# Authentication Procedures

X.509 also includes three alternative authentication procedures that are intended for use across a variety of applications. All these procedures make use of public-key signatures. It is assumed that the two parties know each other’s public key, either by obtaining each other’s certificates from the directory or because the certificate is included in the initial message from each side. These procedures are treated as strong contrary to simple authentication procedures in which a client is authenticated to a server by sending him its identifier and password in clear or hashed together with a timestamp and nonce:





# Authentication Procedures (Cont 1)



# One-Way Authentication

One way authentication involves a single transfer of information from one user (A) to another (B), and establishes the following:

1. The identity of A and that the message was generated by A
2. That the message was intended for B
3. The integrity and originality (it has not been sent multiple times) of the message

Note that only the identity of the initiating entity is verified in this process, not that of the responding entity.

At minimum, the message includes a timestamp, , a nonce , and the identity of B and is signed with A’s private key. The timestamp consists of an optional generation time and an expiration time. This prevents delayed delivery of messages. The nonce can be used to detect replay attacks. The nonce value must be unique within the expiration time of the message. Thus, B can store the nonce

# One-Way Authentication (Cont 1)

until it expires and reject any new messages with the same nonce.

For pure authentication, the message is used simply to present credentials to B. The message may also include information to be conveyed. This information, sgnData, is included within the scope of the signature, guaranteeing its authenticity and integrity. The message may also be used to convey a session key to B, encrypted with B’s public key.



# Two-Way Authentication

In addition to the three elements just listed, two-way authentication establishes the following elements:

1. The identity of B and that the reply message was generated by B
2. That the message was intended for A
3. The integrity and originality of the reply

The reply message includes the nonce from A, to validate the reply. It also includes a timestamp and nonce generated by B. As before, the message may include signed additional information and a session key encrypted with A’s public key.

# Three-Way Authentication

Here, a final message from A to B is included, which contains a signed copy of the nonce . The intent of this design is that timestamps need not be checked: Because both nonces are echoed back by the other side, each side can check the returned nonce to detect replay attack. This approach is needed when synchronized clocks are not available. It is not shown in Figure 14.5, but the response from A also includes B to counter meet-in-the middle attack when opponent C intercepts messages between A and B

( <http://nob.cs.ucdavis.edu/classes/ecs153-1997-01/Postscript/kerbiso.ps> ).





Timestamps in the protocol above are zeros because in the three-way authentication clocks are not used.

# One-Time Password (OTP)

We follow [**L. Lamport, Password authentication with insecure communication. – Communications of ACM, 1981, v. 24, No 11, 770-772,** [**http://cmpe.emu.edu.tr/chefranov/cmpe552-06/Lecture%20Notes/Lamport81.pdf**](http://cmpe.emu.edu.tr/chefranov/cmpe552-06/Lecture%20Notes/Lamport81.pdf)].

Passwords can be compromised when they cross a network or from a server’s database. In previous considerations, we assumed that there is a trusted third party and/or password database on the server side is safe. But this may not be true, and OTP schemas give solution for such cases.

Solution is strongly based on one-way hash functions.

Lamport’s schema allows having some finite number,, of authentications of a user to a server before the initialization procedure will be required. In initialization procedure, the user and server exchange securely secret information (by means of some special channel, personally, by ordered mail, courier, or in some other secure way). Schema assumes that a password never crosses insecure network, and the server’s password database might be compromised, but can’t be changed by an intruder. The server and user use one and the same hash function, , in the authentication procedure. The server authenticates the user applying the hash function to a current “password” value in its database. Actually, this value is derived from the passport, and is used as a current password, which changes from one authentication to another. That’s why the schema is called “one-time password”. Current password depends on the authentication number and can’t exceed . OTP schemas represent “challenge-response” schemas.

# Initialization Procedure

The client selects a password, , a number, , calculates

,

where

.

The client securely delivers to the server (, and the servers saves it into () tuple.

# Authentication Procedure

When the client, C, requests authentication by the server, S, the following proceeds:

1. C -> S: C\_ID //client sends his ID

2. S -> C: Counter(C\_ID) //server responds by respective Counter value

3. C -> S: C\_ID, 

4. S: If  then {

 S authenticates C, and sets ()=()

 }

 Else C is not authenticated

After authentications,  will become equal to one, and on the next authentication, C should pass his password in clear. Hence, maximal number of authentications without sending of the secret client’s password is . The trick with the schema is that due to one-wayness of the hash function, it is not feasible to find a value such that its hash will be equal to the stored by S value. As far as this value changes from one authentication to another, knowledge of the current password does not allow using it because rate of cracking may be less than that of password update. The schema has a problem that if is large (to avoid often initialization) then C should perform on average large amount of computations (Step 3 of authentication procedure)

# MD5 Message Digest Algorithm

MD5 (<http://www.faqs.org/rfcs/rfc1321.html> ) was developed by Ron Rivest at MIT in 1991. Until 1996, when a flaw was found in it, MD5 was the most widely used secure hash algorithm. In description, we follow Stallings, Cryptography and Network Security textbook.

# MD5 Logic

The algorithm takes as input a message of arbitrary length and produces as output a 128-bit message digest. The input is processed in 512-bit blocks.

Figure 12.1 depicts the overall processing of a message to produce a digest. The processing consists of the following steps:

1. **Append padding bits**: The message is padded so that its length in bits is congruent to 448 modulo 512 (). That is, the length of the padded message is 64 bits less than an integer multiple of 512 bits. Padding is always added, even if the message is already of the desired length. For example, if the message is 448 bits long, it is padded with 512 bits to a length of 960 bits. Thus, the number of padding bits is in the range of 1 to 512. The padding consists of a single 1-bit followed by the necessary number of 0-bits.
2. **Append length**: A 64-bit representation of the length in bits of the original message (before the padding) is appended to the result of Step 1 (least significant byte first). If the original

length is greater than , then only the lower-order 64 bits of the length are used. Thus, the field contains the length of the original message, modulo .

The outcome of the first two steps yields a message that is an integer multiple of 512 bits in length. In Figure 12.1,

# MD5 Logic (Cont 1)

 the expanded message is represented as the sequence of 512-bit blocks , so that the total length of the expanded message is  bits. Equivalently, the result is a multiple of 16 32-bit words. Let M[0..N-1] denote the words of the resulting message with N an integer multiple of 16. Thus, .

1. **Initialize MD buffer**: A 128-bit buffer is used to hold intermediate and final results of the hash function. The buffer can be represented as four 32-bit registers (A,B,C,D). These registers are initialized to the following 32-bit integers (hexadecimal values):

A=67452301

B=EFCDAB89

C=98BADCFE

D=10325476

These values are stored in little-endian format, which is the least-significant byte of a word in the low-address byte position. As 32-bit strings, the initialization values (in hexadecimal format) appear as follows:

Word A: 01 23 45 67

Word B: 89 AB CD EF

Word C: FE DC BA 98

Word D: 76 54 32 10

# MD5 Logic (Cont 2)

1. **Process message in 512-bit (16-word) blocks**: The heart of the algorithm is a compression function that consists of four “rounds” of processing; this module is labeled in Figure 12.1, and its logic is illustrated in Figure 12.2. The four rounds have a similar structure, but each uses a different primitive logical function, referred to as F, G, H, and I in the specification.



Each round takes as input the current 512-bit block being processed () and the 128-bit buffer value ABCD and updates the contents of the buffer. Each round also makes use of one-fourth of a 64-element table T[1..64], constructed from the sine function. The *i-*th element of T, denoted

T[i], has the value equal to the integer part of , where *i* is in radians. Because , each element of T is an integer that can be represented in 32 bits. The table provides a “randomized “ set of 32-bit patterns, which should eliminate any regularities in the input data. Table 12.1b lists values of T.

# MD5 Logic (Cont 3)

The output of the fourth round is added to the input to the first round () to produce . The addition is done independently for each of the four words in the buffer with each of the corresponding words in , using addition modulo .

1. Output: After all L 512-bit blocks have been processed, the output from the L-th stage is the 128-bit message digest.

We can summarize the behavior of MD5 as follows:



Where

- initial value of the ABCD buffer, defined in Step 3

 - the *q*-th 512-bit block of the message

*L* – the number of blocks in the message (including padding and length fields)

 - chaining variable processed with the *q*-th block of the message

 - round function using primitive logic function *x*

 - final message digest value

 - addition modulo performed separately on each word of the pair of inputs

# MD5 Logic (Cont 4)



# MD5 Compression Function

Let’s look in more detail at the logic of the four rounds of the processing of one 512-bit block. Each round consists of a sequence of 16 steps operating on the buffer ABCD. Each step is of the form



Where

*a,b,c,d* – the four words of the buffer, in a specified order that varies across steps

*g* – one of the primitive functions F,G,H,I

*<<<s* – circular left shift (rotation) of the 32-bit argument by *s* bits

*X[k] – M[q**16+k] – k*-th 32-bit word in the *q*-th 512-bit of the message

*T[i]* – the *i*-th 32-bit word in matrix *T*

+ - addition modulo 

Figure 12.3 illustrates the step operation



The order in which the four words (*a,b,c,d*) are used produces a word-level circular right shift for each step.

One of the four primitive logical functions is used for each of the four rounds of the algorithm. Each primitive function takes three 32-bit words as input and produces a 32-bit output. Each

# MD5 Compression Function (Cont 1)

function performs a set of logical operations; that is, the *n*-th bit of the output is a function of the three inputs. The functions can be summarized as follows:

|  |  |  |
| --- | --- | --- |
| Round | Primitive function g | g(b,c,d) |
| 1 | F(b,c,d) |  |
| 2 | G(b,c,d) |  |
| 3 | H(b,c,d) |  |
| 4 | I(b,c,d) |  |

Figure 12.4 adapted from RFC 1321, defines the processing algorithm of step 4. The array of 32-bit words *X[0..15]* holds the value of the current 512-bit input block being processed. Within a round, each of the 16 words of *X[i]* is used exactly once, during one step; the order in which these words are used varies from round to round. In the first round, they are used in the original order. The following permutations are defined for rounds 2 through 4:



Consider permutation:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 1 | 6 | 11 | 0 | 5 | 10 | 15 | 4 | 9 | 14 | 3 | 8 | 13 | 2 | 7 | 12 |

Each of the 64 32-bit word elements of *T* is used exactly once, during one step of one round. Also, note that for each step, only one of the 4 bytes of the *ABCD* buffer is updated. Hence, each byte of the buffer is updated four times during the round and then a final time at the end to produce the final output of this block. Finally, note that four different circular left shift amounts are used each round and are different from round to round. The point of all this complexity is to make it very difficult to generate collisions (two 512-bit blocks that produce the same output).

# MD5 Compression Function (Cont 2)



We see that in Figure 12.4, order of *k* values in Round 2 follows specified above permutation .