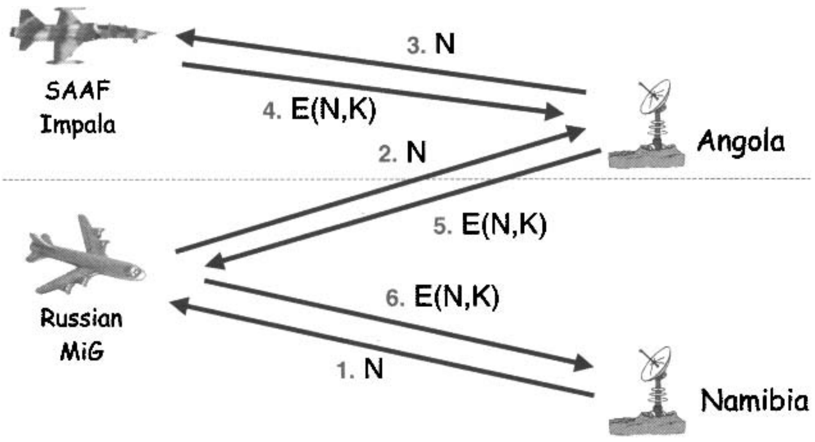

Figure 9.2: MiG-in-the-Middle

9.3 Authentication Protocols

"I can't explain myself, I'm afraid, Sir," said Alice,
"because I'm not myself you see."
— Lewis Carroll, *Alice in Wonderland*

Suppose that Alice must prove to Bob that she's Alice, where Alice and Bob are communicating over a network. Keep in mind that Alice can be a human or a machine, and ditto for Bob. In fact, in this networked scenario, Alice and Bob will almost invariably be machines, which has important implications that we'll consider in a moment.

In many cases, it's sufficient for Alice to prove her identity to Bob, without Bob proving his identity to Alice. But sometimes *mutual authentication* is necessary, that is, Bob must also prove his identity to Alice. It seems obvious that if Alice can prove her identity to Bob, then precisely the same protocol can be used in the other direction for Bob to prove his identity to Alice. We'll see that, in security protocols, the obvious approach is not always secure.

In addition to authentication, a *session key* is inevitably required. A session key is a symmetric key that will be used to protect the confidentiality and/or integrity of the current session, provided the authentication succeeds. Initially, we'll ignore the session key so that we can concentrate on authentication.

In certain situations, there may be other requirements placed on a security protocol. For example, we might require that the protocol use public keys, or symmetric keys, or hash functions. In addition, some situations might call for a protocol that provides anonymity or plausible deniability (discussed below) or other not-so-obvious features.

We've previously considered the security issues associated with authentication on standalone computer systems. While such authentication presents its own set of challenges (hashing, salting, etc.), from the protocol perspective, it's straightforward. In contrast, authentication over a network requires very careful attention to protocol issues. When a network is involved, numerous attacks are available to Trudy that are generally not a concern on a standalone computer. When messages are sent over a network, Trudy can passively observe the messages and she can conduct various active attacks such as replaying old messages, inserting, deleting, or changing messages. In this book, we haven't previously encountered anything comparable to these types of attacks.

Our first attempt at authentication over a network is the protocol in Figure 9.3. This three-message protocol requires that Alice (the client) first initiate contact with Bob (the server) and state her identity. Then Bob asks for proof of Alice's identity, and Alice responds with her password. Finally, Bob uses Alice's password to authenticate Alice.

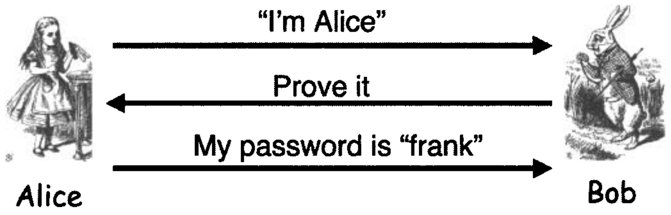

Figure 9.3: Simple Authentication

Although the protocol in Figure 9.3 is certainly simple, it has some major flaws. For one thing, if Trudy is able to observe the messages that are sent, she can later replay the messages to convince Bob that she is Alice, as illustrated in Figure 9.4. Since we are assuming these messages are sent over a network, this replay attack is a serious threat.

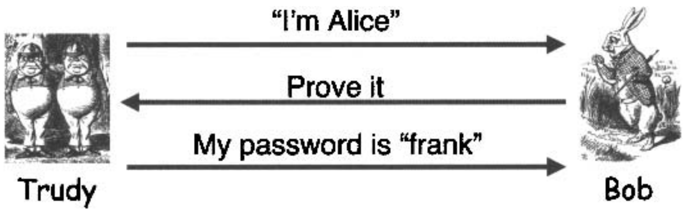

Figure 9.4: Replay Attack

Another issue with the too-simple authentication in Figure 9.3 is that Alice's password is sent in the clear. If Trudy observes the password when it is sent from Alice's computer, then Trudy knows Alice's password. This is even worse than a replay attack since Trudy can then pose as Alice on any site where Alice has reused this particular password. Another password issue with this protocol is that Bob must know Alice's password before he can authenticate her.

This simple authentication protocol is also inefficient, since the same effect could be accomplished in a single message from Alice to Bob. So, this protocol is a loser in every respect. Finally, note that the protocol in Figure 9.3 does not attempt to provide mutual authentication, which may be required in some cases.

For our next attempt at an authentication protocol, consider Figure 9.5. This protocol solves some of the problems of our previous simple authentication protocol. In this new-and-improved version, a passive observer, Trudy, will not learn Alice's password and Bob no longer needs to know Alice's password—although he must know the hash of Alice's password.

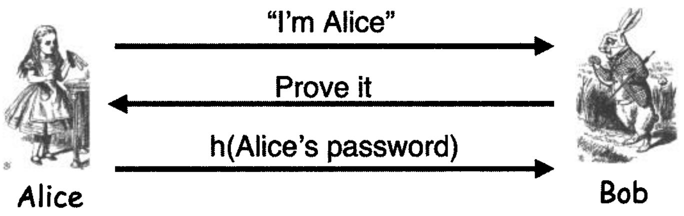

Figure 9.5: Simple Authentication with a Hash

The major flaw in the protocol of Figure 9.5 is that it's still subject to a replay attack, where Trudy records Alice's messages and later replays them to Bob. In this way, Trudy could be authenticated as Alice, without knowledge of Alice's password.

To authenticate Alice, Bob will need to employ a *challenge-response* mechanism. That is, Bob will send a challenge to Alice, and the response from Alice must be something that only Alice can provide and that Bob can verify. To prevent a replay attack, Bob can incorporate a "number used once," or *nonce*, in the challenge. That is, Bob will send a unique challenge each time, and the challenge will be used to compute the appropriate response. Bob can thereby distinguish the current response from a replay of a previous response. In other words, the nonce is used to ensure the freshness of the response. This approach to authentication with replay prevention is illustrated in Figure 9.6.

First, we'll design an authentication protocol using Alice's password. A password is something only Alice should know and Bob can verify—assuming that Bob knows Alice's password, that is.

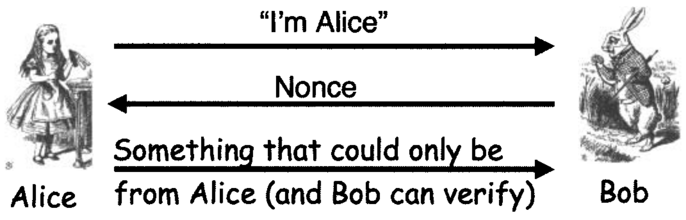
"I'm Alice"
Nonce
Something that could only be from Alice (and Bob can verify)
Alice
Bob

Figure 9.6: Generic Authentication

Our first serious attempt at an authentication protocol that is resistant to replay appears is Figure 9.7. In this protocol, the nonce sent from Bob to Alice is the challenge. Alice must respond with the hash of her password together with the nonce, which, assuming Alice's password is secure, serves to prove that the response was generated by Alice. Note that the nonce proves that the response is fresh and not a replay.

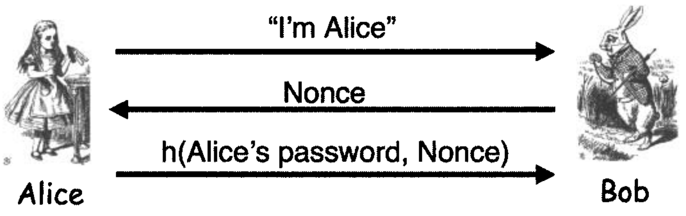

Figure 9.7: Challenge-Response

One problem with the protocol in Figure 9.7 is that Bob must know Alice's password. Furthermore, Alice and Bob typically represent machines rather than users, so it makes no sense to use passwords. After all, passwords are little more than a crutch used by humans because we are incapable of remembering keys. That is, passwords are about the closest thing to a key that humans can remember. So, if Alice and Bob are actually machines, they should be using keys instead of passwords.

9.3.1 Authentication Using Symmetric Keys

Having liberated ourselves from passwords, let's design a secure authentication protocol based on symmetric key cryptography. Recall that our notation for encrypting is $C = E(P, K)$ where P is plaintext, K is the key, and C is the ciphertext, while the notation for decrypting is $P = D(C, K)$. When discussing protocols, we are primarily concerned with attacks on protocols, not attacks on the cryptography used in protocols. Consequently, in this chapter we'll assume that the underlying cryptography is secure.

Suppose that Alice and Bob share the symmetric key K_{AB}. As in symmetric cryptography, we assume that nobody else has access to K_{AB}. Alice will authenticate herself to Bob by proving that she knows the key, without revealing the key to Trudy. In addition, the protocol must provide protection against a replay attack.

Our first symmetric key authentication protocol appears in Figure 9.8. This protocol is analogous to our previous password-based challenge-response protocol, but instead of hashing a nonce with a password, we've encrypted the nonce R with the shared symmetric key K_{AB}.

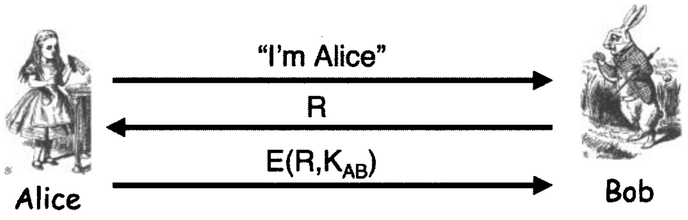

Figure 9.8: Symmetric Key Authentication Protocol

The symmetric key authentication protocol in Figure 9.8 allows Bob to authenticate Alice, since Alice can encrypt R with K_{AB}, Trudy cannot, and Bob can verify that the encryption was done correctly—Bob knows K_{AB}. This protocol prevents a replay attack, thanks to the nonce R, which ensures that each response is fresh. The protocol lacks mutual authentication, so our next task will be to develop a mutual authentication protocol based on symmetric keys.

Our first attempt at mutual authentication appears in Figure 9.9. This protocol is certainly efficient, and it does use symmetric key cryptography, but it has an obvious flaw. The third message in this protocol is simply a replay of the second, and consequently it proves nothing about the sender, be it Alice or Trudy.

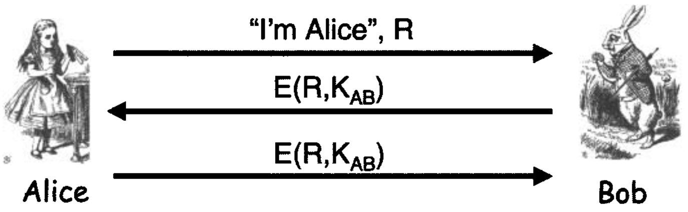

Figure 9.9: Mutual Authentication?

A more plausible approach to mutual authentication would be to use the secure authentication protocol in Figure 9.8 and repeat the process twice, once for Bob to authenticate Alice and once more for Alice to authenticate Bob. We've illustrated this approach in Figure 9.10, where we've combined some messages for the sake of efficiency.

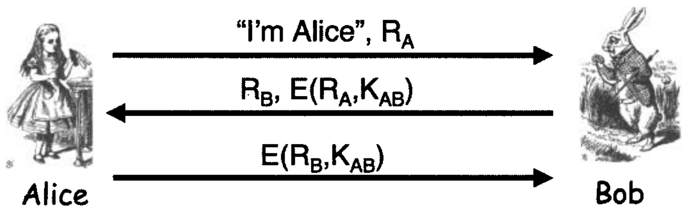

Figure 9.10: Secure Mutual Authentication?

Perhaps surprisingly, the protocol in Figure 9.10 is insecure—it is subject to an attack that is analogous to the MiG-in-the-middle attack discussed previously. In this attack, which is illustrated in Figure 9.11, Trudy initiates a conversation with Bob by claiming to be Alice and sends a challenge R_A to Bob. Following the protocol, Bob encrypts the challenge R_A and sends it, along with his challenge R_B, to Trudy. At this point Trudy appears to be stuck, since she doesn't know the key K_{AB}, and therefore she can't respond appropriately to Bob's challenge. However, Trudy cleverly opens a new connection to Bob where she again claims to be Alice and this time sends Bob his own "random" challenge R_B. Bob, following the protocol, responds with $E(R_B, K_{AB})$, which Trudy can now use to complete the first connection. Trudy can leave the second connection to time out, since she has—in the first connection—convinced Bob that she is Alice.

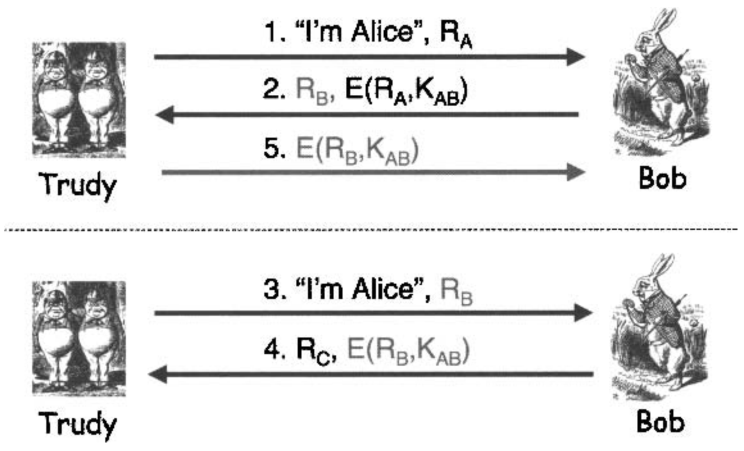

Figure 9.11: Trudy's Attack

The conclusion is that a non-mutual authentication protocol may not be secure for mutual authentication. Another conclusion is that protocols (and attacks on protocols) can be subtle. Yet another conclusion is that "obvious" changes to protocols can cause unexpected security problems.

In Figure 9.12, we've made a couple of minor changes to the insecure mutual authentication protocol of Figure 9.10. In particular, we've encrypted the user's identity together with the nonce. This change is sufficient to prevent Trudy's previous attack since she cannot use a response from Bob for the third message—Bob will realize that he encrypted it himself.

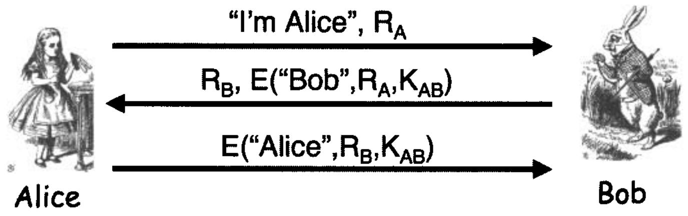

Figure 9.12: Strong Mutual Authentication Protocol

One lesson here is that it's a bad idea to have the two sides in a protocol do exactly the same thing, since this might open the door to an attack. Another lesson is that small changes to a protocol can result in big changes in its security.

9.3.2 Authentication Using Public Keys

In the previous section we devised a secure mutual authentication protocol using symmetric keys. Can we accomplish the same thing using public key cryptography? First, recall our public key notation. Encrypting a message M with Alice's public key is denoted $C = \{M\}_{\text{Alice}}$ while decrypt C with Alice's private key, and thereby recovering the plaintext M, is denoted $M = [C]_{\text{Alice}}$. Signing is also a private key operation. Of course, encryption and decryption are inverse operation, as are signing and signature verification, that is

$$[\{M\}_{\text{Alice}}]_{\text{Alice}} = M \text{ and } \{[M]_{\text{Alice}}\}_{\text{Alice}} = M.$$

It's always important to remember that in public key cryptography, anybody can do public key operations, while only Alice can use her private key.[4]

Our first attempt at authentication using public key cryptography appears in Figure 9.13. This protocol allows Bob to authenticate Alice, since only Alice can do the private key operation that is necessary to reply with R in the third message. Also, assuming that the nonce R is chosen (by Bob) at

[4]Repeat to yourself 100 times: The public key is public.

random, a replay attack is not feasible. That is, Trudy cannot replay R from a previous iteration of the protocol, since the random challenge will almost certainly not be the same in a subsequent iteration.

However, if Alice uses the same key pair to encrypt as she uses for authentication, then there is a potential problem with the protocol in Figure 9.13. Suppose Trudy has previously intercepted a message encrypted with Alice's public key, say, $C = \{M\}_{\text{Alice}}$. Then Trudy can pose as Bob and send C to Alice in message two, and Alice will decrypt it and send the plaintext back to Trudy. From Trudy's perspective, it doesn't get any better than that. The moral of the story is that you should not use the same key pair for signing as you use for encryption.

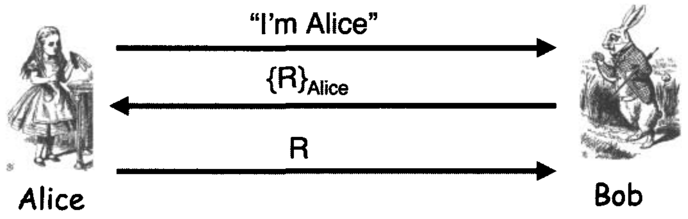

Figure 9.13: Authentication with Public Key Encryption

The authentication protocol in Figure 9.13 uses public key encryption. Is it possible to accomplish the same feat using digital signatures? In fact, it is, as illustrated in Figure 9.14.

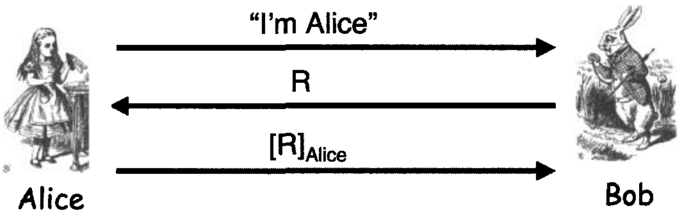

Figure 9.14: Authentication via Digital Signature

The protocol in Figure 9.14 has similar security issues as the public key encryption protocol in Figure 9.13. In Figure 9.14, if Trudy can pose as Bob, she can get Alice to sign anything. Again, the solution to this problem is to always use different key pairs for signing and encryption. Finally, note that, from Alice's perspective, the protocols in Figures 9.13 and 9.14 are identical, since in both cases she applies her private key to whatever shows up in message two.

9.3.3 Session Keys

Along with authentication, we invariably require a session key. Even when a symmetric key is used for authentication, we want to use a distinct session keys to encrypt data within each connection. The purpose of a session key is to limit the amount of data encrypted with any one particular key, and it also serves to limit the damage if one session key is compromised. A session key is used to provide confidentiality or integrity protection (or both) to the messages.

We want to establish the session key as part of the authentication protocol. That is, when the authentication is complete, we will also have securely established a shared symmetric key. Therefore, when analyzing an authentication protocol, we need to consider attacks on the authentication itself, as well as attacks on the session key.

Our next goal is to design an authentication protocol that also provides a shared symmetric key. It looks to be straightforward to include a session key in our secure public key authentication protocol. Such a protocol appears in Figure 9.15.

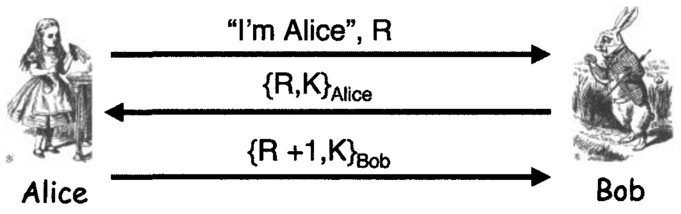

Figure 9.15: Authentication and a Session Key

One possible concern with the protocol of Figure 9.15 is that it does not provide for mutual authentication—only Alice is authenticated.[5] But before we tackle that issue, can we modify the protocol in Figure 9.15 so that it uses digital signatures instead of public key encryption? This also seems straightforward, and the result appears in Figure 9.16.

However, there is a fatal flaw in the protocol of Figure 9.16. Since the key is signed, anybody can use Bob's (or Alice's) public key and find the session key K. A session key that is public knowledge is definitely not secure. But before we dismiss this protocol entirely, note that it does provide mutual authentication, whereas the public key encryption protocol in Figure 9.15 does not. Can we combine these protocols so as to achieve both mutual

[5]One strange thing about this protocol is that the key K acts as Bob's challenge to Alice and the nonce R is useless. But there is a method to the madness, which will become clear shortly.

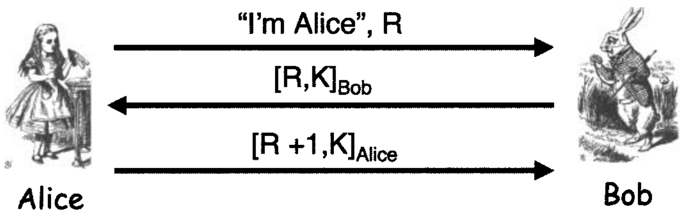

Figure 9.16: Signature-Based Authentication and Session Key

authentication and a secure session key? The answer is yes, and there are a couple of ways to do so.

Suppose that, instead of signing or encrypting the messages, we sign and encrypt the messages. Figure 9.17 illustrates such a sign and encrypt protocol. This appears to provide the desired secure mutual authentication and a secure session key.

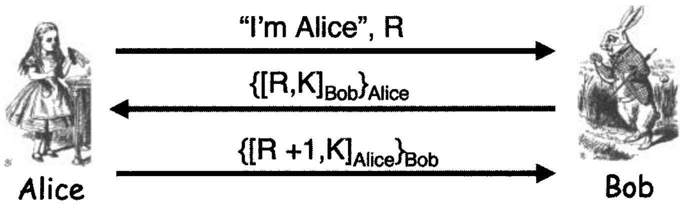

Figure 9.17: Mutual Authentication and Session Key

Since the protocol in Figure 9.17 provides mutual authentication and a session key using sign and encrypt, surely encrypt and sign must work, too. An encrypt and sign protocol appears in Figure 9.18.

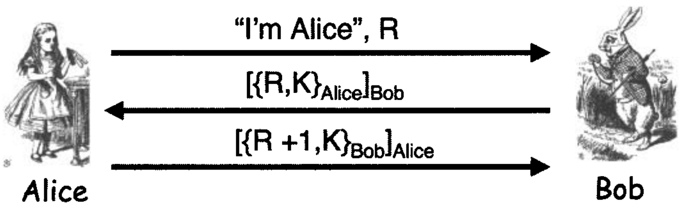

Figure 9.18: Encrypt and Sign Mutual Authentication

Note that the values $\{R, K\}_{\text{Alice}}$ and $\{R + 1, K\}_{\text{Bob}}$ in Figure 9.18 are available to anyone who has access to Alice's or Bob's public keys (which, by assumption, is anybody who wants them). Since this is not the case in Figure 9.17, it might seem that sign and encrypt somehow reveals less information than encrypt and sign. However, it appears that an attacker must break the public key encryption to recover K in either case and, if so, there is no security difference between the two. Recall that when analyzing protocols, we assume all crypto is strong, so breaking the encryption is not an option for Trudy.

9.3.4 Perfect Forward Secrecy

Now that we have conquered mutual authentication and session key establishment (using public keys), we turn our attention to *perfect forward secrecy*, or PFS. What is PFS? Rather than answer directly, we'll look at an example that illustrates what PFS is not. Suppose that Alice encrypts a message with a shared symmetric key K_{AB} and sends the resulting ciphertext to Bob. Trudy can't break the cipher to recover the key, so out of desperation she simply records all of the messages encrypted with the key K_{AB}. Now suppose that at some point in the future Trudy manages to get access to Alice's computer, where she finds the key K_{AB}. Then Trudy can decrypt the recorded ciphertext messages. While such an attack may seem unlikely, the problem is potentially significant since, once Trudy has recorded the ciphertext, the encryption key remains a vulnerability into the future. To avoid this problem, Alice and Bob must both destroy all traces of K_{AB} once they have finished using it. This might not be as easy as it seems, particularly if K_{AB} is a long-term key that Alice and Bob will need to use in the future. Furthermore, even if Alice is careful and properly manages her keys, she would have to rely on Bob to do the same (and vice versa).

PFS makes such an attack impossible. That is, even if Trudy records all ciphertext messages and she later recovers all long-term secrets (symmetric keys and/or private keys), she cannot decrypt the recorded messages. While it might seem that this is an impossibility, it is not only possible, but actually fairly easy to achieve in practice.

Suppose Bob and Alice share a long-term symmetric key K_{AB}. Then if they want PFS, they definitely can't use K_{AB} as their encryption key. Instead, Alice and Bob must agree on a session key K_S and forget K_S after it's no longer needed, i.e., after the current session ends. So, as in our previous protocols, Alice and Bob must find a way to agree on a session key K_S, by using their long-term symmetric key K_{AB}. However, for PFS we have the added condition that if Trudy later finds K_{AB}, she cannot determine K_S, even if she recorded all of the messages exchanged by Alice and Bob.

Suppose that Alice generates a session key K_S and sends $E(K_S, K_{AB})$ to Bob, that is, Alice simply encrypts the session key and sends it to Bob. If we are not concerned with PFS, this would be a sensible way to establish a session key in conjunction with an authentication protocol. However, this approach, which is illustrated in Figure 9.19, does not provide PFS. If Trudy records all of the messages and later recovers K_{AB}, she can decrypt $E(K_S, K_{AB})$ to recover the session key K_S, which she can then use to decrypt the recorded ciphertext messages. This is precisely the attack that PFS is supposed to prevent.

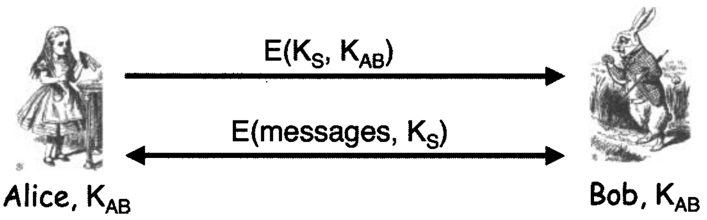

Figure 9.19: Naïve Attempt at PFS

There are actually several ways to achieve PFS, but the most elegant approach is to use an *ephemeral Diffie-Hellman* key exchange. As a reminder, the standard Diffie-Hellman key exchange protocol appears in Figure 9.20. In this protocol, g and p are public, Alice chooses her secret exponent a and Bob chooses his secret exponent b. Then Alice sends g^a mod p to Bob and Bob sends g^b mod p to Alice. Alice and Bob can each compute the shared secret g^{ab} mod p. Recall that the crucial weakness with Diffie-Hellman is that it is subject to a man-in-the-middle attack, as discussed in Section 4.4 of Chapter 4.

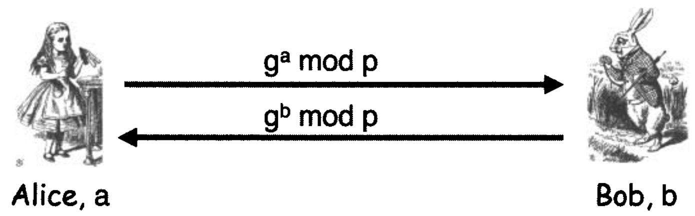

Figure 9.20: Diffie-Hellman

If we are to use Diffie-Hellman for PFS,[6] we must prevent the man-in-the-middle attack, and, of course, we must somehow assure PFS. The aforementioned ephemeral Diffie-Hellman can accomplish both. To prevent the MiM attack, Alice and Bob can use their shared symmetric key K_{AB} to encrypt the Diffie-Hellman exchange. Then to get PFS, all that is required is that, once Alice has computed the shared session key $K_S = g^{ab}$ mod p, she must forget

[6]Your acronym-loving author was tempted to call this protocol DH4PFS or maybe EDH4PFS but, for once, he showed restraint.

her secret exponent a and, similarly, Bob must forget his secret exponent b. This protocol is illustrated in Figure 9.21.

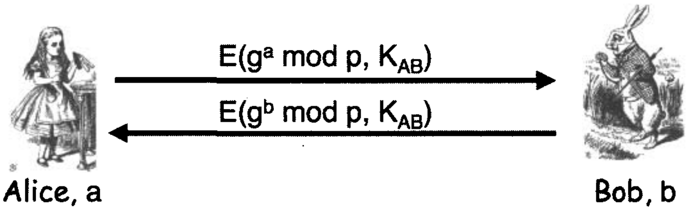

Figure 9.21: Ephemeral Diffie-Hellman for PFS

One interesting feature of the PFS protocol in Figure 9.21 is that once Alice and Bob have forgotten their respective secret exponents, even they can't reconstruct the session key K_S. If Alice and Bob can't recover the session key, certainly Trudy can be no better off. If Trudy records the conversation in Figure 9.21 and later is able to find K_{AB}, she will not be able to recover the session key K_S unless she can break Diffie-Hellman. Assuming the underlying crypto is strong, we have satisfied our requirements for PFS.

9.3.5 Mutual Authentication, Session Key, and PFS

Now let's put it all together and design a mutual authentication protocol that establishes a session key with PFS. The protocol in Figure 9.22, which is a slightly modified form of the encrypt and sign protocol from Figure 9.18, appears to fill the bill. It is a good exercise to give convincing arguments that Alice is actually authenticated (explaining exactly where and how that happens and why Bob is convinced he's talking to Alice), that Bob is authenticated, that the session key is secure, that PFS is provided, and that there are no obvious attacks.

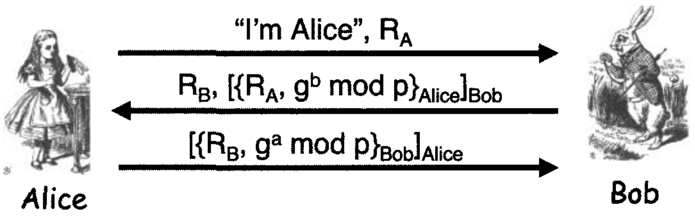
"I'm Alice", R_A
R_B, [{R_A, g^b mod p}_Alice]_Bob
[{R_B, g^a mod p}_Bob]_Alice
Alice
Bob

Figure 9.22: Mutual Authentication, Session Key and PFS

Now that we've developed a protocol that satisfies all of our security requirements, we can turn our attention to questions of efficiency. That is, we'll try to reduce the number of messages in the protocol or increase the efficiency in some other way, such as by reducing the number of public key operations.

9.3.6 Timestamps

A *timestamp T* is a time value, typically expressed in milliseconds. With some care, a timestamp can be used in place of a nonce, since a current timestamp ensures freshness. The benefit of a timestamp is that we don't need to waste any messages exchanging nonces, assuming that the current time is known to both Alice and Bob. Timestamps are used in many real-world security protocols, such as Kerberos, which we discuss in the next chapter.

Along with the potential benefit of increased efficiency, timestamps create some security issues as well.[7] For one thing, the use of timestamps implies that time is a security-critical parameter. For example, if Trudy can attack Alice's system clock (or whatever Alice relies on for the current time), she may cause Alice's authentication to fail. A related problem is that we can't rely on clocks to be perfectly synchronized, so we must allow for some *clock skew*, that is, we must accept any timestamp that is close to the current time. In general, this can open a small window of opportunity for Trudy to conduct a replay attack—if she acts within the allowed clock skew a replay will be accepted. It is possible to close this window completely, but the solution puts an additional burden on the server (see Problem 27). In any case, we would like to minimize the clock skew without causing excessive failures due to time inconsistencies between Alice and Bob.

To illustrate the benefit of a timestamp, consider the authentication protocol in Figure 9.23. This protocol is essentially the timestamp version of the sign and encrypt protocol in Figure 9.17. Note that by using a timestamp, we're able to reduce the number of messages by a third.

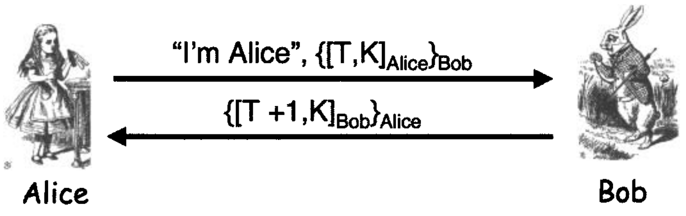

Figure 9.23: Authentication Using a Timestamp

The authentication protocol in Figure 9.23 uses a timestamp together with sign and encrypt and it appears to be secure. So it would seem obvious that the timestamp version of encrypt and sign must also be secure. This protocol is illustrated in Figure 9.24.

[7]This is yet another example of the "no free lunch" principle.

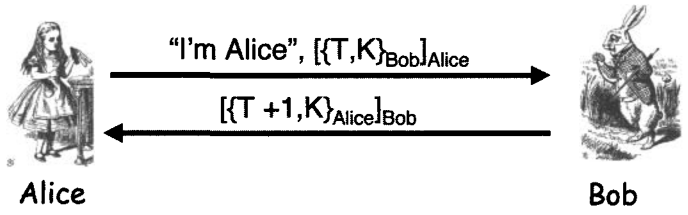

Figure 9.24: Encrypt and Sign Using a Timestamp

Unfortunately, with protocols, the obvious is not always correct. In fact, the protocol in Figure 9.24 is subject to attack. Trudy can recover $\{T, K\}_{\text{Bob}}$ by applying Alice's public key. Then Trudy can open a connection to Bob and send $\{T, K\}_{\text{Bob}}$ in message one, as illustrated in Figure 9.25. Following the protocol, Bob will then send the key K to Trudy in a form that Trudy can decrypt. This is not good, since K is the session key shared by Alice and Bob.

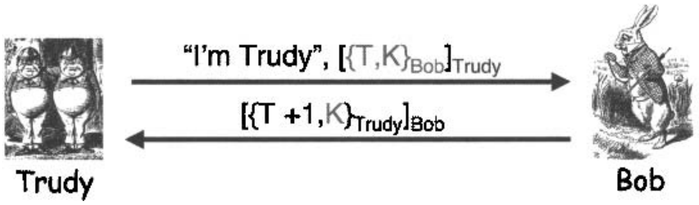

Figure 9.25: Trudy's Attack on Encrypt and Sign

The attack in Figure 9.25 shows that our encrypt and sign protocol is not secure when we use a timestamp. But our sign and encrypt protocol is secure when a timestamp is used. In addition, the nonce versions of both sign and encrypt as well as encrypt and sign are secure (see Figures 9.17 and 9.18). These examples nicely illustrate that, when it comes to security protocols, we should never take anything for granted.

Is the flawed protocol in Figure 9.24 fixable? In fact, there are several minor modifications that will make this protocol secure. For example, there's no reason to return the key K in the second message, since Alice already knows K and the only purpose of this message is to authenticate Bob. The timestamp in message two is sufficient to authenticate Bob. This secure version of the protocol is illustrated in Figure 9.26 (see also Problem 21).

In the next chapter, we'll discuss several well-known, real-world security protocols. These protocols use the concepts that we've presented in this chapter. But before moving on to the real world of Chapter 10, we briefly look at a couple of additional protocol topics. First, we'll consider a weak

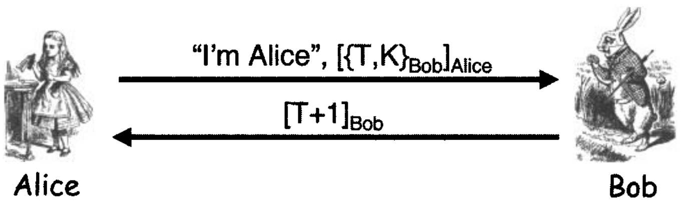

Figure 9.26: Secure Encrypt and Sign with a Timestamp

form of authentication that relies on TCP which, unfortunately, is sometimes used in practice. Finally, we discuss the Fiat-Shamir zero knowledge protocol. We'll encounter Fiat-Shamir again in the final chapter.

9.4 Authentication and TCP

In this section we'll take a quick look at how TCP is sometimes used for authentication. TCP was not designed to be used in this manner and, not surprisingly, this authentication method is not secure. But it does illustrate some interesting network security issues.

There is an undeniable temptation to use the IP address in a TCP connection for authentication.[8] If we could make this work, then we wouldn't need any of those troublesome keys or pesky authentication protocols.

Below, we'll give an example of TCP-based authentication and we illustrate an attack on the scheme. But first we briefly review the TCP three-way handshake, which is illustrated in Figure 9.27. The first message is a synchronization request, or SYN, whereas the second message, which acknowledges the synchronization request, is a SYN-ACK, and the third message—which can also contain data—acknowledges the previous message, and is simply known as an ACK.

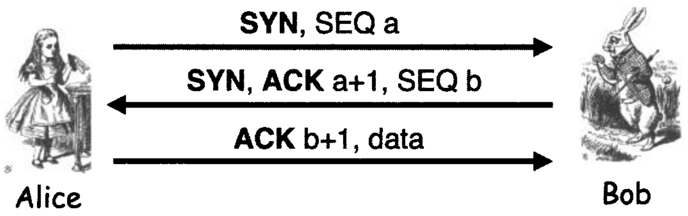

Figure 9.27: TCP 3-Way Handshake

[8]As we'll see in the next chapter, the IPSec protocol relies on the IP address for user identity in one of its modes. So, even people who should know better cannot always resist the temptation.