MESSAGE AUTHENTICATION
&
HASH FUNCTION

AUTHENTICATION REQUIREMENTS

- Confidentiality
  - Disclosure: Release of message contents to any person or process not processing the appropriate cryptographic key.
  - Traffic Analysis: Discovery of the pattern of traffic between two parties.

- Message Authentication
  - Masquerade: Insertion of messages into the network from a fraudulent source.
  - Content Modification
  - Sequence Modification
  - Timing Modification
- Digital Signature
  - Repudiation: Denial of receipt of message by either party.

**SUMMARY**

- Message Authentication:
  - It is a procedure to verify that received message comes from the alleged source and have not been altered.

- Digital Signature:
  - It is an authentication technique that also includes measures to counter repudiation by either source or destination.
TWO-LEVEL AUTHENTICATION

- Lower-Level:
  - There must be some sort of function that produces an authenticator – a value to be used to authenticate a message.

- Higher-Level:
  - The lower level function must be used as primitive to enable a receiver to verify the authenticity of a message.

TYPES OF FUNCTION

- Types of functions that may be used to produce authenticator may be grouped into three classes.

- Message Encryption:
  - The ciphertext 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:
  ○ A public function that maps a message of any length into a fixed-length hash value, which serves as the authenticator.

• Example: To improve authentication, one solution is to force a plaintext to have some structure that is easily recognized but that cannot be replicated without recourse to the encryption function.
(a) Symmetric encryption: confidentiality and authentication

(b) Public-key encryption: confidentiality

Figure 11.1 Basic Uses of Message Encryption
Figure 11.1 Basic Uses of Message Encryption

(c) Public-key encryption: authentication and signature

(d) Public-key encryption: confidentiality, authentication, and signature
Figure 11.2 Internal and External Error Control
An error control code is just one example; in fact, any sort of structuring added to the transmitted message serves to strengthen the authentication capability.

Such structure is provided by the use of a communications architecture consisting of layered protocols.

As an example, consider the structure of messages transmitted using the TCP/IP protocol architecture.
Figure 11.3 TCP Segment
Message Authentication Code

• Uses a secret key;
• Generates a small fixed-size block of data, known as a cryptographic checksum or MAC;
• Accepts variable size message;
• Assumes two communicating parties share a common secret key;
Figure 11.4 Basic Uses of Message Authentication Code (MAC)
Hash Function

• A variation on the MAC is the one-way hash function;
• Accepts a variable size message $M$ as input;
• Produces fixed size output, referred to as a hash code $H(M)$;
• Unlike MAC, it does not use a key but is a function of $M$
Figure 11.5 Basic Uses of Hash Function (page 1 of 2)
Figure 11.5 Basic Uses of Hash Function (page 2 of 2)
Data Authentication Algorithm

- Uses DES
Hash Function

A Hash Value

- A hash value is generated by a function $H$ of the form

$$h = H(M);$$

where $M$ is a variable-length message and $H(M)$ is the fixed-length hash value

- The hash value is appended to the message at the source at a time when the message is assumed or known to be correct

- The receiver authenticates that message by re-computing the hash value

- The hash function itself is not considered to be secret. Therefore, hash value must be protected.
REQUIREMENTS FOR A HASH FUNCTION

• The purpose of a hash function is to produce a “fingerprint” of a file, message or other block of data

• A hash function H must have the following properties:

  Adaptive requirement: H can be applied to a block of data of any size

  Length requirement: H produces a fixed-length output

  Complexity requirement: 
  H(x) is relatively easy to compute for any given x, make both hardware and software implementations practical

  One-way property requirement: 
  For any given code h, it is computationally infeasible to find x such that H(x) = h.
**Weak collision resistance requirement:**
For any given block $x$, it is computationally infeasible to find $y \neq x$ with $H(y) = H(x)$.

**Strong collision resistance requirement:**
It is computationally infeasible to find any pair $(x, y)$ such that $H(x) = H(y)$. 
Simple Hash Functions

- All hash functions operate using the following general principles:

The input is viewed as a sequence of n-bit blocks.
The input is processed one block at a time in an iterative fashion to produce an n-bit hash function.

- One of the simplest hash functions is the bit-by-bit exclusive-OR (XOR) of every block. This can be expressed as follows:

\[ C_i = b_{i1} \oplus b_{i2} \oplus \ldots \oplus b_{im} \]

Where \( C_i \) = \( i^{th} \) bit of the hash code, \( 1 \leq i \leq n \)

\( m = \) number of n-bit blocks in the input

\( b_{ij} = i^{th} \) bit in \( j^{th} \) block

\( \oplus = \) XOR operation
<table>
<thead>
<tr>
<th>block 1</th>
<th>block 2</th>
<th>block m</th>
<th>hash code</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>bit 1</strong></td>
<td><strong>bit 2</strong></td>
<td><strong>bit n</strong></td>
<td></td>
</tr>
<tr>
<td>$b_{11}$</td>
<td>$b_{21}$</td>
<td></td>
<td>$b_{n1}$</td>
</tr>
<tr>
<td>$b_{12}$</td>
<td>$b_{22}$</td>
<td></td>
<td>$b_{n2}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$b_{1m}$</td>
<td>$b_{2m}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$b_{nm}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$C_1$</td>
<td>$C_2$</td>
<td></td>
<td>$C_n$</td>
</tr>
</tbody>
</table>

**Figure 11.7 Simple Hash Function Using Bitwise XOR**
Figure 11.8 Two Simple Hash Functions

XOR with 1-bit rotation to the right

XOR of every 16-bit block
Hash Algorithms

Two Important Hash Functions

- MD5: Message Digest Algorithm
- SHA-1: Secure Hash Algorithm
Figure 12.1 Message Digest Generation Using MD5
MD5 Algorithm Description

- Input:
  - Arbitrary length message
  - Processed in 512-bit blocks

- Output:
  - 128-bit message digest

- Assume we have a b-bit message as input, and that we wish to find its message digest. Here b is an arbitrary nonnegative integer; b may be zero, it need not be a multiple of eight, and it may be arbitrarily large.

  \[ m_0 \ m_1 \ ... \ m_{b-1} \]

- The following five steps are performed to compute the message digest of the message.
**Step 1: Append Padding Bits**

- The message is "padded" (extended) so that its length (in bits) is congruent to 448, modulo 512.
- That is, the message is extended so that it is just 64 bits shy of being a multiple of 512 bits long.
- Padding is always performed, even if the length of the message is already congruent to 448, modulo 512.
- Padding is performed as follows: a single "1" bit is appended to the message, and then "0" bits are appended so that the length in bits of the padded message becomes congruent to 448, modulo 512.
- In all, at least one bit and at most 512 bits are appended.
**STEP 2: APPEND LENGTH**

- A 64-bit representation of b (the length of the message before the padding bits were added) is appended to the result of the previous step.

- In the unlikely event that b is greater than $2^{64}$, then only the low-order 64 bits of b are used. (These bits are appended as two 32-bit words and appended low-order word first in accordance with the previous conventions.)

- At this point the resulting message (after padding with bits and with b) has a length that is an exact multiple of 512 bits.

- Equivalently, this message has a length that is an exact multiple of 16 (32-bit) words.

- Let $M[0 \ldots N-1]$ denote the words of the resulting message, where $N$ is a multiple of 16. Thus $N = L \times 16$. 
**Step 3: Initialize MD Buffer**

A four-word (128 bit = 3x32-bit word) buffer (A,B,C,D) is used to hold intermediate and final results of the hash function and compute the message digest.

Here each of A, B, C, D is a 32-bit register. These registers are initialized to the following 32-bit integers (in hexadecimal values):

\[
\begin{align*}
A &= \text{67452301} \\
B &= \text{EFCDAB89} \\
C &= \text{98BADCFE} \\
D &= \text{10325476}
\end{align*}
\]

These values are stored in little-endian format, which is the least significant byte of a word in the low-address byte position:

\[
\begin{align*}
\text{word A:} &\quad 01 \quad 23 \quad 45 \quad 67 \\
\text{word B:} &\quad 89 \quad \text{AB} \quad \text{CD} \quad \text{EF} \\
\text{word C:} &\quad \text{FE} \quad \text{DC} \quad \text{BA} \quad 98 \\
\text{word D:} &\quad 76 \quad 54 \quad 32 \quad 10
\end{align*}
\]
Figure 12.2 MD5 Processing of a Single 512-bit Block
Step 4: Process Message in 512-bit (16-word) blocks

We first define four auxiliary functions that each take as input three 32-bit words and produce as output one 32-bit word.

\[ F(X, Y, Z) = (X \land Y) \lor (\neg X \land Z) \]
\[ G(X, Y, Z) = (X \land Z) \lor (Y \land \neg Z) \]
\[ H(X, Y, Z) = X \oplus Y \oplus Z \]
\[ I(X, Y, Z) = Y \oplus (X \lor \neg Z) \]

In each bit position F acts as a conditional: if \( X \) then \( Y \) else \( Z \).
This step uses a 64-element table \( T[1 \ldots 64] \) constructed from the sine function.

Let \( T[i] \) denote the \( i \)-th element of the table, which is equal to the integer part of \( 2^{32} \times \text{abs}(\sin(i)) \), where \( i \) is in radians.

That is

\[
T[i] = 4294967296 \times \text{abs}(\sin(i))
\]
Figure 12.3  Elementary MD5 Operation (single step)
Do the following:

/* Process each 16-word block. */
For i = 0 to N/16-1 do

    /* Copy block i into X. */
    For j = 0 to 15 do
        Set X[j] to M[i*16+j].
    end /* of loop on j */

/* Save A as AA, B as BB, C as CC, and D as DD. */
AA = A
BB = B
CC = C
DD = D
/* Round 1. */

/* Let [abcd k s i] denote the operation
   \[ a = b + ((a + F(b,c,d) + X[k] + T[i]) \ll s) \]. */

/* Do the following 16 operations. */

[ABCD 0 7 1]    [DABC 1 12 2]    [CDAB 2 17 3]
    [BCDA 3 22 4]
[ABCD 4 7 5]    [DABC 5 12 6]    [CDAB 6 17 7]
    [BCDA 7 22 8]
[ABCD 8 7 9]    [DABC 9 12 10]   [CDAB 10 17 11]
    [BCDA 11 22 12]
[ABCD 12 7 13]  [DABC 13 12 14]  [CDAB 14 17 15]
    [BCDA 15 22 16]
/* Round 2. */

/* Let \([abcd\; k\; s\; i]\) denote the operation
\[ a = b + ((a + G(b,c,d) + X[k] + T[i]) \ll s). */

/* Do the following 16 operations. */

<table>
<thead>
<tr>
<th>ABCD</th>
<th>1 5 17</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCD</td>
<td>0 20 20</td>
</tr>
<tr>
<td>ABCD</td>
<td>5 21</td>
</tr>
<tr>
<td>BCD</td>
<td>4 20 24</td>
</tr>
<tr>
<td>ABCD</td>
<td>9 25</td>
</tr>
<tr>
<td>BCD</td>
<td>8 20 28</td>
</tr>
<tr>
<td>ABCD</td>
<td>13 29</td>
</tr>
<tr>
<td>BCD</td>
<td>12 20 32</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DABC</th>
<th>6 9 18</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAB</td>
<td>10 9 22</td>
</tr>
<tr>
<td>DBC</td>
<td>15 14 23</td>
</tr>
<tr>
<td>DAB</td>
<td>14 9 26</td>
</tr>
<tr>
<td>DBC</td>
<td>7 14 31</td>
</tr>
<tr>
<td>DAB</td>
<td>2 9 30</td>
</tr>
</tbody>
</table>
/* Round 3. */
/* Let \( [abcd \ k \ s \ t] \) denote the operation
\[
  a = b + ((a + H(b,c,d) + X[k] + T[i]) \ll s).
\] */
/* Do the following 16 operations. */
[ABCD  5  4  33]  [DABC  8  11  34]  [CDAB 11  16  35]
  [BCDA 14  23  36]
[ABCD  1  4  37]  [DABC  4  11  38]  [CDAB  7  16  39]
  [BCDA 10  23  40]
[ABCD 13  4  41]  [DABC  0  11  42]  [CDAB  3  16  43]
  [BCDA  6  23  44]
[ABCD  9  4  45]  [DABC 12  11  46]  [CDAB 15  16  47]
  [BCDA  2  23  48]
/* Round 4. */
/* Let \([abcd k s t]\) denote the operation
\[a = b + ((a + I(b,c,d) + X[k] + T[i]) \ll s). */
/* Do the following 16 operations. */
[ABCD 0 6 49] [DABC 7 10 50] [CDAB 14 15 51]
   [BCDA 5 21 52]
[ABCD 12 6 53] [DABC 3 10 54] [CDAB 10 15 55]
   [BCDA 1 21 56]
[ABCD 8 6 57] [DABC 15 10 58] [CDAB 6 15 59]
   [BCDA 13 21 60]
[ABCD 4 6 61] [DABC 11 10 62] [CDAB 2 15 63]
   [BCDA 9 21 64]

/* Then perform the following additions. (That is increment
each of the four registers by the value it had before
this block was started.) */
A = A + AA
B = B + BB
C = C + CC
D = D + DD

end /* of loop on i */
**Step 5: Output**

The message digest produced as output is A, B, C, D. That is, we begin with the low-order byte of A, and end with the high-order byte of D.

This completes the description of MD5. A reference implementation in C is given in a separate program.