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 connection less 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 non receipt 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 connection less application, an individual message (e.g., datagram) could be delayed or replayed.

7. Repudiation: Denial of receipt of message by destination or denial of transmission of message by source.

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 the attacks listed under items 3 through 6.

'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. A digital signature is an authentication technique that also includes measures to counter repudiation by either source or destination.

AUTHENTICATION FUNCTIONS

Any message authentication or digital signature mechanism can he viewed as having fundamentally two levels. At the lower level, there must he some sort of function that produces an authenticator: a value to be used to authenticate a message.

This lower-level function is then used as primitive in a higher-level authentication protocol that enables a receiver to verify the authenticity of a message.

The types of functions that may be used to produce an authenticator may be grouped into three classes, as follows:

- **Message encryption**: The cipher text of the entire message serves as its authenticator
- **Message authentication code (MAC):** A public function of the message and a secret key that produces a fixed-length value that serves as the authenticator
- **Hash function (Message Digest)**: A public function that maps a message of any length into a fixed-length hash value (known as message digest), which serves as the authenticator

Message Encryption

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

Conventional Encryption

Consider the straightforward use of conventional encryption (Figure 4.1a). A message transmitted from source A to destination B is encrypted using a secret key K shared by A and B. If no other party knows the key, then confidentiality is provided: No other party can recover the plaintext of the message.

In addition, we may say that B is assured that the message came was generated by A , because A is the only other party that possesses K and therefore the only other party with the information necessary to construct cipher text that can be decrypted with K.

So we may say that conventional encryption provides authentication as well as confidentiality. However, this flat statement needs to be qualified. Consider exactly what is happening at B. Given a decryption function D and a secret key K, the destination will accept *any* input X and produce output Y = DK(X). If X is the cipher text of a legitimate message M produced by the corresponding encryption function, then Y is some plaintext message M. Otherwise, Y will be a meaningless sequence of bits. There may need to be some automated means of determining at B whether Y is legitimate plaintext and therefore must have come from A.

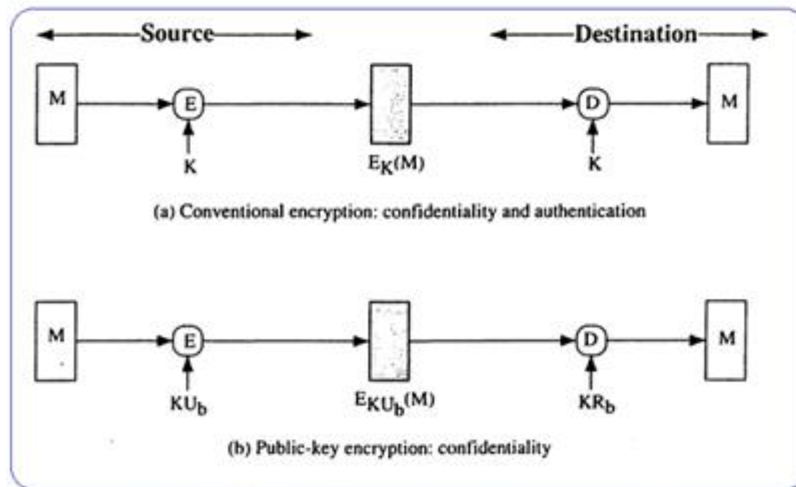

FIG : 4.1 Basic Uses of Message encryption

Public-Key Encryption

The straightforward use of public-key encryption (Figure 4.1 b) provides confidentiality but not authentication. The source (A) uses the public key KU_b of the destination (B) to encrypt M. Because only B has the corresponding private key KR_b, only B can decrypt the message. This scheme provides no authentication because any opponent could also use B's public key to encrypt a message, claiming to be A.

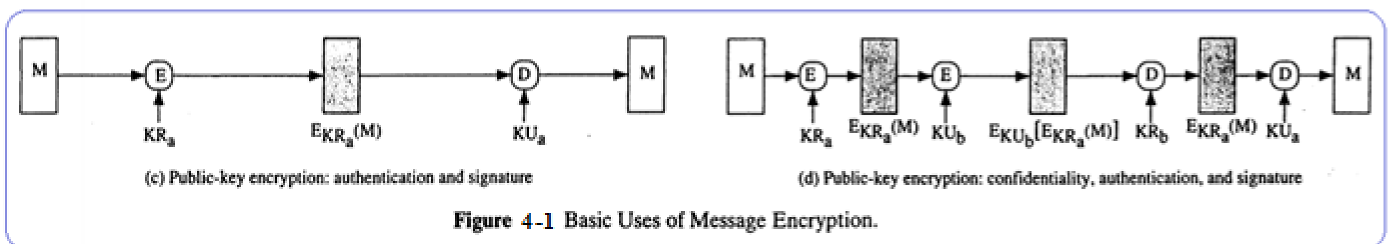

Figure 4-1 Basic Uses of Message Encryption.

To provide authentication, A uses its private key to encrypt the message, and B uses A's public key to decrypt (Figure 4.1c), This provides a measure of authentication using the same type of reasoning as in the conventional encryption case: The message must have come from A because A is the only party that possesses KRa and therefore the only party with the information necessary to construct cipher text that can be decrypted with

KUa• Again, the same reasoning as before applies: There must be some internal structure to the plaintext so that the receiver can distinguish between well-formed plaintext and random bits.

Assuming there is such structure, then the scheme of Figure 4.1 c does provide authentication. Note that this scheme does not provide confidentiality. Anyone in possession of A's public key can decrypt the ciphertext. 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 (Figure 4.1 d).

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 a cryptographic checksum or MAC, that is appended to the message.

Let us assume that the sender A wants to send a message M to a receiver B. How the MAC processing works is shown in Fig.4.2

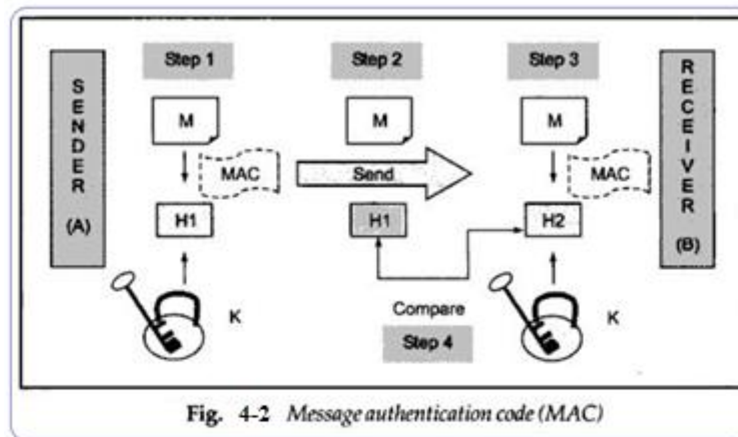

Fig. 4-2 *Message authentication code (MAC)*

I. A and B share a symmetric (secret) key K, which is not known to anyone else. A calculates the MAC by applying key K to the message M.

2. A then sends the original message M and the MAC Hl to B.

3. When B receives the message, B also uses K to calculate its own MAC H2 over M.

4. B now compares HI with H2. If the two match, B concludes that the message M has not been changed during transit. However, if HI ≠ H2, B rejects the message, realizing that the message was changed during transit.
(Note: The process just described provides authentication but not confidentiality, because the message as a whole is transmitted in the clear.) The significances of a MAC are as follows:

1. The MAC assures the receiver (in this case, B) that the message is not altered. This is because if an attacker alters the message but does not alter the MAC (in this case, H 1), then the receiver's calculation of the MAC (in this case, H2) will differ from it. Why does the attacker then not also alter the MAC? Well, as we know, the key used in the calculation of the MAC (in this case, K) is assumed to be known only to the sender and the receiver (in this case, A and B). Therefore, the attacker does not know the key, K and therefore, she cannot alter the MAC.

2. The receiver (in this case, B) is assured that the message indeed came from the correct sender (in this case, A). Since only the sender and the receiver (A and B, respectively, in this case) know the secret key (in this case. K), no one else could have calculated the MAC (in this case, HI) sent by the sender (in this case, A).

3. If the message includes a sequence number (such as is used with HDLC, X.25,and TCP), then the receiver can be assured of the proper sequence because an attacker cannot successfully alter the sequence number.

Note: Although the calculation of the MAC seems to be quite similar to an encryption process, it is actually different in one important respect. As we know, in symmetric key cryptography, the cryptographic process must be reversible. That is, the encryption and decryption are the mirror images of each other. However that in the

case of MAC, both the sender and the receiver are performing encryption process only. Thus, a MAC algorithm need not be reversible- it is sufficient to be a one-way function (encryption) only.

Requirements of MAC

In assessing the security of a MAC function, we need to consider the types of attacks that may be mounted against it. With that in mind, let us state the requirements for the function. Assume that an opponent knows the MAC function but does not know K. Then the MAC function should satisfy the following requirements.

1. If an opponent observes M and MAC (K,M) , it should be computationally infeasible for the opponent to construct a message M'such that MAC(K, M') = MAC(K, M)

2. MAC(*K*, *M*) should be uniformly distributed in the sense that for randomly chosen messages, M and M', the probability that MAC (K,M) = MAC(K,M') is 2^{-n}, where is the number of bits in the tag.

3. Let M' be equal to some known transformation on M. That is M' = f(M), . For example, f may involve inverting one or more specific bits. In that case,

$$\Pr[\mathrm{MAC}(K, M) = \mathrm{MAC}(K, M')] = 2^{-n}$$

The first requirement speaks to the earlier example, in which an opponent is able to construct a new message to match a given tag, even though the opponent does not know and does not learn the key.

The second requirement deals with the need to thwart a brute-force attack based on chosen plaintext. That is, if we assume that the opponent does not know K but does have access to the MAC function and can present messages for MAC generation, then the opponent could try various messages until finding one that matches a given tag. If the MAC function exhibits uniform distribution, then a brute-force method would require, on average $2^{(n-1)}$ attempts before finding a message that fits a given tag.

The final requirement dictates that the authentication algorithm should not be weaker with respect to certain parts or bits of the message than others. If this were not the case, then an opponent who had M and MAC (K,M) could attempt variations on M at the known "weak spots" with a likelihood of early success at producing a new message that matched the old tags.

Hash function (Message Digest):

A message digest is a *fingerprint* or the summary of a message. It is similar to the concepts of *Longitudinal Redundancy Check (LRC)* or *Cyclic Redundancy Check (CRC).* That is, it is used to verify the *integrity* of the data (i.e. to ensure that a message has not been tampered with after it leaves the sender but before it reaches the receiver).

Let us understand this with the help of an LRC example An example of LRC calculation at the sender's end is shown in Fig.4-3 . As shown, a block of bits is organized in the form of a list (as rows) in the *Longitudinal Redundancy Check (LRC).*

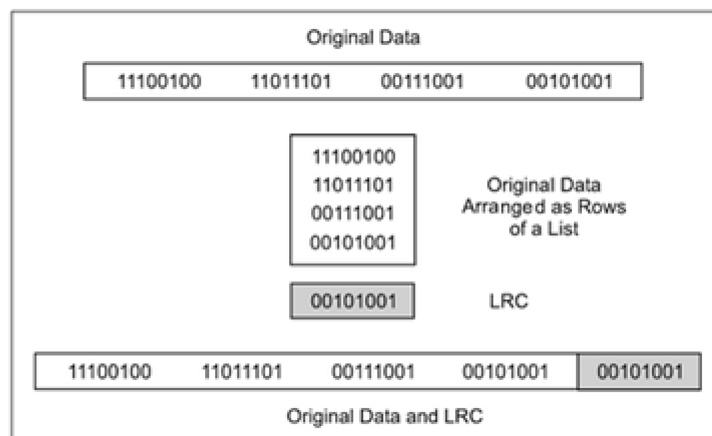

Fig 4-3 Longitudinal Redundancy Check (LRC)

Here, for instance, if we want to send 32 bits, we arrange them into a list of four (horizontal) rows. Then we count how many 1 bits occur in each of the 8 (vertical) columns. [If the number of 1s in the column is odd then we say that the column has *odd parity* (indicated by a 1 bit in the shaded LRC row); otherwise if the number of 1s in the column is even, we call it *even parity* (indicated by a 0 bit in the shaded LRC row).] For instance, in the first column, we have two 1s, indicating an even parity, and therefore, we have a 0 in the shaded LRC row for the first column. Similarly, for the last column, we have three 1s, indicating an odd parity, and therefore, we have a 1 in the shaded LRC row for the last column. Thus, the parity bit for each column is calculated and a new row of eight parity bits is created. These become the parity bits for the whole block. Thus, the LRC is actually a *fingerprint* of the original

The data along with the LRC is then sent to the receiver. The receiver separates the data block from the LRC block (shown shaded). It performs its own LRC on the data block alone. It then compares its LRC values with the ones received from the sender. If the two LRC values match then the receiver has a reasonable confidence that the message sent by the sender has not been changed, while in transit.

Idea* of a *Message Digest

The concept of message digests is based on similar principles. However, it is slightly wider in scope. For instance, suppose that we have a number 4000 and we divide it by 4 to get] 000. Thus, 4 becomes a fingerprint of the number 4000. Dividing 4000 by 4 will always yield 1000. If we change either 4000 or 4, the result will not be 1000.

Another important point is, if we are simply given the number 4, but are not given any further information; we would not be able to trace back the equation 4 x 1000 = 4000. Thus, we have one more important concept here. The fingerprint of a message (in this case, the number 4) does not tell anything about the original message (in this case, the number 4000). This is because there are infinite other possible equations, which can produce the result 4.

- **Original number is 7391743**

Operation	Result
Multiply 7 by 3	21
Discard first digit	1
Multiply 1 by 9	9
Multiply 9 by 1	9
Multiply 9 by 7	63
Discard first digit	3
Multiply 3 by 4	12
Discard first digit	2
Multiply 2 by 3	6

- **Message digest is 6**

Fig. 4-4 Simplistic example of a message digest

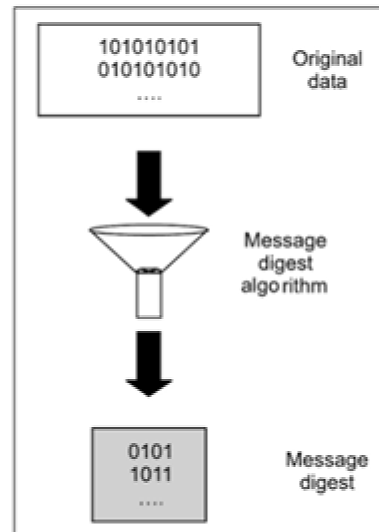

Fig. 4-5 Message-digest concept

Another simple example of a message digest is shown in Fig.4-4. Let us assume that we want to calculate the message digest of a number 7391753. Then, we multiply each digit in the number with the next digit (excluding it if it is 0), and disregarding the first digit of the multiplication operation if the result is a two digit number.

Thus, we perform a hashing operation (or a message digest algorithm) over a block of data to produce its hash or message digest, which is smaller in size than the original message. This concept is shown in Fig.4-5. So far, we are considering very simple cases of message digests. Actually, the message digests are not so small and straightforward to compute. Message digests usually consist of 128 or more bits. This means that the chance of any two message digests being the same is anything between 0 to at least 2^{128}. The message-digest length is chosen to be so long with a purpose. This ensures that the scope for two message digests is the same.

Requirements* of *a Message Digest

We can summarize the requirements of the message digest concept, as follows:

(1) Given a message, it should be very easy to find its corresponding message digest. This is shown in Fig.4-6 Also, for a given message, the message digest must always be the same.

(2) Given a message digest, it should be very difficult to find the original message for which the digest was created. This is shown in Fig.4-7

(3) Given any two messages, if we calculate their message digests. the two message digests must be different, This is shown in Fig. 4-8

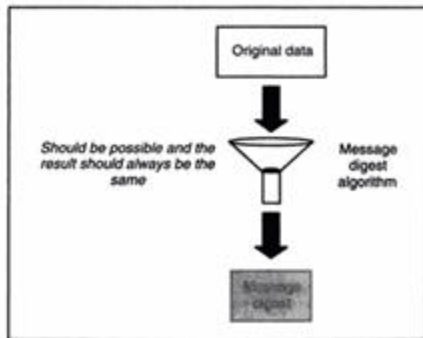

Fig. 4-6 Message digest for the same original data should always be the same

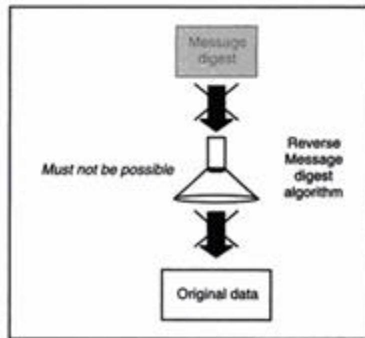

Fig. 4-7 Message digest should not work in the opposite direction

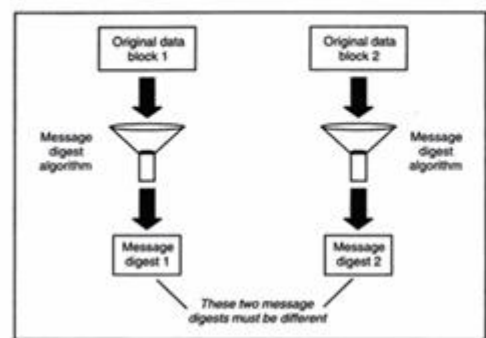

Fig. 4-8 Message digests of two different messages must be different

If any two messages produce the same message digest, thus violating our principle, it is called as a collision. That is, if two message digests *collide,* they meet at the digest! The message digest algorithms usually produce a message digest of length 128 bits or 160 bits. This means that the chances of any two message digests being the same are one in 2^{128} or 2^{160}, respectively. Clearly, this seems possible only in theory, but extremely rare in practice.

MD5

Introduction

MD5 is a message digest algorithm developed by Ron Rivest. MD5 actually has its roots in a series of message digest algorithms, which were the predecessors of MD5 all developed by Rivest. The original message digest algorithm was called as MD. MD5 is its 5th version.

MDS is quite fast, and produces 128-bit message digests. Over the years, researchers have developed potential weaknesses in MOS. However, so far, MOS has been able to successfully defend itself against collisions. After some initial processing, the input text is processed in 5l2-bit blocks (which are further divided into 16 sub-blocks each of 32 bits). The output of the algorithm is a set of four 32 bit blocks, which make up the 128-bit message digest.

How MD5 works

Step 1: Padding

The first step in MDS is to add padding bits to the original message. The aim of this step is to make the length of the original message equal LO a value, which is 64 bits less than an exact multiple of 512. For example. if the length of the original message is 1000 bits, we add a padding of 472 bits to make the length of the message 1472 bits. This is because, if we add 64 to 1472, we get 1536, which is a multiple of 5I2 (because 1536 = 5I2 x 3).

Thus, after padding, the original message will have a length of 448 bits (64 bits less than 512). 960 bits (64 bits less than 1024), 1472 bits (64 bits less than 1536), etc.

The padding consists of a single l-bit, followed by as many 0-bits, as required. Note that padding is always added. even if the message length is already 64 bits less than a multiple of 512. Thus, if the message were already of length say 448 bits, we will add a padding of 512 bits to make its length 960 bits. Thus, the padding length is any value between 1 and 512. The padding process is shown in Fig. 4.9.

Step 2: Append length

After padding bits are added, the next step is to calculate the original length of the message and add it to the end of the message, after padding. The length of the message is calculated, excluding the padding bits (i.e. it is

the length before the padding bits were added). This length of the original message is now expressed as a 64-bit value. and these 64 bits are appended to the end of the original message + padding. This is shown in Fig. 4-10.

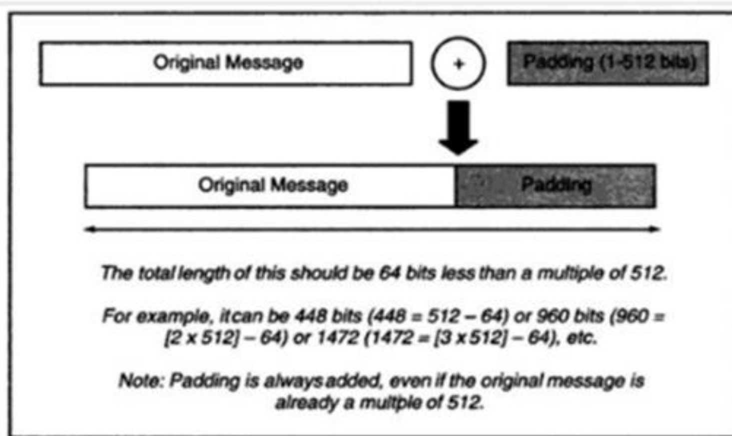

Fig. 4-9 Padding process

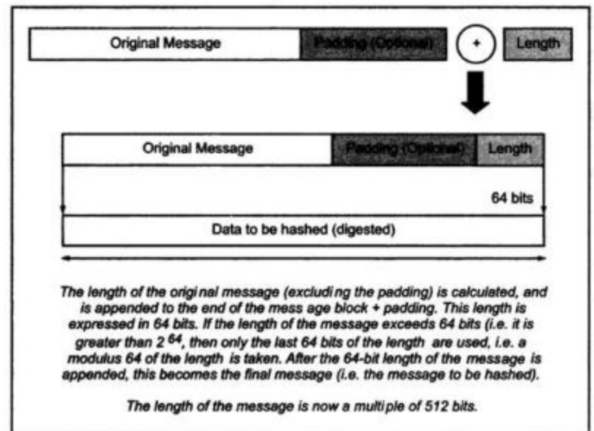

Fig. 4-10 Append length

Step 3: Divide the input into 512-bit blocks

Now, we divide the input message into blocks. each of length 512 bits. This is shown in Fig.4-11.

Step 4: Initialize chaining variables

In this step, four variables (called as chaining variables) are initialized. They are called as A. B, C and D. Each of these is a 32-bit number. The initial hexadecimal values of these chaining variables are shown in Table 4.1.

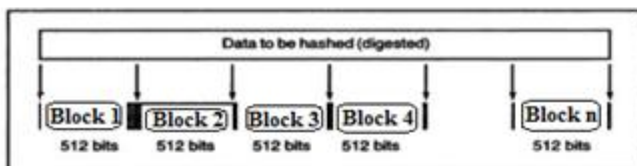

Fig. 4-11 Data is divided into 512-bit blocks

Table 4-1 *Chaining Variables*

A	Hex	01	23	45	67
B	Hex	89	AB	CD	EF
C	Hex	FE	DC	BA	98
D	Hex	76	54	32	10

Step 5: Process blocks

After all the initializations, the real algorithm begins. There is a loop that runs for as many 512-bit blocks as are in the message. Divide the current 512·bit block into 16 sub-blocks. Thus, each sub-block contains 32 bits, as shown in fig. 4-.12. Now, we have four *rounds.* In each round, we process all the 16 sub-blocks belonging to a block. The inputs to each round are: (a) all the 16 sub-blocks, (b) the variables a, b, c, d, and (c) some constants, designated as't'. This is shown in fig. 4-13

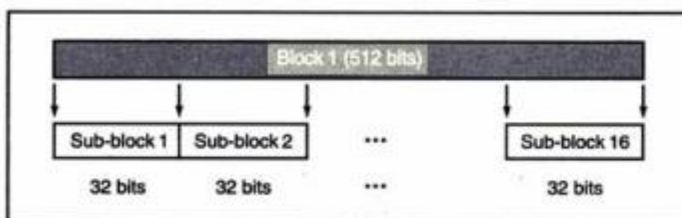

Fig. 4.: 12 Sub-blocks within a block

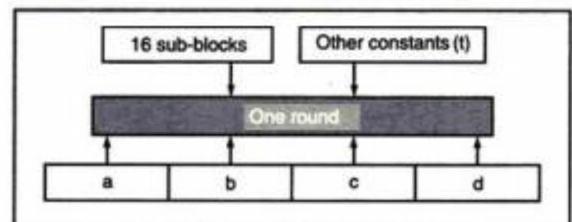

Fig. 4. 13 Conceptual process within a round

All the four rounds vary in one major way: step 1 of the four rounds has different processing. The other steps in all the four rounds are the same.

• In each round, we have 16 input sub-blocks, named M[0], M[l], ... M[151, or in general, M[i], where i varies from 0 to 15. As we know, each sub-block consists of 32 bits.

• Also, L is an array of constants. It contains 64 elements, with each element consisting of 32 bits. We denote the elements of this array L as t[1l, t[2l .... t[64], or in general as t[k], where k varies from 1 LO 64. Since there are four rounds, we use 16 out of the 64 values of t in each round.

Let us summarize these iterations of all the four rounds. In each case, the output of the intermediate as well as the final iteration is copied into the register abed. Note that we have 16 such iterations in each round.

1. A process P is first performed on b, c and d. This process P is different in all the four rounds.

2. The variable a is added to the output of the process P (i.e. to the register abcd).

3. The message sub-block M[i] is added to the output of step 2 (i.e. to the register abcd).

4. The constant t[k] is added to the output of step 3 (i.e. to the register abcd).

5. The output of step 4 (i.e. the contents of register abed) is circular-left shifted by s bits. (The value of s keeps changing.).

6. The variable b is added to the output of step 5 (i.e, to the register abcd).

7. The output of step 6 becomes the new abcd for the next step.

This is shown in Fig. 4.14.

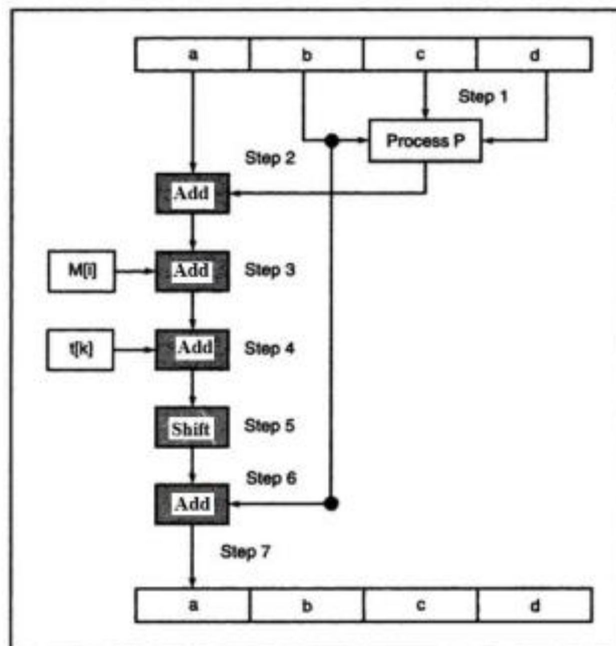

Fig. 4-14 *One MD5 operation*

We can mathematically express a single MD5 operation as follows:

a = b + ((a + Process P (b, c, d) + M[i] + T[k]) < < < s)

The *strength* of *MD5*

The attempt of Rivest was to add as much of complexity and randomness as possible to the MD5 algorithm, so that no two message digests produced by MD5 on any TWO different message are equal. MD5 has a property that every bit of the message digest is some function of every bit in the input. The possibility that two messages produce the same message digest using MD5 is in the order of 2^{64} operations. Given a message digest working backwards to find the original message can lead up to 2^{128} operations.

HMAC

Introduction

HMAC stands for **Hash-based Message Authentication Code**. HMAC has been chosen as a mandatory security implementation for the Internet Protocol (IP) security, and is also used in the Secure Socket Layer (SSL) protocol, widely used on the Internet. The fundamental idea behind HMAC is to reuse the existing message digest algorithms, such as MD5 or SHA-l. HMAC treats the message digest as a black box. Additionally it uses the shared symmetric key to encrypt the message digest, which produces the output MAC. This is shown in Fig. 4-15

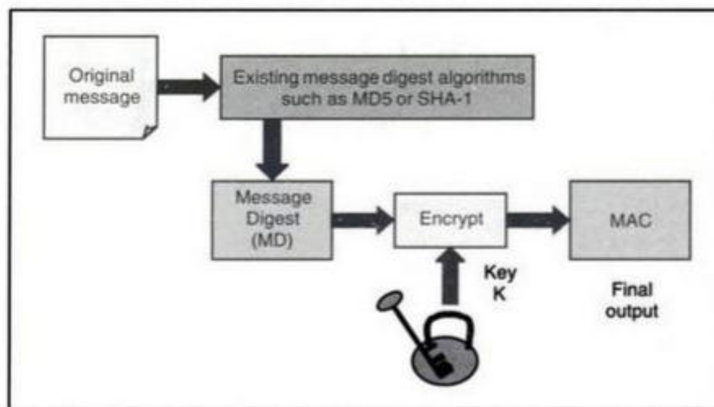

Fig. 4. 15 *HMAC Concept*

Working of HMAC

Different variables involved in the calculation of HMAC are as listed below.

MD	=	The message digest/hash function used (e.g. M05, SHA-I, etc.)
M	=	The input message whose MACis to be calculated
L	=	The number of blocks in the message M
B	=	The number of bits in each block
K	=	The shared symmetric key to be used in HMAC
Ipad	=	A string 00110110 repeated b/8 times (Hex equivalent 36)
Opad	=	A string 01011010 repeated b/8 Limes (Hex equivalent 5A)

HMAC operation.(Step-by-step approach)

Step 1: Make the length of K equal to b

The algorithm starts with three possibilities, depending on the length of the key K:

Length of K < b

In this case. we need to expand the key (K) LO make the length of K equal LO the number of bits in the original message block (i.e, b). For this, we add as many 0 bits as required to the left of K. For 'example, if the initial length of K = 170 bits, and b = 512, then we add 342 bits, all with a value 0, to the left of K. We shall continue tovcall this modified key as K.

• ***Length of K = b***

In this case, we do not take any action, and proceed to step 2.

• ***Length of K > b***

In this case, we need to trim K to make the length of K equal to the number of bits in the original message block (i.e. b). For this, we pass K through the message digest algorithm (H) selected for this particular instance of HMAC, which will give us a key K, trimmed so that its length is equal to b.

Step 2: XOR K with ipad to produce S1

We XOR K (the output of step I) and ipad to produce a variable called as S1.

Step 3: Append M to S1

We now take the original message (M) and simply append it to the end of S1 (which was calculated in step 2). This is shown in Fig. 4.16.

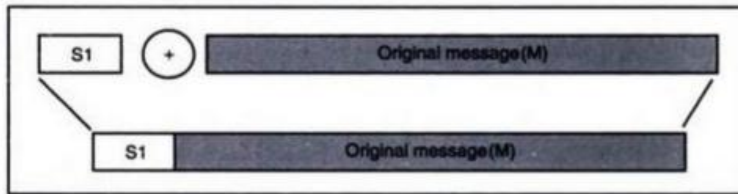

Fig. 4.16 *Step 3 of HMAC*

Step 4: Message digest algorithm

Now, the selected message digest algorithm (e.g. MD5. SHA-1, etc.) is applied to the output of step 3 (i.e. to the combination of S1 and M). Let us call the output of this operation as H.

Step 5: XOR K with opad to produce S2

Now, we XOR K (the output of step 1) with opad to produce a variable called as S2.

Step 6: Append H to S2

In this step, we take the message digest calculated in step 4 (i.e, H) and simply append it to the end of S2 (which was calculated in step 5).

Step 7: Message digest algorithm

Now. the selected message digest algorithm (e.g, MD5, SHA·1. etc.) is applied to the output or step 6 (i.e. to the concatenation of S2 and H). This is the final MAC that we want.

Let us summarize the seven steps of HMAC, as shown in Fig. 4.17

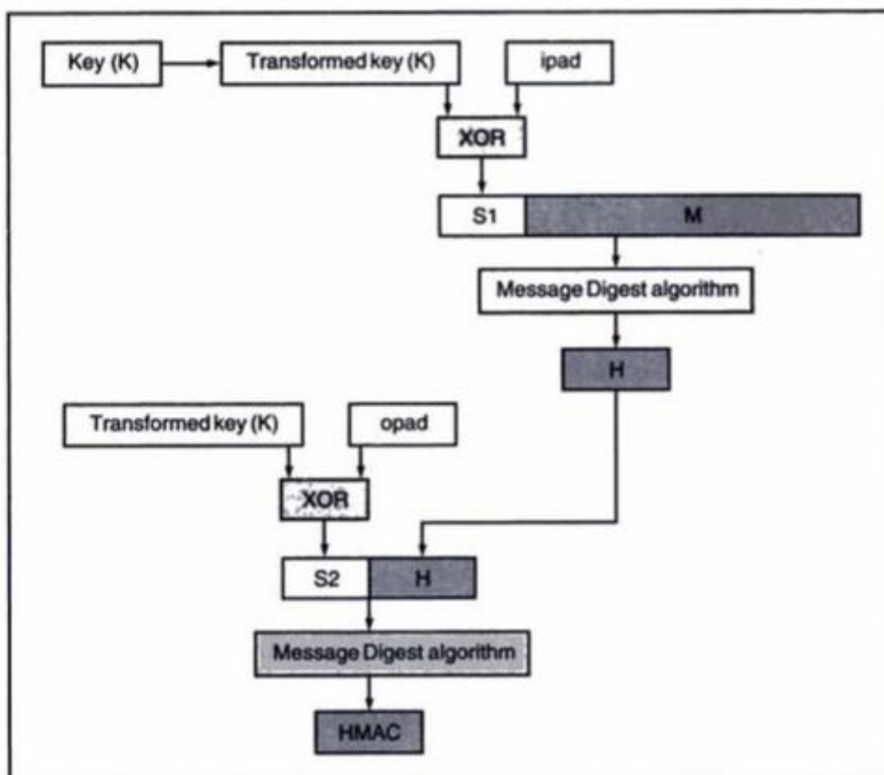

Fig. 4-17 *Complete HMAC operation*

Security of Hash functions

The security offered by an Hash function can be judged with its collision resistance capability. This can be successfully tested with two categories of attacks on hash functions: brute-force attacks and cryptanalysis.

A brute-force attack does not depend on the specific algorithm but depends only on bit length. In the case of a hash function, a brute-force attack depends only on the bit length of the hash value. A cryptanalysis, in contrast, is an attack based on weaknesses in a particular cryptographic algorithm.

We look first at brute-force attacks.

If collision resistance is required (and this is desirable for a general-purpose secure hash code), then the value $2^{m/2}$ determines the strength of the hash code against brute-force attacks. Van Oorschot and Wiener presented a design for a $10 million collision search machine for MD5, which has a 128-bit hash length that could find a collision in 24 days. Thus, a 128-bit code may be viewed as inadequate. The next step up, if a hash code is treated as a sequence of 32 bits, is a 160-bit hash length. With a hash length of 160 bits (as in the case of SHA 1 algorithm), the same search machine would require over four thousand years to find a collision.

Cryptanalysis attacks

As with encryption algorithms, cryptanalytic attacks on hash functions seek to exploit some property of the algorithm to perform some attack other than an exhaustive search. The way to measure the resistance of a hash algorithm to cryptanalysis is to compare its strength to the effort required for a brute-force attack. That is, an ideal hash algorithm will require a cryptanalytic effort greater than or equal to the brute-force effort.

Security of MAC

Just as with encryption algorithms and hash functions, we can group attacks on MACs into two categories: brute-force attacks and cryptanalysis.

Brute-Force Attacks

A brute-force attack on a MAC is a more difficult undertaking than a brute-force attack on a hash function because it requires known message-tag pairs. Let us see why this is so. To attack a hash code, we can proceed in the following way. Given a fixed message x with n-bit hash code h=H(x), a brute-force method of finding a collision is to pick a random bit string and check if H(y) = H(x). The attacker can do this repeatedly off line. Whether an off-line attack can be used on a MAC algorithm depends on the relative size of the key and the tag

There are two lines of attack possible: attack the key space and attack the MAC value. If an attacker can determine the MAC key, then it is possible to generate a valid MAC value for any input. Suppose the key size is k bits and that the attacker has one known text–tag pair. Then the attacker can compute the n-bit tag on the known text for all possible keys. At least one key is guaranteed to produce the correct tag.

This phase of the attack takes a level of effort proportional to 2^k (that is, one operation for each of the 2^k possible key values). However MAC is a many-to-one mapping, there may be other keys that produce the correct value. Thus, if more than one key is found to produce the correct value, additional text–tag pairs must be tested. It can be shown that the level of effort drops off rapidly with each additional text–MAC pair and that the overall level of effort is roughly 2^k

An attacker can also work on the tag without attempting to recover the key. Here, the objective is to generate a valid tag for a given message or to find a message that matches a given tag. In either case, the level of effort is comparable to that for attacking the one-way or weak collision-resistant property of a hash code, or 2^n .In the case of the MAC, the attack cannot be conducted off line without further input

To summarize, the level of effort for brute-force attack on a MAC algorithm can be expressed as min $(2^k, 2^n)$. The assessment of strength is similar to that for symmetric encryption algorithms. It would appear reasonable to require that the key length and tag length satisfy a relationship such as min $(k, n) \geq N$, where N is perhaps in the range of 128 bits.

Cryptanalysis

As with encryption algorithms and hash functions, cryptanalytic attacks on MAC algorithms seek to exploit some property of the algorithm to perform some attack other than an exhaustive search. The way to measure the resistance of a MAC algorithm to cryptanalysis is to compare its strength to the effort required for a brute-force attack. That is, an ideal MAC algorithm will require a cryptanalytic effort greater than or equal to the brute-force effort.