-key authority
• Public-key certificates

Public Announcement of Public Keys:

A
B
PU_a
PU_a
PU_a
PU_a
PU_b
PU_b
PU_b
PU_b

Figure 14.10 Uncontrolled Public-Key Distribution

On the face of it, the point of public-key encryption is that the public key is public.Thus, if there is some broadly accepted public-key algorithm, such as RSA, any participant can send his or her public key to any other participant or broadcast the key to the community at large (Figure 14.10). it has a major weakness. Anyone can forge such a public announcement. That is, some user could pretend to be user A and send a public key to another participant or broadcast such a public key. Until such time as user A discovers the forgery and alerts other participants, the forger is able to read all encrypted messages intended for A and can use the forged keys for authentication .

Publicly Available Directory:

A greater degree of security can be achieved by maintaining a publicly available dynamic directory of public keys. Maintenance and distribution of the public directory would have to be the responsibility of some trusted entity or organization (Figure 14.11).

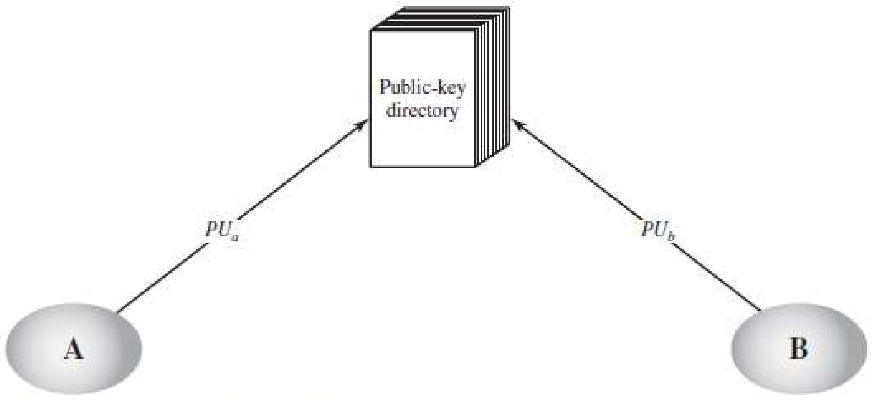


Figure 14.11 Public-Key Publication

Such a scheme would include the following elements:

1. The authority maintains a directory with a {name, public key} entry for each participant.
2. Each participant registers a public key with the directory authority. Registration would have to be in person or by some form of secure authenticated communication.
3. A participant may replace the existing key with a new one at any time, either because of the desire to replace a public key that has already been used for a large amount of data, or because the corresponding private key has been compromised in some way.
4. Participants could also access the directory electronically. For this purpose, secure, authenticated communication from the authority to the participant is mandatory.

This scheme is clearly more secure than individual public announcements but still has vulnerabilities. If an adversary succeeds in obtaining or computing the private key of the directory authority, the adversary could authoritatively pass out counterfeit public keys and subsequently impersonate any participant and eavesdrop on messages sent to any participant. Another way to achieve the same end is for the adversary to tamper with the records kept by the authority.

Public-Key Authority

Stronger security for public-key distribution can be achieved by providing tighter control over the distribution of public keys from the directory. The following steps (matched by number to Figure 14.12) occur.

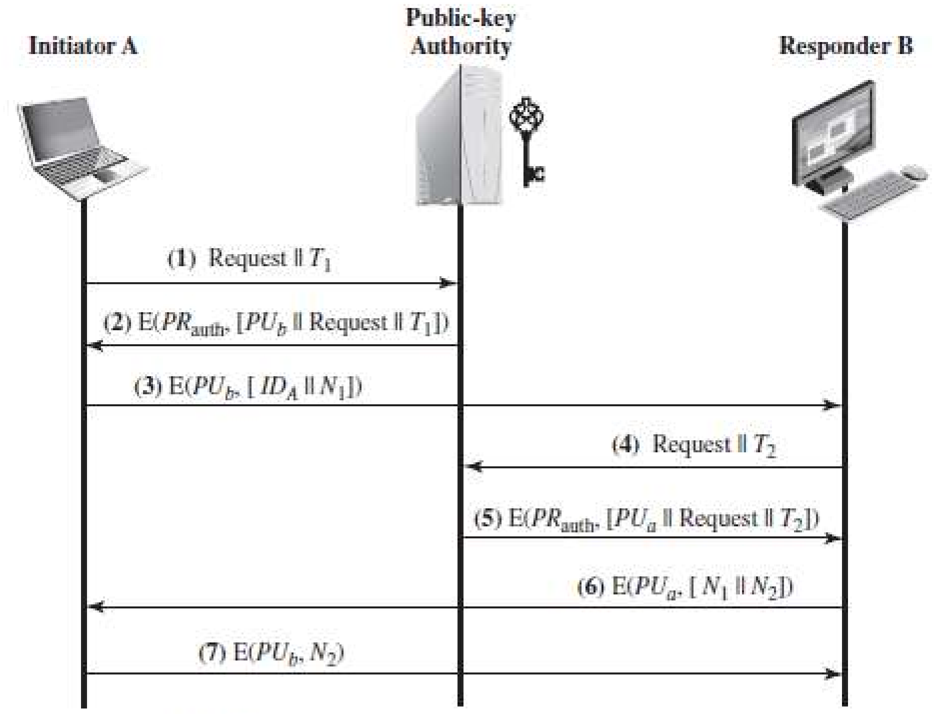

Initiator A
Public-key Authority
Responder B
(1) Request ‖ T_1
(2) E(PR_{auth}, [PU_b ‖ Request ‖ T_1])
(3) E(PU_b, [ID_A ‖ N_1])
(4) Request ‖ T_2
(5) E(PR_{auth}, [PU_a ‖ Request ‖ T_2])
(6) E(PU_a, [N_1 ‖ N_2])
(7) E(PU_b, N_2)

Figure 14.12 Public-Key Distribution Scenario

1. A sends a time stamped message to the public-key authority containing a request for the current public key of B.
2. The authority responds with a message that is encrypted using the authority's private key, *PR*auth. Thus, A is able to decrypt the message using the authority's public key. Therefore, A is assured that the message originated with the authority. The message includes the following:

- B’s public key, *PUb*, which A can use to encrypt messages destined for B
- The original request used to enable A to match this response with the corresponding earlier request and to verify that the original request was not altered before reception by the authority
- The original timestamp given so A can determine that this is not an old message from the authority containing a key other than B’s current public key

3. A stores B’s public key and also uses it to encrypt a message to B containing an identifier of A (*IDA*) and a nonce (*N*1), which is used to identify this transaction uniquely.
4, 5. B retrieves A’s public key from the authority in the same manner as A retrieved B’s public key.
At this point, public keys have been securely delivered to A and B, and they may begin their protected exchange. However, two additional steps are desirable:
6. B sends a message to A encrypted with *PUa* and containing A’s nonce (*N*1) as well as a new nonce generated by B (*N*2). Because only B could have decrypted message (3), the presence of *N*1 in message (6) assures A that the correspondent is B.
7. A returns *N*2, which is encrypted using B’s public key, to assure B that its correspondent is A.

Thus, a total of seven messages are required. However, the initial five messages need be used only infrequently because both A and B can save the other’s public key for future use—a technique known as caching. Periodically, a user should request fresh copies of the public keys of its correspondents to ensure currency.

Public-Key Certificates

The scenario of Figure 14.12 is attractive, yet it has some drawbacks. The public-key authority could be somewhat of a bottleneck in the system, for a user must appeal to the authority for a public key for every other user that it wishes to contact. As before, the directory of names and public keys maintained by the authority is vulnerable to tampering.

An alternative approach, first suggested by Kohnfelder [KOHN78], is to use **certificates** that can be used by participants to exchange keys without contacting a public-key authority, in a way that is as reliable as if the keys were obtained directly from a public-key authority. In essence, a certificate consists of a public key, an identifier of the key owner, and the whole block signed by a trusted third party.

Typically, the third party is a certificate authority, such as a government agency or a financial institution, that is trusted by the user community. A user can present his or her public key to the authority in a secure manner and obtain a certificate. The user can then publish the certificate. Anyone needing this user's public key can obtain the certificate and verify that it is valid by way of the attached trusted signature.

A participant can also convey its key information to another by transmitting its certificate. Other participants can verify that the certificate was created by the authority. We can place the following requirements on this scheme:

1. Any participant can read a certificate to determine the name and public key of the certificate's owner.
2. Any participant can verify that the certificate originated from the certificate authority and is not counterfeit.
3. Only the certificate authority can create and update certificates.
4. Any participant can verify the currency of the certificate.

A certificate scheme is illustrated in Figure 14.13. Each participant applies to the certificate authority, supplying a public key and requesting a certificate. Application must be in person or by some form of secure authenticated communication.

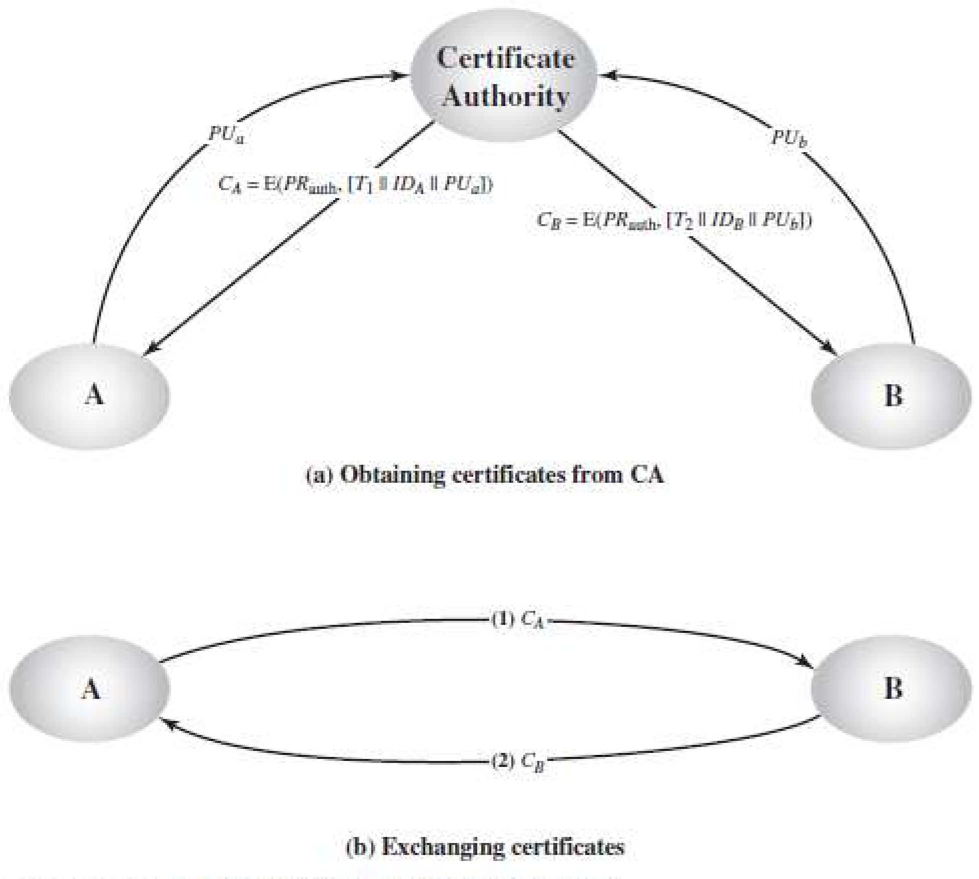

Certificate Authority
PU_a
PU_b
$C_A = E(PR_{auth}, [T_1 \| ID_A \| PU_a])$
$C_B = E(PR_{auth}, [T_2 \| ID_B \| PU_b])$
A
B
(a) Obtaining certificates from CA
(1) C_A
(2) C_B
A
B
(b) Exchanging certificates

Figure 14.13 Exchange of Public-Key Certificates

For participant A, the authority provides a certificate of the form

$$CA = E(PR\text{auth}, [T\} \; IDA \; \}PUa])$$

where *PR*auth is the private key used by the authority and *T* is a timestamp. A may then pass this certificate on to any other participant, who reads and verifies the certificate as follows:

$$D(PU\text{auth}, CA) = D(PU\text{auth}, E(PR\text{auth}, [T\} \; IDA \; \}PUa])) = (T\} \; IDA \; \}PUa)$$

- The recipient uses the authority's public key, *PU*auth, to decrypt the certificate. Because the certificate is readable only using the authority's public key, this verifies that the certificate came from the certificate authority.
- The elements *IDA* and *PUa* provide the recipient with the name and public key of the certificate's holder.
- The timestamp *T* validates the currency of the certificate.
- The timestamp counters the following scenario. A's private key is learned by an adversary. A generates a new private/public key pair and applies to the certificate authority for a new certificate. Meanwhile, the adversary replays the old certificate to B. If B then encrypts messages using the compromised old public key, the adversary can read those messages.

In this context, the compromise of a private key is comparable to the loss of a credit card. The owner cancels the credit card number but is at risk until all possible communicants are aware that the old credit card is obsolete. Thus, the timestamp serves as something like an expiration date. If a certificate is sufficiently old, it is assumed to be expired.

X.509 authentication service

X.509 Certificate

- ITU-T recommendation X.509 is part of the X.500 series of recommendations that define a directory service.
- The directory is, in effect, a server or distributed set of servers that maintains a database of information about users.
- The information includes a mapping from user name to network address, as well as other attributes and information about the users.
- X.509 defines a framework for the provision of authentication services by the X.500 directory to its users.

- The directory may serve as a repository of public-key certificates. Each certificate contains the public key of a user and is signed with the private key of a trusted certification authority.
- In addition, X.509 defines alternative authentication protocols based on the use of public-key certificates.

X.509 is based on the use of public-key cryptography and digital signatures. The standard does not dictate the use of a specific algorithm but recommends RSA. The digital signature scheme is assumed to require the use of a hash function Figure 14.14 illustrates the generation of a public-key certificate.

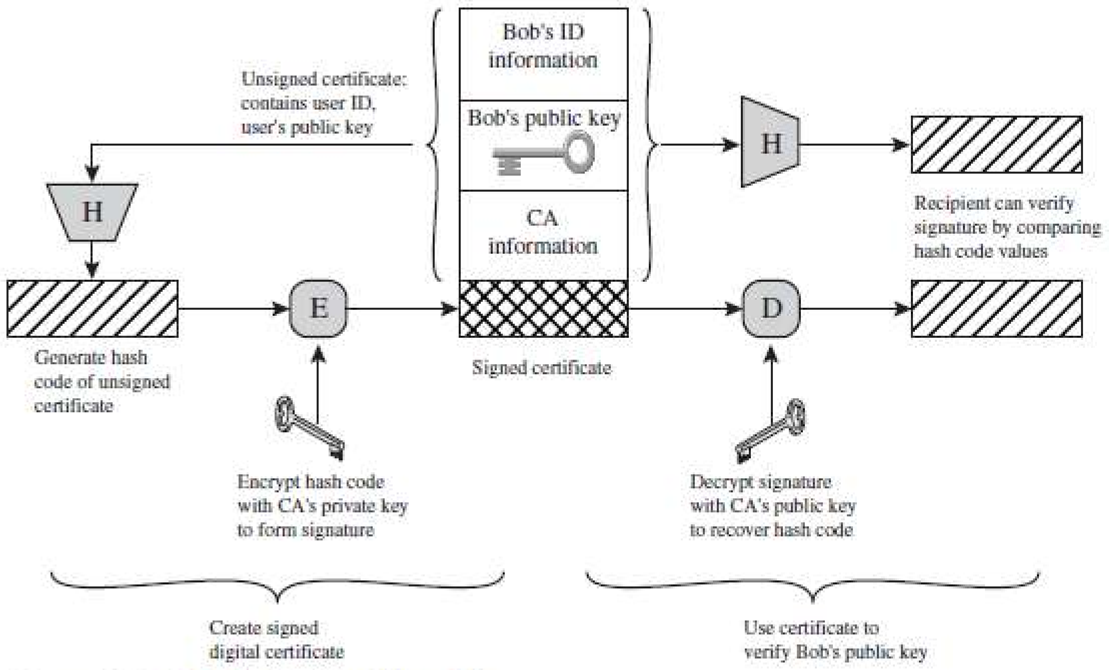

Bob's ID information
Bob's public key
CA information
Unsigned certificate: contains user ID, user's public key
H
H
Recipient can verify signature by comparing hash code values
E
D
Signed certificate
Generate hash code of unsigned certificate
Encrypt hash code with CA's private key to form signature
Decrypt signature with CA's public key to recover hash code
Create signed digital certificate
Use certificate to verify Bob's public key

Figure 14.14 Public-Key Certificate Use

Certificates

The heart of the X.509 scheme is the public-key certificate associated with each user. These user certificates are assumed to be created by some trusted certification authority (CA) and placed in the directory by the CA or by the user. The directory server itself is not responsible for the creation of public keys or for the certification function; it merely provides an easily accessible location for users to obtain certificates. Figure 14.15a shows the general format of a certificate, which includes the following elements.

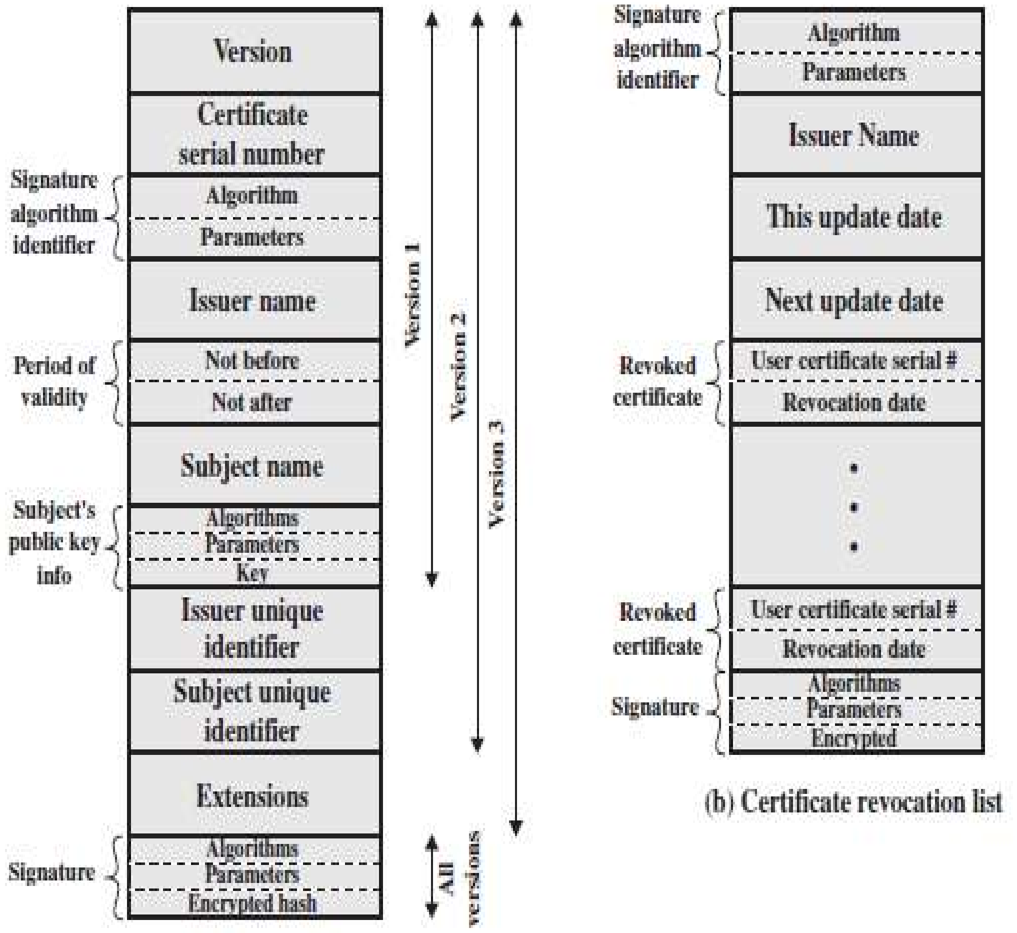


Figure 14.15 X.509 Formats

- **Version:** Differentiates among successive versions of the certificate format; the default is version 1. If the *issuer unique identifier* or *subject unique identifier* are present, the value must be version 2. If one or more extensions are present, the version must be version 3.
- **Serial number:** An integer value unique within the issuing CA that is unambiguously associated with this certificate.
- **Signature algorithm identifier:** The algorithm used to sign the certificate together with any associated parameters. Because this information is repeated in the signature field at the end of the certificate, this field has little, if any, utility.
- **Issuer name:** X.500 name of the CA that created and signed this certificate.
- **Period of validity:** Consists of two dates: the first and last on which the certificate is valid.
- **Subject name:** The name of the user to whom this certificate refers. That is, this certificate certifies the public key of the subject who holds the corresponding private key.

- **Subject’s public-key information:** The public key of the subject, plus an identifier of the algorithm for which this key is to be used, together with any associated parameters.
- **Issuer unique identifier:** An optional-bit string field used to identify uniquely the issuing CA in the event the X.500 name has been reused for different entities.
- **Subject unique identifier:** An optional-bit string field used to identify uniquely the subject in the event the X.500 name has been reused for different entities.
- **Extensions:** A set of one or more extension fields. Extensions were added in version 3 and are discussed later in this section.
- **Signature:** Covers all of the other fields of the certificate; it contains the hash code of the other fields encrypted with the CA’s private key. This field includes the signature algorithm identifier.The unique identifier fields were added in version 2 to handle the possible reuse of subject and/or issuer names over time. These fields are rarely used.

The standard uses the following notation to define a certificate:

CA V A W = CA {V, SN, AI, CA, UCA, A, UA, Ap, TA}

where

Y VXW = the certificate of user X issued by certification authority Y

Y {I} = the signing of I by Y. It consists of I with an encrypted hash code appended

V = version of the certificate

SN = serial number of the certificate

AI = identifier of the algorithm used to sign the certificate

CA = name of certificate authority

UCA = optional unique identifier of the CA

A = name of user A

UA = optional unique identifier of the user A

Ap = public key of user A

TA = period of validity of the certificate

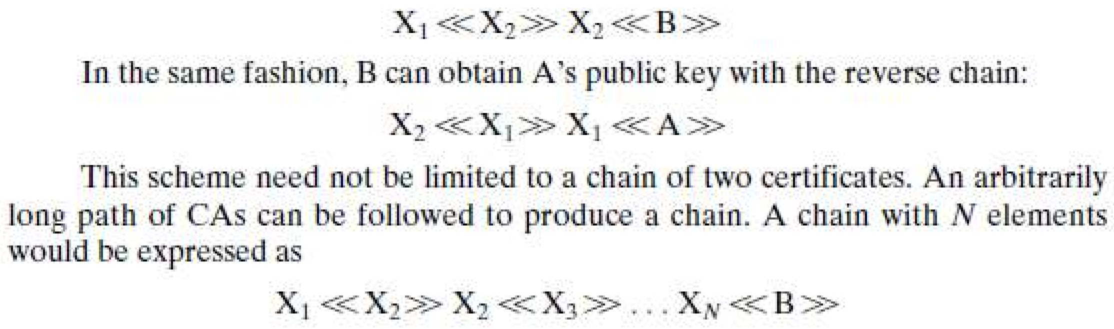
The CA signs the certificate with its private key. If the corresponding public key is known to a user, then that user can verify that a certificate signed by the CA is valid.

Obtaining a User's Certificate :

- If there is a large community of users, it may not be practical for all users to subscribe to the same CA. Because it is the CA that signs certificates, each participating user must have a copy of the CA's own public key to verify signatures.
- Thus, with many users, it may be more practical for there to be a number of CAs, each of which securely provides its public key to some fraction of the users.
- Now suppose that A has obtained a certificate from certification authority X1
- B has obtained a certificate from CA X2.
- If A does not securely know the public key of X2, then B's certificate, issued by X2, is useless to A.
- A can read B's certificate, but A cannot verify the signature.
- However, if the two CAs have securely exchanged their own public keys, the following procedure will enable A to obtain B's public key.

Step 1 A obtains from the directory the certificate of X2 signed by X1. Because A securely knows X1's public key, A can obtain X2's public key from its certificate and verify it by means of X1's signature on the certificate.
Step 2 A then goes back to the directory and obtains the certificate of B signed by X2. Because A now has a trusted copy of X2's public key, A can verify the signature and securely obtain B's public key.

A has used a chain of certificates to obtain B's public key. In the notation of X.509, this chain is expressed as



$$X_1 \ll X_2 \gg X_2 \ll B \gg$$

In the same fashion, B can obtain A's public key with the reverse chain:

$$X_2 \ll X_1 \gg X_1 \ll A \gg$$

This scheme need not be limited to a chain of two certificates. An arbitrarily long path of CAs can be followed to produce a chain. A chain with *N* elements would be expressed as

$$X_1 \ll X_2 \gg X_2 \ll X_3 \gg \dots X_N \ll B \gg$$

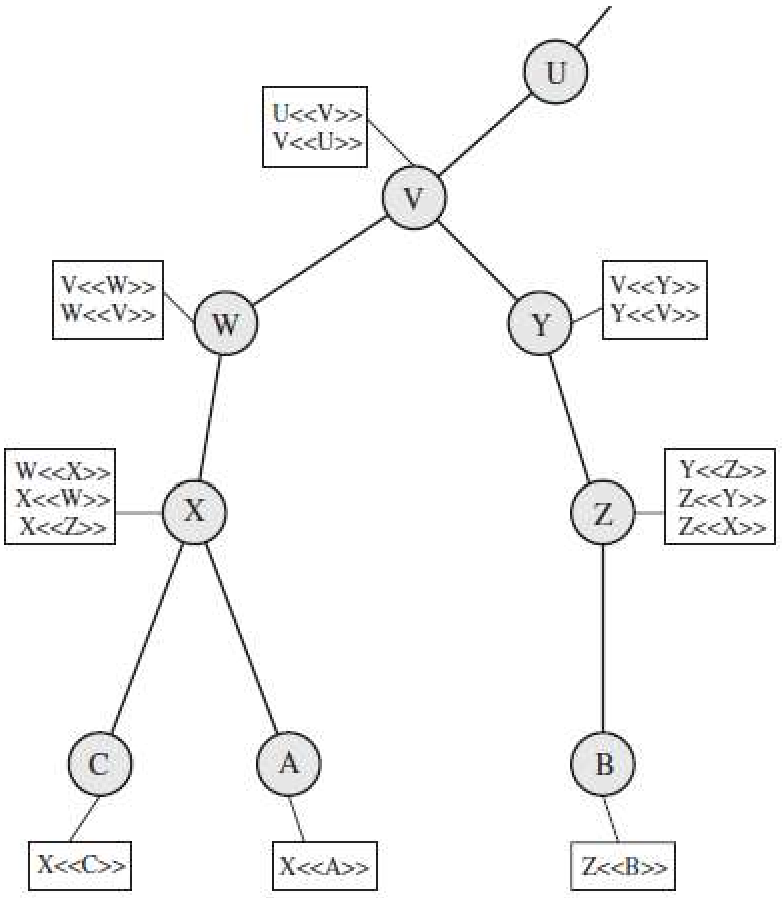


Figure 14.16 X.509 Hierarchy: A Hypothetical Example

In this example, user A can acquire the following certificates from the directory to establish a certification path to B:

$$X \ll W \gg W \ll V \gg V \ll Y \gg Y \ll Z \gg Z \ll B \gg$$

When A has obtained these certificates, it can unwrap the certification path in sequence to recover a trusted copy of B's public key. Using this public key, A can send encrypted messages to B. If A wishes to receive encrypted messages back from B, or to sign messages sent to B, then B will require A's public key, which can be obtained from the following certification path:

$$Z \ll Y \gg Y \ll V \gg V \ll W \gg W \ll X \gg X \ll A \gg$$

B can obtain this set of certificates from the directory, or A can provide them as part of its initial message to B.

Revoc ation of Certificates: Recall from Figure 14.15 that each certificate includes a period of validity, much like a credit card. Typically, a new certificate is issued just before the expiration of the old one. In addition, it may be desirable on occasion to revoke a certificate before it expires, for one of the following reasons.

1. The user's private key is assumed to be compromised.
2. The user is no longer certified by this CA. Reasons for this include that the subject's name has changed, the certificate is superseded, or the certificate was not issued in conformance with the CA's policies.
3. The CA's certificate is assumed to be compromised.

Each CA must maintain a list consisting of all revoked but not expired certificates issued by that CA, including both those issued to users and to other CAs. These lists should also be posted on the directory. Each certificate revocation list (CRL) posted to the directory is signed by the issuer and includes (Figure 14.15b) the issuer's name, the date the list was created, the date the next CRL is scheduled to be issued, and an entry for each revoked certificate. Each entry consists of the serial number of a certificate and revocation date for that certificate. Because serial numbers are unique within a CA, the serial number is sufficient to identify the certificate. When a user receives a certificate in a message, the user must determine whether the certificate has been revoked. The user could check the directory each time a certificate is received. To avoid the delays (and possible costs) associated with directory searches, it is likely that the user would maintain a local cache of certificates and lists of revoked certificates.

X.509 Version 3:

The X.509 version 2 format does not convey all of the information that recent design and implementation experience has shown to be needed. [FORD95] lists the following requirements not satisfied by version 2.

1. The subject field is inadequate to convey the identity of a key owner to a public- key user. X.509 names may be relatively short and lacking in obvious identification details that may be needed by the user.
2. The subject field is also inadequate for many applications, which typically recognize entities by an Internet e-mail address, a URL, or some other Internetrelated identification.
3. There is a need to indicate security policy information. This enables a security application or function, such as IPSec, to relate an X.509 certificate to a given policy.
4. There is a need to limit the damage that can result from a faulty or malicious CA by setting constraints on the applicability of a particular certificate.

5. It is important to be able to identify different keys used by the same owner at different times. This feature supports key lifecycle management: in particular, the ability to update key pairs for users and CAs on a regular basis or under exceptional circumstances.

The certificate extensions fall into three main categories: key and policy information, subject and issuer attributes, and certification path constraints.

Key and Policy Information These extensions convey additional information about the subject and issuer keys, plus indicators of certificate policy. A certificate policy is a named set of rules that indicates the applicability of a certificate to a particular community and/or class of application with common security requirements. For example, a policy might be applicable to the authentication of electronic data interchange (EDI) transactions for the trading of goods within a given price range.
This area includes:
• **Authority key identifier:** Identifies the public key to be used to verify the signature on this certificate or CRL. Enables distinct keys of the same CA to be differentiated. One use of this field is to handle CA key pair updating.
• **Subject key identifier:** Identifies the public key being certified. Useful for subject key pair updating. Also, a subject may have multiple key pairs and, correspondingly, different certificates for different purposes (e.g., digital signature and encryption key agreement).
• **Key usage:** Indicates a restriction imposed as to the purposes for which, and the policies under which, the certified public key may be used. May indicate one or more of the following: digital signature, nonrepudiation, key encryption, data encryption, key agreement, CA signature verification on certificates, CA signature verification on CRLs.
• **Private-key usage period:** Indicates the period of use of the private key corresponding to the public key. Typically, the private key is used over a different period from the validity of the public key. For example, with digital signature keys, the usage period for the signing private key is typically shorter than that for the verifying public key.
• **Certificate policies:** Certificates may be used in environments where multiple policies apply. This extension lists policies that the certificate is recognized as supporting, together with optional qualifier information.
• **Policy mappings:** Used only in certificates for CAs issued by other CAs. Policy mappings allow an issuing CA to indicate that one or more of that issuer's policies can be considered equivalent to another policy used in the subject CA's domain.

__**Certificate Subject and Issuer Attributes**__ : These extensions support alternative names, in alternative formats, for a certificate subject or certificate issuer and can convey additional information about the certificate subject to increase a certificate user's confidence that the certificate subject is a particular person or entity. For example, information such as postal address, position within a corporation, or picture image may be required.

The extension fields in this area include:

• **Subject alternative name:** Contains one or more alternative names, using any of a variety of forms. This field is important for supporting certain applications, such as electronic mail, EDI, and IPSec, which may employ their own name forms.
• **Issuer alternative name:** Contains one or more alternative names, using any of a variety of forms.
• **Subject directory attributes:** Conveys any desired X.500 directory attribute values for the subject of this certificate.

__**Certification Path Constraints**__: These extensions allow constraint specifications to be included in certificates issued for CAs by other CAs. The constraints may restrict the types of certificates that can be issued by the subject CA or that may occur subsequently in a certification chain.

The extension fields in this area include:
• **Basic constraints:** Indicates if the subject may act as a CA. If so, a certification path length constraint may be specified.
• **Name constraints:** Indicates a name space within which all subject names in subsequent certificates in a certification path must be located.
• **Policy constraints:** Specifies constraints that may require explicit certificate policy identification or inhibit policy mapping for the remainder of the certification path.

Kerberos

Kerberos4 is an authentication service developed as part of Project Athena at MIT. The problem that Kerberos addresses is this: Assume an open distributed environment in which users at workstations wish to access services on servers distributed throughout the network. We would like for servers to be able to restrict access to authorized users and to be able to authenticate requests for service.

In this environment, a workstation cannot be trusted to identify its users correctly to network services. In particular, the following three threats exist:

1. A user may gain access to a particular workstation and pretend to be another user operating from that workstation.
2. A user may alter the network address of a workstation so that the requests sent from the altered workstation appear to come from the impersonated workstation.
3. A user may eavesdrop on exchanges and use a replay attack to gain entrance to a server or to disrupt operations.

In any of these cases, an unauthorized user may be able to gain access to services and data that he or she is not authorized to access. Rather than building in elaborate authentication protocols at each server, Kerberos provides a centralized authentication server whose function is to authenticate users to servers and servers to users.

Kerberos relies exclusively on symmetric encryption, making no use of public-key encryption. Two versions of Kerberos are in common use. Version 4 [MILL88, STEI88] implementations still exist. Version 5 [KOHL94] corrects some of the security deficiencies of version 4 and has been issued as a proposed Internet Standard (RFC 4120) and RFC 4121).

.Motivation

If a set of users is provided with dedicated personal computers that have no network connections, then a user's resources and files can be protected by physically securing each personal computer. When these users instead are served by a centralized time-sharing system, the time-sharing operating system must provide the security. The operating system can enforce access-control policies based on user identity and use the logon procedure to identify users.

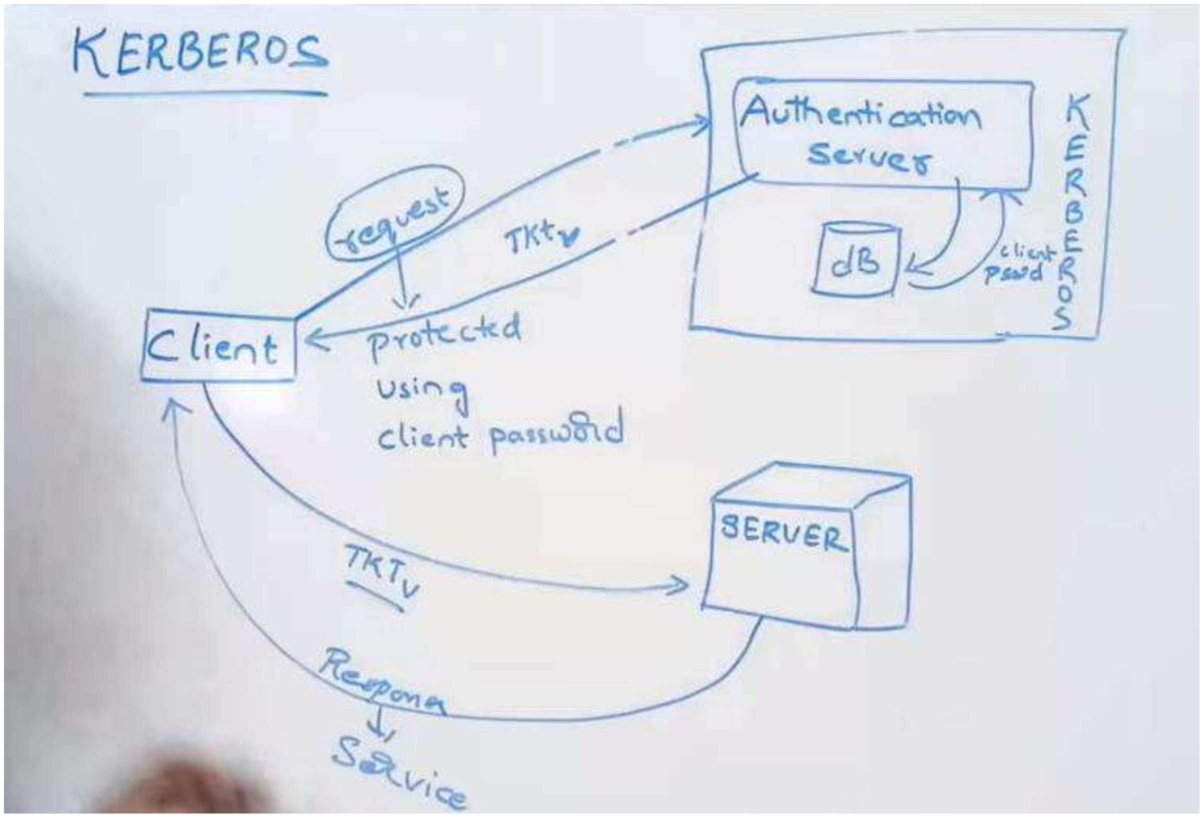
The first published report on Kerberos [STEI88] listed the following requirements.

• **Secure:** A network eavesdropper should not be able to obtain the necessary information to impersonate a user. More generally, Kerberos should be strong enough that a potential opponent does not find it to be the weak link.
• **Reliable:** For all services that rely on Kerberos for access control, lack of availability of the Kerberos service means lack of availability of the supported services. Hence, Kerberos should be highly reliable and should employ a distributed server architecture with one system able to back up another.
• **Transparent:** Ideally, the user should not be aware that authentication is taking place beyond the requirement to enter a password.
• **Scalable:** The system should be capable of supporting large numbers of clients and servers. This suggests a modular, distributed architecture. To support these requirements, the overall scheme of Kerberos is that of a trusted third-party authentication service that uses a protocol based on that proposed by Needham and Schroeder [NEED78], It is trusted in the sense that clients and servers trust Kerberos to mediate their mutual authentication. Assuming the Kerberos protocol is well designed, then the authentication service is secure if the Kerberos server itself is secure.7

Kerberos Version 4

Version 4 of Kerberos makes use of DES, in a rather elaborate protocol, to provide the authentication service. Viewing the protocol as a whole, it is difficult to see the need for the many elements contained therein. Therefore, we adopt a strategy used by Bill Bryant of Project Athena [BRYA88] and build up to the full protocol by looking first at several hypothetical dialogues. Each successive dialogue adds additional complexity to counter security vulnerabilities revealed in the preceding dialogue. After examining the protocol, we look at some other aspects of version 4.

A Simple Authentication Dialogue



Consider the following hypothetical dialogue:

$$\begin{aligned}
&\textbf{(1)}\ \mathrm{C} \rightarrow \mathrm{AS}: \quad ID_C \| P_C \| ID_V \\
&\textbf{(2)}\ \mathrm{AS} \rightarrow \mathrm{C}: \quad Ticket \\
&\textbf{(3)}\ \mathrm{C} \rightarrow \mathrm{V}: \quad ID_C \| Ticket \\
&Ticket = E(K_v, [ID_C \| AD_C \| ID_V])
\end{aligned}$$

where

C	= client
AS	= authentication server
V	= server
ID_C	= identifier of user on C
ID_V	= identifier of V
P_C	= password of user on C
AD_C	= network address of C
K_v	= secret encryption key shared by AS and V

In this scenario,

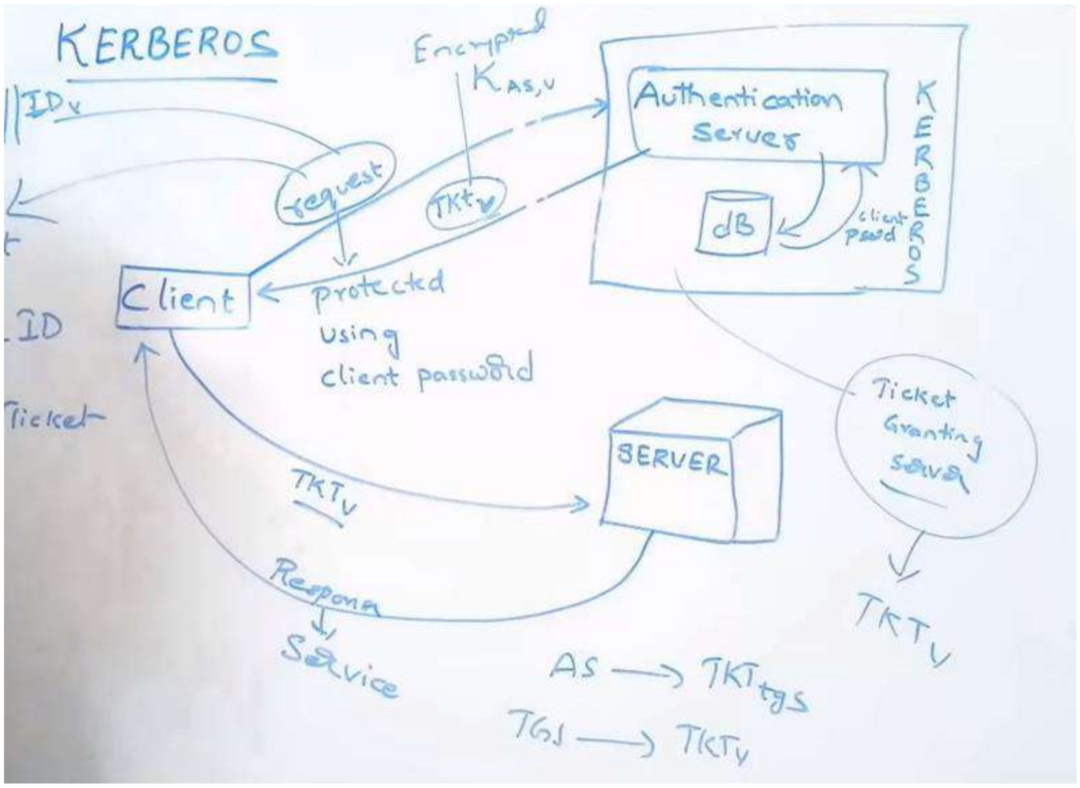
- The user logs on to a workstation and requests access to server V.

- The client module C in the user’s workstation requests the user’s password and then sends a message to the AS that includes the user’s ID, the server’s ID, and the user’s password.
- The AS checks its database to see if the user has supplied the proper password for this user ID and whether this user is permitted access to server V. If both tests are passed, the AS accepts the user as authentic and must now convince the server that this user is authentic.
- To do so, the AS creates a **ticket** that contains the user’s ID and network address and the server’s ID. This ticket is encrypted using the secret key shared by the AS and this server. This ticket is then sent back to C. Because the ticket is encrypted, it cannot be altered by C or by an opponent. With this ticket, C can now apply to V for service. C sends a message to V containing C’s ID and the ticket.
- V decrypts the ticket and verifies that the user ID in the ticket is the same as the unencrypted user ID in the message.
- If these two match, the server considers the user authenticated and grants the requested service.

A More Secure Authentication Dialogue : Although the foregoing scenario solves some of the problems of authentication in an open network environment, **problems** remain. Two in particular stand out.

First problem, we would like to minimize the number of times that a user has to enter a password. Suppose each ticket can be used only once. If user C logs on to a workstation in the morning and wishes to check his or her mail at a mail server, C must supply a password to get a ticket for the mail server. If C wishes to check the mail several times during the day, each attempt requires reentering the password. We can improve matters by saying that tickets are reusable. For a single logon session, the workstation can store the mail server ticket after it is received and use it on behalf of the user for multiple accesses to the mail server. However, under this scheme, it remains the case that a user would need a new ticket for every different service. If a user wished to access a print server, a mail server, a file server, and so on, the first instance of each access would require a new ticket and hence require the user to enter the password.

The **second problem** is that the earlier scenario involved a plaintext transmission of the password [message (1)]. An eavesdropper could capture the password and use any service accessible to the victim. To solve these additional problems, we introduce a scheme for avoiding plaintext passwords and a new server, known as the **ticket-granting server** (TGS).



The new (but still hypothetical) scenario is as follows.

Once per user logon session:

(1) $C \rightarrow AS$: $ID_C \| ID_{tgs}$

(2) $AS \rightarrow C$: $E(K_c, Ticket_{tgs})$

Once per type of service:

(3) $C \rightarrow TGS$: $ID_C \| ID_V \| Ticket_{tgs}$

(4) $TGS \rightarrow C$: $Ticket_v$

Once per service session:

(5) $C \rightarrow V$: $ID_C \| Ticket_v$

$$Ticket_{tgs} = E(K_{tgs}, [ID_C \| AD_C \| ID_{tgs} \| TS_1 \| Lifetime_1])$$
$$Ticket_v = E(K_v, [ID_C \| AD_C \| ID_v \| TS_2 \| Lifetime_2])$$

The new service, TGS, issues tickets to users who have been authenticated to AS. Thus, the user first requests a ticket-granting ticket (*Tickettgs*) from the AS.

The client module in the user workstation saves this ticket. Each time the user requires access to a new service, the client applies to the TGS, using the ticket to authenticate itself. The TGS then grants a ticket for the particular service.

The client saves each service-granting ticket and uses it to authenticate its user to a server each time a particular service is requested.

Let us look at the details of this scheme:

1. The client requests a ticket-granting ticket on behalf of the user by sending its user's ID to the AS, together with the TGS ID, indicating a request to use the TGS service.

2. The AS responds with a ticket that is encrypted with a key that is derived from the user's password (*Kc*), which is already stored at the AS. When this response arrives at the client, the client prompts the user for his or her password, generates the key, and attempts to decrypt the incoming message. If the correct password is supplied, the ticket is successfully recovered.

3. The client requests a service-granting ticket on behalf of the user. For this purpose, the client transmits a message to the TGS containing the user's ID, the ID of the desired service, and the ticket-granting ticket.

4. The TGS decrypts the incoming ticket using a key shared only by the AS and the TGS (*Ktgs*) and verifies the success of the decryption by the presence of its ID. It checks to make sure that the lifetime has not expired. Then it compares the user ID and network address with the incoming information to authenticate the user. If the user is permitted access to the server V, the TGS issues a ticket to grant access to the requested service.

5. The client requests access to a service on behalf of the user. For this purpose, the client transmits a message to the server containing the user's ID and the service granting ticket. The server authenticates by using the contents of the ticket.

This new scenario satisfies the two requirements of only one password query per user session and protection of the user password.

The Version 4 Authentication Dialogue Although the foregoing scenario enhances security compared to the first attempt, two additional problems remain. The heart of the first problem is the lifetime associated with the ticket-granting ticket. If this lifetime is very short (e.g., minutes), then the user will be repeatedly asked for a password. If the lifetime is long (e.g., hours), then an opponent has a greater opportunity for replay. An opponent could eavesdrop on the network and capture a copy of the ticket-granting ticket and then wait for the legitimate user to log out.

Then the opponent could forge the legitimate user's network address and send the message of step (3) to the TGS. This would give the opponent unlimited access to Similarly, if an opponent captures a service-granting ticket and uses it before it expires, the opponent has access to the corresponding service. Thus, we arrive at an additional requirement. A network service (the TGS or an application service) must be able to prove that the person using a ticket is the same person to whom that ticket was issued.

The second problem is that there may be a requirement for servers to authenticate themselves to users. Without such authentication, an opponent could sabotage the configuration so that messages to a server were directed to another location. The false server would then be in a position to act as a real server and capture any information from the user and deny the true service to the user.

Kerberos protocol. Figure 15.1 provides a simplified overview.

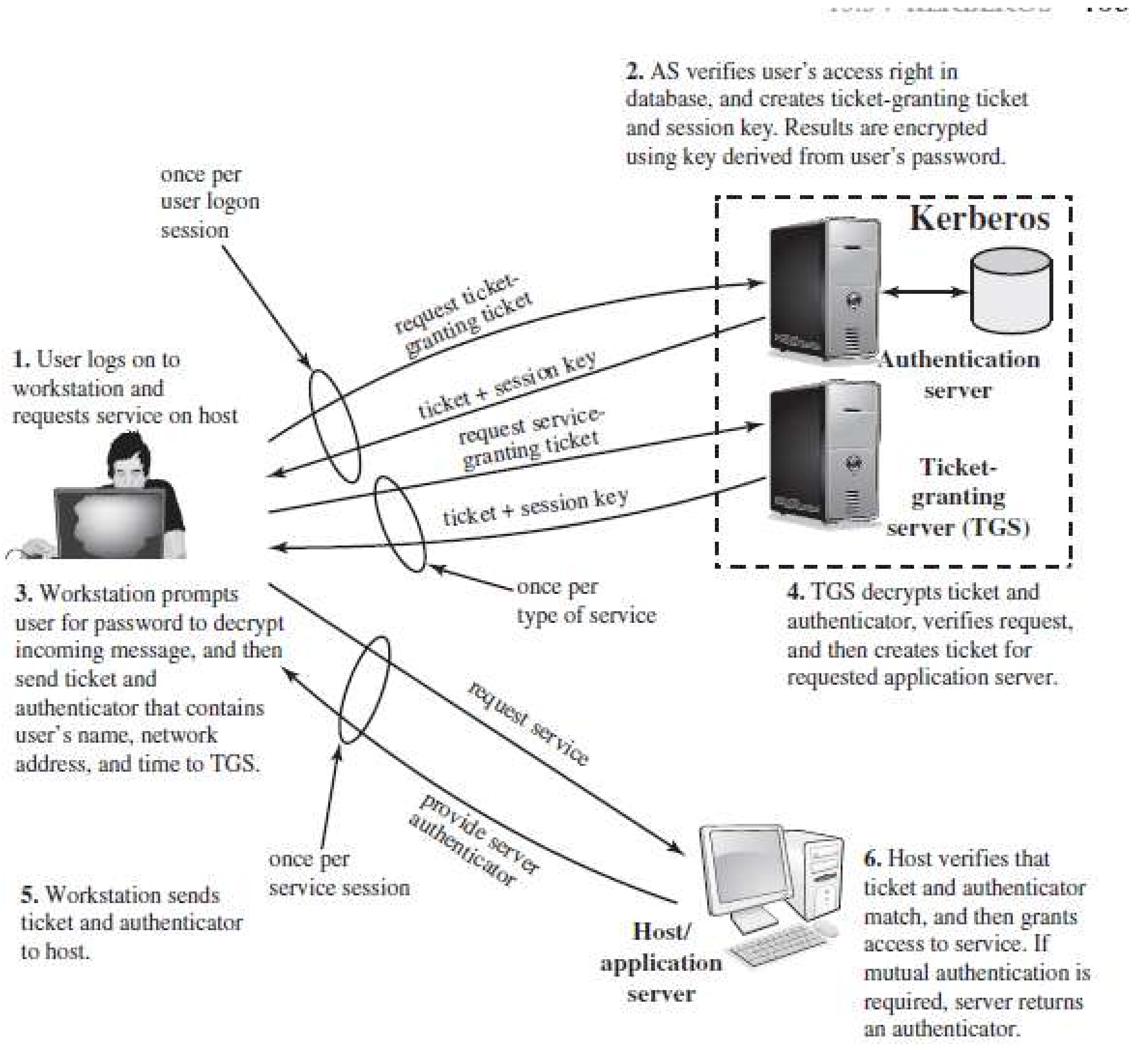

2. AS verifies user's access right in database, and creates ticket-granting ticket and session key. Results are encrypted using key derived from user's password.
once per user logon session
Kerberos
request ticket-granting ticket
ticket + session key
Authentication server
1. User logs on to workstation and requests service on host
request service-granting ticket
ticket + session key
Ticket-granting server (TGS)
once per type of service
3. Workstation prompts user for password to decrypt incoming message, and then send ticket and authenticator that contains user's name, network address, and time to TGS.
4. TGS decrypts ticket and authenticator, verifies request, and then creates ticket for requested application server.
request service
provide server authenticator
once per service session
5. Workstation sends ticket and authenticator to host.
Host/ application server
6. Host verifies that ticket and authenticator match, and then grants access to service. If mutual authentication is required, server returns an authenticator.

Table 15.1 Summary of Kerberos Version 4 Message Exchanges

(1) C → AS $ID_c \| ID_{tgs} \| TS_1$

(2) AS → C $E(K_c, [K_{c,tgs} \| ID_{tgs} \| TS_2 \| Lifetime_2 \| Ticket_{tgs}])$

$$Ticket_{tgs} = E(K_{tgs}, [K_{c,tgs} \| ID_C \| AD_C \| ID_{tgs} \| TS_2 \| Lifetime_2])$$

(a) Authentication Service Exchange to obtain ticket-granting ticket

(3) C → TGS $ID_v \| Ticket_{tgs} \| Authenticator_c$

(4) TGS → C $E(K_{c,tgs}, [K_{c,v} \| ID_v \| TS_4 \| Ticket_v])$

$$Ticket_{tgs} = E(K_{tgs}, [K_{c,tgs} \| ID_C \| AD_C \| ID_{tgs} \| TS_2 \| Lifetime_2])$$

$$Ticket_v = E(K_v, [K_{c,v} \| ID_C \| AD_C \| ID_v \| TS_4 \| Lifetime_4])$$

$$Authenticator_c = E(K_{c,tgs}, [ID_C \| AD_C \| TS_3])$$

(b) Ticket-Granting Service Exchange to obtain service-granting ticket

(5) C → V $Ticket_v \| Authenticator_c$

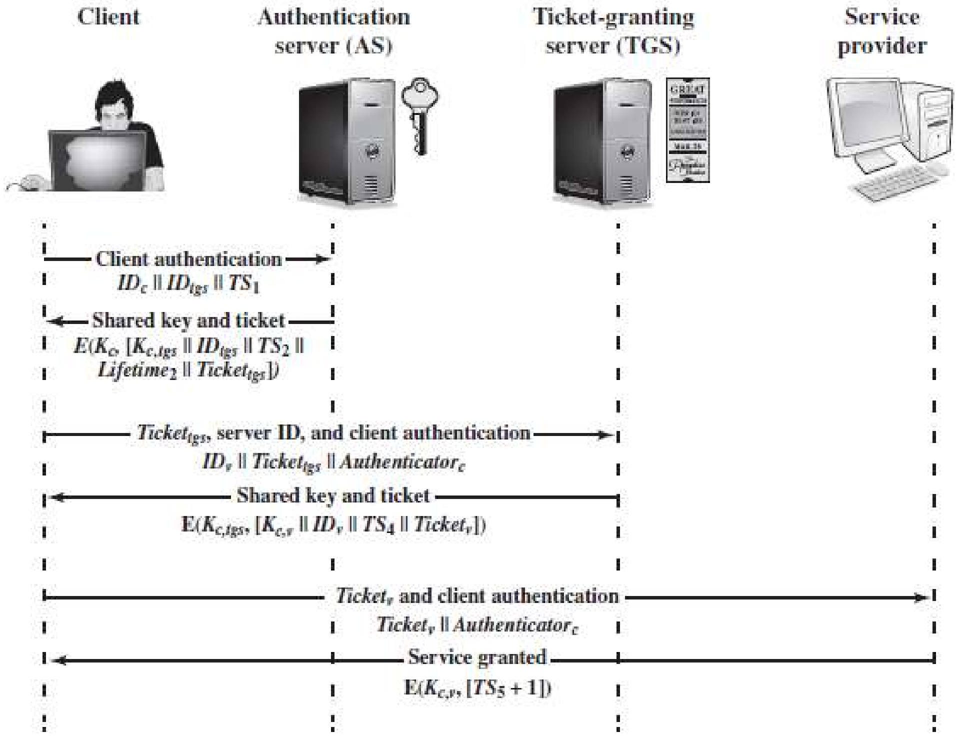
(6) V → C $E(K_{c,v}, [TS_5 + 1])$ (for mutual authentication)

$$Ticket_v = E(K_v, [K_{c,v} \| ID_C \| AD_C \| ID_v \| TS_4 \| Lifetime_4])$$

$$Authenticator_c = E(K_{c,v}, [ID_C \| AD_C \| TS_5])$$

(c) Client/Server Authentication Exchange to obtain service

Figure 15.2 illustrates the Kerberos exchanges among the parties. Table 15.2 summarizes the justification for each of the elements in the Kerberos protocol.



Client
Authentication server (AS)
Ticket-granting server (TGS)
Service provider
Client authentication
$ID_c \| ID_{tgs} \| TS_1$
Shared key and ticket
$E(K_c, [K_{c,tgs} \| ID_{tgs} \| TS_2 \| Lifetime_2 \| Ticket_{tgs}])$
$Ticket_{tgs}$, server ID, and client authentication
$ID_v \| Ticket_{tgs} \| Authenticator_c$
Shared key and ticket
$E(K_{c,tgs}, [K_{c,v} \| ID_v \| TS_4 \| Ticket_v])$
$Ticket_v$ and client authentication
$Ticket_v \| Authenticator_c$
Service granted
$E(K_{c,v}, [TS_5 + 1])$

Figure 15.2 Kerberos Exchanges

Kerberos Version 5

Kerberos version 5 is specified in RFC 4120 and provides a number of improvements over version 4 [KOHL94]. To begin, we provide an overview of the changes from version 4 to version 5 and then look at the version 5 protocol.

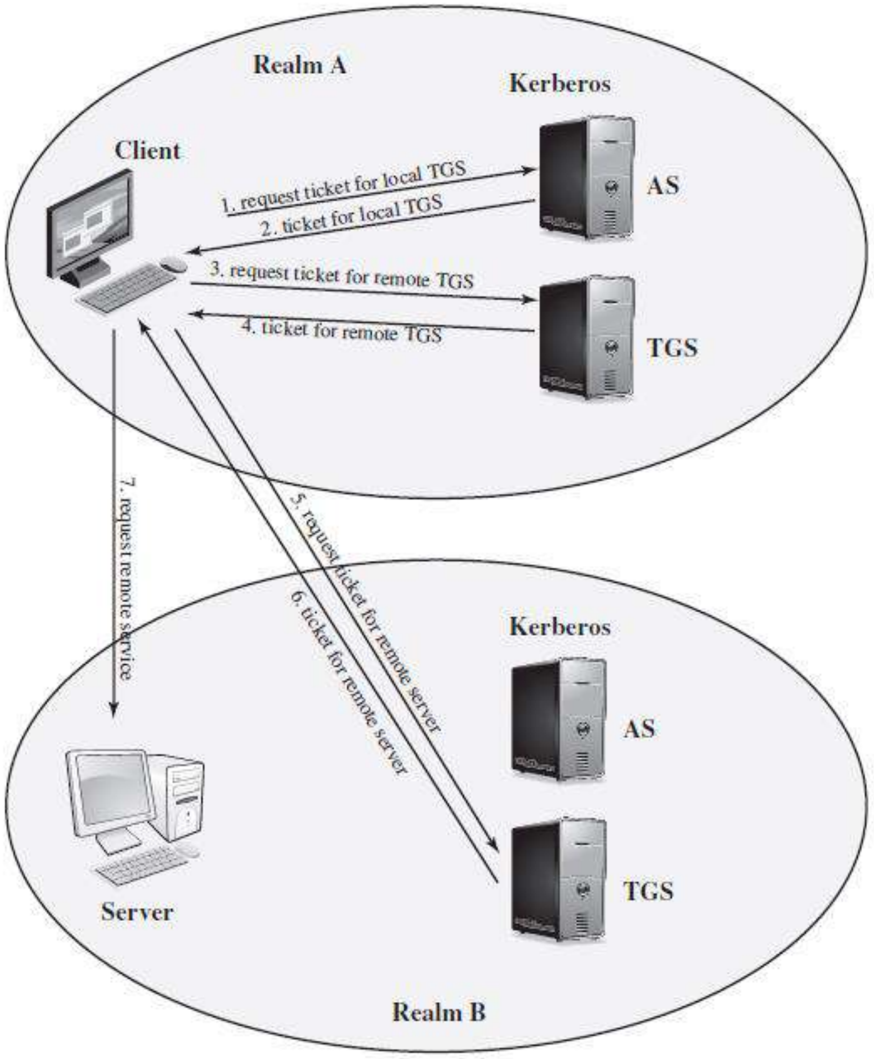
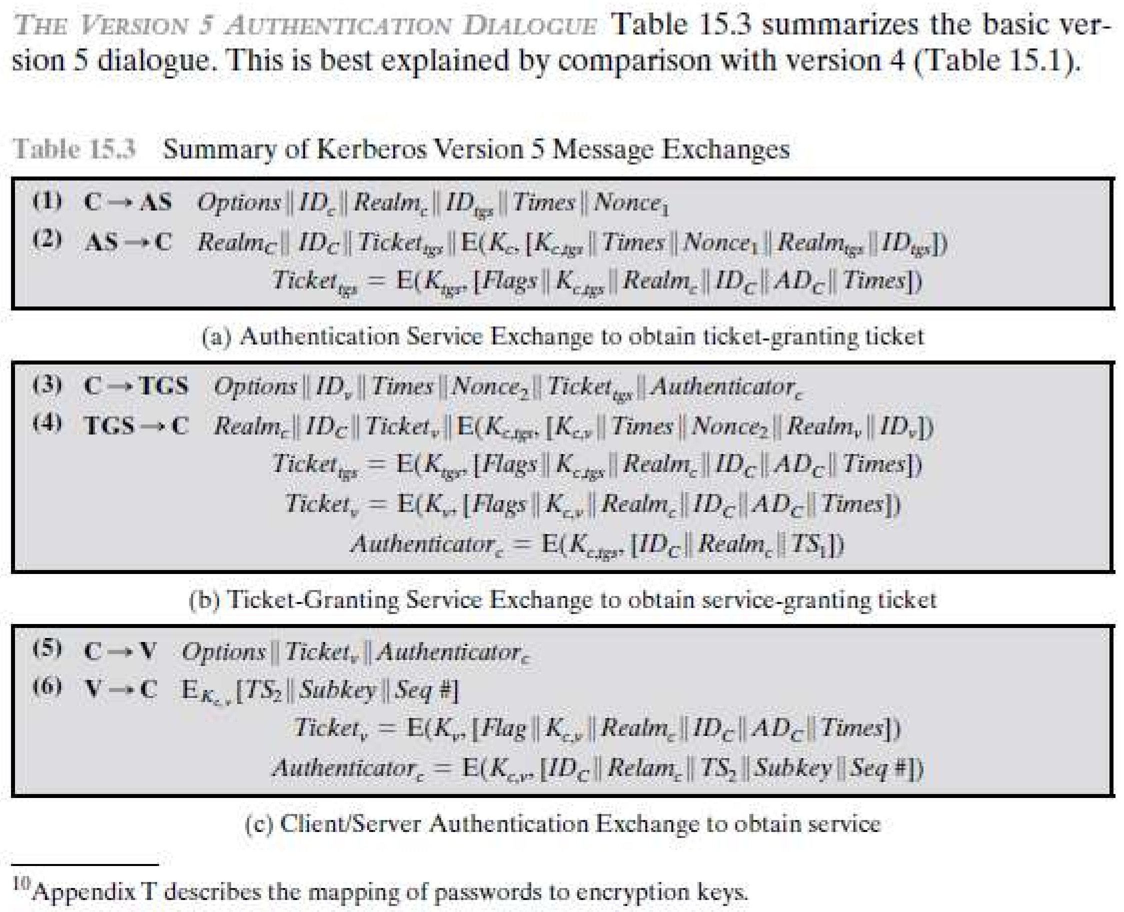

Realm A
Kerberos
Client
AS
1. request ticket for local TGS
2. ticket for local TGS
3. request ticket for remote TGS
4. ticket for remote TGS
TGS
7. request remote service
5. request ticket for remote server
6. ticket for remote server
Kerberos
AS
Server
TGS
Realm B

Figure 15.3 Request for Service in Another Realm



THE VERSION 5 AUTHENTICATION DIALOGUE Table 15.3 summarizes the basic version 5 dialogue. This is best explained by comparison with version 4 (Table 15.1).

Table 15.3 Summary of Kerberos Version 5 Message Exchanges

(1)	**C → AS**	$Options \| ID_c \| Realm_c \| ID_{tgs} \| Times \| Nonce_1$
(2)	**AS → C**	$Realm_C \| ID_C \| Ticket_{tgs} \| \mathrm{E}(K_c, [K_{c,tgs} \| Times \| Nonce_1 \| Realm_{tgs} \| ID_{tgs}])$
		$Ticket_{tgs} = \mathrm{E}(K_{tgs}, [Flags \| K_{c,tgs} \| Realm_c \| ID_C \| AD_C \| Times])$

(a) Authentication Service Exchange to obtain ticket-granting ticket

(3)	**C → TGS**	$Options \| ID_v \| Times \| Nonce_2 \| Ticket_{tgs} \| Authenticator_c$
(4)	**TGS → C**	$Realm_c \| ID_C \| Ticket_v \| \mathrm{E}(K_{c,tgs}, [K_{c,v} \| Times \| Nonce_2 \| Realm_v \| ID_v])$
		$Ticket_{tgs} = \mathrm{E}(K_{tgs}, [Flags \| K_{c,tgs} \| Realm_c \| ID_C \| AD_C \| Times])$
		$Ticket_v = \mathrm{E}(K_v, [Flags \| K_{c,v} \| Realm_c \| ID_C \| AD_C \| Times])$
		$Authenticator_c = \mathrm{E}(K_{c,tgs}, [ID_C \| Realm_c \| TS_1])$

(b) Ticket-Granting Service Exchange to obtain service-granting ticket

(5)	**C → V**	$Options \| Ticket_v \| Authenticator_c$
(6)	**V → C**	$\mathrm{E}_{K_{c,v}}[TS_2 \| Subkey \| Seq \#]$
		$Ticket_v = \mathrm{E}(K_v, [Flag \| K_{c,v} \| Realm_c \| ID_C \| AD_C \| Times])$
		$Authenticator_c = \mathrm{E}(K_{c,v}, [ID_C \| Relam_c \| TS_2 \| Subkey \| Seq \#])$

(c) Client/Server Authentication Exchange to obtain service

[10]Appendix T describes the mapping of passwords to encryption keys.

Differences between Versions 4 and 5 Version 5 is intended to address the limitations of version 4 in two areas: environmental shortcomings and technical deficiencies.Let us briefly summarize the improvements in each area.

1. **Encryption system dependence:** Version 4 requires the use of DES. Export restriction on DES as well as doubts about the strength of DES were thus of concern. In version 5, ciphertext is tagged with an encryption-type identifier so that any encryption technique may be used. Encryption keys are tagged with a type and a length, allowing the same key to be used in different algorithms and allowing the specification of different variations on a given algorithm.
2. **Internet protocol dependence:** Version 4 requires the use of Internet Protocol (IP) addresses. Other address types, such as the ISO network address, are not accommodated. Version 5 network addresses are tagged with type and length, allowing any network address type to be used.
3. **Message byte ordering:** In version 4, the sender of a message employs a byte ordering of its own choosing and tags the message to indicate least significant byte in lowest address or most significant byte in lowest address. This techniques works but does not follow established conventions. In version 5, all message structures are defined using Abstract Syntax Notation One (ASN.1) and Basic Encoding Rules (BER), which provide an unambiguous byte ordering.

4. **Ticket lifetime:** Lifetime values in version 4 are encoded in an 8-bit quantity in units of five minutes. Thus, the maximum lifetime that can be expressed is 28 * 5 = 1280 minutes (a little over 21 hours). This may be inadequate for some applications (e.g., a long-running simulation that requires valid Kerberos credentials throughout execution). In version 5, tickets include an explicit start time and end time, allowing tickets with arbitrary lifetimes.

5. **Authentication forwarding:** Version 4 does not allow credentials issued to one client to be forwarded to some other host and used by some other client. This capability would enable a client to access a server and have that server access another server on behalf of the client. For example, a client issues a request to a print server that then accesses the client's file from a file server, using the client's credentials for access. Version 5 provides this capability.

6. **Interrealm authentication:** In version 4, interoperability among *N* realms requires on the order of *N*2 Kerberos-to-Kerberos relationships, as described earlier. Version 5 supports a method that requires fewer relationships

Public-Key Infrastructure (PKI):

RFC 4949 (*Internet Security Glossary*) defines public-key infrastructure (PKI) as the set of hardware, software, people, policies, and procedures needed to create, manage, store, distribute, and revoke digital certificates based on asymmetric cryptography. The principal objective for developing a PKI is to enable secure, convenient, and efficient acquisition of public keys. The Internet Engineering Task Force (IETF) Public Key Infrastructure X.509 (PKIX) working group has been the driving force behind setting up a formal (and generic) model based on X.509 that is suitable for deploying a certificate-based architecture on the Internet. This section describes the PKIX model. Figure 14.17 shows the interrelationship among the key elements of the PKIX model.

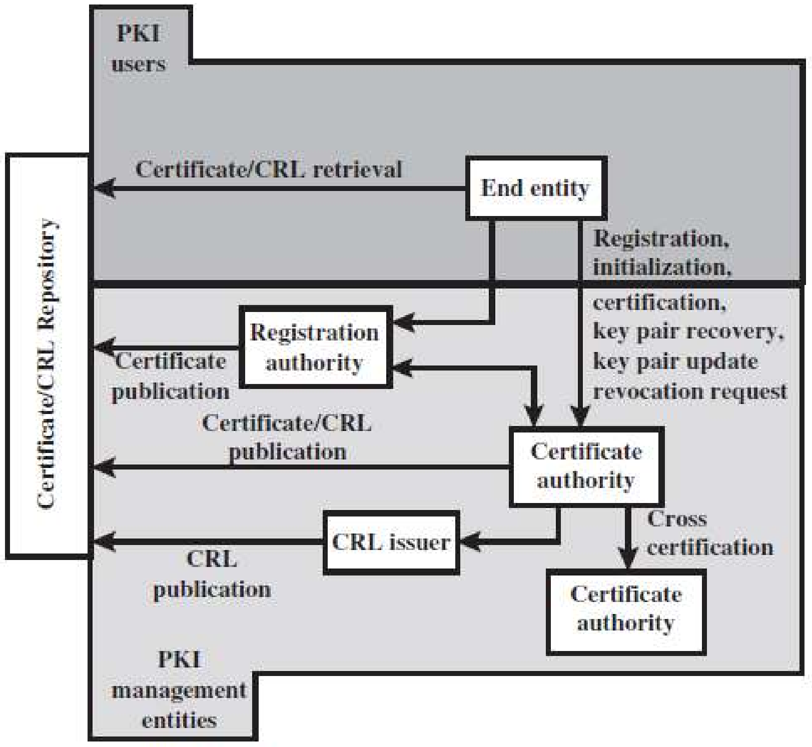

PKI users
Certificate/CRL retrieval
End entity
Registration, initialization, certification, key pair recovery, key pair update revocation request
Certificate/CRL Repository
Registration authority
Certificate publication
Certificate/CRL publication
Certificate authority
CRL issuer
CRL publication
Cross certification
Certificate authority
PKI management entities

Figure 14.17 PKIX Architectural Model

These elements are

• **End entity:** A generic term used to denote end users, devices (e.g., servers, routers), or any other entity that can be identified in the subject field of a public-key certificate. End entities typically consume and/or support PKI-related services.

• **Certification authority (CA):** The issuer of certificates and (usually) certificate revocation lists (CRLs). It may also support a variety of administrative functions, although these are often delegated to one or more Registration Authorities.

• **Registration authority (RA):** An optional component that can assume a number of administrative functions from the CA. The RA is often associated with the end entity registration process but can assist in a number of other areas as well.

• **CRL issuer:** An optional component that a CA can delegate to publish CRLs.

• **Repository:** A generic term used to denote any method for storing certificates and CRLs so that they can be retrieved by end entities.

<u>PKIX Management Functions</u>

PKIX identifies a number of management functions that potentially need to be supported by management protocols. These are indicated in Figure 14.17 and include the following:

• **Registration:** This is the process whereby a user first makes itself known to a CA (directly or through an RA), prior to that CA issuing a certificate or certificates for that user. Registration begins the process of enrolling in a PKI. Registration usually involves some offline or online procedure for mutual authentication. Typically, the end entity is issued one or more shared secret keys used for subsequent authentication.

• **Initialization:** Before a client system can operate securely, it is necessary to install key materials that have the appropriate relationship with keys stored elsewhere in the infrastructure. For example, the client needs to be securely initialized with the public key and other assured information of the trusted CA(s), to be used in validating certificate paths.

• **Certification:** This is the process in which a CA issues a certificate for a user's public key, returns that certificate to the user's client system, and/or posts that certificate in a repository.

• **Key pair recovery:** Key pairs can be used to support digital signature creation and verification, encryption and decryption, or both. When a key pair is used for encryption/decryption, it is important to provide a mechanism to recover the necessary decryption keys when normal access to the keying material is no longer possible, otherwise it will not be possible to recover the encrypted data. Loss of access to the decryption key can result from forgotten passwords/ PINs, corrupted disk drives, damage to hardware tokens, and so on. Key pair recovery allows end entities to restore their encryption/decryption key pair from an authorized key backup facility (typically, the CA that issued the end entity's certificate).

• **Key pair update:** All key pairs need to be updated regularly (i.e., replaced with a new key pair) and new certificates issued. Update is required when the certificate lifetime expires and as a result of certificate revocation.

• **Revocation request:** An authorized person advises a CA of an abnormal situation requiring certificate revocation. Reasons for revocation include private key compromise, change in affiliation, and name change.

• **Cross certification:** Two CAs exchange information used in establishing a cross-certificate. A cross-certificate is a certificate issued by one CA to another CA that contains a CA signature key used for issuing certificates.

<u>PKIX Management Protocols</u>

The PKIX working group has defines two alternative management protocols between PKIX entities that support the management functions listed in the preceding subsection. RFC 2510 defines the certificate management protocols (CMP). Within CMP, each of the management functions is explicitly identified by specific protocol exchanges. CMP is designed to be a flexible protocol able to accommodate a variety of technical, operational, and business models.

RFC 2797 defines certificate management messages over CMS (CMC), where CMS refers to RFC 2630, cryptographic message syntax. CMC is built on earlier work and is intended to leverage existing implementations. Although all of the PKIX functions are supported, the functions do not all map into specific protocol exchanges.