# Chapter 13
# Key Establishment

With the cryptographic mechanisms that we have learned so far, in particular symmetric and asymmetric encryption, digital signatures and message authentication codes (MACs), one can relatively easily achieve the basic security services (cf. Sect. 10.1.3):

- Confidentiality (with encryption algorithms)
- Integrity (with MACs or digital signatures)
- Message authentication (with MACs or digital signatures)
- Non-repudiation (with digital signatures)

Similarly, identification can be accomplished through protocols which make use of standard cryptographic primitives.

However, all cryptographic mechanisms that we have introduced so far assume that keys are properly distributed between the parties involved, e.g., between Alice and Bob. The task of key establishment is in practice one of the most important and often also most difficult parts of a security system. We already learned some ways of distributing keys, in particular Diffie–Hellman key exchange. In this chapter we will learn many more methods for establishing keys between remote parties. You will learn about the following important issues:

- How keys can be established using symmetric cryptosystems
- How keys can be established using public-key cryptosystems
- Why public-key techniques still have shortcomings for key distribution
- What certificates are and how they are used
- The role that public-key infrastructures play

C. Paar, J. Pelzl, *Understanding Cryptography*,

DOI 10.1007/978-3-642-04101-3_13, 

## 13.1 Introduction

In this section we introduce some terminology, some thoughts on key freshness and a very basic key distribution scheme. The latter is helpful for motivating the more advanced methods which will follow in this chapter.

### 13.1.1 Some Terminology

Roughly speaking, key establishment deals with establishing a shared secret between two or more parties. Methods for this can be classified into *key transport* and *key agreement* methods, as shown in Fig. 13.1. A key transport protocol is a technique where one party securely transfers a secret value to others. In a key agreement protocol two (or more) parties derive the shared secret where all parties contribute to the secret. Ideally, none of the parties can control what the final joint value will be.

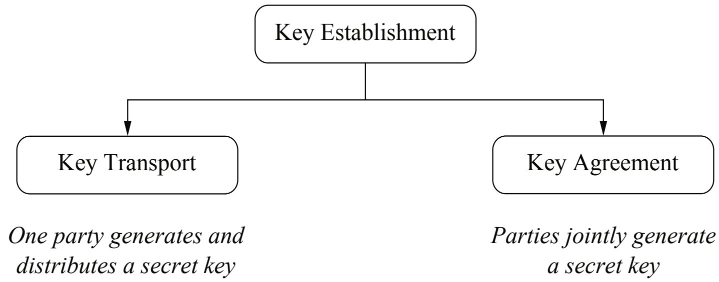


**Fig. 13.1** Classification of key establishment schemes

Key establishment itself is strongly related to identification. For instance, you may think of attacks by unauthorized users who join the key establishment protocol with the aim of masquerading as either Alice or Bob with the goal of establishing a secret key with the other party. To prevent such attacks, each party must be assured of the identity of the other entity. All of these issues are addressed in this chapter.

### 13.1.2 Key Freshness and Key Derivation

In many (but not all) security systems it is desirable to use cryptographic keys which are only valid for a limited time, e.g., for one Internet connection. Such keys are called *session keys* or *ephemeral keys*. Limiting the period in which a cryptographic key is used has several advantages. A major one is that there is less damage if the key is exposed. Also, an attacker has less ciphertext available that was generated under one key, which can make cryptographic attacks much more difficult. Moreover, an attacker is forced to recover several keys if he is interested in decrypting larger parts of plaintext. Real-world examples where session keys are frequently generated include voice encryption in GSM cell phones and video encryption in pay-TV satellite systems; in both cases new keys are generated within a matter of minutes or sometimes even seconds.

The security advantages of *key freshness* are fairly obvious. However, the question now is, how can key updates be realized? The first approach is to simply execute the key establishment protocols shown in this chapter over and over again. However, as we see later, there are always certain costs associated with key establishment, typically with respect to additional communication connections and computations. The latter holds especially in the case of public-key algorithms which are very computationally intensive.

The second approach to key update uses an already established joint secret key to *derive* fresh session keys. The principal idea is to use a key derivation function (KDF) as shown in Fig. 13.2. Typically, a non-secret parameter $r$ is processed together with the joint secret $k_{AB}$ between the users Alice and Bob.

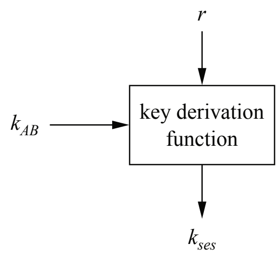


**Fig. 13.2** Principle of key derivation

An important characteristic of the key derivation function is that it should be a one-way function. The one-way property prevents an attacker from deducing $k_{AB}$ should any of the session keys become compromised, which in turn would allow the attacker to compute all other session keys.

One possible way of realizing the key derivation function is that one party sends a nonce, i.e., a numerical value that is used only once, to the other party. Both users encrypt the nonce using the shared secret key $k_{AB}$ by means of a symmetric cipher such as AES. The corresponding protocol is shown below.

**Key Derivation with Nonces**

| **Alice** | | **Bob** |
|---|---|---|
| | | generate nonce $r$ |
| | $\xleftarrow{\quad r \quad}$ | |
| derive key | | derive key |
| $k_{ses} = e_{k_{AB}}(r)$ | | $k_{ses} = e_{k_{AB}}(r)$ |

An alternative to encrypting the nonce is hashing it together with $k_{AB}$. One way of achieving this is that both parties perform a HMAC computation with the nonce serving as the "message":

$$k_{ses} = HMAC_{k_{AB}}(r)$$

Rather than sending a nonce, Alice and Bob can also simply encrypt a counter *cnt* periodically, where the ciphertext again forms the session key:

$$k_{ses} = e_{k_{AB}}(cnt)$$

or compute the HMAC of the counter:

$$k_{ses} = HMAC_{k_{AB}}(cnt)$$

Using a counter can save Alice and Bob one communication session because, unlike the case of the nonce-based key derivation, no value needs to be transmitted. However, this holds only if both parties know exactly when the next key derivation needs to take place. Otherwise, a counter synchronization message might be required.

### 13.1.3 The $n^2$ Key Distribution Problem

Until now we mainly assumed that the necessary keys for symmetric algorithms are distributed via a "secure channel", as depicted in the beginning of this book in Fig. 1.5. Distributing keys this way is sometimes referred to as *key predistribution* or *out-of-band transmission* since it typically involves a different mode (or band) of communication, e.g., the key is transmitted via a phone line or in a letter. Even though this seems somewhat clumsy, it can be a useful approach in certain practical situations, especially if the number of communicating parties is not too large. However, key predistribution quickly reaches its limits even if the number of entities in a network is only moderately large. This leads to the well-known $n^2$ key distribution problem.

We assume a network with $n$ users, where every party is capable of communicating with every other one in a secure fashion, i.e., if Alice wants to communicate with Bob, these two share a secret key $k_{AB}$ which is only known to them but not to any of the other $n-2$ parties. This situation is shown for the case of a network with $n = 4$ participants in Fig. 13.3.

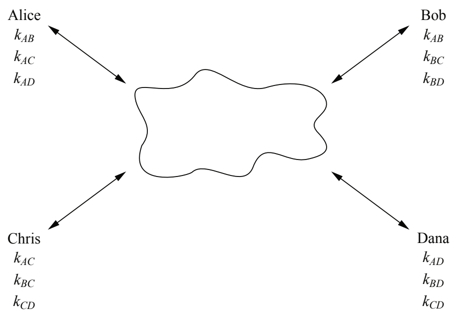


**Fig. 13.3** Keys in a network with $n = 4$ users

We can extrapolate several features of this simple scheme for the case of $n$ users:

- Each user must store $n-1$ keys.
- There is a total of $n(n-1) \approx n^2$ keys in the network.
- A total of $n(n-1)/2 = \binom{n}{2}$ symmetric key pairs are in the network.
- If a new user joins the network, a secure channel must be established with every other user in order to upload new keys.

The consequences of these observations are not very favorable if the number of users increases. The first drawback is that the number of keys in the system is roughly $n^2$. Even for moderately sized networks, this number becomes quite large. All these keys must be generated securely at one location, which is typically some type of trusted authority. The other drawback, which is often more serious in practice, is that adding one new user to the system requires updating the keys at all existing users. Since each update requires a secure channel, this is very burdensome.

*Example 13.1.* A mid-size company with 750 employees wants to set up secure e-mail communication with symmetric keys. For this purpose, $750 \times 749/2 = 280,875$ symmetric key pairs must be generated, and $750 \times 749 = 561,750$ keys must be distributed via secure channels. Moreover, if employee number 751 joins the company, all 750 other users must receive a key update. This means that 751 secure channels (to the 750 existing employees and to the new one) must be established.
$\diamond$

Obviously, this approach does not work for large networks. However, there are many cases in practice where the number of users is (i) small and (ii) does not change frequently. An example could be a company with a small number of branches which all need to communicate with each other securely. Adding a new branch does not happen too often, and if this happens it can be tolerated that one new key is uploaded to any of the existing branches.

## 13.2 Key Establishment Using Symmetric-Key Techniques

Symmetric ciphers can be used to establish secret (session) keys. This is somewhat surprising because we assumed for most of the book that symmetric ciphers themselves need a secure channel for establishing their keys. However, it turns out that it is in many cases sufficient to have a secure channel only when a new user joins the network. This is in practice often achievable for computer networks because at set-up time a (trusted) system administrator might be needed in person anyway who can install a secret key manually. In the case of embedded devices, such as cell phones, a secure channel is often given during manufacture, i.e., a secret key can be loaded into the device "in the factory".

The protocols introduced in the following all perform key transport and not key agreement.

### *13.2.1 Key Establishment with a Key Distribution Center*

The protocols developed in the following rely on a *Key Distribution Center (KDC)*. This is a server that is fully trusted by all users and that shares a secret key with each user. This key, which is named the *Key Encryption Key* (KEK), is used to securely transmit session keys to users.

#### Basic Protocol

A necessary prerequisite is that each user $U$ shares a unique secret key KEK $k_U$ with the key distribution center which predistributed through a secure channel. Let's look what happens if one party requests a secure session from the KDC, e.g., Alice wants to communicate with Bob. The interesting part of this approach is that the KDC **encrypts the session key** that will eventually be used by Alice and Bob. In a basic protocol, the KDC generates two messages, $y_A$ and $y_B$, for Alice and Bob, respectively:

$$y_A = e_{k_A}(k_{ses})$$
$$y_B = e_{k_B}(k_{ses})$$

Each message contains the session key encrypted with one of the two KEKs. The protocol looks like this:

**Basic Key Establishment Using a Key Distribution Center**

| **Alice** | | **KDC** | | **Bob** |
|---|---|---|---|---|
| KEK: $k_A$ | | KEK: $k_A$, $k_B$ | | KEK: $k_B$ |
| | $\xrightarrow{\text{RQST}(ID_A, ID_B)}$ | | | |
| | | generate random $k_{ses}$<br>$y_A = e_{k_A}(k_{ses})$<br>$y_B = e_{k_B}(k_{ses})$ | | |
| | $\xleftarrow{y_A}$ | | $\xrightarrow{y_B}$ | |
| $k_{ses} = e^{-1}_{k_A}(y_A)$ | | | | $k_{ses} = e^{-1}_{k_B}(y_B)$ |
| $y = e_{k_{ses}}(x)$ | $\xrightarrow{y}$ | | | $x = e^{-1}_{k_{ses}}(y)$ |

The protocol begins with a request message $\text{RQST}(ID_A, ID_B)$, where $ID_A$ and $ID_B$ simply indicate the users involved in the session. The actual key establishment protocol is executed subsequently in the upper part of the drawing. Below the solid line is, as an example, shown how Alice and Bob can now communicate with each other securely using the session key.

It is important to note that two types of keys are involved in the protocol. The KEKs $k_A$ and $k_B$ are long-term keys that do not change. The session key $k_{ses}$ is an ephemeral key that changes frequently, ideally for every communication session. **In order to understand this protocol more intuitively, one can view the predistributed KEKs as forming a secret channel between the KDC and each user.** With this interpretation, the protocol is straightforward: The KDC simply sends a session key to Alice and Bob via the two respective secret channels.

Since the KEKs are long-term keys, whereas the session keys have typically a much shorter lifetime, in practice sometimes different encryption algorithms are used with both. Let's consider the following example. In a pay-TV system AES might be used with the long-term KEKs $k_U$ for distributing session keys $k_{ses}$. The session keys might only have a lifetime of, say, one minute. The session keys are used to encrypt the actual plaintext (the digital TV signal in this example) with a fast stream cipher. A stream cipher might be required to assure real-time decryption. The advantage of this arrangement is that even if a session key becomes compromised, only one minute's worth of multimedia data can be decrypted by an adversary. Thus, the cipher that is used with the session key does not necessarily need to have the same cryptographic strength as the algorithm which is used for distributing the session keys. On the other hand, if one of the KEKs becomes compromised, all prior and future traffic can be decrypted by an eavesdropper.

It is easy to modify the above protocol such that we save one communication session. This is shown in the following:

**Key Establishment Using a Key Distribution Center**

| **Alice** | | **KDC** | | **Bob** |
|---|---|---|---|---|
| KEK: $k_A$ | | KEK: $k_A, k_B$ | | KEK: $k_B$ |
| | $\xrightarrow{\mathrm{RQST}(ID_A,ID_B)}$ | | | |
| | | generate random $k_{ses}$<br>$y_A = e_{k_A}(k_{ses})$<br>$y_B = e_{k_B}(k_{ses})$ | | |
| | $\xleftarrow{y_A,y_B}$ | | | |
| $k_{ses} = e^{-1}_{k_A}(y_A)$<br>$y = e_{k_{ses}}(x)$ | | | | |
| | $\xrightarrow{y,y_B}$ | | | |
| | | | | $k_{ses} = e^{-1}_{k_B}(y_B)$<br>$x = e^{-1}_{k_{ses}}(y)$ |

Alice receives the session key encrypted with both KEKs, $k_A$ and $k_B$. She is able to compute the session key $k_{ses}$ from $y_A$ and can use it subsequently to encrypt the actual message she wants to send to Bob. The interesting part of the protocol is that Bob receives both the encrypted message $y$ as well as $y_B$. He needs to decrypt the latter one in order to recover the session key which is needed for computing $x$.

Both of the KDC-based protocols have the advantage that there are only $n$ long-term symmetric key pairs in the system, unlike the first naïve scheme that we encountered, where about $n^2/2$ key pairs were required. The $n$ long-term KEKS only need to be stored by the KDC, while each user only stores his or her own KEK. Most importantly, if a new user Noah joins the network, a secure channel only needs to be established once between the KDC and Noah to distribute the KEK $k_N$.

## Security

Even though the two protocols protect against a passive attacker, i.e, an adversary that can only eavesdrop, there are attacks if an adversary can actively manipulate messages and create faked ones.

**Replay Attack** One weakness is that a *replay attack* is possible. This attack makes use of the fact that neither Alice nor Bob know whether the encrypted session key they receive is actually a new one. If an old one is reused, key freshness is violated. This can be a particularly serious issue if an old session key has become compromised. This could happen if an old key is leaked, e.g., through a hacker, or if the encryption algorithm used with an old key has become insecure due to cryptanalytical advances.

If Oscar gets hold of a previous session key, he can impersonate the KDC and resend old messages $y_A$ and $y_B$ to Alice and Bob. Since Oscar knows the session key, he can decipher the plaintext that will be encrypted by Alice or Bob.

**Key Confirmation Attack** Another weakness of the above protocol is that Alice is not assured that the key material she receives from the KDC is actually for a session between her and Bob. This attack assumes that Oscar is also a legitimate (but malicious) user. By changing the session-request message Oscar can trick the KDC and Alice to set up session between him and Alice as opposed to between Alice and Bob. Here is the attack:

**Key Confirmation Attack**

| **Alice** | | **Oscar** | | **KDC** | **Bob** |
|---|---|---|---|---|---|
| KEK: $k_A$ | | KEK: $k_O$ | | KEK: $k_A, k_B, k_O$ | KEK: $k_B$ |
| | $\xrightarrow{\text{RQST}(ID_A, ID_B)}$ | | | | |
| | | ↯ substitute | | | |
| | | | $\xrightarrow{\text{RQST}(ID_A, ID_O)}$ | | |
| | | | | random $k_{ses}$<br>$y_A = e_{k_A}(k_{ses})$<br>$y_O = e_{k_O}(k_{ses})$ | |
| | | | $\xleftarrow{y_A, y_O}$ | | |
| $k_{ses} = e^{-1}_{k_A}(y_A)$<br>$y = e_{k_{ses}}(x)$ | | | | | |
| | $\xrightarrow{y, y_O}$ | | | | |
| | | ↯ intercept<br>$k_{ses} = e^{-1}_{k_O}(y_O)$<br>$x = e^{-1}_{k_{ses}}(y)$ | | | |

The gist of the attack is that the KDC believes Alice requests a key for a session between Alice and Oscar, whereas she really wants to communicate with Bob. Alice assumes that the encrypted key "$y_O$" is "$y_B$", i.e., the session key encrypted under Bob's KEK $k_B$. (Note that if the KDC puts a header $ID_O$ in front of $y_O$ which associates it with Oscar, Oscar might simply change the header to $ID_B$.) In other words, Alice has no way of knowing that the KDC prepared a session with her and Oscar; instead she still thinks she is setting up a session with Bob. Alice continues with the protocol and encrypts her actual message as $y$. If Oscar intercepts $y$, he can decrypt it.

The underlying problem for this attack is that there is *no* key confirmation. If key confirmation were given, Alice would be assured that Bob and no other user knows the session key.

### *13.2.2 Kerberos*

A more advanced protocol that protects against both replay and key confirmation attacks is Kerberos. It is, in fact, more than a mere key distribution protocol; its main purpose is to provide user authentication in computer networks. Kerberos was standardized as an RFC 1510 in 1993 and is in widespread use. It is also based on a KDC, which is named the “authentication sever” in Kerberos terminology. Let’s first look at a simplified version of the protocol.

**Key Establishment Using a Simplified Version of Kerberos**

| **Alice** | | **KDC** | | **Bob** |
|---|---|---|---|---|
| KEK: $k_A$ | | KEK: $k_A$, $k_B$ | | KEK: $k_B$ |
| generate nonce $r_A$ | | | | |
| | $\xrightarrow{\mathrm{RQST}(ID_A, ID_B, r_A)}$ | | | |
| | | generate random $k_{ses}$<br>generate lifetime $T$<br>$y_A = e_{k_A}(k_{ses}, r_A, T, ID_B)$<br>$y_B = e_{k_B}(k_{ses}, ID_A, T)$ | | |
| | $\xleftarrow{y_A, y_B}$ | | | |
| $k_{ses}, r'_A, T, ID_B = e^{-1}_{k_A}(y_A)$<br>verify $r'_A = r_A$<br>verify $ID_B$<br>verify lifetime $T$<br>generate time stamp $T_S$<br>$y_{AB} = e_{k_{ses}}(ID_A, T_S)$ | | | | |
| | $\xrightarrow{y_{AB}, y_B}$ | | | |
| | | | | $k_{ses}, ID_A, T = e^{-1}_{k_B}(y_B)$<br>$ID_A', T_S = e^{-1}_{k_{ses}}(y_{AB})$<br>verify $ID_A' = ID_A$<br>verify lifetime $T$<br>verify time stamp $T_S$ |
| $y = e_{k_{ses}}(x)$ | $\xrightarrow{y}$ | | | $x = e^{-1}_{k_{ses}}(y)$ |

Kerberos assures the *timeliness* of the protocol through two measures. First, the KDC specifies a lifetime $T$ for the session key. The lifetime is encrypted with both session keys, i.e., it is included in $y_A$ and $y_B$. Hence, both Alice and Bob are aware of the period during which they can use the session key. Second, Alice uses a time stamp $T_S$, through which Bob can be assured that Alice’s messages are recent and are not the result of a replay attack. For this, Alice’s and Bob’s system clocks must be synchronized, but not with a very high accuracy. Typical values are in the range of a few minutes. The usage of the lifetime parameter $T$ and the time stamp $T_S$ prevent replay attacks by Oscar.

Equally important is that Kerberos provides key confirmation and user authentication. In the beginning, Alice sends a random nonce $r_A$ to the KDC. This can be considered as a *challenge* because she challenges the KDC to encrypt it with their joint KEK $k_A$. If the returned challenge $r'_A$ matches the sent one, Alice is assured that the message $y_A$ was actually sent by the KDC. This method to authenticate users is known as *challenge-response protocol* and is widely used, e.g., for authentication of smart cards.

Through the inclusion of Bob’s identity $ID_B$ in $y_A$ Alice is assured that the session key is actually meant for a session between herself and Bob. With the inclusion of Alice’s identity $ID_A$ in both $y_B$ and $y_{AB}$, Bob can verify that (i) the KDC included a session key for a connection between him and Alice and (ii) that he is currently actually talking to Alice.

### *13.2.3 Remaining Problems with Symmetric-Key Distribution*

Even though Kerberos provides strong assurance that the correct keys are being used and that users are authenticated, there are still drawbacks to the protocols discussed so far. We now describe remaining general problems that exist for KDC-based schemes.

**Communication requirements** One problem in practice is that the KDC needs to be contacted if a new secure session is to be initiated between any two parties in the network. Even though this is a performance rather than a security problem, it can be a serious hindrance in a system with very many users. In Kerberos, one can alleviate this potential problem by increasing the lifetime $T$ of the key. In practice, Kerberos can run with tens of thousands of users. However, it would be a problem to scale such an approach to "all" Internet users.

**Secure channel during initialization** As discussed earlier, all KDC-based protocols require a secure channel at the time a new user joins the network for transmitting that user's key encryption key.

**Single point of failure** All KDC-based protocols, including Kerberos, have the security drawback that they have a *single point of failure*, namely the database that contains the key encryption keys, the KEKs. If the KDC becomes compromised, all KEKs in the entire system become invalid and have to be re-established using secure channels between the KDC and each user.

**No perfect forward secrecy** If any of the KEKs becomes compromised, e.g., through a hacker or Trojan software running on a user's computer, the consequences are serious. First, all future communication can be decrypted by the attacker who eavesdrops. For instance, if Oscar got a hold of Alice's KEK $k_A$, he can recover the session key from all messages $y_A$ that the KDC sends out. **Even more dramatic is the fact that Oscar can also decrypt past communications if he stored old messages $y_A$ and $y$.** Even if Alice immediately realizes that her KEK has been compromised and she stops using it right away, there is nothing she can do to prevent Oscar from decrypting her *past* communication. Whether a system is vulnerable if long-term keys are compromised is an important feature of a security system and there is a special terminology used:

**Definition 13.1.** A cryptographic protocol has *perfect forward secrecy* (PFS) if the compromise of long-term keys does not allow an attacker to obtain past session keys.

Neither Kerberos nor the simpler protocols shown earlier offer PFS. The main mechanism to assure PFS is to employ public-key techniques, which we study in the following sections.

## 13.3 Key Establishment Using Asymmetric Techniques

Public-key algorithms are especially suited for key establishment protocols since they don't share most of the drawbacks that symmetric key approaches have. In fact, next to digital signatures, key establishment is the other major application domain of public-key schemes. They can be used for both key transport and key agreement. For the former, Diffie–Hellman key exchange, elliptic curve Diffie–Hellman or related protocols are often used. For key transport, any of the public-key encryption schemes, e.g., RSA or Elgamal, is often used. We recall at this point that public-key primitives are quite slow, and that for this reason actual data encryption is usually done with symmetric primitives like AES or 3DES, after a key has been established using asymmetric techniques.

At this moment it looks as though public-key schemes solve all key establishment problems. It turns out, however, that they all require what is termed an *authenticated channel* to distribute the public keys. The remainder of this chapter is chiefly devoted to solving the problem of authenticated public key distribution.

### *13.3.1 Man-in-the-Middle Attack*

The *man-in-the-middle attack*[1] is a serious attack against public-key algorithms. The basic idea of the attack is that the adversary, Oscar, replaces the public keys sent out by the participants with his own keys. This is possible whenever public keys are not authenticated. The man-in-the-middle (MIM) attack has far-reaching consequences for asymmetric cryptography. For didactical reasons we will study the MIM attack against the Diffie–Hellman key exchange (DHKE). However, it is extremely important to bear in mind that the attack is applicable against any asymmetric scheme unless the public-keys are protected, e.g., through certificates, a topic that is discussed in Sect. 13.3.2.

We recall that the DHKE allows two parties who never met before to agree on a shared secret by exchanging messages over an insecure channel. For convenience, we restate the DHKE protocol here:

[1] The "man-in-the-middle attack" should not be confused with the similarly sounding but in fact entirely different "meet-in-the-middle attack" against block ciphers which was introduced in Sect. 5.3.1.

**Diffie–Hellman Key Exchange**

| **Alice** | | **Bob** |
|---|---|---|
| choose random $a = k_{pr,A}$<br>compute $A = k_{pub,A} \equiv \alpha^a \bmod p$ | | choose random $b = k_{pr,B}$<br>compute $B = k_{pub,B} \equiv \alpha^b \bmod p$ |
| | $\xrightarrow{\quad A \quad}$ | |
| | $\xleftarrow{\quad B \quad}$ | |
| $k_{AB} \equiv B^a \bmod p$ | | $k_{AB} \equiv A^b \bmod p$ |

As we discussed in Sect. 8.4, if the parameters are chosen carefully, which includes especially a prime $p$ with a length of 1024 or more bit, the DHKE is secure against eavesdropping, i.e., passive attacks. We consider now the case that an adversary is not restricted to only listening to the channel. Rather, Oscar can also actively take part in the message exchange by intercepting, changing and generating messages. The underlying idea of the MIM attack is that Oscar replaces both Alice's and Bob's public key by his own. The attack is shown here:

**Man-in-the-Middle Attack Against the DHKE**

| **Alice** | | **Oscar** | | **Bob** |
|---|---|---|---|---|
| choose $a = k_{pr,A}$<br>$A = k_{pub,A} \equiv \alpha^a \bmod p$ | | | | choose $b = k_{pr,B}$<br>$B = k_{pub,B} \equiv \alpha^b \bmod p$ |
| | $\xrightarrow{\quad A \quad}$ ↯ | substitute $\tilde{A} \equiv \alpha^o$ | $\xrightarrow{\quad \tilde{A} \quad}$ | |
| | $\xleftarrow{\quad \tilde{B} \quad}$ ↯ | substitute $\tilde{B} \equiv \alpha^o$ | $\xleftarrow{\quad B \quad}$ | |
| $k_{AO} \equiv (\tilde{B})^a \bmod p$ | | $k_{AO} \equiv A^o \bmod p$<br>$k_{BO} \equiv B^o \bmod p$ | | $k_{BO} \equiv (\tilde{A})^b \bmod p$ |

Let's look at the keys that are being computed by the three players, Alice, Bob and Oscar. The key Alice computes is:

$$k_{AO} = (\tilde{B})^a \equiv (\alpha^o)^a \equiv \alpha^{oa} \bmod p$$

which is identical to the key that Oscar computes as $k_{AO} = A^o \equiv (\alpha^a)^o \equiv \alpha^{ao} \bmod p$. At the same time Bob computes:

$$k_{BO} = (\tilde{A})^b \equiv (\alpha^o)^b \equiv \alpha^{ob} \bmod p$$

which is identical to Oscar's key $k_{BO} = B^o \equiv (\alpha^b)^o \equiv \alpha^{bo} \bmod p$. Note that the two malicious keys that Oscar sends out, $\tilde{A}$ and $\tilde{B}$, are in fact the same values. With use different names here merely to stress the fact that Alice and Bob assume that they have received each other‘s public keys.

What happens in this attack is that two DHKEs are being performed simultaneously, one between Alice and Oscar and another one between Bob and Oscar. As a result, Oscar has established a joined key with Alice, which we termed $k_{AO}$, and another one with Bob, which we named $k_{BO}$. **However, neither Alice nor Bob is aware of the fact that they share a key with Oscar and not with each other!** Both assume that they have computed a joint key $k_{AB}$.

From here on, Oscar has much control over encrypted traffic between Alice and Bob. As an example, here is how he can read encrypted messages in a way that goes unnoticed by Alice and Bob:

**Message Manipulation After a Man-in-the-Middle Attack**

| **Alice** | | **Oscar** | | **Bob** |
|---|---|---|---|---|
| message $x$ | | | | |
| $y = AES_{k_{AO}}(x)$ | | | | |
| | $\xrightarrow{y}$ | $\lightning$ intercept | | |
| | | decrypt $x = AES^{-1}_{k_{AO}}(x)$ | | |
| | | re-encrypt $y' = AES_{k_{BO}}(x)$ | | |
| | | | $\xrightarrow{y'}$ | |
| | | | | decrypt $x = AES^{-1}_{k_{BO}}(y')$ |

For illustrative purposes, we assumed that AES is used for the encryption. Of course, any other symmetric cipher can be used as well. Please note that Oscar can not only read the plaintext $x$ but can also alter it prior to re-encrypting it with $k_{BO}$. This can have serious consequences, e.g., if the message $x$ describes a financial transaction.

## *13.3.2 Certificates*

The underlying problem of the man-in-the-middle attack is that public keys are not authenticated. We recall from Sect. 10.1.3 that message authentication ensures that the sender of a message is authentic. However, in the scenario at hand Bob receives a public key which is supposedly Alice's, but he has no way of knowing whether that is in fact the case. To make this point clear, let's examine how a key of a user Alice would look in practice:

$$k_A = (k_{pub,A}, ID_A),$$

where $ID_A$ is identifying information, e.g., Alice's IP address or her name together with date of birth. The actual public key $k_{pub,A}$, however, is a mere binary string, e.g., 2048 bit. If Oscar performs a MIM attack, he would change the key to:

$$k_A = (k_{pub,O}, ID_A).$$

Since everything is unchanged except the anonymous actual bit string, the receiver will not be able to detect that it is in fact Oscar's. This observation has far-reaching consequences which can be summarized in the following statement:

**Even though public-key schemes do not require a secure channel, they require authenticated channels for the distribution of the public keys.**

We would like to stress here again that the MIM attack is not restricted to the DHKE, but is in fact applicable to any asymmetric crypto scheme. The attack always proceeds the same way: Oscar intercepts the public key that is being sent and replaces it with his own.

The problem of trusted distribution of private keys is central in modern public-key cryptography. There are several ways to address the problem of key authentication. The main mechanism is the use of *certificates*. The idea behind certificates is quite easy: Since the authenticity of the message $(k_{pub,A}, ID_A)$ is violated by an active attack, we apply a cryptographic mechanism that provides authentication. More specifically, we use digital signatures.[2] Thus, a certificate for a user Alice in its most basic form is the following structure:

$$\text{Cert}_A = [(k_{pub,A}, ID_A), \text{sig}_{k_{pr}}(k_{pub,A}, ID_A)]$$

The idea is that the receiver of a certificate verifies the signature prior to using the public key. We recall from Chap. 10 that the signature protects the signed message — which is the structure $(k_{pub,A}, ID_A)$ in this case — against manipulation. If Oscar attempts to replace $k_{pub,A}$ by $k_{pub,O}$ it will be detected. Thus, it is said that **certificates bind the identity of a user to their public key**.

Certificates require that the receiver has the correct verification key, which is a public key. If we were to use Alice's public key for this, we would have the same problem that we are actually trying to solve. Instead, the signatures for certificates are provided by a mutually trusted third party. This party is called the *Certification Authority* commonly abbreviated as *CA*. It is the task of the CA to generate and issue certificates for all users in the system. For certificate generation, we can distinguish between two main cases. In the first case, the user computes her own asymmetric key pair and merely requests the CA to sign the public key, as shown in the following simple protocol for a user named Alice:

[2] MACs also provide authentication and could, in principle, also be used for authenticating public keys. However, because MACs themselves are symmetric algorithms, we would again need a secure channel for distributing the MAC keys with all the associated drawbacks.

**Certificate Generation with User-Provided Keys**

| **Alice** | | **CA** |
|---|---|---|
| generate $k_{pr,A}, k_{pub,A}$ | | |
| | $\xrightarrow{\text{RQST}(k_{pub,A}, ID_A)}$ | |
| | | verify $ID_A$ <br> $s_A = \text{sig}_{k_{pr,CA}}(k_{pub,A}, ID_A)$ <br> $\text{Cert}_A = [(k_{pub,A}, ID_A), s_A]$ |
| | $\xleftarrow{\text{Cert}_A}$ | |

From a security point of view, the first transaction is crucial. It must be assured that Alice's message $(k_{pub,A}, ID_A)$ is sent via an authenticated channel. Otherwise, Oscar could request a certificate in Alice's name.

In practice it is often advantageous that the CA not only signs the public keys but also generates the public–private key pairs for each user. In this case, a basic protocol looks like this:

**Certificate Generation with CA-Generated Keys**

| **Alice** | | **CA** |
|---|---|---|
| request certificate | $\xrightarrow{\text{RQST}(ID_A)}$ | |
| | | verify $ID_A$ <br> generate $k_{pr,A}, k_{pub,A}$ <br> $s_A = \text{sig}_{k_{pr,CA}}(k_{pub,A}, ID_A)$ <br> $\text{Cert}_A = [(k_{pub,A}, ID_A), s_A]$ |
| | $\xleftarrow{\text{Cert}_A, k_{pr,A}}$ | |

For the first transmission, an authenticated channel is needed. In other words: The CA must be assured that it is really Alice who is requesting a certificate, and not Oscar who is requesting a certificate in Alice's name. Even more sensitive is the second transmission consisting of $(\text{Cert}_A, k_{pr,A})$. Because the private key is being sent here, not only an authenticated but a secure channel is required. In practice, this could be a certificate delivered by mail on a CD-ROM.

Before we discuss CAs in more detail, let's have a look at the DHKE which is protected with certificates:

**Diffie–Hellman Key Exchange with Certificates**

| **Alice** | | **Bob** |
|---|---|---|
| $a = k_{pr,A}$ | | $b = k_{pr,B}$ |
| $A = k_{pub,A} \equiv \alpha^a \bmod p$ | | $B = k_{pub,B} \equiv \alpha^B \bmod p$ |
| $\mathrm{Cert}_A = [(A, ID_A), s_A]$ | | $\mathrm{Cert}_B = [(B, ID_B), s_B]$ |
| | $\xrightarrow{\mathrm{Cert}_A}$ | |
| | $\xleftarrow{\mathrm{Cert}_B}$ | |
| verify certificate: | | verify certificate: |
| $\mathrm{ver}_{k_{pub,CA}}(\mathrm{Cert}_B)$ | | $\mathrm{ver}_{k_{pub,CA}}(\mathrm{Cert}_A)$ |
| compute session key: | | compute session key: |
| $k_{AB} \equiv B^a \bmod p$ | | $k_{AB} \equiv A^b \bmod p$ |

One very crucial point here is the verification of the certificates. Obviously, without verification, the signatures within the certificates would be of no use. As can be seen in the protocol, verification requires the public key of the CA. This key must be transmitted via an authenticated channel, otherwise Oscar could perform MIM attacks again. It looks like we haven't gained much from the introduction of certificates since we again require an authenticated channel! **However, the difference from the former situation is that we need the authenticated channel only once, at set-up time.** For instance, public verification keys are nowadays often included in PC software such as Web browsers or Microsoft software products. The authenticated channel is here assumed to be given through the installation of original software which has not been manipulated. What's happening here from a more abstract point of view is extremely interesting, namely a **transfer of trust**. We saw in the earlier example of DHKE without certificates, that Alice and Bob have to trust each other's public keys directly. With the introduction of certificates, they only have to trust the CA's public key $k_{pub,CA}$. If the CA signs other public keys, Alice and Bob know that they can also trust those. This is called a *chain of trust*.

### *13.3.3 Public-Key Infrastructures (PKI) and CAs*

The entire system that is formed by CAs together with the necessary support mechanisms is called a *public-key infrastructure*, usually referred to as *PKI*. As the reader can perhaps start to imagine, setting up and running a PKI in the real world is a complex task. Issues such as identifying users for certificate issuing and trusted distribution of CA keys have to be solved. There are also many other real-world issues; among the most complex are the existence of many different CAs and revocation of certificates. We discuss some aspects of using certificate systems in practice in the following.

### X.509 Certificates

In practice, certificates not only include the ID and the public key of a user, they tend to be quite complex structures with many additional fields. As an example, we look at the a X.509 certificate in Fig. 13.4. X.509 is an important standard for network authentication services, and the corresponding certificates are widely used for Internet communication, i.e., in S/MIME, IPsec and SSL/TLS.

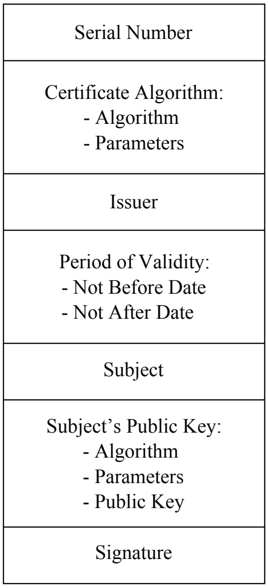


**Fig. 13.4** Detailed structure of an X.509 certificate

Discussing the fields defined in a X.509 certificate gives us some insight into many aspects of PKIs in the real world. We discuss the most relevant ones in the following:

1. *Certificate Algorithm*: Here it is specified which signature algorithm is being used, e.g., RSA with SHA-1 or ECDSA with SHA-2, and with which parameters, e.g., the bit lengths.
2. *Issuer*: There are many companies and organizations that issue certificates. This field specifies who generated the one at hand.
3. *Period of Validity*: In most cases, a public key is not certified indefinitely but rather for a limited time, e.g., for one or two years. One reason for doing this is that private keys which belong to the certificate may become compromised. By limiting the validity period, there is only a certain time span during which an attacker can maliciously use the private key. Another reason for a restricted lifetime is that, especially for certificates for companies, it can happen that the

user ceases to exist. If the certificates, and thus the public keys, are only valid for limited time, the damage can be controlled.

4. *Subject*: This field contains what was called $ID_A$ or $ID_B$ in our earlier examples. It contains identifying information such as names of people or organizations. Note that not only actual people but also entities like companies can obtain certificates.
5. *Subject's Public Key*: The public key that is to be protected by the certificate is here. In addition to the binary string which is the public key, the algorithm (e.g., Diffie–Hellman) and the algorithm parameters, e.g., the modulus $p$ and the primitive element $\alpha$, are stored.
6. *Signature*: The signature over all other fields of the certificate.

We note that for every signature two public key algorithms are involved: the one whose public key is protected by the certificate and the algorithm with which the certificate is signed. These can be entirely different algorithms and parameter sets. For instance, the certificate might be signed with an RSA 2048-bit algorithm, while the public key within the certificate could belong to a 160-bit elliptic curve scheme.

### Chain of Certificate Authorities (CAs)

In an ideal world, there would be one CA which issues certificates for, say, all Internet users on planet Earth. Unfortunately, that is not the case. There are many different entities that act as CAs. First of all, many countries have their own "official" CA, often for certificates that are used for applications that involve government business. Second, certificates for websites are currently issued by more than 50 mostly commercial entities. (Most Web browsers have the public key of those CAs pre-installed.) Third, many corporations issue certificate for their own employees and external entities who do business with them. It would be virtually impossible for a user to have the private keys of all these different CAs at hand. What is done instead is that CAs certify each other.

Let's look at an example where Alice's certificate is issued by CA1 and Bob's by CA2. At the moment, Alice is only in possession of the public key of "her" CA1, and Bob has only $k_{pub,CA2}$. If Bob sends his certificate to Alice, she cannot verify Bob's public key. This situation looks like this:

**Two Users with Different Certificate Authorities**

| **Alice** | | **Bob** |
|---|---|---|
| $k_{pub,CA1}$ | | $k_{pub,CA2}$ |
| | | $\mathrm{Cert}_B = [(k_{pub,B}, ID_B), \mathrm{sig}_{k_{pr,CA2}}(k_{pub,B}, ID_B)]$ |
| | $\xleftarrow{\mathrm{Cert}_B}$ | |

Alice can now request CA2's public key, which is itself contained in a certificate that was signed by Alice's CA1:

**Verification of a CA Public Key**

| **Alice** | | **CA2** |
|---|---|---|
| | $\xrightarrow{\text{RQST}(\text{Cert}_{CA2})}$ | |
| | $\xleftarrow{\text{Cert}_{CA2}}$ | |
| $\text{ver}_{k_{pub,CA1}}(\text{Cert}_{CA2})$<br>$\Rightarrow k_{pub,CA2}$ is valid<br>$\text{ver}_{k_{pub,CA2}}(\text{Cert}_B)$<br>$\Rightarrow k_{pub,B}$ is valid | | |

The structure $\text{Cert}_{CA2}$ contains the public key of CA2 signed by CA1, which looks like this:

$$\text{Cert}_{CA2} = [(k_{pub,CA2}, ID_{CA2}), \text{sig}_{k_{pr,CA1}}(k_{pub,CA2}, ID_{CA2})]$$

The important outcome of the process is that Alice can now verify Bob's key.

What's happening here is that a certificate chain is being established. CA1 trusts CA2 which is expressed by CA1 signing the public key $k_{pub,CA2}$. Now Alice can trust Bob's public key since it was signed by CA1. This situation is called a *chain of trust*, and it is said that *trust is delegated*.

In practice, CAs can be arranged hierarchically, where each CA signes the public key of the certificate authorities one level below. Alternatively, CAs can cross-certify each other without a strict hierarchical relationship.

### Certificate Revocation Lists

One major issue in practice is that it must be possible to revoke certificates. A common reason is that a certificate is stored on a smart card which is lost. Another reason could be that a person left an organization and one wants to make sure that she is not using the public key that was given to her. The solution in these situations seems easy: Just publish a list with all certificates that are currently invalid. Such a list is called a *certificate revocation list*, or *CRL*. Typically, the serial numbers of certificates are used to identify the revoked certificates. Of course, a CRL must be signed by the CA since otherwise attacks are possible.

The problem with CLRs is how to transmit them to the users. The most straightforward way is that every user contacts the issuing CA every time a certificate of another user is received. The major drawback is that now the CA is involved in every session set-up. This was one major drawback of KDC-based, i.e., symmetric-key, approaches. The promise of certificate-based communication was that no online contact to a central authority was needed.

An alternative is that CRLs are sent out periodically. The problem with this approach is that there is always a period during which a certificate is invalid but users have not yet been informed. For instance, if the CRL is sent out at 3:00 am every morning (a time with relatively little network traffic otherwise), a dishonest person could have almost a whole day where a revoked certificate is still valid. To counter this, the CRL update period can be shortened, say to one hour. However, this would be a tremendous burden on the bandwidth of the network. This is an instructive example for the tradeoff between costs in the form of network traffic on one hand, and security on the other hand. In practice, a reasonable compromise must be found.

In order to keep the size of CRLs moderate, often only the changes from the last CRL broadcast are sent out. These update-only CRLs are referred to as *delta CRLs*.

## 13.4 Discussion and Further Reading

**Key Establishment Protocols** In most modern network security protocols, public-key approaches are used for establishing keys. In this book, we introduced the Diffie–Hellman key exchange and described a basic key transport protocol in Chap. 6 (cf. Fig. 6.5). In practice, often considerably more advanced asymmetric protocols are used. However, most of them are based on either the Diffie–Hellman or a key transport protocol. A comprehensive overview on this area is given in [33].

We now give a few examples of generic cryptographic protocols that are often preferred over the basic Diffie–Hellman key exchange. The *MTI* (Matsumoto–Takashima–Imai) protocols are an ensemble of authenticated Diffie–Hellman key exchanges which were already published in 1986. Good descriptions can be found in [33] and [120]. Another popular Diffie–Hellman extension is the station-to-station (STS) protocol. It uses certificates and provides both user and key authentication. A discussion about STS variants can be found in [60]. A more recent protocol for authenticated Diffie–Hellman is the MQV protocol which is discussed in [108]. It is typically used with elliptic curves.

A prominent practical example for a key establishment protocol is the Internet Key Exchange (IKE) protocol. IKE provides key material for IPsec, which is the “official” security mechanism for Internet traffic. IKE is quite complex and offers many options. At its core, however, is a Diffie–Hellman key agreement followed by an authentication. The latter can either be achieved with certificates or with pre-shared keys. A good starting point for more information on IPsec and IKE is the RFC [128] and, more accessibly, reference [161, Chapter 16].

**Certificates and Alternatives** During the second half of the 1990s there was a belief that essentially every Internet user would need a certificate in order to communicate securely, e.g., for doing ebusiness transactions. “PKI” was a buzzword for some time, and many companies were formed that provided certificates and PKI services. However, it turned out that there are major technical and practical hurdles to a PKI that truly encompasses all or most Internet users. What has happened instead is that nowadays many servers are authenticated with certificates, for instance Internet retailers, whereas most individual users are not. The needed CA verification keys are often preinstalled in users' Web browsers. This asymmetric set-up — the server is authenticated but the user is not — is acceptable since the user is typically the one who provides crucial information such as her credit card number. A comprehensive introduction to the large field of PKI and certificates is given in the book [2]. An interesting and entertaining discussion about the alleged shortcomings of PKI is given in [74], and an equally instructive rebuttal is online at [107].

We introduced certificates and a public-key infrastructure as the main method for authenticating public keys. Such hierarchical organized certificates are only one possible approach, though this is the most widely used one. Another concept is the *web of trust* that relies entirely on trust relationships between parties. The idea is as follows: If Alice trusts Bob, it is assumed that she also wants to trust all other users whom Bob trusts. This means that every party in such a web of trust implicitly trusts parties whom it does not know (or has never met before). The most popular example for such a system are *Pretty Good Privacy (PGP)* and *Gnu Privacy Guard (GPG)*, which are widely used for signing and encrypting emails.

## 13.5 Lessons Learned

- A key transport protocol securely transfers a secret key to other parties.
- In a key agreement protocol, two or more parties negotiate a common secret key.
- In most common symmetric protocols, the key exchange is coordinated by a trusted third party. A secure channel between the third party and each user is only required at set-up time.
- Symmetric key establishment protocols do not scale well to networks with large numbers of users and they provide typically no perfect forward secrecy.
- The most widely used asymmetric key establishment protocol is the Diffie–Hellman key exchange.
- All asymmetric protocols require that the public keys are authenticated, e.g., with certificates. Otherwise man-in-the-middle attacks are possible.

## Problems

**13.1.** In this exercise, we want to analyze some variants of key derivation. In practice, one *masterkey* $k_{MK}$ is exchanged in a secure way (e.g. certificate-based DHKE) between the involved parties. Afterwards, the session keys are regularly updated by use of key derivation. For this purpose, three different methods are at our disposal:

(1) $k_0 = k_{MK}$; $k_{i+1} = k_i + 1$
(2) $k_0 = h(k_{MK})$; $k_{i+1} = h(k_i)$
(3) $k_0 = h(k_{MK})$; $k_{i+1} = h(k_{MK}||i||k_i)$

where $h()$ marks a (secure) hash function, and $k_i$ is the $i$th session key.

1. What are the main differences between these three methods?
2. Which method provides *Perfect Forward Secrecy*?
3. Assume Oscar obtains the $n$th session key (e.g., via brute-force). Which sessions can he now decrypt (depending on the chosen method)?
4. Which method remains secure if the masterkey $k_{MK}$ is compromised? Give a rationale!

**13.2.** Imagine a peer-to-peer network where 1000 users want to communicate in an authenticated and confidential way without a central Trusted Third Party (TTP).

1. How many keys are collectively needed, if symmetric algorithms are deployed?
2. How are these numbers changed, if we bring in a central instance (Key Distribution Center, KDC)?
3. What is the main advantage of a KDC against the scenario without a KDC?
4. How many keys are necessary if we make use of asymmetric algorithms?

Also differentiate between keys which *every* user has to store and keys which are collectively necessary.

**13.3.** You have to choose the cryptographic algorithms for a KDC where two different classes of encryption occur:

- $e_{k_{U,KDC}}()$, where $U$ denotes an arbitrary network node (user),
- $e_{k_{ses}}()$ for the communication between two users.

You have the choice between two different algorithms, DES and 3DES (Triple-DES), and you are advised to use distinct algorithms for both encryption classes. Which algorithm do you use for which class? Justify your answer including aspects of security as well as celerity.

**13.4.** This exercise considers the security of key establishment with the aid of a KDC. Assume that a hacker performs a successful attack against the KDC at the point of time $t_x$, where all keys are compromised. The attack is detected.

1. Which (practical) measures have to be taken in order to prevent decryption of future communication between the network nodes?

2. Which steps did the attacker have to take in order to decipher data transmissions which occurred at an earlier time ($t < t_x$)? Does such a KDC system provide Perfect Forward Secrecy (PFS) or not?

**13.5.** We will now analyze an improved KDC system. In contrast to the previous problem, all keys $e_{k_{U,KDC}}()$ are now refreshed in relatively short intervals:

- The KDC generates a new (random) key: $k_{U,KDC}^{(i+1)}$
- The KDC transmits the new key to user $U$, encrypted with the old one: $e_{k_{U,KDC}^{(i)}}(k_{U,KDC}^{(i+1)})$

Which decryptions are possible, if a staff member of the KDC is corruptible and "sells" all recent keys $e_{k_{U,KDC}^{(i)}}$ of the KDC at the point of time $t_x$? We assume that this circumstance is not detected until the point of time $t_y$ which could be much later, e.g., one year.

**13.6.** Show a key confirmation attack against the basic KDC protocol introduced in Sect. 13.2.1. Describe each step of the attack. Your drawing should look similar to the one showing a key confirmation attack against the second (modified) KDC-based protocol.

**13.7.** Show that PFS is in fact not given in the simplified Kerberos protocol. Show how Oscar can decrypt past and future communications if:

1. Alice's KEK $k_A$ becomes compromised
2. Bob's KEK $k_B$ becomes compromised

**13.8.** Extend the Kerberos protocol such that a mutual authentication between Alice and Bob is performed. Give a rationale that your solution is secure.

**13.9.** People at your new job are deeply impressed that you worked through this book. As the first job assignment you are asked to design a digital pay-TV system which uses encryption to prevent service theft through wire tapping. As key exchange protocol, a strong Diffie–Hellman with, e.g., 2048-bit modulus is being used. However, since your company wants to use cheap legacy hardware, only DES is available for data encryption algorithm. You decide to use the following key derivation approach:

$$K^{(i)} = f(K_{AB} \parallel i). \tag{13.1}$$

where $f$ is an irreversible function.

1. First we have to determine whether the attacker can store an entire movie with reasonable effort (in particular, cost). Assume the data rate for the TV link is 1 Mbit/s, and that the longest movies we want to protect are 2 hours long. How many Gbytes (where 1M = $10^6$ and 1G = $10^9$) of data must be stored for a 2-hour film (don't mix up bit and byte here)? Is this realistic?

2. We assume that an attacker will be able to find a DES key in 10 minutes using a brute-force attack. Note that this is a somewhat optimistic assumption from an attacker's point of view, but we want to provide some medium-term security by assuming increasingly faster key searches in the future.
   How frequently must a key be derived if the goal is to prevent an offline decryption of a 2-hour movie in less than 30 days?

**13.10.** We consider a system in which a key $k_{AB}$ is established using the Diffie–Hellman key exchange protocol, and the encryption keys $k^{(i)}$ are then derived by computing:

$$k^{(i)} = h(k_{AB} \parallel i) \tag{13.2}$$

where $i$ is just an integer counter, represented as a 32-bit variable. The values of $i$ are public (e.g., the encrypting party always indicates which value for $i$ was used in a header that precedes each ciphertext block). The derived keys are used for the actual data encryption with a symmetric algorithm. New keys are derived every 60 sec during the communication session.

1. Assume the Diffie–Hellman key exchange is done with a 512-bit prime, and the encryption algorithm is AES. Why doesn't it make cryptographic sense to use the key derivation protocol described above? Describe the attack that would require the least computational effort from Oscar.
2. Assume now that the Diffie–Hellman key exchange is done with a 2048-bit prime, and the encryption algorithm is DES. Describe in detail what the advantages are that the key derivation scheme offers compared to a system that just uses the Diffie–Hellman key for DES.

**13.11.** We reconsider the Diffie–Hellman key exchange protocol. Assume now that Oscar runs an active man-in-the-middle attack against the key exchange as explained in Sect. 13.3.1. For the Diffie–Hellman key exchange, use the parameters $p = 467$, $\alpha = 2$, and $a = 228$, $b = 57$ for Alice and Bob, respectively. Oscar uses the value $o = 16$. Compute the key pairs $k_{AO}$ and $k_{BO}$ (i) the way Oscar computes them, and (ii) the way Alice and Bob compute them.

**13.12.** We consider the Diffie–Hellman key exchange scheme with certificates. We have a system with the three users Alice, Bob and Charley. The Diffie–Hellman algorithm uses $p = 61$ and $\alpha = 18$. The three secret keys are $a = 11$, $b = 22$ and $c = 33$. The three IDs are ID(A)=1, ID(B)=2 and ID(C)=3.

For signature generation, the Elgamal signature scheme is used. We apply the system parameters $p' = 467$, $d' = 127$, $\alpha' = 2$ and $\beta$. The CA uses the ephemeral keys $k_E = 213$, 215 and 217 for Alice's, Bob's and Charley's signatures, respectively. (In practice, the CA should use a better pseudorandom generator to obtain the $k_E$ values.)

To obtain the certificates, the CA computes $x_i = 4 \times b_i + \mathrm{ID}(i)$ and uses this value as input for the signature algorithm. (Given $x_i$, $\mathrm{ID}(i)$ follows then from $\mathrm{ID}(i) \equiv x_i \bmod 4$.)

1. Compute three certificates $Cert_A$, $Cert_B$ and $Cert_C$.

2. Verify all three certificates.
3. Compute the three session keys $k_{AB}$, $k_{AC}$ and $k_{BC}$.

**13.13.** Assume Oscar attempts to use an active (substitution) attack against the Diffie–Hellman key exchange with certificates in the following ways:

1. Alice wants to communicate with Bob. When Alice obtains C(B) from Bob, Oscar replaces it with (a valid!) C(O). How will this forgery be detected?
2. Same scenario: Oscar tries now to replace only Bob's public key $b_B$ with his own public key $b_O$. How will this forgery be detected?

**13.14.** We consider certificate generation with CA-generated keys. Assume the second transmission of $(\mathrm{Cert}_A, k_{pr,A})$ takes place over an authenticated but insecure channel, i.e., Oscar can read this message.

1. Show how he can decrypt traffic which is encrypted by means of a Diffie–Hellman key that Alice and Bob generated.
2. Can he also impersonate Alice such that he computes a DH key with Bob without Bob noticing?

**13.15.** Given is a user domain in which users share the Diffie–Hellman parameters $\alpha$ and $p$. Each user's public Diffie–Hellman key is certified by a CA. Users communicate securely by performing a Diffie–Hellman key exchange and then encrypting/decrypting messages with a symmetric algorithm such as AES.

Assume Oscar gets hold of the CA's signature algorithm (and especially its private key), which was used to generate certificates. Can he now decrypt old ciphertexts which were exchanged between two users before the CA signature algorithm was compromised, and which Oscar had stored? Explain your answer.

**13.16.** Another problem in certificate systems is the authenticated distribution of the CA's public key which is needed for certificate verification. Assume Oscar has full control over all of Bob's communications, that is, he can alter all messages to and from Bob. Oscar now replaces the CA's public key with his own (note that Bob has no means to authenticate the key that he receives, so he thinks that he received the CA public key.)

1. (Certificate issuing) Bob requests a certificate by sending a request containing (1) Bob's ID $ID(B)$ and (2) Bob's public key $B$ from the CA. Describe exactly what Oscar has to do so that Bob doesn't find out that he has the wrong public CA key.
2. (Protocol execution) Describe what Oscar has to do to establish a session key with Bob using the authenticated Diffie–Hellman key exchange, such that Bob thinks he is executing the protocol with Alice.

**13.17.** Draw a diagram that shows a key transport protocol shown in Fig. 6.5 from Sect. 6.1, in which RSA encryption is used.

**13.18.** We consider RSA encryption with certificates in which Bob has the RSA keys. Oscar manages to send Alice a verification key $k_{pr,CA}$ which is, in fact, Oscar's key. Show an active attack in which he can decipher encrypted messages that Alice sends to Bob. Should Oscar run a MIM attack or should he set up a session only between himself and Alice?

**13.19.** Pretty Good Privacy (PGP) is a widespread scheme for electronic mail security to provide authentication and confidentiality. PGP does not necessarily require the use of certificate authorities. Describe the trust model of PGP and how the public-key management works in practice.