Chapter 6

Authentication Schemes

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, or 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-orientated 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.
7. **Repudiation:** Denial of receipt of message by destination or denial of transmission of message by source.

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.

Any message authentication or digital signature mechanism can be viewed as having fundamentally two levels. At the lower level, there must be 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.

This section is concerned with the types of functions that may be used to produce an authenticator. These functions may be grouped into three classes, as follows:

1. **Message Encryption:** The ciphertext of the entire message serves as its authenticator.
2. **Message Authentication Code**[1] **(MAC):** A public function of the message and a secret key that produces a fixed length value that serves as the authenticator.
3. **Hash Functions:** A public function that maps a message of any length into a fixed length hash value, which serves as the authenticator.

We will mainly be concerned with the last class of function however it must be noted that hash functions and MACs are very similar except that a hash code doesn't require a secret key. With regard to the first class, this can be seen to provide authentication by virtue of the fact that only the sender and receiver know the key. Therefore the message could only have come from the sender. However there is also the problem that the plaintext message should be recognisable as plaintext message (for example if it was some sort of digitised X-rays it mightn't be).

6.1 Hash Functions

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 (also referred to as a message digest or hash code). Figures 10.1 and 10.2 shows the basic uses of the hash function whereas figure 10.3 shows the general structure of a hash code.

The hash value is appended to the message at the source at the time when the message is assumed or known to be correct. The receiver authenticates that message by recomputing the hash value. Because the hash function itself is not considered to be secret, some means is required to protect the hash value (see figures 10.1 and 10.2).

We begin by examining the requirements for a hash function to be used for message authentication. Because hash functions are, typically, quite complex, it is useful to examine next some very simple hash functions to get a feel for the issues involved. We then look at several approaches to hash function design.

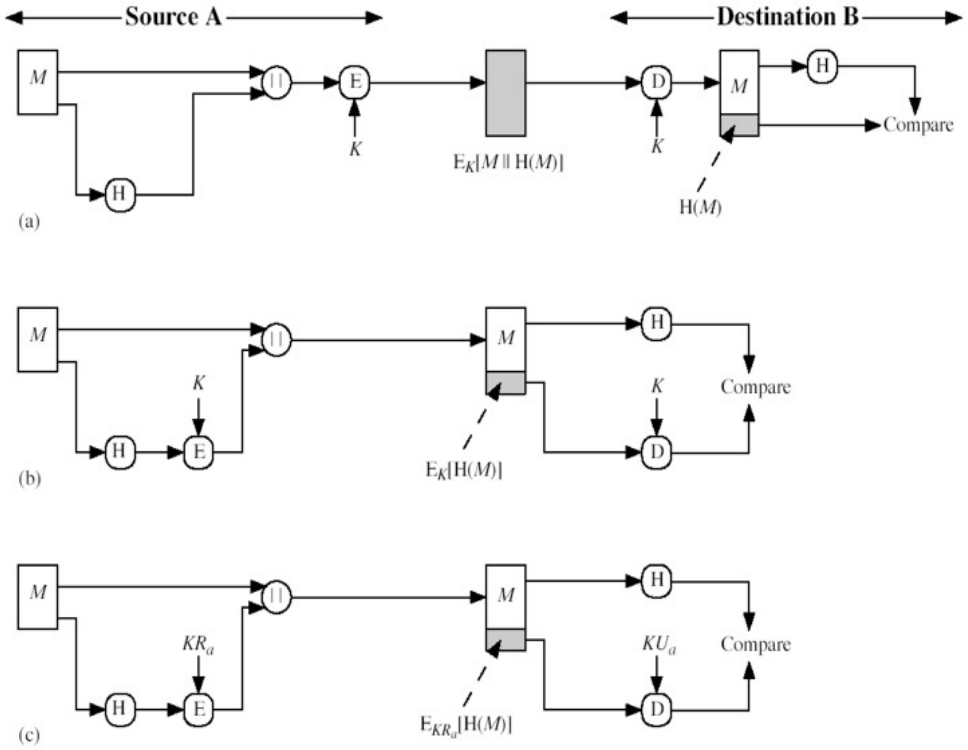


Figure 6.1: Basic uses of the hash function.

The purpose of a hash function is to produce a "fingerprint" of a file, message, or other block of data. To be useful for message authentication, a hash function H must have the following properties:

1. H can be applied to a block of data of any size.
2. H produces a fixed-length output.
3. $H(x)$ is relatively easy to compute for any given x, making both hardware and software implementations practical.
4. For any given code h, it is computationally infeasible to find x such that $H(x) = h$.
5. For any given block x, it is computationally infeasible to find $y \neq x$ with H(y) = H(x) (sometimes referred to as **weak collision property**).
6. It is computationally infeasible to find any pair (x, y) such that $H(x) = H(y)$ (sometimes referred to as **strong collision property**).

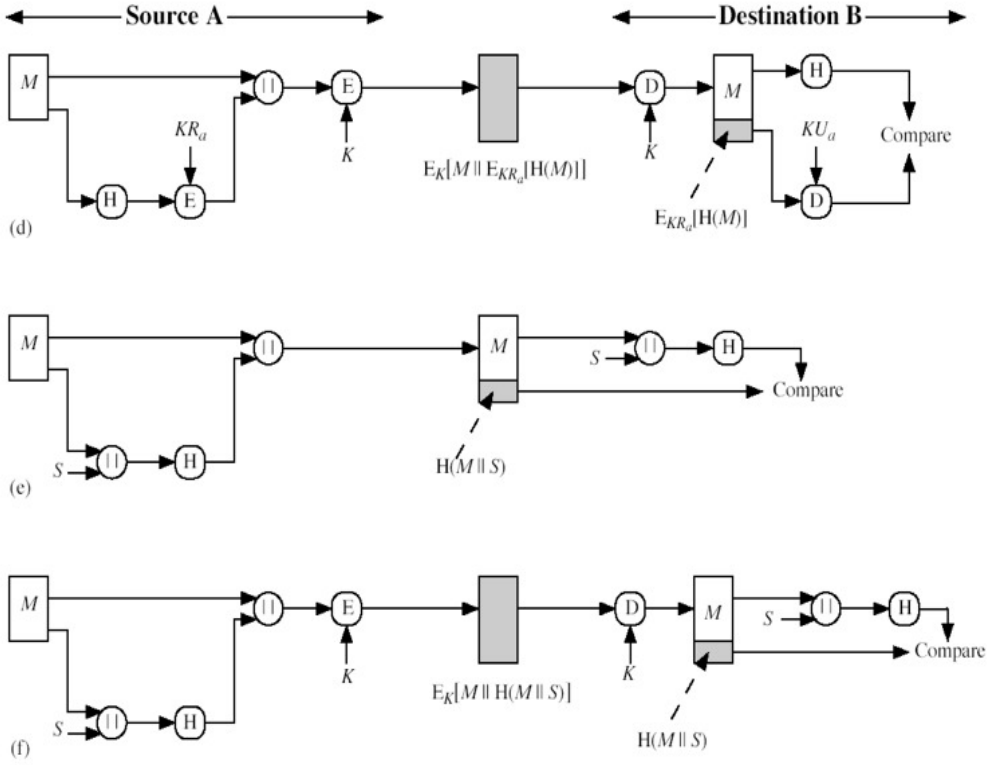


Figure 6.2: Basic uses of the hash function (cont.).

The first three properties are requirements for the practical application of a hash function to message authentication. The fourth property is the "one-way" property; it is easy to generate a code given a message but virtually impossible to generate a message given a code. This property is important if the authentication technique involves the use of a secret value (see figure 6.2e). The secret value itself is not sent; however, if the hash function is not one-way, an attacker can easily discover the secret value. If the attacker can observe or intercept a transmission, the attacker obtains the message M and the hash code $C = H(S_{AB}||M)$. The attacker then inverts the hash function to obtain $S_{AB}||M = H^{-1}(C)$. Because the attacker now has both M and $S_{AB}||M$, it is a trivial matter to recover S_{AB}.

The fifth property guarantees that an alternative message hashing to the same value as a given message cannot be found. This prevents forgery when an encrypted hash code is used (see figures 6.1b and c). For these cases, the opponent can read the message and therefore generate its hash code. But, because the opponent does not have the secret key, the opponent should not be able to alter the message without detection. If this property were not true, an attacker would be capable of the following sequence:

1. Observe or intercept a message plus its encrypted hash code.

2. Generate an unencrypted hash code from the message.

3. Generate an alternate message with the same hash code.

A hash function that satisfies the first five properties in the preceding list is referred to as a **weak hash function**. If the sixth property is also satisfied, then it is referred to as a **strong hash function**. The sixth property protects against a sophisticated class of attack known as the **birthday attack** which we will be looking at later in the notes. Figure 6.3 shows the general structure of a secure hash code. In the next section we are going to study a specific algorithm (SHA-1) which will be seen to have this format. Notice this has a similar structure to the CBC mode used for symmetric algorithms.

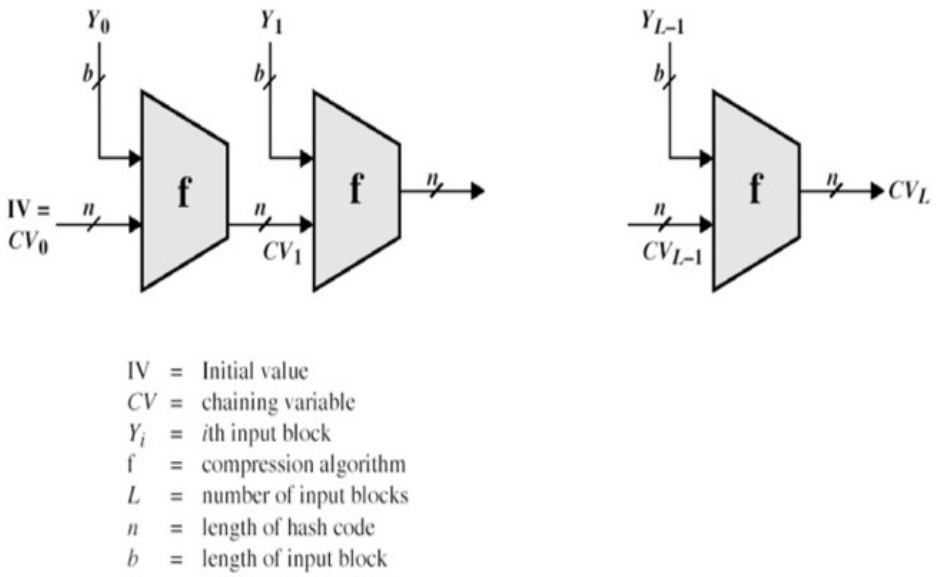


Figure 6.3: General Structure of Secure Hash Code.

6.2 The Secure Hash Algorithm

The **Secure Hash Algorithm (SHA)** was developed by the National Institute of Standards and Technology (NIST) and published as a federal information processing standard (FIPS 180) in 1993; a revised version was issued as FIPS 180-1 in 1995 and is generally referred to as **SHA-1**. The actual standards document is entitled Secure Hash Standard. SHA is based on the MD4 algorithm which is a message digest algorithm that was developed by Ron Rivest at MIT (the "R" in the RSA (Rivest-Shamir-Adelman) public key encryption algorithm). MD4 was later replaced with the popular MD5 algorithm also by Ron Rivest however advances in cryptanalysis and computing power have led to their decline in popularity. Both MD4 and MD5 produce a 128 bit

message digest whereas SHA-1 produces a 160 bit as will be seen. In 2002, NIST produced a revised version of the standard, FIPS180-2, 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. These new versions have the same underlying structure and use the same types of modular arithmetic and logical binary operations as SHA-1. In 2005, NIST announced the intention to phase out approval of SHA-1 and move to a reliance on the other SHA versions by 2010. Shortly thereafter, a research team described an attack in which two separate messages could be found that deliver the same SHA-1 hash using 2^{69} operations, far fewer that the 2^{80} operations previously thought needed to find a collision with an SHA-1 hash. This result should hasten the transition to the other versions of SHA however we will still concentrate on SHA-1 as the underlying structures of the others is the same.

SHA-1 takes as input a message with a maximum length of less than 2^{64} bits and produces as output a 160 bit message digest. The input is processed in 512-bit blocks. Figure 10.4 depicts the overall processing of a message to produce a digest. Although this diagram has MD5 as the hash function the structure is exactly the same for SHA-1 with the exception that the message length is limited in size (its isn't for MD5) and the hash value (and intermediate values CV_i) are 160 bits and not 128 as shown (which is the case for MD5).

The processing consists of the following 5 steps:

1. **Append padding bits:** The message is padded so that its length is congruent to 448 modulo 512 (length $\equiv 448 \pmod{512}$). 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. Thus, the number of padding bits is in the range of 1 to 512 bits. The padding consists of a single 1-bit followed by the necessary number of 0-bits.

2. **Append length:** A block of 64 bits is appended to the message. This block is treated as an unsigned 64-bit integer (most significant byte first) and contains the length of the original message (before padding).

3. **Initialize MD buffer:** A 160 bit buffer is used to hold intermediate values and final results of the Hash function represented as 5, 32 bit registers (A, B, C, D, E) initialized as follows:

$$A = 67452301$$
$$B = EFCDAB89$$
$$C = 98BADCFE$$
$$D = 10325476$$
$$E = C3D2E1F0$$

4. **Process message in 512 bit (16 word) blocks:** The heart of the algorithm is a module which consists of four "rounds" of processing of 20 steps each (see fig-

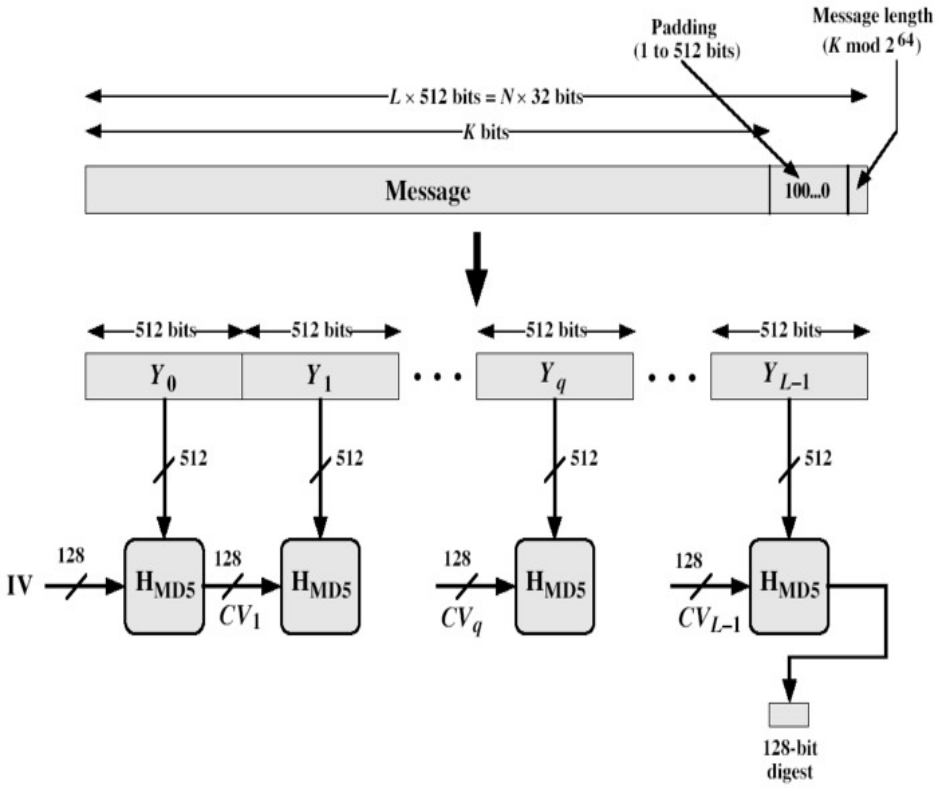


Figure 6.4: Message Digest Generation Using MD5 (equally applicable to SHA-1 with 160 bits instead of 128 etc.).

ure 6.5). Each round has similar structure but uses a different primitive logical function (f_1, f_2, f_3 and f_4). Each round takes as input the current 512-bit block being processed (Y_q) and the 160-bit buffer value ABCDE and updates the contents of the buffer. Each round also makes use of an additive constant K_t where $0 \leq t \leq 79$ indicates one of the 80 steps across four rounds. In fact, only four distinct constants are used (one for $0 \leq t \leq 19$, $20 \leq t \leq 39$, $40 \leq t \leq 59$ and $60 \leq t \leq 79$). The output of the fourth round is added (modulo 2^{32}) to the input to the first round (CV_q) to produce CV_{q+1}.

5. Output after all L 512 bit blocks have been processed the output from the Lth stage is the 160 bit digest.

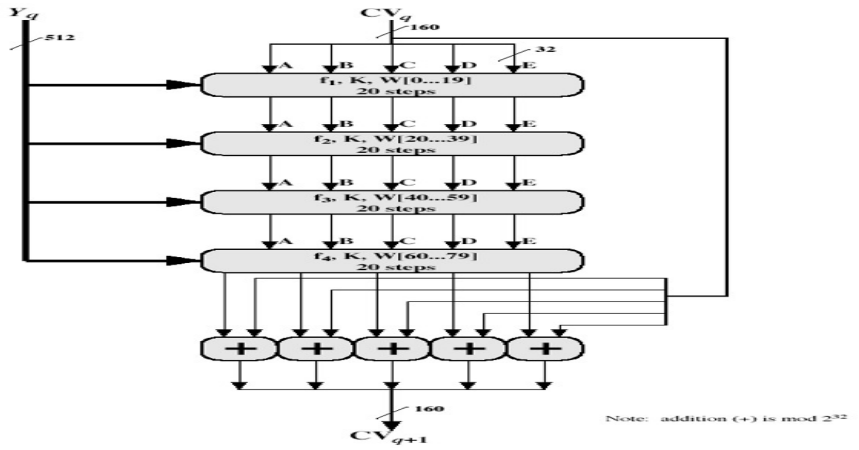


Figure 6.5: SHA-1 Processing of a Single 512-bit Block (SHA-1 compression function).

We can summarise the behavior SHA-1 as follows:

$$\begin{aligned} \text{CV}_0 &= \text{IV} \\ \text{CV}_{q+1} &= \text{SUM}_{32}(CV_q, \text{ABCDE}_q) \\ \text{MD} &= \text{CV}_L \end{aligned}$$

where

IV	=	initial value of the ABCDE buffer, defined in step 3.
ABCDE_q	=	the output of the last round of processing of the qth message block.
L	=	the number of blocks in the message (including padding and length fields).
SUM_{32}	=	Addition modulo 2^{32} performed separately on each word of the pair of inputs.
MD	=	final message digest value.

6.3 Digital Signatures

Message authentication protects two parties who exchange messages from any third party. However, it does not protect the two parties against each other. Several forms of dispute between the two are possible. For example, suppose that John sends an authenticated message to Mary using one of the schemes described earlier. Consider the following disputes that could arise:

- Mary may forge a different message and claim that it came from John. Mary would simply have to create a message and append an authentication code using the key that John and Mary share.
- John can deny sending the message. Because it is possible for Mary to forge a message, there is no way to prove that John did in fact send the message.

Both scenarios are of legitimate concern. In situations where there is not complete trust between sender and receiver, something more than authentication is needed. The most attractive solution to this problem is the **digital signature**.

The digital signature is analogous to the handwritten signature. It 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. On the basis of these properties, we can formulate the following requirements for a digital signature:

- The signature must be a bit pattern that depends on the message being signed.
- 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 recognise 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 of the digital signature in storage.

One of the most popular algorithms for implementing digital signatures is discussed next.

6.3.1 Digital Signature Standard (DSS)

The National Institute of Standards and Technology (NIST) has published FIPS 186 known as the Digital Signature Standard (DSS). The DSS makes use of the Secure Hash Algorithm (SHA) that we just discussed and presents a new digital signature technique, the Digital Signature Algorithm (DSA). The DSS was originally proposed in 1991 and revised in 1993 in response to public feedback concerning the security of the scheme. There was a further minor revision in 1996. In 2000, an expanded version of the standard was issued as FIPS 186-2. This latest version also incorporates digital signature algorithms based on RSA and on elliptic curve cryptography. In this section, we discuss the original DSS algorithm.

The DSS uses an algorithm that is designed to provide only the digital signature function. Unlike RSA, it cannot be used for encryption or key exchange. Nevertheless, it is a public-key technique. It is based on the difficulty of computing discrete logarithms (as is the Diffie Hellman key exchange).

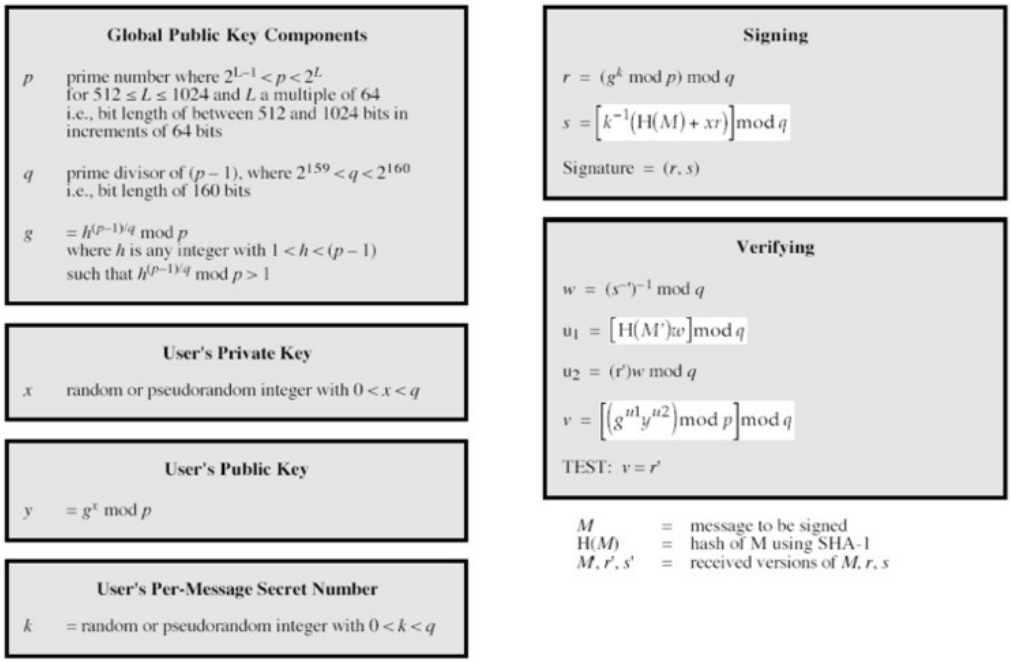


Figure 6.6: The Digital Signature Algorithm (DSA).

The overall scheme is seen in figure 6.6. There are three parameters that are public and can be common to a group of users. A 160-bit prime number q is chosen. Next, a prime number p is selected with a length between 512 and 1024 bits such that q divides $(p-1)$. Finally, g is chosen to be of the form $h^{(p-1)/q} \bmod p$, where h is an integer between 14 and $(p-1)$ with the restriction that g must be greater than 1.

With these numbers in hand, each user selects a private key and generates a public key. The private key x must be a number from 1 to $(p-1)$ and should be chosen randomly or

pseudorandomly. The public key is calculated from the private key as $y = g^x \bmod p$. The calculation of y is relatively straightforward. However finding x given the other parameters appears not to be.

To create a signature, a user calculates two quantities, r and s, that are functions of the public key components (p, q, g), the user's private key (x), the hash code of the message H(M), and an additional integer k that should be generated randomly or pseudorandomly and be unique for each signing.

At the receiving end, verification is performed using the formulas shown in Figure 6.6. The receiver generates a quantity v that is a function of the public key component, the sender's public key, and the hash code of the incoming message. If this quantity matches the r component of the signature, then the signature is validated.

Figure 6.7 depicts the functions of signing and verifying. The structure of the algorithm, as revealed in the figure, is quite interesting. Note that the test at the end is on the value r, which does not depend on the message at all. Instead, r is a function of k and the three global public-key components. The multiplicative inverse of $k \bmod q$ is passed to a function that also has as inputs the message hash code and the user's private key. The structure of this function is such that the receiver can recover r using the incoming message and signature, the public key of the user, and the global public key. It is certainly not obvious from figure 10.7 that such a scheme would work. However it has been proven that is does.

Given the difficulty of taking discrete logarithms, it is infeasible for an opponent to recover K from r or to recover x from s.

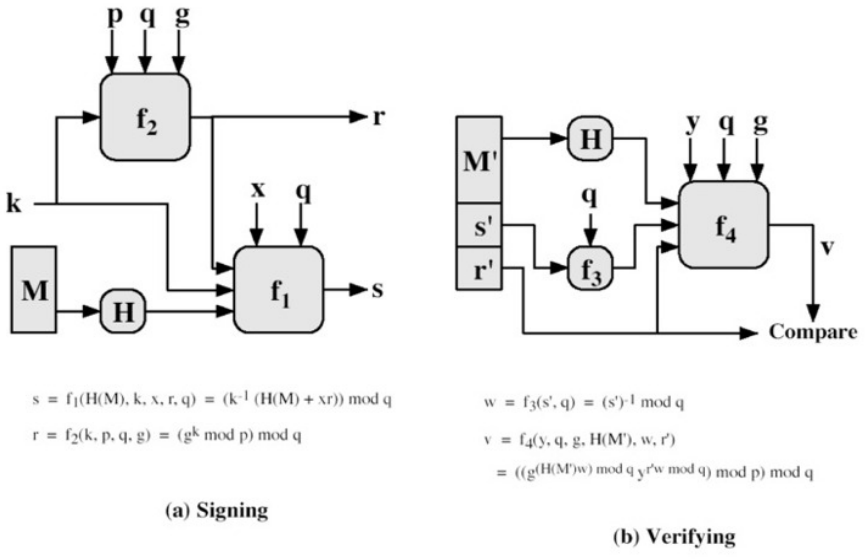


Figure 6.7: DSS Signing and Verifying.

Central Authentication Service (**CAS**)

The **Central Authentication Service** (**CAS**) is a single sign-on protocol for the web developed by Shawn Bayern. Its purpose is to permit a user to access multiple applications while providing their credentials (such as user ID and password) only once. It also allows web applications to authenticate users without gaining access to a user's security credentials, such as a password. The name *CAS* also refers to a software package that implements this protocol.

The CAS protocol involves at least three parties:

- a *client* web browser
- the web *application* requesting authentication and
- the *CAS server*.

It may also involve a *back-end service*, such as a database server, that does not have its own HTTP interface but communicates with a web application.

When the client visits an application requiring authentication, the application redirects it to CAS. CAS validates the client's authenticity, usually by checking a username and password against a database (such as Kerberos, LDAP or Active Directory).

If the authentication succeeds, CAS returns the client to the application, passing along a service ticket. The application then validates the ticket by contacting CAS over a secure connection and providing its own service identifier and the ticket. CAS then gives the application trusted information about whether a particular user has successfully authenticated.

CAS allows multi-tier authentication via proxy address. A cooperating *back-end* service, like a database or mail server, can participate in CAS, validating the authenticity of users via information it receives from web applications. Thus, a webmail client and a webmail server can all implement CAS.

Kerberos

Kerberos is a computer network security protocol developed by MIT that authenticates service requests between two or more trusted hosts across an untrusted network, like the internet. It uses secret-key cryptography and a trusted third party for authenticating client-server applications and verifying users' identities.

Initially developed by the Massachusetts Institute of Technology (MIT) for Project Athena in the late '80s, Kerberos is now the default authorization technology used by Microsoft Windows. Kerberos implementations also exist for other operating systems such as Apple OS, FreeBSD, UNIX, and Linux.

Microsoft rolled out its version of Kerberos in Windows 2000, and it's become the go-to protocol for websites and single sign-on implementations over different platforms. The Kerberos Consortium maintains the Kerberos as an open-source project.

The protocol derives its name from the legendary three-headed dog Kerberos (also known as Cerberus) from Greek myths, the canine guardian to the entrance to the underworld. Kerberos had a snake tail and a particularly bad temper and, despite one notable exception, was a very useful guardian.

But in the protocol's case, the three heads of Kerberos represent the client, the server, and the Key Distribution Center (KDC). The latter functions as the trusted third-party authentication service.

Users, machines, and services that use Kerberos depend on the KDC alone, which works as a single process that provides two functions: authentication and ticket-granting. KDC "tickets" offer authentication to all parties, allowing nodes to verify their identity securely. The Kerberos authentication process employs a conventional shared secret cryptography that prevents packets

traveling across the network from being read or altered, as well as protecting messages from eavesdropping and replay (or playback) attacks.

What is Kerberos Used For?

Although Kerberos is found everywhere in the digital world, it is employed heavily on secure systems that depend on reliable auditing and authentication features. Kerberos is used in Posix authentication, and Active Directory, NFS, and Samba. It's also an alternative authentication system to SSH, POP, and SMTP.

As a part of the learning flow of what Kerberos is, let us next learn about the Kerberos protocol flow.

Kerberos Protocol Flow Overview

Let's take a more detailed look at what Kerberos authentication is and how it works by breaking it down into its core components.

Here are the principal entities involved in the typical Kerberos workflow:

Client. The client acts on behalf of the user and initiates communication for a service request

Server. The server hosts the service the user wants to access

Authentication Server (AS). The AS performs the desired client authentication. If the authentication happens successfully, the AS issues the client a ticket called TGT (Ticket Granting Ticket). This ticket assures the other servers that the client is authenticated

Key Distribution Center (KDC). In a Kerberos environment, the authentication server logically separated into three parts: A database (db), the Authentication Server (AS), and the Ticket Granting Server (TGS). These three parts, in turn, exist in a single server called the Key Distribution Center

Ticket Granting Server (TGS). The TGS is an application server that issues service tickets as a service

Now let's break down the protocol flow.

First, there are three crucial secret keys involved in the Kerberos flow. There are unique secret keys for the client/user, the TGS, and the server shared with the AS.

Client/user. Hash derived from the user's password

TGS secret key. Hash of the password employed in determining the TGS

Server secret key. Hash of the password used to determine the server providing the service.

 # Steps

The protocol flow consists of the following steps:

Step 1: Initial client authentication request. The user asks for a Ticket Granting Ticket (TGT) from the authentication server (AS). This request includes the client ID.

Step 2: KDC verifies the client's credentials. The AS checks the database for the client and TGS's availability. If the AS finds both values, it generates a client/user secret key, employing the user's password hash.

The AS then computes the TGS secret key and creates a session key (SK1) encrypted by the client/user secret key. The AS then generates a TGT containing the client ID, client network address, timestamp, lifetime, and SK1. The TGS secret key then encrypts the ticket.

Step 3: The client decrypts the message. The client uses the client/user secret key to decrypt the message and extract the SK1 and TGT, generating the authenticator that validates the client's TGS.

Step 4: The client uses TGT to request access. The client requests a ticket from the server offering the service by sending the extracted TGT and the created authenticator to TGS.

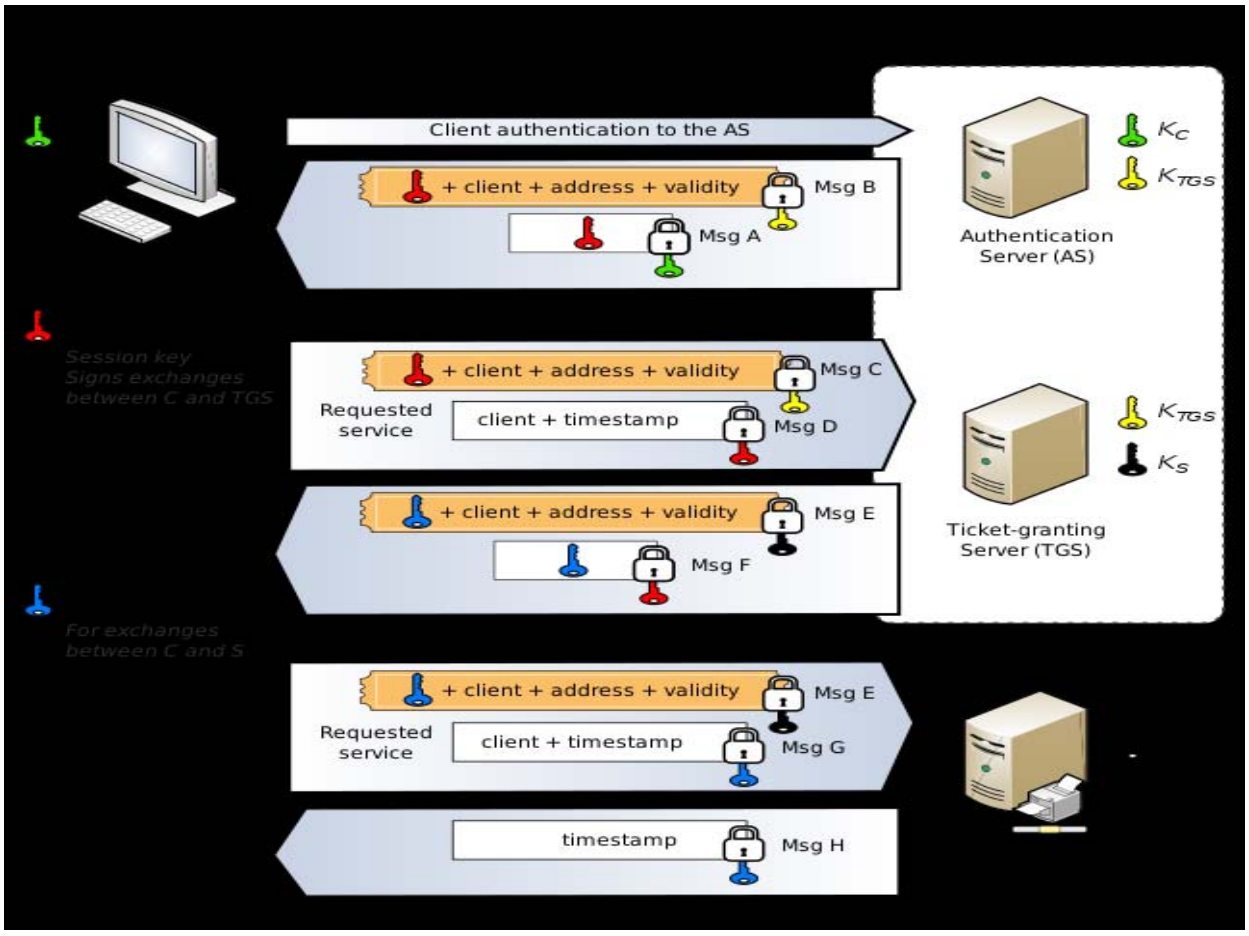
Step 5: The KDC creates a ticket for the file server. The TGS then uses the TGS secret key to decrypt the TGT received from the client and extracts the SK1. The TGS decrypts the authenticator and checks to see if it matches the client ID and client network address. The TGS also uses the extracted timestamp to make sure the TGT hasn't expired.

If the process conducts all the checks successfully, then the KDC generates a service session key (SK2) that is shared between the client and the target server.

Finally, the KDC creates a service ticket that includes the client id, client network address, timestamp, and SK2. This ticket is then encrypted with the server's secret key obtained from the db. The client receives a message containing the service ticket and the SK2, all encrypted with SK1.

Step 6: The client uses the file ticket to authenticate. The client decrypts the message using SK1 and extracts SK2. This process generates a new authenticator containing the client network address, client ID, and timestamp, encrypted with SK2, and sends it and the service ticket to the target server.

Step 7: The target server receives decryption and authentication. The target server uses the server's secret key to decrypt the service ticket and extract the SK2. The server uses SK2 to decrypt the authenticator, performing checks to make sure the client ID and client network address from the authenticator and the service ticket match. The server also checks the service ticket to see if it's expired.



Once the checks are met, the target server sends the client a message verifying that the client and the server have authenticated each other. The user can now engage in a secure session.

After coming so far in learning what Kerberos is, let us next look into the topic if Kerberos is infallible.

Is Kerberos Infallible?

No security measure is 100% impregnable, and Kerberos is no exception. Since it's been around for so long, hackers have had the opportunity over the years to find ways around it, usually by forging tickets, making repeated attempts to guess passwords (brute force/credential stuffing), and using malware to downgrade the encryption.

Despite this, Kerberos is still the best security access protocol available today. The protocol is flexible enough to employ more robust encryption algorithms to help combat new threats, and if users practice good password choice policies, you should be fine!