2.2.1 Mutual Authentication

Suppose that Alice and Bob wish to verify each other's identities; we call this problem "mutual authentication". They could perhaps do this by running Protocol 2.2 twice (i.e., Alice verifies Bob's identity and then Bob verifies Alice's identity. However, it is generally more efficient to design a single protocol to accomplish both goals at once. We will discuss these kinds of protocols in this section.

In order to prove that a mutual authentication protocol is "secure", we need to carefully consider the objectives of the protocol, the attack model and the adversarial goal. To begin with, we stipulate that both Alice and Bob are required to "accept" if a session of the protocol is to be considered complete. Furthermore, the adversary could be trying to fool Alice, Bob, or both of them into accepting when they should not do so.

We need to introduce a couple of new ideas before proceeding further. A participant (Alice or Bob) is said to be an *honest participant* if he or she follows the protocol, performs correct computations, and does not reveal information to the adversary. The adversary is a *passive adversary* if he observes the information being exchanged in sessions of the protocol, but does not create or change any messages himself. The adversary is an *active adversary* if he creates new messages, or if he changes messages sent by Alice and Bob. An active adversary could be taking the place of Alice or Bob. Alternatively, he may be just changing the messages sent by Alice to Bob — or those sent by Bob to Alice — after they are placed in the communication channel. Also, an (active) adversary may have observed several previous sessions, in a passive way, before he carries out his active attack.

The following conditions specify what the outcome of the protocol should be, depending on actions performed by the adversary.

1. If Alice and Bob are the two participants in a session of the protocol, they are both honest, and the adversary is passive, then Alice and Bob will both output "accept".
2. If the adversary is active during a given flow of the protocol, then no honest participant will "accept" after that flow.

The reason for the second, slightly more complicated condition given above, namely, condition 2., is that the adversary might be inactive in a particular session until after one participant accepts, and then become active. Therefore it is entirely possible that one honest participant "accepts " and the other honest participant "rejects". In this situation, the session has not completed successfully. It is the adversary's goal to make all honest participants "accept", so the adversary does not achieve his goal in this case.

There are several ways in which the adversary could be active in a protocol. We list some of these now:

1. The adversary impersonates Alice, hoping to cause Bob to accept.

2. The adversary impersonates Bob, hoping to cause Alice to accept.
3. The adversary is active in some session involving Alice and Bob, and he is trying to cause both Alice and Bob to accept.

As mentioned above, Alice and Bob can verify each other's identities by means of two sessions of Protocol 2.2. What if they try to combine the two sessions into a single protocol, in the obvious way? This reduces the number of flows required (compared to running the original protocol twice) from four to three. However, it turns out that the resulting mutual identification protocol is flawed and can be attacked. Here is the protocol . . .

Protocol 2.3: INSECURE MUTUAL CHALLENGE-AND-RESPONSE

1. Bob chooses a random challenge, r_1, which he sends to Alice.
2. Alice chooses a random challenge, r_2. She also computes

$$y_1 = MAC_K(Alice \mid\mid r_1)$$

and sends r_2 and y_1 to Bob.

3. Bob computes

$$y_1' = MAC_K(Alice \mid\mid r_1).$$

If $y_1' = y_1$, then Bob outputs "accept"; otherwise, Bob outputs "reject". Bob also computes

$$y_2 = MAC_K(Bob \mid\mid r_2)$$

and sends y_2 to Alice.

4. Alice computes

$$y_2' = MAC_K(Bob \mid\mid r_2).$$

If $y_2' = y_2$, then Alice outputs "accept"; otherwise, Alice outputs "reject".

. . . and here is a parallel session attack on it:

Alice		Oscar		Bob
	$\xleftarrow{\quad r_1 \quad}$			
	$\xrightarrow{MAC_K(Alice \parallel r_1),\, r_2}$			
			$\xrightarrow{\quad r_2 \quad}$	
			$\xleftarrow{MAC_K(Bob \parallel r_2),\, r_3}$	
	$\xleftarrow{MAC_K(Bob \parallel r_2)}$			

Oscar, pretending to be Bob, initiates a session with Alice. When Oscar receives Alice's challenge, r_2, in the second flow, he "accepts", and then he initiates a second session (pretending to be Alice) with Bob. In this second session, Oscar sends r_2 to Bob as his challenge in the first flow. When Oscar receives Bob's response (in the second flow in the second session), he forwards it to Alice as the third flow in the first session. Alice will "accept", and therefore Oscar has successfully impersonated Bob in this session. (The second session is dropped, i.e., it is never completed.) This constitutes a successful attack, because all the honest participants in the first session (namely, Alice) accepted.

Clearly, the attack is based on using on re-using a flow from one session in a different flow in another session. It is not difficult to rectify the problem, and there are in fact several ways to modify the protocol so that it is secure. Basically, what is required is to design the flows so that each flow has a different format. Here is one solution along these lines:

Protocol 2.4: (SECURE) MUTUAL CHALLENGE-AND-RESPONSE

1. Bob chooses a random challenge, r_1, which he sends to Alice.
2. Alice chooses a random challenge, r_2. She also computes

$$y_1 = MAC_K(Alice \parallel r_1 \parallel r_2)$$

and sends r_2 and y_1 to Bob.
3. Bob computes

$$y_1' = MAC_K(Alice \parallel r_1 \parallel r_2).$$

If $y_1' = y_1$, then Bob outputs "accept"; otherwise, Bob outputs "reject". Bob also computes

$$y_2 = MAC_K(Bob \parallel r_2)$$

and sends y_2 to Alice.
4. Alice computes

$$y_2' = MAC_K(Bob \parallel r_2).$$

If $y_2' = y_2$, then Alice outputs "accept"; otherwise, Alice outputs "reject".

The only change is in the definition of y_1 in step 2. Now this MAC depends on two challenges, r_1 and r_2. This serves to distinguish the second flow from the third flow (in which the MAC depends only on the challenge r_2).

Protocol 2.4 can be analyzed in a similar fashion as Protocol 2.2. The analysis is a bit more complicated, however, because the adversary could try to play the role of Bob (fooling Alice) or Alice (fooling Bob). The probability that a value y_1 or y_2 can be "reused" from a previous session can be computed, as can the probability that the adversary can compute a new MAC from scratch. It can be shown that Oscar's probability of deceiving Alice or Bob is at most $q/2^k + 2\epsilon$; the details of the proof are omitted.

2.3 Challenge-and Response in the Public-key Setting

Now we turn to the public-key setting, where we do not assume that Alice and Bob have a prior secret key known to both of them. We assume that Alice and Bob are members of a network, in which every participant has public and private keys for certain prespecified cryptosystems and/or signature schemes. In a setting such as this, it is always necessary to provide a mechanism to authenticate the public keys of other people in the network. This requires some kind of "public key infrastructure" (denoted "PKI"). In this chapter, we assume that there is a certain *trusted authority*, denoted TA, who signs the public keys of all people in the network. The (public) verification key of the TA, denoted ver_{TA}, is assumed to be known "by magic" to everyone in the network. This simplified setting is perhaps not completely realisitc, but it allows us to concentrate on the design of the protocols. The general topic of public key infrastructures will be studied in more detail in a later chapter.

2.3.1 Certificates

A *certificate* for someone in the network will consist of some identifying information for that person (e.g., their name, email address, etc.), their public key(s), and the signature of the TA on that information. A certificate allows network users to verify the authenticity of each other's keys.

Suppose, for example, that Alice wants to obtain a certificate from the TA which contains a copy of Alice's public verification key for a signature scheme. The steps in Protocol 2.5 are carried out.

Protocol 2.5: ISSUING A CERTIFICATE TO ALICE

1. The TA establishes Alice's identity by means of conventional forms of identification such as a birth certificate, passport, etc. Then the TA forms a string, denoted $ID(Alice)$, which contains Alice's identification information.
2. A private signing key for Alice, sig_{Alice}, and a corresponding public verification key, ver_{Alice}, are determined.
3. The TA generates its signature

$$s = sig_{TA}(ID(Alice) \parallel ver_{Alice})$$

on Alice's identity string and verification key. The certificate

$$\mathbf{Cert}(Alice) = (ID(Alice) \parallel ver_{Alice} \parallel s)$$

is given to Alice.

We are not specifying exactly how Alice identifies herself to the TA, nor do we specify the precise format of $ID(Alice)$, or how the public and private keys of Alice are selected. In general, these implementation details could vary from one PKI to another.

It is possible for anyone who knows the TA's verification key, ver_{TA}, to verify anyone else's certificate. Suppose that Bob wants to be assured that Alice's public key is authentic. Alice can give her certificate to Bob. Bob can then verify the signature of the TA by checking that

$$ver_{TA}(ID(Alice) \parallel ver_{Alice}, s) = true.$$

The security of a certificate follows immediately from the security of the signature scheme used by the TA.

As mentioned above, the purpose of verifying a certificate is to authenticate someone's public key. The certificate itself does not provide any kind of proof of identity, because certificates contain only public information. Certificates can be distributed to anyone, and possession of a certificate does not imply ownership of it.

2.3.2 Public-key Identification Schemes

We now look at mutual identification schemes in the public-key setting. Our strategy is to modify Protocol 2.4 by reaplcing MACs by signatures. Another difference is that, in the secret-key setting, we included the name of the person who produced the MAC in each MAC (this was important because a secret key K, being known to two parties, allows either party to create MACs). In the public-key setting, only one person can create signatures using a specified private signing key, namely, the person possessing that key. Therefore we do not need to explicitly designate who created a particular signature.

The mutual identification protocol, modified to fit the public-key setting, is as follows:

Protocol 2.6: PUBLIC-KEY MUTUAL AUTHENTICATION

1. Bob chooses a random challenge, r_1, which he sends to Alice.
2. Alice chooses a random challenge, r_2. She also computes
$$y_1 = \mathit{sig}_{Alice}(r_1 \parallel r_2)$$
and sends $\mathbf{Cert}(Alice)$, r_2 and y_1 to Bob.
3. Bob verifies Alice's public key, ver_{Alice}, on the certificate $\mathbf{Cert}(Alice)$. Then he checks that
$$\mathit{ver}_{Alice}(r_1 \parallel r_2, y_1) = true.$$
If so, then Bob outputs "accept"; otherwise, Bob outputs "reject". Bob also computes
$$y_2 = \mathit{sig}_{Bob}(r_2)$$
and sends $\mathbf{Cert}(Bob)$ and y_2 to Alice.
4. Alice verifies Bob's public key, ver_{Bob}, on the certificate $\mathbf{Cert}(Bob)$. Then she checks that
$$\mathit{ver}_{Bob}(r_2, y_2) = true.$$
If so, then Alice outputs "accept"; otherwise, Alice outputs "reject".

The above scheme can be proven to be secure. However, it is instructive to consider a slight modification of the scheme, which turns out to be insecure. The modification is as follows:

Protocol 2.7: INSECURE PUBLIC-KEY MUTUAL AUTHENTICATION

1. Bob chooses a random challenge, r_1, which he sends to Alice.
2. Alice chooses a random challenge, r_2. She also computes

$$y_1 = sig_{Alice}(r_1 \parallel r_2)$$

   and sends $\mathbf{Cert}(Alice)$, r_2 and y_1 to Bob.
3. Bob verifies Alice's public key, ver_{Alice}, on the certificate $\mathbf{Cert}(Alice)$. Then he checks that

$$ver_{Alice}(r_1 \parallel r_2, y_1) = true.$$

   If so, then Bob outputs "accept"; otherwise, Bob outputs "reject". Bob also chooses a random number r_3, computes

$$y_2 = sig_{Bob}(r_2 \parallel r_3)$$

   and sends $\mathbf{Cert}(Bob)$, r_3 and y_2 to Alice.
4. Alice verifies Bob's public key, ver_{Bob}, on the certificate $\mathbf{Cert}(Bob)$. Then she checks that

$$ver_{Bob}(r_2 \parallel r_3, y_2) = true.$$

   If so, then Alice outputs "accept"; otherwise, Alice outputs "reject".

The modification incorporates a random value, r_3, which is chosen by Bob and signed by him (along with r_2) in the third flow of the protocol. However, including this extra piece of information in the signature makes the scheme insecure! This is easily seen by constructing a parallel session attack that is similar to the previous attack of this type:

$$
\begin{array}{ccccc}
\text{Alice} & & \text{Oscar} & & \text{Bob} \\
& \xleftarrow{\quad r_1 \quad} & & & \\
& \xrightarrow{sig_{Alice}(r_1 \parallel r_2),\, r_2} & & & \\
& & & \xrightarrow{\quad r_2 \quad} & \\
& & & \xleftarrow{MAC_{Bob}(r_2 \parallel r_3),\, r_3} & \\
& \xleftarrow{MAC_{Bob}(r_2 \parallel r_3),\, r_3} & & &
\end{array}
$$

The inclusion of the random value r_3 in the third flow makes the third flow similar syntactically to the second flow. This allows the parallel session attack to be carried out successfully.