authentication, there is no real message communication involved until Alice is authenticated by Bob. Alice needs to be online and to take part in the process. Only after she is authenticated can messages be communicated between Alice and Bob. Data-origin authentication is required when an email is sent from Alice to Bob. Entity authentication is required when Alice gets cash from an automatic teller machine.

2. Second, message authentication simply authenticates one message; the process needs to be repeated for each new message. Entity authentication authenticates the claimant for the entire duration of a session.

Verification Categories

In entity authentication, the claimant must identify herself to the verifier. This can be done with one of three kinds of witnesses: *something known, something possessed,* or *something inherent.*

- **Something known.** This is a secret known only by the claimant that can be checked by the verifier. Examples are a password, a PIN, a secret key, and a private key.
- **Something possessed.** This is something that can prove the claimant's identity. Examples are a passport, a driver's license, an identification card, a credit card, and a smart card.
- **Something inherent.** This is an inherent characteristic of the claimant. Examples are conventional signatures, fingerprints, voice, facial characteristics, retinal pattern, and handwriting.

Entity Authentication and Key Management

This chapter discusses entity authentication. The next chapter discusses key managment. These two topics are very closely related; most key management protocols use entity authentication protocols. This is why these two topics are discussed together in most books. In this book they are treated separately for clarity.

14.2 PASSWORDS

The simplest and oldest method of entity authentication is the **password-based authentication,** where the password is something that the **claimant** *knows*. A password is used when a user needs to access a system to use the system's resources (login). Each user has a user identification that is public, and a password that is private. We can divide these authentication schemes into two groups: the **fixed password** and the **one-time password.**

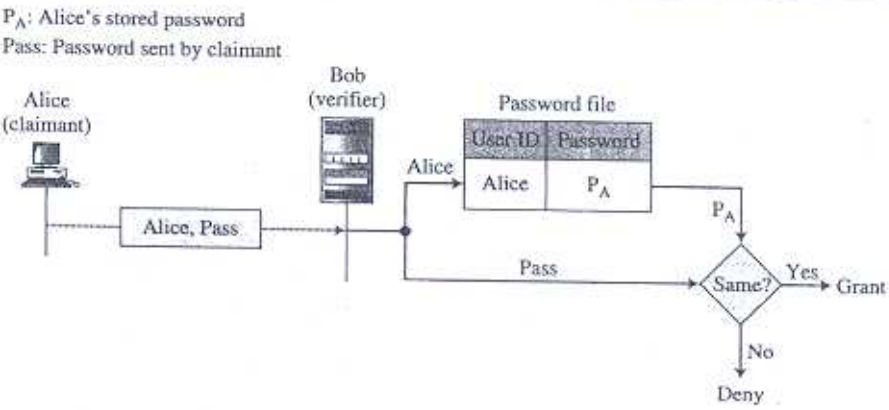
Fixed Password

A fixed password is a password that is used over and over again for every access. Several schemes have been built, one upon the other.

First Approach

In the very rudimentary approach, the system keeps a table (a file) that is sorted by user identification. To access the system resources, the user sends her user identification and password, in plaintext, to the system. The system uses the identification to find the password in the table. If the password sent by the user matches the password in the table, access is granted; otherwise, it is denied. Figure 14.1 shows this approach.

Figure 14.1 *User ID and password file*



Attacks on the First Approach This approach is subject to several kinds of attack.

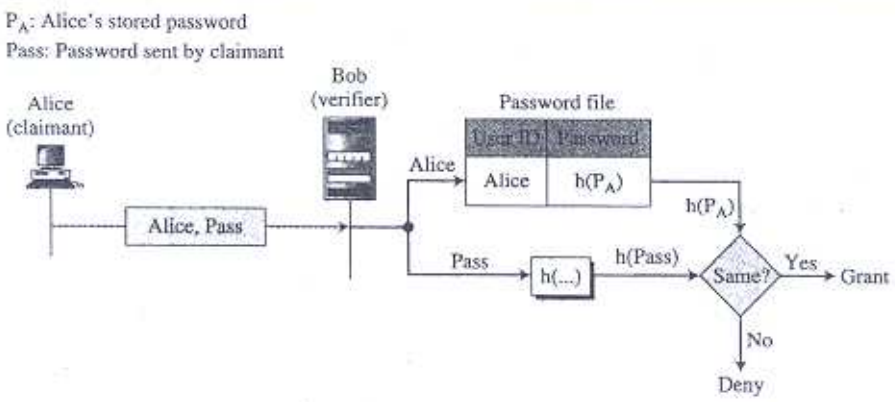
- **Eavesdropping.** Eve can watch Alice when she types her password. Most systems, as a security measure, do not show the characters a user types. Eavesdropping can take a more sophisticated form. Eve can listen to the line and intercept the message, thereby capturing the password for her own use.
- **Stealing a password.** The second type of attack occurs when Eve tries to physically steal Alice's password. This can be prevented if Alice does not write down the password and instead she just commits it to memory. For this reason the password should be very simple or else related to something familiar to Alice. But this makes the password vulnerable to other types of attacks.
- **Accessing a password file.** Eve can hack into the system and get access to the ID/password file. Eve can read the file and find Alice's password or even change it. To prevent this type of attack, the file can be read/write protected. However, most systems need this type of file to be readable by the public. We will see how the second approach can protect the file from this type of attack.
- **Guessing.** Using a guessing attack, Eve can log into the system and try to guess Alice's password by trying different combinations of characters. The password is particularly vulnerable if the user is allowed to choose a short password (a few characters). It is also vulnerable if Alice has chosen something trivial, such as her birthday, her child's name, or the name of her favorite actor. To prevent

guessing, a long random password is recommended, something that is not very obvious. However, the use of such a random password may also create a problem. Because she could easily forget such a password, Alice might store a copy of it somewhere, which makes the password subject to stealing.

Second Approach

A more secure approach is to store the hash of the password (instead of the plaintext password) in the password file. Any user can read the contents of the file, but, because the hash function is a one-way function, it is almost impossible to guess the value of the password. Figure 14.2 shows the situation. When the password is created, the system hashes it and stores the hash in the password file.

Figure 14.2 *Hashing the password*



When the user sends the ID and the password, the system creates a hash of the password and then compares the hash value with the one stored in the file. If there is a match, the user is granted access; otherwise, access is denied. In this case, the file does not need to be read protected.

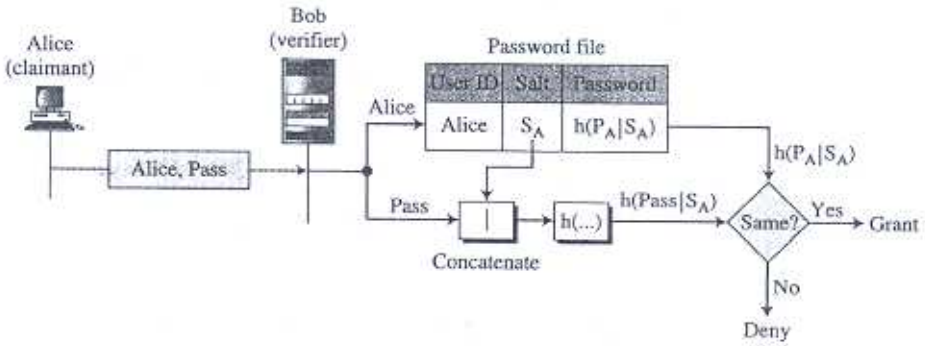
Dictionary Attack The hash function prevents Eve from gaining access to the system even though she has the password file. However, there is still the possibility of **dictionary attack.** In this attack, Eve is interested in finding one password, regardless of the user ID. For example, if the password is 6 digits, Eve can create a list of 6-digit numbers (000000 to 999999), and then apply the hash function to every number; the result is a list of one million hashes. She can then get the password file and search the second-column entries to find a match. This could be programmed and run offline on Eve's private computer. After a match is found, Eve can go online and use the password to access the system. The third approach shows how to make this attack more difficult.

Third Approach

The third approach is called **salting** the password. When the password string is created, a random string, called the salt, is concatenated to the password. The salted password is then hashed. The ID, the salt, and the hash are then stored in the file. Now, when a user asks for access, the system extracts the salt, concatenates it with the received password, makes a hash out of the result, and compares it with the hash stored in the file. If there is a match, access is granted; otherwise, it is denied (see Figure 14.3).

Figure 14.3 *Salting the password*

P_A: Alice's password
S_A: Alice's salt
Pass: Password sent by claimant



Salting makes the dictionary attack more difficult. If the original password is 6 digits and the salt is 4 digits, then hashing is done over a 10-digit value. This means that Eve now needs to make a list of 10 million items and create a hash for each of them. The list of hashes has 10 million entries, and the comparison takes much longer. Salting is very effective if the salt is a very long random number. The UNIX operating system uses a variation of this method.

Fourth Approach

In the fourth approach, two identification techniques are combined. A good example of this type of authentication is the use of an ATM card with a PIN (personal identification number). The card belongs to the category "*something possessed*" and the PIN belongs to the category "*something known*". The PIN is a password that enhances the security of the card. If the card is stolen, it cannot be used unless the PIN is known. The PIN number, however, is traditionally very short so it is easily remembered by the owner. This makes it vulnerable to the guessing type of attack.

One-Time Password

A **one-time password** is a password that is used only once. This kind of password makes eavesdropping and salting useless. Three approaches are discussed here.

First Approach

In the first approach, the user and the system agree upon a *list of passwords*. Each password on the list can be used only once. There are some drawbacks to this approach. First, the system and the user must keep a long list of passwords. Second, if the user does not use the passwords in sequence; the system needs to perform a long search to find the match. This scheme makes eavesdropping and reuse of the password useless. The password is valid only once and cannot be used again.

Second Approach

In the second approach, the user and the system agree to *sequentially update the password*. The user and the system agree on an original password, P_1, which is valid only for the first access. During the first access, the user generates a new password, P_2, and encrypts this password with P_1 as the key. P_2 is the password for the second access. During the second access, the user generates a new password, P_3, and encrypts it with P_2; P_3 is used for the third access. In other words, P_i is used to create P_{i+1}. Of course, if Eve can guess the first password (P_1), she can find all of the subsequent ones.

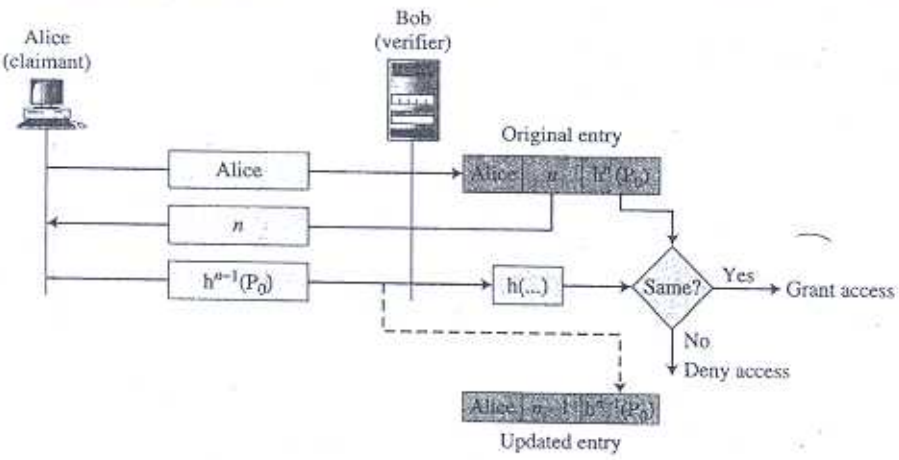
Third Approach

In the third approach, the user and the system create a sequentially updated password using a hash function In this approach, elegantly devised by Leslie Lamport, the user and the system agree upon an original password, P_0, and a counter, n. The system calculates $h^n(P_0)$, where h^n means applying a hash function n times. In other words,

$$h^n(x) = h(h^{n-1}(x)) \quad h^{n-1}(x) = h(h^{n-2}(x)) \quad \ldots \quad h^2(x) = h(h(x)) \quad h^1(x) = h(x)$$

The system stores the identity of Alice, the value of n, and the value of $h^n(P_0)$. Figure 14.4 shows how the user accesses the system the first time.

Figure 14.4 *Lamport one-time password*



When the system receives the response of the user in the third message, it applies the hash function to the value received to see if it matches the value stored in the entry. If there is a match, access is granted; otherwise, it is denied. The system then decrements the value of n in the entry and replaces the old value of the password $h^n(P_0)$ with the new value $h^{n-1}(P_0)$.

When the user tries to access the system for the second time, the value of the counter it receives is $n - 1$. The third message from the user is now $h^{n-2}(P_0)$. When the system receives this message, it applies the hash function to get $h^{n-1}(P_0)$, which can be compared with the updated entry.

The value of n in the entry is decremented each time there is an access. When the value becomes 0, the user can no longer access the system; everything must be set up again. For this reason, the value of n is normally chosen as a large number such as 1000.

14.3 CHALLENGE-RESPONSE

In password authentication, the claimant proves her identity by demonstrating that she knows a secret, the password. However, because the claimant reveals this secret, it is susceptible to interception by the adversary. In **challenge-response authentication,** the claimant proves that she *knows* a secret without sending it. In other words, the claimant does not send the secret to the verifier; the verifier either has it or finds it.

In challenge-response authentication, the claimant proves that she knows a secret without sending it to the verifier.

The *challenge* is a time-varying value such as a random number or a timestamp that is sent by the verifier. The claimant applies a function to the challenge and sends the result, called a *response,* to the verifier. The response shows that the claimant knows the secret.

The challenge is a time-varying value sent by the verifier; the response is the result of a function applied on the challenge.

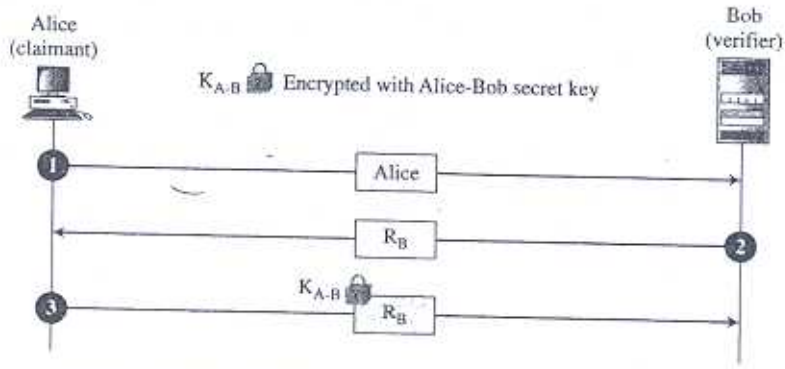
Using a Symmetric-Key Cipher

Several approaches to challenge-response authentication use symmetric-key encryption. The secret here is the shared secret key, known by both the claimant and the verifier. The function is the encrypting algorithm applied on the challenge.

First Approach

In the first approach, the verifier sends a **nonce,** a random number used only once, to challenge the claimant. A nonce must be time-varying; every time it is created, it is different. The claimant responds to the challenge using the secret key shared between the claimant and the verifier. Figure 14.5 shows this first approach.

Figure 14.5 *Nonce challenge*



The first message is not part of challenge-response, it only informs the verifier that the claimant wants to be challenged. The second message is the challenge. R_B is the nonce randomly chosen by the verifier (Bob) to challenge the claimant. The claimant encrypts the nonce using the shared secret key known only to the claimant and the verifier and sends the result to the verifier. The verifier decrypts the message. If the nonce obtained from decryption is the same as the one sent by the verifier, Alice is granted access.

Note that in this process, the claimant and the verifier need to keep the symmetric key used in the process secret. The verifier must also keep the value of the nonce for claimant identification until the response is returned.

The reader may have noticed that use of a nonce prevents a replay of the third message by Eve. Eve cannot replay the third message and pretend that it is a new request for authentication by Alice, because once Bob receives the response, the value of R_B is not valid any more. The next time a new value is used.

Second Approach

In the second approach, the time-varying value is a timestamp, which obviously changes with time. In this approach the challenge message is the current time sent from the verifier to the claimant. However, this supposes that the client and the server clocks are synchronized; the claimant knows the current time. This means that there is no need for the challenge message. The first and third messages can be combined. The result is that authentication can be done using one message, the response to an implicit challenge, the current time. Figure 14.6 shows the approach.

Third Approach

The first and second approaches are for unidirectional authentication. Alice is authenticated to Bob, but not the other way around. If Alice also needs to be sure about Bob's identity, we need bidirectional authentication. Figure 14.7 shows a scheme.

Figure 14.6 *Timestamp challenge*

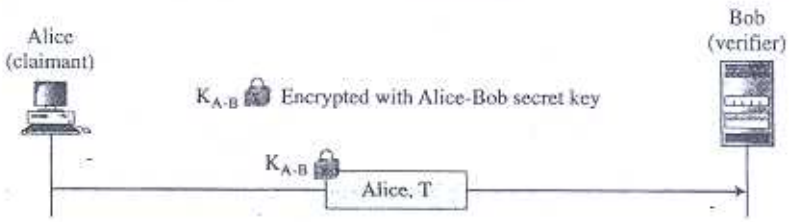
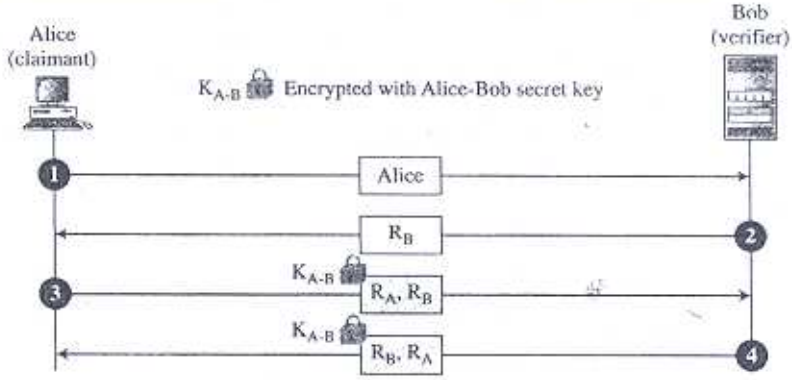


Figure 14.7 *Bidirectional authentication*



The second message R_B is the challenge from Bob to Alice. In the third message, Alice responds to Bob's challenge and at the same time, sends her challenge R_A to Bob. The third message is Bob's response. Note that in the fourth message the order of R_A and R_B are switched to prevent a replay attack of the third message by an adversary.

Using Keyed-Hash Functions

Instead of using encryption/decryption for entity authentication, we can also use a keyed-hash function (MAC). One advantage to the scheme is that it preserves the integrity of challenge and response messages and at the same time uses a secret, the key.

Figure 14.8 shows how we can use a keyed-hash function to create a challenge response with a timestamp.

Note that in this case, the timestamp is sent both as plaintext and as text scrambled by the keyed-hash function. When Bob receives the message, he takes the plaintext T, applies the keyed-hash function, and then compares his calculation with what he received to determine the authenticity of Alice.