MODULE 4

10 Key Management

Key management is crucial to the security of any cryptosystem. Without secure procedures for the handling of cryptographic keys throughout their lifecycle, the benefits of the use of strong cryptographic primitives are potentially lost. Indeed, it could be argued that if key management is not performed correctly then there is no point in using cryptography at all.

Cryptographic primitives are rarely compromised through weaknesses in their design. However they are often compromised through poor key management. Thus, from a practical perspective, there is a strong case for arguing that this is the most important chapter in this book.

This chapter is intended to provide an understanding of the fundamental principles of key management. However, key management is a complex and difficult aspect of any cryptosystem. Since it is essentially the interface between cryptographic mechanisms and the security of a real system, key management must be closely tailored to the needs of a particular application or organisation. For example, different solutions will be needed for managing the keys of a bank, a military organisation, a mobile telephone network, and a home personal computer. *There is no one correct way of managing keys*. As such, the discussion in this chapter cannot be prescriptive. Nonetheless, the following treatment of key management will hopefully explain the main issues and provide useful guidelines.

At the end of this chapter you should be able to:

- Identify some fundamental principles of key management.
- Explain the main phases in the lifecycle of a cryptographic key.
- Discuss a number of different techniques for implementing the different phases in the lifecycle of a cryptographic key.
- Identify appropriate key management techniques for specific application environments.
- Appreciate the need for secure key management policies, practices and procedures.

10.1 Key management fundamentals

In this section we provide an introduction to key management. Most importantly, we identify the scope of key management and introduce the key lifecycle, which we will use to structure the discussion in the remainder of the chapter.

10.1.1 What is key management?

The scope of key management is perhaps best described as the *secure administration of cryptographic keys*. This is a deliberately broad definition, because key management involves a wide range of quite disparate processes, all of which must come together coherently if cryptographic keys are to be securely managed.

The important thing to remember is that cryptographic keys are just special pieces of *data*. Key management thus involves most of the diverse processes associated with information security. These include:

Technical controls. These can be used in various aspects of key management. For example, special hardware devices may be required for storing cryptographic keys, and special cryptographic protocols are necessary in order to establish keys.

Process controls. Policies, practices and procedures play a crucial role in key management. For example, business continuity processes may be required in order to cope with the potential loss of important cryptographic keys.

Environmental controls. Key management must be tailored to the environment in which it will be practiced. For example, the physical location of cryptographic keys plays a big role in determining the key management techniques that are used to administer them.

Human factors. Key management often involves people doing things. Every security practitioner knows that whenever this is the case, the potential for problems occurring is high. Many key management systems rely, at their very highest level, on manual processes.

Thus, while cryptographic keys represent an extremely small percentage of the data that an organisation needs to manage, much of the wider information security issues that the organisation has to deal with (such as physical security, access control, network security, security policy, risk management and disaster recovery) interface with key management. Paradoxically, we will also see that while key management exists to support the use of cryptography, we will also need to use cryptography in order to provide key management.

The good news is that much of key management is about applying 'common sense'. The bad news, of course, is that applying 'common sense' is often much more complex than we first imagine. This is certainly true for key management.

Recall from Section 3.2.1 that if key management was easy then we would all probably be using a one-time pad for encryption!

Note that while all cryptographic keys need to be managed, the discussion in this chapter is primarily aimed at *keys that need to be kept secret*. In other words, we focus on symmetric keys and the private keys of a public-key pair (that is, private decryption or signature keys). Public keys have such special key management issues that we devote Chapter 11 to considering them. That said, many of the broader issues discussed in this chapter also apply to public keys.

10.1.2 The key lifecycle

Another way of defining the scope of key management is to consider the *key lifecycle*, which identifies the various processes concerning cryptographic keys throughout their lifetime. The main phases of the key lifecycle are depicted in Figure 10.1.

Key generation concerns the creation of keys. We discuss key generation in Section 10.3.

Key establishment is the process of making sure that keys reach the end points where they will be used. This is arguably the most difficult phase of the key lifecycle to implement. We discuss key establishment in Section 10.4.

Key storage deals with the safekeeping of keys. It may also be important to conduct *key backup* so that keys can be recovered in the event of loss of a key and, ultimately, *key archival*. These are all discussed in Section 10.5.

Key usage is about how keys are used. As part of this discussion we will consider *key change*. We will also look at how a key's life ends in *key destruction*. These are all discussed in Section 10.6.

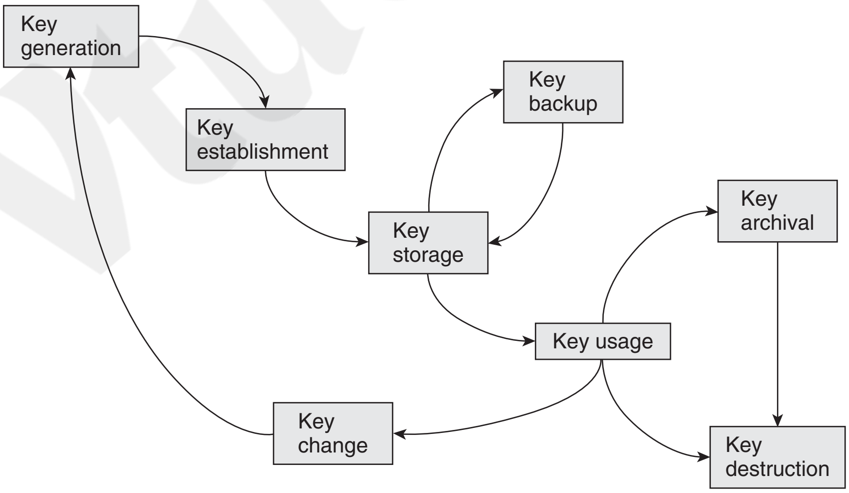

Figure 10.1. The key lifecycle

Note that the various phases in the key lifecycle are not always cleanly separated. For example, we saw in Section 9.4.2 how key agreement protocols such as the Diffie–Hellman protocol simultaneously generate and establish a symmetric key. Also, some phases are not always relevant. For example, key backup and key archival may not be necessary in some applications.

10.1.3 Fundamental key management requirements

There are two fundamental key management requirements that apply throughout the various phases of the key lifecycle:

Secrecy of keys. Throughout the key lifecycle, secret keys (in other words, symmetric keys and private keys) must remain secret from all parties except those that are authorised to know them. This clearly impacts all phases of the key lifecycle since, for example:

- if a weak key generation mechanism is used then it might be possible to determine information about a secret key more easily than intended;
- secret keys are vulnerable when they are 'moved around', thus secure key distribution mechanisms must be used;
- secret keys are perhaps most vulnerable when they are 'sitting around', thus key storage mechanisms must be strong enough to resist an attacker who has access to a device on which they reside;
- if secret keys are not destroyed properly then they can potentially be recovered after the supposed time of destruction.

Assurance of purpose of keys. Throughout the key lifecycle, those parties relying on a key must have *assurance of purpose* of the key. In other words, someone in possession of a key should be confident that they can use that key for the purpose that they believe it to be for. This 'purpose', may include some, or all, of the following:

- information concerning which entities are associated with the key (in symmetric cryptography this is likely to be more than one entity, whereas for public-key cryptography each key is normally only associated with one entity);
- the cryptographic algorithm that the key is intended to be used for;
- key usage restrictions, for example, that a symmetric key can only be used for creating and verifying a MAC, or that a signature key can only be used for digitally signing transactions of less than a certain value.

The 'assurance' necessarily includes some degree of data integrity that links (ideally binds) the above information to the key itself, otherwise it cannot be relied upon. Sometimes, perhaps rather casually, assurance of purpose is referred to as *authentication* of the key. However, assurance of purpose is often much more than identification of the entity associated with the key. Assurance of purpose impacts all phases of the key lifecycle since, for example:

- if a key is established without providing assurance of purpose then it might later be used for a purpose other than that for which it was originally intended (we will show how this could happen in Section 10.6.1);
- if a key is used for the wrong purpose then there could be very serious consequences (we will see an example of this in Section 10.6.1).

In some applications we require an even stronger requirement that assurance of purpose is *provable to a third party*, which might be the case for verification keys for digital signature schemes.

The need for secrecy of keys is self-evident and much of our subsequent discussion about the key lifecycle will be targeted towards providing it. Assurance of purpose of keys is more subtle and is often provided *implicitly*. For example, in the AKE protocol from ISO 9798-2 that we discussed in Section 9.4.3, Alice and Bob establish a shared AES encryption (say) key using a TTP, in each case receiving the key encrypted using a key known only to Alice or Bob and the TTP. In this case the assurance of purpose is implicitly provided through a combination of the facts that:

1. the key arrives shortly after a specific request for a shared AES encryption key;
2. the key has clearly come from the TTP (this case was argued in the protocol analysis in Section 9.4.3);
3. the name of the other communicating party is included in the associated ciphertext.

Assurance of purpose in the above example is largely facilitated by the fact that the parties relying on the key, Alice and Bob, are both part of a ‘closed’ system where they both share long-term symmetric keys with a TTP. In most environments where symmetric key cryptography is used, assurance of purpose is provided through similar implicit arguments. In contrast, public-key cryptography facilitates the use of cryptography in ‘open’ environments where there are no sources of implicit assurance of purpose of keys. Public keys can, literally, be public items of data. By default there are no assurances of whether a public key is correct, with whom it can be associated, or what it can be used for. Thus key management of public keys needs to focus much more explicitly on assurance of purpose of public keys. This is the main subject of Chapter 11.

Finally, the purpose of a key is not always intuitive. For example, we saw in Section 7.2.3 that a user who has a MAC key might not be allowed to use it both for MAC creation and verification. Similarly, a user might not be allowed to use a symmetric key for both encryption and decryption. We will also see in

Section 10.6.1 an example of a symmetric key that can only ever be used by anyone for encryption, never decryption.

10.1.4 Key management systems

We will use the term *key management system* to describe any system for managing the various phases of the key lifecycle introduced in Section 10.1.2. While throughout our discussion of cryptographic primitives we have cautioned against using anything other than a few respected, and standardised, cryptographic primitives, we have to take a more pragmatic view when it comes to key management systems. This is because a key management system needs to be aligned with the functionality and priorities of the organisation that implements it (we use the term 'organisation' in such a loose sense that this could be an individual). For example, a key management system may depend on:

Network topology. Key management is much simpler if it is only needed to support two parties who wish to communicate securely, rather than a multinational organisation that wishes to establish the capability for secure communication between any two employees.

Cryptographic mechanisms. As we will see in this chapter and Chapter 11, some of the key management system requirements of symmetric and public-key cryptography differ.

Compliance restrictions. For example, depending on the application, there may be legal requirements for key recovery mechanisms or key archival (see Section 10.5.5).

Legacy issues. Large organisations whose security partly depends on that of other related organisations may find that their choice of key management system is restricted by requirements to be compatible with business partners, some of whom might be using older technology.

Thus an organisation will, almost inevitably, have to think carefully about how to design and implement a key management system that meets its own needs. That said, there are a number of important standards relating to key management that provide useful guidance. There are inevitably many proprietary key management systems around, some closely related to these standards and others completely non-standard. It is also worth noting that most international standards for key management are lengthy documents, covering a multitude of key management areas, so that full compliance with such standards is extremely hard to achieve. Many organisations will comply with the 'spirit' of a standard rather than the 'letter' of the standard.

Key management is hard, and there are many different ways of approaching it. This raises a major concern: how can we get some level of assurance that a key management system is doing its job effectively? We will briefly revisit this issue in Section 10.7.

10.2 Key lengths and lifetimes

Before we discuss the lifecycle of cryptographic keys, we need to consider a couple of properties of the keys themselves, most significantly the key length.

We already know that, in general (but certainly not by default), longer keys are better from a security perspective. Longer symmetric keys take more time to exhaustively search for and longer public-key pairs tend to make the underlying computational problem on which a public-key cryptosystem is based harder to solve. So there is certainly a case for making keys as long as possible.

However, a cryptographic computation normally takes more time if the key is longer. In addition, longer keys involve greater storage and distribution overheads. Hence longer keys are less efficient in several important respects. Thus key length tends to be based on an efficiency–security tradeoff. We normally want keys to be 'long enough', but not more than that.

10.2.1 Key lifetimes

The issue of key length is closely linked to the intended *lifetime* (also often referred to as the *cryptoperiod*) of a cryptographic key. By this we mean that the key can only be used for a specified period of time, during which it is regarded as being *live*. Once that lifetime has been exceeded, the key is regarded as *expired* and should no longer be used. At this point it may need to be *archived* or perhaps *destroyed* (we discuss this in more detail in Section 10.6.4).

There are many reasons why cryptographic keys have finite lifetimes. These include:

Mitigation against key compromise. Having a finite lifetime prevents keys being used beyond a time within which they might reasonably be expected to be compromised, for example by an exhaustive key search or compromise of the storage medium.

Mitigation against key management failures. Finite key lifetimes help to mitigate against failures in key management. For example, forcing an annual key change will guarantee that personnel who leave an organisation during the year, but for some reason retain keys, do not have access to valid keys the following year.

Mitigation against future attacks. Finite key lifetimes help to mitigate against future advances in the attack environment. For this reason, keys are normally set to expire well before current knowledge suggests that they need to.

Enforcement of management cycles. Finite lifetimes enforce a key change process, which might be convenient for management cycles. For example, if keys provide access to electronic resources that are paid for on an annual subscription basis, then having a one-year key lifetime allows access to keys to be directly linked to subscriptions to the service.

Flexibility. Finite key lifetimes introduce an additional 'variable' which can be adjusted to suit application requirements. For example, a relatively short key (which is relatively inexpensive to generate, distribute and store) could be adopted under the pretext that the key lifetime is also suitably short. We will discuss an example of this in Section 10.4.1, where *data keys* are relatively short but expire quickly in comparison to *master keys*.

Limitation of key exposure. At least in theory, some information relating to a key is 'leaked' to an attacker every time the attacker sees a cryptographic value computed using that key. This is because the result of every cryptographic computation provides the attacker with information that they did not have before they saw the ciphertext. We refer to this as *key exposure*. However, this information is normally of little (often no) use to an attacker if the cryptographic algorithm is strong, hence in many applications key exposure is not a significant issue. Nonetheless, finite key lifetimes limit key exposure, should this be regarded as a concern.

10.2.2 Choosing a key length

The length of a cryptographic key is commensurate with the key lifetime, which in turn is related to the length of time for which cryptographic protection is required. We discussed one aspect of this issue in Section 3.2.2 when we considered the notion of cover time. Note, however, that there is an element of 'the chicken and the egg' in the relationship between key length and key lifetime:

- in an ideal world the key lifetime would be chosen and then a suitable key length selected;
- in the real world the key length may be dictated (for example, the key management system is based on the use of 128-bit AES keys) and thus the key lifetime is set to an appropriate time period.

Fortunately, the decision on key length is made much simpler by the limited number of standard cryptographic algorithms, which in turn present limited options for key lengths. Nonetheless, decisions will need to be made, particularly since some of the most popular cryptographic algorithms such as AES (see Section 4.5) and RSA (see Section 5.2) have variable key lengths.

The most important advice on selecting key length is to listen to the experts. From time to time, influential bodies will issue guidance on advised key lengths, and these should be carefully adhered to. Note that:

- Key length recommendations for symmetric cryptography tend to be algorithm-independent, since the security of respected symmetric encryption algorithms should be benchmarked against the difficulty of an exhaustive key search.

- Key length recommendations for public-key cryptography tend to be algorithm-specific, since the security of a public-key cryptosystem depends upon the perceived difficulty of the hard computational problem on which the algorithm is based (for example, factoring in the case of RSA).

Key length recommendations are usually presented in terms of a combination of potential attack environments and cover times. Table 10.1 provides an example, showing protection profiles and recommended symmetric key lengths.

There are several issues worth noting from Table 10.1:

1. Some of the key length recommendations are specifically linked to maximum recommended key lifetimes.
2. Although these recommendations are largely algorithm-independent, some further specific advice is given by ECRYPT II on the use of Triple DES, since Triple DES has a much weaker security than that suggested by its key length (see Section 4.4.4).

It should also be noted that:

Advice on key length is not unanimous. Ultimately these are subjective opinions, albeit hopefully informed ones. Before choosing a key length it is advisable to seek recommendations from more than one source.

Table 10.1: ECRYPT II protection profiles and symmetric key lengths (2011)

	Protection	Notes	Key length
1	Vulnerable to real-time attacks by individuals	Limited use	32
2	Very short term protection against small organisations	Not for new applications	64
3	Short-term protection against medium organisations; medium-term protection against small organisations		72
4	Very short term protection against agencies; long-term protection against small organisations	Protection to 2012	80
5	Legacy standard level	Protection to 2020	96
6	Medium-term protection	Protection to 2030	112
7	Long-term protection	Protection to 2040	128
8	'Foreseeable future'	Good protection against quantum computers	256

Advice on key length changes over time. It is wise to seek the latest and most accurate information before deciding on key lengths. It is possible, for example, that the advice in Table 10.1, and the key length comparisons in Table 5.2, may no longer be accurate.

10.3 Key generation

We now begin our discussion of the various phases in the key lifecycle. This begins with key generation, which is the creation of cryptographic keys. This is a critical phase of the key lifecycle. As we indicated at the start of Section 8.1, many cryptosystems have been found to have weaknesses because they do not generate their keys in a sufficiently secure manner.

Key generation processes for symmetric and public-key cryptography are fundamentally different. We will first look at ways of generating a symmetric key.

10.3.1 Direct key generation

Symmetric keys are just randomly generated numbers (normally bit strings). The most obvious method for generating a cryptographic key is thus to randomly generate a number, or more commonly a pseudorandom number. We have already discussed random number generation in Section 8.1 and any of the techniques discussed there are potentially appropriate for key generation. The choice of technique will depend on the application. Obviously, the strength of the technique used should take into consideration the importance of the cryptographic key that is being generated. For example, use of a hardware-based non-deterministic generator might be appropriate for a master key, whereas a software-based non-deterministic generator based on mouse movements might suffice for generating a local key to be used to store personal files on a home PC (see Section 8.1.3).

The only further issue to note is that for certain cryptographic algorithms there are sporadic choices of key that some people argue should not be used. For example, as mentioned in Section 4.4.3, DES has some keys that are defined to be *weak*. In the rare event that such keys are generated by a key generation process, some guidance suggests that they should be rejected. Issues such as this are algorithm-specific and the relevant standards should be consulted for advice.

10.3.2 Key derivation

The term *key derivation* is sometimes used to describe the generation of cryptographic keys from other cryptographic keys or secret values. Such 'key laundering' might at first seem a strange thing to do, but there are several significant advantages of deriving keys from other keys:

Efficiency. Key generation and establishment can be relatively expensive processes. Generating and establishing one key (sometimes called a *base key*), and then using it to derive many keys, can be an effective technique for saving on these costs. For example, many applications require both confidentiality and data origin authentication. If separate cryptographic mechanisms are to be used to provide these two security services then they require an encryption key and a MAC key (see Section 6.3.6 for a wider discussion of this, and other options). As we will see in Section 10.6.1, it is good practice to make sure that the keys used for each of these mechanisms are different. Rather than generating and establishing two symmetric keys for this purpose, a cost-efficient solution is to generate and establish one key K and then derive two keys K_1 and K_2 from it. For example, a very simple key derivation process might involve computing:

$$K_1 = h(K||0) \quad \text{and} \quad K_2 = h(K||1),$$

where h is a hash function.

Longevity. In some applications, long-term symmetric keys are preloaded onto devices before deployment. Using these long-term keys directly to encrypt data exposes them to cryptanalysis (as indicated in Section 10.2.1). However, randomly generating a new key requires a key establishment mechanism to be used, which may not always be possible or practical. A good solution is to derive keys for use from the long-term key. In this way, so long as the key derivation process is understood by all parties requiring access to a key, no further key establishment mechanism is required and the long-term key is not exposed through direct use.

Key derivation must be based on a derivation function that is one-way (see Section 6.2.1). This protects the base key in the event that keys derived from it are later compromised. This is important because often many different keys are derived using a single base key, hence the impact of subsequently compromising the base key could be substantial.

There are standards for key derivation. For example, PKCS#5 defines how a key can be derived from a password or a PIN, which can be regarded as a relatively insecure type of cryptographic key, but one which is often long term (such as the PIN associated with a payment card). Key derivation in this case is defined as a function $f(P, S, C, L)$, where:

- f is a key derivation function that explains how to combine the various inputs in order to derive a key;
- P is the password or PIN;
- S is a string of (not necessarily all secret) pseudorandom bits, used to enable P to be used to derive many different keys;

- C is an iteration counter that specifies the number of 'rounds' to compute (just as discussed for a block cipher in Section 4.4.1, this can be used to tradeoff security against efficiency of the key derivation);
- L is the length of the derived key.

Obviously, the benefits of key derivation come at the cost of an increased impact in the event that a base key becomes compromised. This is just the latest of the many efficiency–security tradeoffs that we have encountered.

In Section 10.4.2 we will see another variation of this idea, where keys are derived from old keys and additional data.

10.3.3 Key generation from components

Direct key generation and key derivation are both processes that can be performed if one entity can be trusted to have full control of a particular key generation process. In many situations this is an entirely reasonable assumption.

However, for extremely important secret keys it may not be desirable to trust one entity with key generation. In such cases we need to distribute the key generation process amongst a group of entities in such a way that no members of the group individually have control over the process, but collectively they do. One technique for facilitating this is to generate a key in *component form*. We illustrate this by considering a simple scenario involving three entities: Alice, Bob and Charlie. Assume that we wish to generate a 128-bit key:

1. Alice, Bob and Charlie each randomly generate a *component* of 128 bits. This component is itself a sort of key, so any direct key generation mechanism could be used to generate it. However, given that key generation based on component form is only likely to be used for sensitive keys, the generation of components should be performed as securely as possible. We denote the resulting components by K_A, K_B and K_C, respectively.
2. Alice, Bob and Charlie securely transfer their components to a secure *combiner*. In most applications this combiner will be represented by a hardware security module (see Section 10.5.3). In many cases the 'secure transfer' of these components will be by manual delivery. For some large international organisations, this might even involve several of the components being physically flown across the world. The input of the components to the secure combiner is normally conducted according to a strict protocol that takes the form of a *key ceremony* (see Section 10.7.2).
3. The secure combiner derives a key K from the separate components. In this example, the best derivation function is XOR. In other words:

$$K = K_A \oplus K_B \oplus K_C.$$

Note that the key K is only reconstructed within the secure combiner and not output to the entities involved in the key derivation process. XOR is the 'best' type of key derivation function since knowledge of even two of the components does not leak any information about the derived key K. To see this, consider the case where Alice and Bob are conspiring to try to learn something about key K. Suppose that Alice and Bob XOR their components together to compute $R = K_A \oplus K_B$. Observe that $K = R \oplus K_C$, which means that $R = K \oplus K_C$. Thus R can be considered as the 'encryption' of K using a one-time pad with key K_C. We know from Section 3.1.3 that the one-time pad offers perfect secrecy, which means that knowing R (the 'ciphertext') does not leak any information about K (the 'plaintext').

Thus Alice, Bob and Charlie are able to jointly generate a key in such a way that all three of their components are necessary for the process to complete. If only two of the components are present then no information about the key can be derived, even if the components are combined. This process easily generalises to any number of entities, all of whom must present their components in order to derive the key. Even more ingenious techniques can be used to implement more complex key generation policies. For example, the *Shamir secret-sharing protocol* allows a key to be generated in component form in such a way that the key can be derived from any k of n components, where k can be any number less than n (in our previous example $k = n = 3$).

Component form can also be used in other phases of the key lifecycle, as we will see in Section 10.4 and Section 10.5.

10.3.4 Public-key pair generation

Since key generation for public-key cryptography is algorithm-specific, we will not treat it in detail here. As for symmetric key generation:

- Public-key pair generation often requires the random generation of numbers.
- Relevant standards should be consulted before generating public-key pairs.

However, in contrast to symmetric key generation:

- Not every number in the 'range' of the keyspace of a public-key cryptosystem is a valid key. For example, for RSA the keys d and e are required to have specific mathematical properties (see Section 5.2.1). If we choose an RSA modulus of 1024 bits then there are, in theory, 2^{1024} candidates for e or d. However, only some of these 2^{1024} numbers *can* be an e or a d, the other choices are ruled out.
- Some keys in public-key cryptosystems are chosen to have a specific format. For example, RSA public keys are sometimes chosen to have a specific format that results in them being 'faster than the average case' when they are used to compute exponentiations, thus speeding up RSA encryptions (or RSA signature

verifications if the public key is a verification key). There is no harm in such a deliberate choice of key since the public key is not a secret value. Clearly, if similar restrictions were placed on a private key then an attacker might benefit from having many fewer candidate private keys to choose from.

- The generation of a key pair can be slow and complex. Some devices, such as smart cards, may not have the computational resources to generate key pairs. In such cases it may be necessary to generate key pairs off the card and import them.

Thus, while key generation is always a delicate part of the cryptographic lifecycle, particular care and attention needs to be paid to the generation of public-key pairs. We will also discuss this issue in Section 11.2.2.

10.4 Key establishment

Key establishment is the process of getting cryptographic keys to the locations where they will be used. This part of the key lifecycle tends either to be relatively straightforward, or very hard, to manage. Key establishment is generally hard when keys need to be shared by more than one party, as is the case for most symmetric keys. It is relatively straightforward when:

The key does not need to be shared. This applies to any keys that can be locally generated and do not need to be transferred anywhere, such as symmetric keys for encrypting data on a local machine. Of course, if such keys are not locally generated then key establishment becomes hard again! We will consider this issue for private keys in Section 11.2.2.

The key does not need to be secret. This applies mainly to public keys. In this case key establishment is more of a logistical problem than a security issue. We also discuss this in Section 11.2.2.

The key can be established in a controlled environment. In some cryptographic applications it is possible to establish all the required keys within a controlled environment before the devices containing the keys are deployed. This is often termed *key predistribution*. While this makes key establishment fairly easy, there are still issues:

- Some key establishment problems are transferred into 'device establishment' problems. However, these may be less sensitive. For example, key predistribution can be used to preload keys onto mobile phones (see Section 12.3.5) or set-top boxes for digital television services (see Section 12.5.4). In this case the provider still needs to keep track of which customer receives which device, but this is likely to be a simpler problem than trying to load cryptographic keys onto a device that is already in the hands of a customer.
- In environments suited to key predistribution, it can be challenging to conduct post-deployment key management operations, such as key change (see Section 10.6.2). In such cases it may be necessary to establish entirely new devices.

We have already discussed some important techniques for key establishment:

- In Section 9.4 we discussed AKE (Authentication and Key Establishment) protocols. Many symmetric keys are established by means of an AKE protocol of some sort. We noted that AKE protocols can be classified into key distribution and key agreement protocols.
- In Section 9.4.2 we discussed the Diffie–Hellman protocol, which forms the basis for the majority of AKE protocols based on key agreement.
- In Section 5.5.2 we discussed hybrid encryption, which is a very common method of key establishment in environments where public-key cryptography is supported.

The rest of this section will focus on some special techniques for conducting symmetric key establishment, all of which could be regarded as being particular types of AKE protocol.

10.4.1 Key hierarchies

One of the most widely used techniques for managing symmetric keys is to use a *key hierarchy*. This consists of a ranking of keys, with high-level keys being more 'important' than low-level keys. Keys at one level are used to encrypt keys at the level beneath. We will see shortly how this concept can be used to facilitate symmetric key establishment.

PHILOSOPHY BEHIND KEY HIERARCHIES

There are two clear advantages of deploying keys in a hierarchy:

Secure distribution and storage. By using keys at one level to encrypt keys at the level beneath, most keys in the system can be protected by the keys above them. This allows keys to be securely distributed and stored in encrypted form.

Facilitating scalable key change. As we will discuss further in Section 10.6.2, there are many reasons why keys may need to be changed. Some of these reasons are concerned with the risk of a key being compromised, which is arguably more likely to happen to 'front-line facing' keys that are directly used to perform cryptographic computations, such as encryption of transmitted data. Use of a key hierarchy makes it relatively easy to change these low-level keys without the need to replace the high-level keys, which are expensive to establish.

However, one significant problem remains: how to distribute and store the keys at the top level of the hierarchy? The use of a key hierarchy focusses the key management problems onto these top-level keys. Effort can thus be concentrated on key management solutions for the top-level keys. The payoff is that if we get

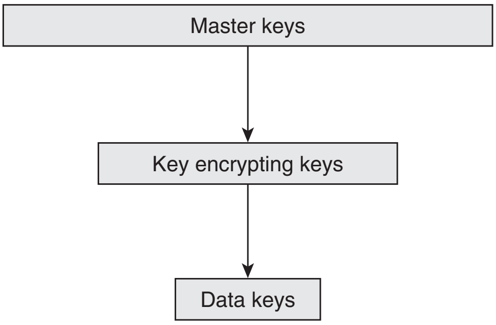

Figure 10.2. A three-level key hierarchy

the management of the top-level keys ‘right’, the management of the rest of the keys can be taken care of using the key hierarchy.

A SIMPLE KEY HIERARCHY

The idea of a key hierarchy is best illustrated by looking at a simple example. The ‘simple’ example shown in Figure 10.2 provides a key hierarchy that is probably good enough (and maybe even more complex than necessary) for the majority of applications. The three levels of this hierarchy consist of:

Master keys. These are the top-level keys that require careful management. They are only used to encrypt key encrypting keys. Since the key management of master keys is expensive, they will have relatively long lifetimes (perhaps several years).

Key encrypting keys. These are distributed and stored in encrypted form using master keys. They are only used to encrypt data keys. Key encrypting keys will have shorter lifetimes than master keys, since they have greater exposure and are easier to change.

Data keys. These are distributed and stored in encrypted form using key encrypting keys. These are the working keys that will be used to perform cryptographic computations. They have high exposure and short lifetimes. This may simply correspond to the lifetime a single session, hence data keys are often referred to as *session keys*.

Since the longevity of the keys in the hierarchy increases as we rise up the hierarchy, it is often the case that so does the length of the respective keys. Certainly keys at one level should be *at least as long* as keys at the level beneath. Note that the ‘middle layer’ of key encrypting keys may well be unnecessary for many applications, where it suffices to have master keys and data keys.

MANAGING THE TOP-LEVEL KEYS

Top-level (master) keys need to be securely managed, or the whole key hierarchy is compromised. Most key management systems using key hierarchies will employ hardware security modules (HSMs) to store master keys. These top-level keys will never leave the HSMs in unprotected form. HSMs will be discussed in more detail in Section 10.5.3.

The generation of master keys is an extremely critical operation. Master keys are commonly generated, established and backed up in component form. We discussed component form in Section 10.3.3 and will discuss the establishment process for a master key from components in Section 10.7.2. If a master key needs to be shared between two different HSMs then one option is to generate the same master key from components separately on each HSM. An alternative is to run a key agreement protocol (see Section 9.4.2) between the two HSMs in order to establish a shared master key.

SCALABLE KEY HIERARCHIES

The notion of a key hierarchy works fine for a relatively simple network, but quickly becomes unmanageable for large networks. Consider a simple two-level hierarchy consisting of only master and data keys. If we have a network of n users, then the number of possible pairs of users is $\frac{1}{2}n(n-1)$. This means that, for example, if there are 100 users then there are $\frac{1}{2} \times 100 \times 99 = 4950$ possible pairs of users. Hence, in the worst case, we might have to establish 4950 separate master keys amongst the 100 HSMs in the network, which is not practical.

Alternatively, we could install the same master key in all HSMs. Data keys for communication between Alice and Bob could then be derived from the common master key and Alice and Bob's identities. However, compromise of Alice's HSM would now not only compromise data keys for use between Alice and Bob, but data keys between *any* pair of users in the network. This is not normally acceptable.

In such cases it is common to deploy the services of a trusted third party, whom all the users trust, which we will refer to as a *key centre* (KC). The idea is that each user in the network shares a key with the KC, which acts as a 'go between' any time any pairs of users require a shared key. In this way we reduce the need for 4950 master keys in a network of 100 users to just 100 master keys, each one shared between a specific user and the KC.

There are two key distribution appoaches to acquiring shared keys from a KC. We illustrate these using a very simple scenario. In each case we assume that Alice wishes to establish a shared data key K with Bob. We will also assume that both Alice and Bob have respectively established master keys K_{AC} and K_{BC} with the KC, and that a simple two-level key hierarchy is being employed. The two approaches are:

Key translation. In this approach the KC simply translates an encrypted key from encryption using one key to encryption using another. In this case the KC is acting as a type of *switch*. This process is depicted in Figure 10.3 and runs as follows:

1. Alice generates a data key K, encrypts it using K_{AC} and sends this to KC.

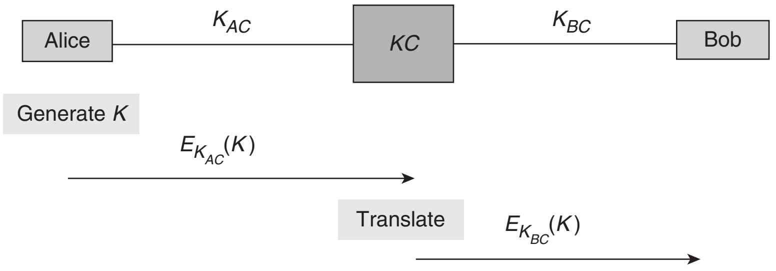

Figure 10.3. Key translation

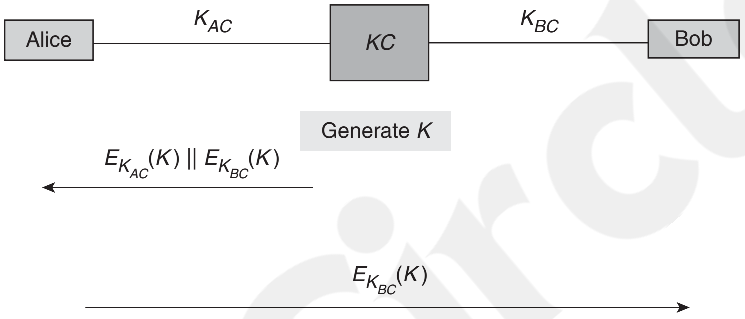

Figure 10.4. Key despatch

2. KC decrypts the encrypted K using K_{AC}, re-encrypts it using K_{BC} and then sends this to Bob.
3. Bob decrypts the encrypted K using K_{BC}.

Key despatch. In this approach the KC generates the data key and produces two encrypted copies of it, one for each user. This process, which we have already encountered in Section 9.4.3, is depicted in Figure 10.4 and runs as follows:

1. KC generates a data key K, encrypts one copy of it using K_{AC} and another copy of it using K_{BC}, and then sends both encrypted copies to Alice.
2. Alice decrypts the first copy using K_{AC} and sends the other copy to Bob.
3. Bob decrypts the second copy using K_{BC}.

The only real difference between these two key distribution approaches is who generates the data key. Both approaches are used in practice.

An alternative way of deploying key hierarchies for networks of many users is to use public-key cryptography and have a master public-key pair. We can think of hybrid encryption (as discussed in Section 5.5.2) as a two-level key hierarchy where the public key plays the role of a master key, which is then used to encrypt data keys. However, it is important to recognise that this approach comes with its own issues and does not do away with the need for a trusted third party of some sort. This now takes the form of a certificate authority, which we discuss in Section 11.1.2.

10.4.2 Unique key per transaction schemes

We now look at a different way of establishing a cryptographic key. *Unique key per transaction* (UKPT) schemes are so called because they establish a new key each time that they are used.

MOTIVATION FOR UKPT SCHEMES

Most of the previous key establishment mechanisms that we have discussed involve one, or both, of the following:

- Use of long-term (top-level) secret keys, for example, the use of master keys or key encrypting keys in key hierarchies.
- A special transfer of data explicitly for the purposes of key establishment. This applies to every technique that we have discussed thus far, except key predistribution.

While these are acceptable features in many environments, they may not be desirable in others. The first requires devices that can securely store and use long-term keys, and the second introduces a communication overhead.

One of the reasons that most of the previous schemes require these features is that the new key that is being established has been generated *independently*, in the sense that it has no relationship with any existing data (including existing keys). An alternative methodology is to generate new keys by deriving them from information already shared by Alice and Bob. We discussed derivation in Section 10.3.2, where the shared information was an existing key known to Alice and Bob. However, importantly, this shared information does not need to be a long-term secret key. Rather it can be a short-term key, other data, or a combination of the two.

If key derivation is used to generate new keys then the processes of key generation and key establishment essentially 'merge'. This brings several advantages:

1. Alice and Bob do not need to store a long-term key;
2. Alice and Bob are not required to engage in any special communication solely for the purpose of key establishment;
3. Key generation and establishment can be 'automated'.

APPLICATION OF UKPT SCHEMES

UKPT schemes adopt the methodology we have just described by updating keys using a key derivation process after each use. A good example of an application of UKPT schemes is retail point-of-sale terminals, which are used by merchants to verify PINs and approve payment card transactions. The advantages of a UKPT scheme all apply to this scenario:

1. Terminals have limited security controls, since they must be cheap enough to deploy widely. In addition, they are typically located in insecure public environments such as stores and restaurants. They are also portable, so that they can easily be moved around, hence easily stolen. (This what we will refer to as a Zone 1 key storage environment in Section 10.5.3.) It is thus undesirable that they contain important top-level keys.
2. Transactions should be processed speedily to avoid delays, hence efficiency is important.
3. Terminals may be managed and operated by unskilled staff, hence full automation of the key establishment process is a necessity.

EXAMPLE UKPT SCHEMES

Consider a UKPT scheme operating between a merchant *terminal* and a *host* (a bank or card payment server). The terminal maintains a *key register*, which is essentially the running 'key' that will be updated after every transaction. We will describe a generic UKPT scheme in terms of the protocol that is run between the terminal and the host during a transaction. Note:

- We assume at the start of the protocol that the terminal and the host share an initial value that is stored in the terminal key register. This may or may not be a secret value (it might just be a seed designed to initiate the process).
- We will describe a simple protocol that uses a single *transaction key* to compute MACs on the exchanged messages. In reality, such protocols may be slightly more complex since, for example, an encryption key might also be needed to encrypt the PIN of the card.

Figure 10.5 illustrates our generic UKPT scheme:

1. The terminal derives the transaction key using the contents of the key register and shared information that will be available to the host.
2. The terminal sends a *request* message to the host. The transaction key is used to compute a MAC on the request message.
3. The host derives the transaction key (the technique for doing this varies between schemes, as we will shortly illustrate).
4. The host validates the MAC on the request message.
5. The host sends a *response* message to the terminal. The transaction key is used to compute a MAC on the response message.
6. The terminal validates the MAC on the response message.
7. The terminal updates the contents of the key register.

In order to produce a real UKPT scheme from the generic UKPT scheme of Figure 10.5, we need to answer three questions:

1. What is the initial value in the terminal key register?

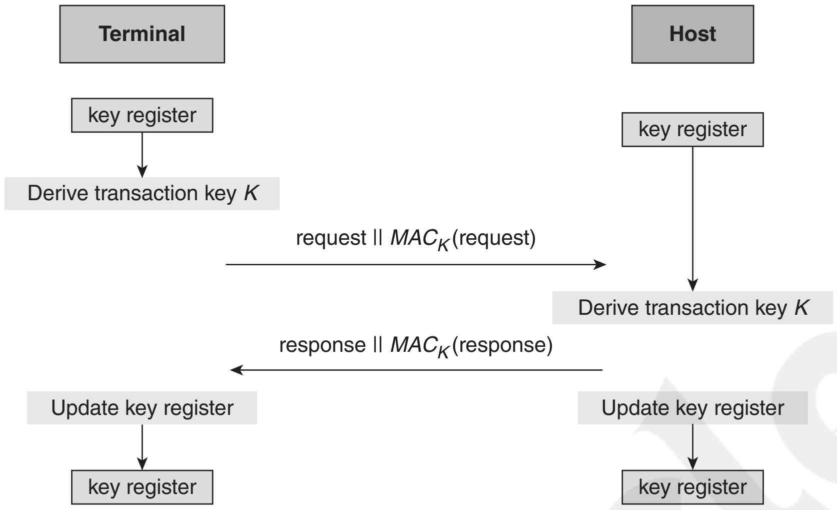
Terminal
Host
key register
key register
Derive transaction key K
request || MAC_K(request)
Derive transaction key K
response || MAC_K(response)
Update key register
Update key register
key register
key register

Figure 10.5. Generic UKPT scheme

2. How should the transaction key be derived so that the terminal and host derive the same key?
3. How should the terminal key register be updated so that the terminal and host update to the same value?

There are various ways in which these operations could be conducted in such a way that the terminal and host stay synchronised. Two examples of real UKPT schemes are:

Racal UKPT scheme. This scheme answers the three questions as follows:

1. The initial value is a secret seed, which is agreed between the terminal and the host.
2. The host maintains an identical key register to the terminal. The transaction key is derived from the key register and the card data (more precisely, the primary account number on the card), both of which are known by the terminal and the host.
3. At the end of the protocol, the new key register value is computed as a function of the old key register value, the card data (primary account number) and the transaction data (more precisely, the two *MAC residues* of the MACs on the request and response messages, both of which can be computed by the host and the terminal but neither of which are transmitted during the protocol, see Section 6.3.3). Both the terminal and the host conduct the same computation to update their key registers.

Derived UKPT scheme. This scheme is supported by Visa, amongst others, and answers the three questions as follows:

1. The initial value is a unique initial key that is installed in the terminal.
2. The transaction key is derived by the terminal from the contents of the terminal key register, a transaction counter, and the terminal's unique identifier. The host has a special base (master) key. The host does not need to maintain a key register, but can calculate this same transaction key from the base key, the transaction counter and the terminal identifier.
3. At the end of the protocol the new terminal key register value is derived from the old key register value and the transaction counter. The host does not need to store this value because it can compute transaction keys directly, as just described.

One of the most attractive features of the Racal UKPT scheme is that it has an in-built audit trail. If a transaction is successful then, since it relies on all previous transactions, this also confirms that the previous transactions were successful. A potential problem with the Racal UKPT scheme is synchronisation in the event that a transaction does not complete successfully.

The Derived UKPT scheme has the significant advantage that the host does not need to maintain key registers and can derive transaction keys directly. However, it suffers from the problem that an attacker who compromises a terminal (and thus obtains the value in the key register) will be able to compute future transaction keys for that terminal. In the Racal UKPT scheme such an attacker would also need to capture all the future card data. The Derived UKPT scheme also requires a careful initialisation process, since compromise of the terminal initial key leads to the compromise of all future transactions keys.

The problems with these UKPT schemes can all be addressed through careful management. UKPT schemes are very effective key management systems for addressing the difficulties associated with key establishment in the types of environment for which they are designed.

10.4.3 Quantum key establishment

We close our discussion of key establishment with a few words about a technique that has captured the public attention, but whose applicability remains to be determined.

MOTIVATION FOR QUANTUM KEY ESTABLISHMENT

In Section 3.1.3 we discussed one-time pads and argued that they represented 'ideal' cryptosystems that offered perfect secrecy. However, in Section 3.2.1 we pointed out substantial practical problems with implementing a one-time pad. These were all essentially key management problems, perhaps the most serious being the potential need to establish long, randomly generated symmetric keys at two different locations.

Tantalisingly, if a way could be found of making key establishment ‘easy’ then perhaps a one-time pad could be used in practice. This is one motivation for *quantum key establishment*, which is an attempt to make the establishment of long, shared, randomly generated symmetric keys ‘easy’.

Note that quantum key establishment is often inappropriately described as ‘quantum cryptography’. The latter name suggests that it is something to do with new cryptographic algorithms that are suitable for use to protect against quantum computers (see Section 5.4.4). Quantum key establishment is in fact a technique for establishing a conventional symmetric key, which can then be used in any symmetric cryptosystem, including a one-time pad. Of course, it does have some relevance to quantum computers, since a one-time pad still offers perfect secrecy if an attacker is in the fortunate position of having a quantum computer, whereas many modern encryption algorithms would no longer be secure (see Section 5.4.4). Nonetheless, quantum key establishment is only what it claims to be, a technique for key establishment.

THE BASIC IDEA

Quantum key establishment takes place over a *quantum channel*. This is typically instantiated by an optical fibre network or free space. Alice and Bob must have devices capable of sending and receiving information that is encoded as quantum states, often termed *qubits*, which are the quantum equivalent of bits on a conventional communication channel. These qubits are represented by *photons*. In a conventional communication channel, one simple way of establishing a symmetric key is for Alice to generate a key and then send it to Bob. The problem with this approach is that an attacker could be listening in on the communication and thus learn the key. Even worse, neither Alice nor Bob would be aware that this has happened.

The basic idea behind quantum key establishment is to take advantage of the fact that in a quantum channel such an attacker cannot ‘listen in’ without changing the information in the channel. This is a very useful property, which Alice and Bob can exploit to test whether an attacker has been listening to their communication.

The most well known quantum key establishment protocol is the *BB84 protocol*. While the following conceptual overview of this protocol is simplified and omits important background information, it should provide a flavour of the basic idea. The BB84 protocol involves the following steps:

1. Alice randomly generates a stream of qubits, and sends these as a stream of polarised photons to Bob.
2. Bob measures them using a *polarisation detector*, which will return either a 0 or a 1 for each photon.
3. Bob contacts Alice over a conventional authenticated channel (perhaps a secure email, a telephone call, or a cryptographically authenticated channel), and Alice then provides him with information that probably results in Bob discarding

approximately 50% of the measurements that he has just taken. This is because there are two different types of polarisation detector that Bob can use to measure each photon, and if he chooses the wrong one then the resulting measurement has only a 50% chance of being correct. Alice advises him over the authenticated channel which polarisation detector she used to encode each qubit, and Bob throws away the returns of all the wrongly measured photons.

4. Alice and Bob now conduct a check over the authenticated channel on the stream of bits that they think they have just agreed upon. They do this by randomly choosing some positions and then check to see if they both agree on the bits in these positions. If they find no discrepancies then they throw away the bits that were used to conduct the check, and form a key from the bits that they have not yet checked.

To understand why this protocol works, consider the position of an attacker. This attacker can take measurements of photons on the quantum channel and can listen in to all the discussion on the authenticated channel. However, if the attacker chooses to measure a photon on the quantum channel, and if the attacker uses the wrong detector (which will happen in approximately 50% of the measurements) then this process changes the polarisation, which in turn leads to Bob obtaining an incorrect output bit. Thus such activity will be detected with high probability by Alice and Bob when they do the 'spot check' of agreed bits. Alice and Bob can set this probability to be as high as they like, simply by increasing the number of bits that they check.

QUANTUM KEY ESTABLISHMENT IN PRACTICE

The theory behind quantum key establishment is certainly intriguing. However, the motivation for quantum key establishment is all about overcoming practical problems. Is quantum key establishment, itself, practical?

There are a number of substantial limitations of quantum key establishment. These include:

Distance limitations. Implementations of quantum key establishment are improving all the time. Nonetheless, it has still only been demonstrated to work over limited distances. For example, in 1988 it was shown to work over a 30 cm distance. This had improved by 2010 to around 150 km in an optical fibre network, and several demonstration networks had been built that used quantum key establishment. It is believed that a significant extension of the underlying mechanisms will be required if distances over 400 km are ever to be achieved in optical fibre networks. In contrast, there are no technical limits on the distance over which most conventional key establishment techniques can be used.

Data rates. There are limits to the rate at which key material can be exchanged over the quantum channel. This is also related to the distance over which the key establishment is being conducted.

Cost. Use of quantum key establishment requires expensive hardware devices and suitable quantum channels. Although the associated costs will doubtless reduce over time, most conventional key establishment techniques do not require such special technology.

The need for conventional authentication. Quantum key establishment requires a conventional means of authentication to be used. For example, in the BB84 protocol it is important that Alice and Bob establish an authenticated channel. How will they do this? One way is, of course, to use symmetric cryptography. So how do they establish the key used for authentication? If a conventional key establishment technique is used then the security of the quantum key establishment relies on the security of conventional key establishment. It could be argued that very little has been gained.

However, the biggest issue with quantum key establishment is really *whether it is needed at all*. Most of the other key establishment mechanisms that we have discussed are all very effective when used with strong cryptographic algorithms such as AES to support them. Are the costs of quantum key establishment really justifiable?

It is worth noting, however, that quantum key establishment does permit the continuous establishment of randomly generated keys. Quantum key establishment is probably best considered as a technique that has potential for high-security applications where it is felt that use of a one-time pad is merited. While it does rely on conventional authentication, it could be argued that this is not a big problem since the authenticated channel is only required for a relatively short period of time. In comparison, data protected using the resulting key may be kept secure for a long time. Nonetheless, it would seem unlikely that we will see widespread adoption of quantum key establishment.

10.5 Key storage

Secret keys need to be protected from exposure to parties other than the intended 'owners'. It is thus very important that they are stored securely. In this section we consider how to store keys. We will also discuss how to manage the potential loss or unavailability of keys.

10.5.1 Avoiding key storage

The best solution of all would be not to store cryptographic keys anywhere and just generate them on the fly whenever they are required. This is possible in some applications. Since the same key must be generated on the fly every time we need to use it, we require a deterministic key generator (see Section 8.1.4) to generate the key. Recall from Section 8.1.4 that deterministic generators require a seed, so we will require this seed to be consistently used each time we generate the key. But the seed also needs to be kept secure, so where do we store the seed?

For most applications that use this technique, the seed is stored inside the human brain in the form of a passphrase or strong password. This is precisely the technique adopted by some cryptographic software to protect private keys, which are encrypted using a key encrypting key (see Section 12.7.1) and then stored. The user generates a passphrase, which they are required to remember. The passphrase is then used as a seed for a deterministic generator that generates the key encrypting key on the fly. The key encrypting key is then used to decrypt the encrypted private key. The obvious drawback of this process is that the security of the stored key is now dependent on the security of the seed (passphrase) that is used to generate the key encrypting key. However, this is a pragmatic solution that represents a balance between security and usability that is appropriate for many types of application.

But it is not always possible to avoid storing a key. For example:

- Suppose that a symmetric key is being used to secure communication between Alice and Bob, who are in different locations. In some applications Alice and Bob may be able to locally generate the key precisely when they require it. However, in many other applications the key will need be stored somewhere, at least for a short while (for example, if Alice and Bob are both issued with the key in advance by a mutually trusted third party).
- Many uses of cryptography require long-term access to certain keys. For example, keys used for secure data storage may themselves need to be stored for a long time in order to facilitate future access to the protected data.
- Public-key pairs are expensive to generate. Generating them precisely when they are needed is inefficient. In many cases this is impossible, since the devices on which the private keys reside (for example, a smartcard) may have no user interface. Thus private keys almost always need to be securely stored.

10.5.2 Key storage in software

One option for storing a cryptographic key is to embed the key into software. As mentioned in Section 3.2.4, conducting any part of the cryptographic process in software comes with inherent risks. However, storing keys in software is much cheaper than storing keys in hardware so, as is often the case, the security risks have to be traded off against the cost benefits.

STORING KEYS IN THE CLEAR

By far the cheapest, and the riskiest, approach is to store keys in the clear in software. In other words, regard keys as pieces of data that are stored on a hard drive as unprotected data. Crazy though this sounds, this is often done. One common approach is to try to 'hide' the keys somewhere in the software. This is 'security by obscurity', which is always dangerous since it relies on the hider of the keys being 'smarter' than any attacker. In addition, there are two fundamental problems with hiding cryptographic keys in software:

1. The developer who designs the software will know where the keys are, so there is at least one potential attacker out there who knows where to look for the keys.
2. Assuming that the hidden keys are specific to different versions (users) of the software, an attacker who obtains two versions of the software could compare them. Any locations where differences are noted are potential locations of material relating to a key.

Even if these fundamental problems do not apply to a specific application, the underlying concerns about unprotected keys being stored in software are sufficiently serious that this approach is normally best avoided. Indeed, software storage of keys in the clear is explicitly forbidden by many key management systems and standards.

STORING KEYS USING CRYPTOGRAPHY

Fortunately, we are already very familiar with a technique that can be employed to protect data that resides in software on a computer system. We can encrypt it! While this might seem the obvious thing to do, it has only moved the goalposts, not removed them. In order to encrypt a key, we require a key encrypting key. So where do we store the key encrypting key? If it is a public key, where do we store the corresponding private key?

There are really only four options:

Encrypt it with yet another key. So where do we store that key?

Generate it on the fly. This is a fairly common approach that we discussed in Section 10.5.1 and is often taken for applications where a hardware-based solution is not viable.

Store it in hardware. This is probably the most common approach but, obviously, requires access to a suitable hardware device. The key encrypting key remains on the hardware device, which is also where all encryption and decryption using this key is performed. We discuss hardware storage of keys in Section 10.5.3.

Store it in component form. We introduced the idea of component form in Section 10.3.3. It can also be used for key storage. By using components we make the task of obtaining a key harder since, in order to recover the key, all of the necessary components need to be obtained. However, we have only partially solved the storage problem, since we still have to store the components somewhere. As components are essentially keys themselves, hence not easily memorised, the most common way to store components is on hardware (such as a smart card). Thus component form is really a strengthening of a hardware-based solution, not an alternative.

10.5.3 Key storage in hardware

The safest medium in which to store a cryptographic key is hardware. There are, of course, different types of hardware device, with varying levels of security.

HARDWARE SECURITY MODULES

The securest hardware storage media for cryptographic keys are *hardware security modules* (HSMs). These dedicated hardware devices that provide key management functionality are sometimes known as *tamper-resistant devices*. Many HSMs can also perform bulk cryptographic operations, often at high speed. An HSM can either be peripheral or can be incorporated into a more general purpose device such as a point-of-sale terminal.

While we have chosen to introduce HSMs as mechanisms for the secure storage of cryptographic keys, it is important to appreciate that HSMs are often used to enforce other phases of the key lifecycle.

Keys stored on HSMs are physically protected by the hardware. If anyone attempts to penetrate an HSM, for example, to extract a key from the device, tamper-resistant circuitry is triggered and the key is normally deleted from the HSM's memory. There are various techniques that can be used to provide tamper resistance. These include:

Micro-switches. A simple mechanism that releases a switch if an HSM is opened. This is not particularly effective, since a clever attacker can always drill a hole and use glue to force the switch off.

Electronic mesh. A fine-gauge electronic mesh that can be attached to the inside of an HSM case. This mesh surrounds the sensitive components. If broken, it activates the tamper-detection circuitry. This mechanism is designed to protect against penetrative attacks, such as drilling.

Resin. A hard substance, such as epoxy resin, that can be used to encase sensitive components. Sometimes electronic mesh is also embedded in resin. Any attempt to drill through the resin, or dissolve the resin using chemicals, will generally damage the components and trigger the tamper-detection circuitry.

Temperature detectors. Sensors that are designed to detect variations in temperature outside the normal operating range. Abnormal temperatures may be an indication of an attack. For example, one type of attack involves, literally, freezing the device memory.

Light-sensitive diodes. Sensors that can be used to detect penetration or opening of an HSM casing.

Movement or tilt detectors. Sensors that can detect if somebody is trying to physically remove an HSM. One approach is to use mercury tilt switches, which interrupt the flow of electrical current if the physical alignment of an HSM changes.

Voltage or current detectors. Sensors that can detect variations in voltage or current outside the normal operating range. Such anomalies may be indication of an attack.

Security chips. Special secure microprocessors that can be used for cryptographic processing within an HSM. Even if an attacker has penetrated all the other defences of an HSM, the keys may still remain protected inside the security chip.

Different HSMs may use different combinations of these techniques to build up a layered defence against attacks. An HSM is also typically backed up by a battery, so that it cannot be attacked simply by switching off the power supply.

KEY STORAGE ON AN HSM

There is at least one key, often referred to as a *local master key* (LMK), that resides inside the HSM at all times. Some HSMs may store many LMKs, each having its own specific use. Any other keys that need to be stored can either be:

1. stored inside the HSM;
2. stored outside the HSM, encrypted using an LMK.

In the latter case, when a key stored outside the HSM needs to be used, it is first submitted to the HSM, where it is recovered using the LMK and then used.

This approach places a great deal of reliance on the LMK. It is thus extremely important to back up the LMK (see Section 10.5.5) in order to mitigate against loss of the LMK. Such loss can occur if the HSM fails, or if it is attacked, since the tamper-resistance controls are likely to delete the HSM memory. Indeed this applies to any keys that are only stored inside the HSM. Thus we can see that the issue of whether to store a key inside or outside the HSM involves a tradeoff between:

Efficiency – storing keys inside the HSM is more efficient in terms of processing speed since they do not need to be imported and then recovered before use.

Need for backups – since every key only stored inside the HSM needs to be securely backed up, perhaps in component form.

OTHER TYPES OF HARDWARE

While HSMs are the securest hardware devices on which to store keys, there are numerous other hardware devices offering less security. Some of these devices might include some of the tamper-resistance measures that we outlined for HSMs, while others may just rely on the hardware itself to provide some resistance to attack.

One class of hardware devices are smart tokens, including smart cards, which we first discussed in Section 8.3.3. Smart tokens are designed to be portable and cheap, so the security measures deployed to protect them are limited. Thus while smart tokens are normally appropriate media for storing keys specific to a user, for example, the type of token used in Section 8.5 for generating dynamic passwords, they are typically not secure enough to store cryptographic keys that are critical for an entire system, such as a system master key.

COMMUNICATING WITH HARDWARE

The use of hardware is a good means of protecting stored keys, however, it relies on a secure interface between the processes outside the hardware and the processes inside the hardware. For example, without a secure interface it might be possible for an unauthorised party to take an encrypted key from a database outside the hardware and 'persuade' the hardware to decrypt and then use this key on their behalf to reveal the result of a cryptographic computation. The problem is that the hardware responds to external calls, which can come from any number of different applications. Most hardware requires an *Application Programming Interface* (API) that contains a large number of different commands such as generate key, verify PIN, validate MAC, etc. It is therefore possible that an attacker could utilise these commands.

The security of this interface relies on access control to applications and devices, which in turn is related to the security of the hardware computing platform and the physical security surrounding it. Thus HSMs for important applications such as banking systems are always located in tightly controlled environments, both logically and physically.

In order to exploit weaknesses in an API, an attacker needs to write an application and have communication access to the hardware. Such an attacker would probably need to be a 'privileged' insider. Nonetheless, some proof-of-concept attacks have been publicised against the APIs of commercial HSMs. We will give an example of an API attack in Section 10.6.1.

EVALUATING HARDWARE

Since hardware is often used for critical components of a key management system, it is essential to have high confidence that it is sufficiently secure to fulfill its role. Obviously, it is not in a security hardware vendor's interests to produce insecure products, so they normally maintain high levels of vigilance regarding the security of their products. They also spend a lot of time reviewing and analysing the associated APIs. There are several organisations that carry out independent evaluations of the physical protection offered by hardware, particularly HSMs. There are also standards for HSM security, the most important of which is FIPS 140, against which most HSMs are evaluated.

10.5.4 Key storage risk factors

The risks to key storage media depend not only on the devices on which keys are stored, but also on the environments within which the devices reside. This relationship is indicated in Figure 10.6, which identifies four zones based on different environmental and device controls.

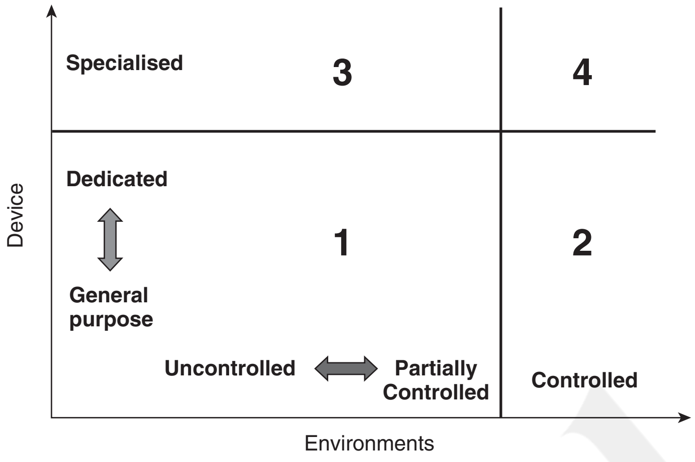

Figure 10.6. Key storage risk zones

The two dimensions depicted in Figure 10.6 represent:

Environments, which range from

Uncontrolled: public environments such as shops and restaurants, where it is not possible to implement strict access control mechanisms;

Partially controlled: environments such as general offices and homes, where it is possible to implement basic access control mechanisms (for example, a physical door key);

Controlled: environments such as high-security offices and military installations, where it is possible to implement strong access control mechanisms (for example, biometric swipe cards).

Devices, which range from

General purpose: general devices running conventional operating systems with their default in-built security controls (for example, a laptop);

Dedicated: dedicated devices that offer some specialist security controls, such as limited tamper resistance (for example, a point-of-sale terminal or a mobile phone);

Specialised: specialised devices whose main functionality is to provide security (for example, an HSM).

The four zones identified in Figure 10.6 are mainly conceptual, but illustrate the importance of both dimensions.

Zone 1. This is the lowest security zone and thus offers the highest risk. However, for many applications this may provide sufficient security. For example, a key stored in encrypted form on the hard disk of a home PC may well be good enough for protection of the user's personal files. Likewise, any keys stored under the limited protection of a portable point-of-sale terminal are probably still secure from anything other than attacks from experts.

Zone 2. The security offered by Zone 1 devices is increased substantially when they are moved into a controlled environment. In the extreme, a key stored in the clear in software on a general PC provides excellent security if the PC is not networked and is kept in a physically secure room with an armed guard at the door! More realistically, encrypted keys stored on PCs that are located in an office with strong physical security (such as smart card access control to the rooms) and good network security controls should have better protection than those on a PC located in a public library or an internet cafe.

Zone 3. Specialised devices sometimes have to be located in insecure environments because of the nature of their application. A good example is provided by Automated Teller Machines (ATMs), which need to be 'customer facing'. Such devices are thus exposed to a range of potentially serious attacks that are made possible by their environment, such as an attacker attempting to physically remove them with the intention of extracting keys back in a laboratory.

Zone 4. The highest-security zone is provided when a specialist device is kept in a controlled environment. This is not just the most secure, but the most expensive zone within which to provide solutions. This level of security is nonetheless appropriate for important keys relating to high-security applications such as data processing centres, financial institutions, and certification authorities.

Note that this conceptual 'model' could easily be extended. For example, we have not considered the different ways in which keys stored on the devices are activated (see Section 10.6.3).

10.5.5 Key backup, archival and recovery

We have spent most of our discussion about cryptography assuming that the use of cryptography brings security benefits. However, there are situations where use of cryptography can potentially have damaging consequences. One such situation arises if a cryptographic key becomes 'lost'. For example:

1. data stored in encrypted form will itself be lost if the corresponding decryption key is lost, since nobody can recover the data from the ciphertext;
2. a digital signature on a message becomes ineffective if the corresponding verification key is lost, since nobody has the ability to verify it.

The first scenario illustrates the potential need for *key backup* of critical secret keys. The second scenario more broadly illustrates the potential need for *key archival*, which is the long-term storage of keys beyond the time of their expiry.

Note that because key archival tends to apply to keys after they have expired, it appears in the key lifecycle of Figure 10.1 as a process occurring after key usage. However, we have included it in this section as it is closely related to key backup.

KEY BACKUP

It can be surprisingly easy to 'lose' critical cryptographic keys. As we discussed in Section 10.5.3, important keys are often stored on HSMs. An obvious attack against a Zone 3 (see Figure 10.6) HSM would be to physically attack the HSM to the point that one of its tamper-resistant triggers is activated and the device wipes its memory. The attacker does not learn the keys stored on the device but, without a backup, the potential impact on the organisation relying on the HSM is high. Even Zone 4 HSMs are subject to risks such as a careless cleaner bumping into a device and accidentally wiping its memory.

As noted at the start of this chapter, cryptographic keys are just pieces of data, so the backup of keys is not technically any more difficult than the backup of more general data. An obvious, but important, point is that the security of a key backup process must be as strong as the security of the key itself. For example, it would be unwise to back up an AES key by encrypting it using a DES key. The backed-up key will need to be stored on media that is subject to at least the same level of device and environmental security controls as the key itself. Indeed for the highest levels of key, the use of component form might be the only appropriate method for key backup.

KEY ARCHIVAL

Key archival is essentially a special type of backup, which is necessary in situations where cryptographic keys may still be required in the period between their expiry and their destruction. Such keys will no longer be 'live' and so cannot be used for any new cryptographic computations, but they may still be required. For example:

- There may be a legal requirement to keep data for a certain period of time. If that data is stored in encrypted form then there will be a legal requirement to keep the keys so that the data can be recovered. As an illustration, the London Stock Exchange requires keys to be archived for seven years.
- A document that has been digitally signed, such as a contract, may require the capability for that digital signature to be verified well beyond the period of expiry of the key that was used to sign it. Hence it may be necessary to archive the corresponding verification key to accommodate future requests. For example, Belgian legislation requires verification keys used for electronic signatures in online banking applications to be archived for five years (see also Section 12.6.5).

Managing the storage of archived keys is just as critical as for key backups. Once a key no longer needs to be archived, it should be destroyed.

KEY RECOVERY

Key recovery is the key management process where a key is recovered from a backup or an archive. Technically this is no harder than retrieving a key from any other type of storage, so the challenges all relate to the management processes that surround key recovery. Clearly it should not be possible to recover a key unless the recovery is suitably authorised.

Note that the term ‘key recovery’ is also associated with initiatives to force a ‘mandatory’ backup, also referred to as *key escrow*. The idea behind key escrow is that if any data is encrypted then a copy of the decryption key is stored (escrowed) by a trusted third party in such a way that, should it be necessary and the appropriate legal authority obtained, the decryption key can be obtained and used to recover the data. Such a situation might arise if the encrypted data is uncovered in the course of a criminal investigation. Many suggested key escrow mechanisms employed component form storage of escrowed keys in an attempt to reassure potential users of their security.

The idea of key escrow is fraught with problems, not the least being how to ‘force’ all users to use a cryptosystem that has an in-built key escrow facility. When routine key escrow was proposed in the early 1990s by the governments of a number of countries, including the UK and the US, business community concerns were sufficiently high that it was not pursued. Nonetheless, the ensuing debate about key escrow did help to raise the profile of the genuine need for key backup and key recovery in many cryptographic applications.

10.6 Key usage

Having considered the generation, establishment and storage of cryptographic keys, we now continue our study of the key lifecycle by looking at issues relating to key usage. The most important of these is key separation. We will also discuss the mechanics of key change, key activation and key destruction.

10.6.1 Key separation

The *principle of key separation* is that *cryptographic keys must only be used for their intended purpose*. In this section we consider why key separation is a good idea and discuss options for enforcing it.

THE NEED FOR KEY SEPARATION

The problems that can arise if key separation is not enforced can be serious. In many applications the need for key separation may be quite obvious. For example, it may be the case that encryption and entity authentication are conducted by distinct processes, each with their own particular requirements regarding key lengths. We will see an example of this in Section 12.2 when we look at WLAN security, where the process for encryption is ‘locked down’ across all applications, but the entity authentication process can be tailored to a specific application environment.

However, in other applications it may be tempting to use a key that has already been established for one purpose and then, for convenience, use it for some other purpose. We illustrate the potential dangers of doing this with two examples.

Example 1. Like passwords, PINs should not be stored anywhere in the clear. Hence PINs are often stored in encrypted form using a symmetric *PIN encrypting key*. This key should only ever be used to encrypt PINs. It should never be used to decrypt an encrypted PIN. In contrast, a normal symmetric data key is used both for encryption and decryption. If these two keys are somehow interchanged within an HSM then we have two serious problems. Firstly, it may become possible to decrypt and reveal a PIN. Secondly, it may not be possible to recover any normal data encrypted with the PIN encrypting key.

Example 2. Suppose we have an HSM with the following two security functions:

Function 1. This generates a four-digit PIN for a payment card by:

1. encrypting the card's 16-digit account number using a *PIN generation key*, and outputting the resulting ciphertext in hex form;
2. scanning the hex output for the first four digits in the range 0 to 9, but ignoring any in the range A to F, which are then used to form the PIN (additional measures need to be taken in the unlikely event that there are insufficient digits generated using this process to form a PIN);
3. outputting the resulting PIN in encrypted form.

Function 2. This generates a MAC on some input data by:

1. computing a simple CBC-MAC (using the version of CBC-MAC depicted in Figure 6.7, which is not recommended in practice) on the input data using a MAC key;
2. outputting the MAC in hex form.

Now suppose that an attacker is able to persuade the HSM to use the key from Function 1 to compute Function 2. In other words, the attacker is able to generate a MAC on the card's account number using the PIN generation key. The result will be that a MAC is output in hex form. Assuming that the same block cipher is used by both functions (in Function 1 to encrypt and in Function 2 to compute CBC-MAC) and since the account number is likely to be a short piece of data less than one block long, the MAC that is output will be the same as the encrypted account number that is produced in the first stage of Function 1. The attacker can then scan the MAC for the first four digits in the range 0 to 9, and hence determine the PIN.

These two examples both illustrate the potential dangers of not enforcing key separation. It might be argued that they are rather artificial examples for several reasons, one of which is that it should not be possible to force keys within an HSM to be used for purposes other than they were intended, especially as the keys

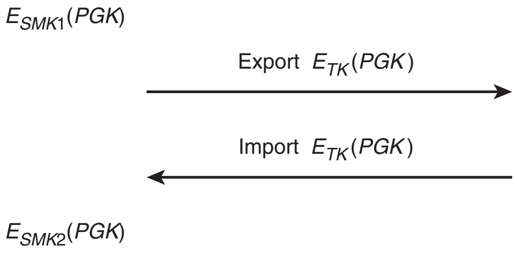

Figure 10.7. Key masquerade attack

involved will never appear in the clear outside the HSM. We now show how this could, at least in theory, happen.

As we will discuss shortly, one method of enforcing key separation in an HSM is to store keys in the HSM encrypted under a master key that is specific to one usage purpose. In this way, access to the key is directly linked to the use of a master key that identifies the key usage purpose. However, many HSMs have *export* and *import* functions that allow keys to be transferred between different HSMs. Keys are encrypted using a *transport key* during export and import. Figure 10.7 shows how this facility could, potentially, be used to change the apparent usage purpose of a key.

1. A PIN generation key *PGK* is stored on the HSM, encrypted by a *storage master key SMK*1, which is the local key on the HSM that is used to store PIN generation keys.
2. The HSM is instructed to export *PGK*. It thus decrypts the encrypted *PGK* using *SMK*1, then re-encrypts *PGK* using the transport key *TK*. This is then exported.
3. The HSM is then instructed by the attacker to import a new MAC key. The attacker submits *PGK*, encrypted under *TK*.
4. The HSM decrypts the encrypted *PGK* using *TK*, then re-encrypts it using storage master key *SMK*2, which is the HSM key used to store MAC keys. The HSM thus now regards *PGK* as a MAC key.

This attack will not be possible if different variants of transport key are used for separate export and import functions. However, due to interoperability issues between different vendors' solutions, transport key variants might not be permitted.

The above difficulties all arise through security weaknesses in the interface between the device on which the keys are stored and the outside world, which we already observed in Section 10.5.3 was an aspect of key management that can be problematic.

ENFORCING KEY SEPARATION

In order to avoid some of the problems that we have just illustrated, mechanisms are required to enforce key separation. This can be regarded as part of the wider provision of assurance of purpose of keys, which we discussed in Section 10.1.3. Keys are often unstructured bit strings, so there is no obvious way of distinguishing the purpose of a key from its basic form. There are two main techniques that can be used to enable the purpose of a key to be distinguished:

Encrypting a key using a specified variant key. This is a hardware-enforced method that we previously mentioned, which involves using specific higher-level keys to encrypt keys for particular purposes. For example, in Figure 10.7 the HSM used the key *SMK*1 to encrypt PIN generation keys, and key *SMK*2 to encrypt MAC keys. The HSM can interpret the usage based on the key encrypting key variant used. This technique can be applied to keys being distributed, as well as keys being stored. This method can be used to enforce any type of key separation (for example, the separation of the different MAC keys required to support the example in Section 7.2.3).

Embedding the key in a larger data block. This involves embedding the key into a larger data object that also includes a statement on the key usage. Three examples of this are:

Employing redundancy. As discussed in Section 4.4, a DES key has an effective length of 56 bits, but is usually a 64-bit value. Thus, there are 8 'spare' bits that can be used for other purposes. The original DES standard recommends that the spare bits be used to provide error detection in the event that a DES key becomes corrupted. Since the standard did not mandate this approach, the idea of *key tagging* was introduced. This allows the eight spare bits to define the key usage. When a key is presented in a command to an HSM, the tagging bits are checked by the HSM to ensure that the key is a valid key for the command that it is being used for.

Key blocks. This is a formatted data string that allows a key to be represented along with other data relating to the key. One example is the *ANSI TR-31* key block, which is depicted in Figure 10.8 and has the following fields:

- the *header* includes information that clarifies the purpose of the key;
- the *optional header* includes optional data such as the expiry date of the key;
- the *key* is encrypted using a suitable key encrypting key;
- the *authenticator* is a MAC on the rest of the key block, which provides data origin authentication (data integrity) of the key block data.

Public-key certificates. These are types of key block used to provide assurance of purpose for public keys. A public-key certificate often includes a field that

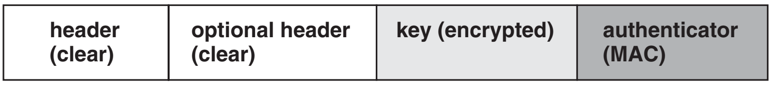

Figure 10.8. ANSI TR-31 key block

defines the key usage. We will discuss public-key certificates in more detail in Section 11.1.2.

It is important to realise that while distinguishing the purpose of a cryptographic key is helpful, it does not *enforce* key separation. Enforcement of key separation also requires procedural controls, which we discuss in Section 10.7.

KEY SEPARATION IN PRACTICE

At an intuitive level, the principle of key separation makes sense. Clearly, in an ideal world, having separate keys for separate purposes keeps things simple. However, the principle of key separation is precisely what it claims to be, namely, a 'principle'. Enforcing it does not come without potential costs. For example, enforcing it may mean that a key management system has more keys to manage than it would have if the principle is not enforced. It thus may be tempting to use keys for more than one purpose, just to reduce the number of keys that have to be managed. Of course, if we decide to use a particular symmetric key for both encryption and computing MACs then we could argue that the principle of key separation is still enforced since the 'purpose' of the key is both encryption and computing MACs! At least *thinking* about the principle of key separation has made us recognise these different uses of the key.

One example of a pragmatic compromise between the principle of key separation and practical issues is key derivation, which we discussed in Section 10.3.2. In this case a single *derivation key* might be stored and then used to derive separate encryption and MAC keys. Technically the same key is being used twice, since derivation is by means of a publicly known process, so the encryption and MAC keys are not as 'different' as we would like in the ideal world. Pragmatically, we are enforcing the principle of key separation by having two keys for the two different cryptographic operations.

The examples in this section have illustrated the dangers of not following the principle. The degree to which the principle of key separation is adhered to in a real application naturally depends on the specific priorities of the application environment. We will comment on key separation issues when we examine applications in Chapter 12.

10.6.2 Key change

Most key management systems require the ability to change keys.

THE NEED FOR KEY CHANGE

The need for a change of key tends to arise in two different types of circumstance:

Planned key changes. These will most likely occur at regular intervals. One reason for a planned key change might be the end of the key lifetime (see Section 10.2.1). Another reason might simply be to regularly practice key change procedures in preparation for an unplanned key change (the equivalent of a 'fire drill'). In some organisations this is the most common planned change, since their key lifetimes are very long.

Unplanned key changes. These may occur for a variety of reasons. Indeed, many of the reasons that we gave in Section 10.2.1 for having finite key lifetimes were to mitigate against unplanned events. An unplanned key change may thus be required if these unplanned events actually occur. For example:

- a key is compromised;
- a security vulnerability becomes apparent with the potential to lead to key compromise (such as an operating system vulnerability, a breakthrough in cryptanalysis, or a failure of a tamper-resistance mechanism in an HSM);
- an employee unexpectedly leaves an organisation.

Note that in some of these cases it may simply be enough to *withdraw* a key (remove it from active use), rather than change it. However, care must be taken before making this type of decision. For example, when an employee unexpectedly leaves an organisation on good terms then it may suffice to withdraw any personal keys that they held, such as any symmetric keys shared only by the employee and a central system, or any public-key pairs relating only to the employee. However, the employee might also have held *group keys* that are shared by several members of staff. It would be advisable to change these keys, since they are likely to remain in use after the employee's departure.

IMPACT OF KEY CHANGE

Key change can be a very expensive process, depending on the importance of the key being changed. An unplanned key change is particularly problematic, especially in the event of a key compromise, since it raises questions about any cryptographic operations that were conducted using the affected key, such as the confidentiality of any encrypted data. One likely consequence is that it will probably be necessary to also change any other keys encrypted using the affected key, which in turn raises questions about any cryptographic operations conducted using them.

The minimum impact of a key change is that a new key needs to be generated and established. However, the impact can be severe, especially in the case of high-level key compromise. For example, if a master key is compromised in a financial system then the resulting costs might include costs of investigation into the compromise, costs related to any 'rogue' transactions conducted using compromised keys, damage to reputation and loss of customer confidence. Recovery from unplanned key changes should be part of an organisation's wider disaster recovery and business continuity processes.

One situation where the damage caused by a key compromise might be limited is when the time of a cryptographic operation is logged and the time of key compromise is known. For example, in the case of a signature key being compromised, it might only be necessary to deem all signatures generated using the key after the time of compromise to be invalid.

MECHANISMS FOR CHANGING KEYS

As mentioned above, key change requires:

- generation and establishment of a new key;
- withdrawing the old key (and potentially destroying or archiving it).

Any of the mechanisms for these operations discussed elsewhere in this chapter could, in theory, be used to conduct these processes. Ideally, planned key changes should happen automatically and require very little intervention. For example, we saw in Section 10.4.2 that UKPT schemes automate planned key changes after every transaction. More intervention may be required in the case of unplanned key changes.

Obviously, high-level key changes are more complex to manage. For example, if a storage master key in an HSM goes through a planned change then all keys encrypted under the old storage master key will need to be decrypted, and then re-encrypted using the new storage master key. In this case, since the storage master key has not been compromised, there is no need to change all the keys that were encrypted using it.

Note that key changes are not always easy to facilitate. Indeed, the *migration* process from one key to another can be particularly challenging and, where possible, needs to be carefully planned for in order to make the transition as smooth as possible.

CHANGING PUBLIC-KEY PAIRS

It is perhaps slightly surprising that key change is, in general, simpler to perform for symmetric keys. This is 'surprising' because key change forces a new key establishment operation, which is usually a more difficult process for symmetric keys. There are two reasons why changing public-key pairs is normally more challenging:

Knowledge of public keys. Since symmetric keys need to be carefully 'positioned' in a network so that entities relying on them have the right keys, a key management system tends to be fully 'in control' of where its symmetric keys are located. This, at least in theory, makes withdrawing a symmetric key straightforward. In contrast, the 'public' nature of a public key means that a key management system may have little control over which entities have knowledge of a public key. Indeed, in open environments such as the Internet, a public key could be known by anyone.

Open application environments. Symmetric cryptography tends to be employed in closed environments. Thus any key management system handling symmetric keys should have mechanisms and controls in place for key establishment that can be reused for key change. In contrast, public-key cryptography tends to be used in open environments where this may be more challenging.

Since private and public keys are interdependent, any requirement to change one of them requires the other also to be changed. Changing a private key is arguably simpler than changing a symmetric key. However, changing public keys requires special mechanisms, which we will discuss in Section 11.2.3.

10.6.3 Key activation

When assessing the security of any key management system, it is important to pay attention to the processes by which keys are *activated*, by which we mean that their use is 'authorised'. We observed in Section 8.3.3 that one problem with using identity information based on a cryptographic key for entity authentication can be that the effective security is not as strong as expected. This problem arose because in the scenario under discussion the key was activated by a less-secure mechanism, such as a password.

This potential problem applies more widely than just to entity authentication. Indeed, in any use of cryptography we have to instruct the device performing the cryptographic computation to select a particular key for use. If this all takes place within the confines of an HSM then we may have little to be concerned about. However, in many applications key activation requires human interaction.

As an example, consider a signature key stored on a computer for digitally signing emails. If RSA is being used then this signature key might, reasonably, be up to 2048 bits long, which is clearly a value that the human owner of the key will not be capable of memorising. When the user decides to digitally sign an email, the user needs to instruct the email client to activate their signature key. Several scenarios may now apply, depending on how the key is stored (if at all) on the computer. These include:

Key stored on the computer in the clear. In this case the user might activate the key simply by entering an instruction, perhaps selecting the key from a list of potential keys stored on the computer. Key activation is thus possible for anyone with access to the computer. In this case the effective security of the keys is simply linked to the security required to access the computer itself, which perhaps just requires a valid username and password.

Key stored on the computer in encrypted form. The user might activate the key in this case by being prompted to provide some secret identity information, such as a passphrase. This passphrase would then be used to generate the key that can be used to recover the signature key. In this case the effective security is linked to the security of the passphrase.

Key generated on the fly. In this case the key is not stored on the computer, but is generated on the fly. The activation of the key is thus linked to the generation of the key. Again, one way of implementing this is to request some identity information such as a passphrase from the user. Thus the effective security of the key is again determined by the security of this passphrase.

Key stored off the computer. Another option is that the key is stored on a peripheral device. The key activation takes place when the user connects the device to the computer. In this case the effective security is linked to the security of the peripheral device. This process may also require a passphrase to be used.

The above scenarios are just examples, but what they all illustrate is that, even though a 2048-bit key is being used to secure the application, the key activation process plays a vital role in determining the effective security that is in place. In particular, the 2048-bit key might be activated by an attacker through:

- compromise of a security mechanism used to activate the key (such as a passphrase);
- access to a device on which the key is stored.

10.6.4 Key destruction

When a key is no longer required for any purpose then it must be destroyed in a secure manner. The point at which key destruction is required may either be:

1. when the key expires (the natural end of the key's lifetime);
2. when the key is withdrawn (before its expiry, in the event of unplanned events such as those discussed in Section 10.6.2);
3. at the end of a required period of key archival.

Since keys are a special type of data, the mechanisms available for destroying keys are precisely those for destroying general data. Since keys are sensitive data, secure mechanisms must be used. Suitable techniques are sometimes referred to as *data erasure* or *data sanitisation* mechanisms.

It goes without saying that simply deleting a key from a device is not sufficient if the key is to be truly destroyed. Not only does this not destroy the key, but operating systems may well have other (temporary) copies of the key in different locations. Even if the key was stored on the device in encrypted form, this may be useful to a determined attacker. Many secure data destruction mechanisms involve repeatedly overwriting the memory containing the key with randomly generated data. The number of overwrites is normally configurable. It should also be noted that any other media storing information about keys, such as paper, should also be destroyed. Relevant standards provide guidance on how to do this.

10.7 Governing key management

We have repeatedly stressed in this chapter that key management is the main interface between the technology of cryptography and the users and systems that rely on it. To this extent, key management is a small, but important, part of the wider management of the security of an information system.

For a private user managing keys on their own machine, key management may simply involve the selection of appropriate techniques for conducting each of the relevant phases of the key lifecycle. However, key management is a much more complex process for an organisation, due to the diversity of processes that affect key management, which we outlined in Section 10.1.1.

Key management within an organisation thus needs to be governed by rules and processes. In this section we will briefly discuss some of the issues involved in governing key management effectively within an organisation.

10.7.1 Key management policies, practices and procedures

Within an organisation, the most common way to govern key management is through the specification of:

Key management policies. These define the overall requirements and strategy for providing key management. For example, a policy might be that all cryptographic keys are stored only in hardware.
Key management practices. These define the tactics that will be used in order to achieve the key management policy goals. For example, that all devices using cryptography will have an in-built HSM.
Key management procedures. These document the step-by-step tasks necessary in order to implement the key management practices. For example, the specification of a key establishment protocol that will be used between two devices.

Clearly, different organisations will have different approaches to the formulation of key management policies, practices and procedures, but the important outcome of this process should be that key management governance is:

By design: in other words, that the entire key management lifecycle has been planned from the outset, and not made up in response to events as they occur.
Coherent: the various phases of the key lifecycle are considered as linked component parts of a larger unified process and designed with this ‘big picture’ in mind.
Integrated: the phases of the key management lifecycle are integrated with the wider requirements and priorities of the organisation.

For commercial organisations, it may also make sense to publicise key management policies and practices, since this can be used as a mechanism for increasing confidence in their security practices. This is particularly relevant for organisations providing cryptographic services, such as Certificate Authorities (see Section 11.2.3).

The formulation of key management policies, practices and procedures also facilitates the auditing of key management, which is part of the wider process of auditing security. This is because not only can the policies, practices and procedures themselves be scrutinised, but the effectiveness of their implementation can then be tested.

10.7.2 Example procedure: key generation ceremony

We illustrate the potential complexities of key management governance by giving an example of an important type of key management procedure that might be required by a large organisation. This is that of a *key ceremony*, which can be used to implement key generation from components (as discussed in Section 10.3.3). Note that the key in question could be a top-level (master) symmetric key or top-level (root) private key, which needs to be installed into an HSM. The key might be:

- a new key being freshly generated;
- an existing key being re-established (from backed-up stored components).

The participants are:

Operation manager: responsible for the physical aspects, including the venue, hardware, software and any media on which components are stored or transported;

Key manager: responsible for making sure that the key ceremony is performed in accordance with the relevant key management policies, practices and procedures;

Key custodians: the parties physically in possession of the key components, responsible for handling them appropriately and following the key ceremony as instructed;

Witnesses: responsible for observing the key ceremony and providing independent assurance that all other parties perform their roles in line with the appropriate policies, practices and procedures (this might involve recording the key ceremony).

The key ceremony itself involves several phases:

Initialisation. The operation manager installs and configures the required hardware and software, including the HSM, within a controlled environment. This process might need to be recorded by witnesses.

Component retrieval. The components required for the key ceremony, held by the relevant key custodians, are transported to the key ceremony location. These key custodians may be from different organisations (departments) and may not be aware of each others' identities.

Key generation/establishment. The key is installed onto the HSM under the guidance of the key manager. This process will involve the various key custodians taking part in the key ceremony, but not necessarily simultaneously (for example, it may be required that the key custodians never meet). Throughout the key ceremony, the witnesses record the events and any deviations from the defined procedure are noted. At the end, an official record is presented to the key manager.

Validation. If necessary, following the completion of the key ceremony, the official record can be scrutinised to validate that the correct procedure was followed (perhaps as part of an audit).

This proposed key ceremony is meant only as an illustration, and precise details of key ceremonies will depend on local requirements. The point, however, is to demonstrate the importance of key management policies, practices and procedures. Regardless of the underlying cryptographic technology and hardware, the security of the key ceremony itself is down to an orchestrated sequence of actions by human beings, which can only be controlled by procedures of this type.

10.8 Summary

In this chapter we have discussed key management, the aspect of cryptography that is of greatest relevance to users of cryptography, since it is the part most likely to require decision making and process design in individual application environments. We have observed that key management is always necessary, but never easy. In particular, we have:

- Stressed the importance of keeping in mind the entire cryptographic key lifecycle, from key generation through to key destruction.
- Examined the various phases of the cryptographic key lifecycle in detail.
- Noted that key management must ultimately be governed by policies, practices and procedures.

This chapter dealt with management of keys that need to be kept secret. In the next chapter we examine further issues that arise when managing key pairs for use in public-key cryptography.

10.9 Further reading

Despite the importance of key management, it is often a topic that is overlooked in introductions to cryptography and there are few dedicated and comprehensive overviews of the topic. Probably the best overall treatment is NIST's recommendations for key management, published as NIST 800-57 [139]. The first part of this special publication provides a thorough grounding in key management and the second part includes advice on key management governance. NIST 800-130 [142] deals with the design of key management systems. The first part of ISO/IEC 11770 [4] presents a general overview and basic model for key management. Another relevant standard is ISO 11568 [3], which covers key management in the important application area of retail banking, as does the influential standard ANSI X9.24 [26]. Dent and Mitchell [55] provide a good overview of the contents of most of these key management standards.

The best place for advice on key lengths is the web portal managed by Giry [89]. We used the 2010 key length recommendations from the European project ECRYPT II [66] in Table 10.1, which is one of the resources linked to by [89]. NIST provide guidance on key derivation in NIST 800-108 [141] and on key derivation from passwords in NIST 800-132 [143], as does PKCS#5 [115]. Key generation by components is conducted using simple secret sharing schemes, a good introduction to which can be found in Stinson [185]. Unique key per transaction schemes are popular in the retail world and are described in banking standards. The Racal UKPT scheme has been standardised in UKPA Standard 70 [193] and the Derived UKPT scheme can be found in ANSI X9.24 [26].

We provided several references to key establishment mechanisms in Chapter 9, including Boyd and Mathuria [40] and ISO/IEC 11770 [4]. The BB84 protocol was first proposed by Bennett and Brassard [31], with an accessible description of it provided in Singh [176]. Scientific American ran a story on quantum key establishment by Stix [186]. There is quite a lot of misinformation around on the usefulness and likely impact of quantum key establishment. We recommend the practical analysis of Moses [126] and Paterson, Piper and Schack [152] as providing interesting perspectives.

Hardware security modules are fundamental components of many key management systems. One of the most influential standards in this area is FIPS 140-2 [79]. HSMs are also treated in the banking standards ISO 11568 [3] and ISO 13491 [5]. Attridge [27] provides a brief introduction to HSMs and their role in cryptographic key management. The zones of risk for key storage depicted in Figure 10.6 are based on ISO 13491 [5]. Ferguson, Schneier and Kohno [75] include a chapter on key storage, and Kenan [107] discusses key storage in the context of cryptographically protected databases. Bond [38] describes fascinating attacks on HSMs that had achieved a high level of FIPS 140 compliance. Dent and Mitchell [55] include a chapter on cryptography APIs.

The key block that we described in Figure 10.8 is from ANSI X9 TR-31 [25]. NIST has a special publication NIST 800-88 [137] relating to data deletion (sanitisation). Finally, the key generation ceremony that we described is loosely based on the ceremony described in [112].

10.10 Activities

1. Provide some examples of attacks that can be conducted if assurance of purpose of cryptographic keys is not provided as part of:
   (a) a fully symmetric hierarchical key management system deployed in a government department;

(b) an open key management system supporting public-key (hybrid) cryptography to provide email security.

2. Guidance on key lengths changes over time.
   (a) Name two reputable sources for key length recommendations (other than the ECRYPT recommendations of Table 10.1) and explain why they are credible sources.
   (b) To what extent do their recommendations for symmetric key lengths 'match' the recommendations shown in Table 10.1?
   (c) Given a public-key cryptosystem, explain how experts might determine which key length for this algorithm is 'equivalent' to a symmetric key length of 128 bits.

3. Which of the following keys do you think should be the longest:
   - a key protecting the PIN of a credit card in a point-of-sale terminal;
   - a transaction (session) key protecting a large money transfer between two banks?

4. For each of the following, give an example (with justification) of an application of cryptography where it might make sense to deploy:
   (a) a flat key hierarchy with just one level;
   (b) a two-level key hierarchy;
   (c) a three-level key hierarchy.

5. UKPT schemes offer support for key management in special application environments.
   (a) Which of the phases of the key management lifecycle shown in Figure 10.1 is a UKPT scheme designed to make more straightforward?
   (b) Compare the impacts on the Racal and Derived UKPT schemes in the event that an attacker compromises a point-of-sale terminal and is able to access any keys stored in the terminal.
   (c) Compare the impacts on the Racal and Derived UKPT schemes if there is a communication error in the middle of a transaction.
   (d) Suggest some key management controls that are designed to overcome the 'weaknesses' of these two UKPT schemes.

6. Quantum key establishment technology is at a relatively early stage of maturity. Explore the 'state of the art' in quantum key establishment by finding out:
   (a) What is the longest distance over which a symmetric key has been established using quantum key establishment?
   (b) What are the best current data rates?
   (c) Which commercial organisations are selling quantum key establishment technology?
   (d) Which applications are deploying quantum key establishment technology?

7. Hardware security modules (HSMs) are commonly used to store cryptographic keys.
   (a) What benchmarks are used for evaluating the security of an HSM?
   (b) Which organisations carry out such evaluations?
   (c) Provide an example of a currently available commercial HSM technology and provide any details that you can about the security features that it uses.

8. Key backup is an important part of the cryptographic key lifecycle.
   (a) Why is it important to back up cryptographic keys?
   (b) In what ways might backup of cryptographic keys differ from backup of more general data on a computer system?
   (c) As system administrator of a small organisation deploying symmetric cryptography for protection of all traffic on the local intranet, suggest what techniques and procedures you will use for the backup (and subsequent management of backed-up) cryptographic keys.

9. In the past, the idea of mandatory key escrow in order to facilitate access to decryption keys during an authorised government investigation has been proposed.
   (a) Explain what is meant by mandatory key escrow.
   (b) What are the main problems with attempting to support mandatory key escrow within a key management system?
   (c) An alternative approach is to provide a legal framework within which targets of an authorised investigation are 'forced' by law to reveal relevant decryption keys. What are the potential advantages and disadvantages of this approach?
   (d) For the jurisdiction in which you currently reside, find out what (if any) mechanisms exist for supporting an authorised government investigation in the event that the investigators require access to data that has been encrypted.

10. Give an example of a real cryptographic application that:
    (a) 'enforces' the principle of key separation (explain why);
    (b) 'abuses' the principle of key separation (justify why, if possible).

11. Cryptographic keys need to be destroyed at the end of their lifetime. Find out what the latest recommended techniques are for destroying:
    (a) a data key that is stored on a laptop;
    (b) a master key that is stored on a server in a bank.

12. Key management must be governed by appropriate policies, practices and procedures.
    (a) Provide one example (each) of an appropriate policy statement, practice and procedure relating to passwords used to access a personal computer in an office.

(b) Give three different examples of things that might go wrong if an organisation fails to govern key management properly.

(c) For each of your chosen examples, indicate how appropriate key management governance might help to prevent the stated problem arising in the first place.

13. Suppose three users, Alice, Bob and Charlie, wish to use symmetric cryptography to protect files that are transferred between their personal computers. They decide:

- not to use any standard secure file transfer package;
- to encrypt files directly using an encryption algorithm implemented on their local machines;
- send the encrypted files over any appropriate insecure channel (there is no need to consider what type of channel is used).

Design a suitable key management system (including all phases of the key lifecycle) for supporting this application.

14. Cryptographic Application Programming Interfaces (APIs) provide services that allow developers to build secure applications based on cryptography.

(a) What are the most popular cryptographic APIs in use today?

(b) For a cryptographic API of your choice, write a short summary of the main services provided by the API, including the range of cryptographic primitives and algorithms that it supports.

(c) What vulnerabilities might arise from potential misuse of a cryptographic API? (You might choose to illustrate this answer by providing examples of potential attacks, of which there are several that have been reported in the media.)

15. A payment card organisation has a key management system in place to manage the PINs that their customers use. It has the following features:

- All PINs are generated using a PIN generation key *PGK*, which is a single length DES key.
- *PGK* is generated from three components PGK_A, PGK_B and PGK_C, all of which are stored on smart cards held in a single safe.
- The components PGK_A and PGK_B are backed up, but PGK_C is not.
- When *PGK* is established from its components, the key generation ceremony specifies that the holders of each of the components must pass the smart card with their component to the internal auditor after they have installed it.
- Some of the retail systems supporting the card payment system store *PGK* is software.
- Customers are permitted to change their PIN using a telephone-based interactive voice recognition service.

(a) What problems can you identify with this key management system?

(b) Propose some changes to the key management system in order to overcome these problems.

16. It is sometimes suggested that a cryptographic key should not be used unnecessarily, since each use of a key 'exposes' its use to an attacker. Suppose that a cryptosystem is being used for encryption purposes.

(a) What does an attacker potentially learn each time an encryption key is used?
(b) Our standard assumptions about a cryptosystem suggest that an attacker knows corresponding pairs of plaintexts and ciphertexts, so do our standard assumptions contradict in any way this 'principle' of minimising key exposure?
(c) To what extent do you think that key exposure is a real risk if the cryptosystem is using AES?
(d) Provide some examples of key management techniques that reduce key exposure.

17. There are many different reasons why cryptographic keys need to be changed from time to time. This can be particularly problematic for long-term keys such as master keys. Suggest different ways in which an organisation could manage the process of changing (migrating) from the use of one master key to another.

18. It will be increasingly important in the future to use resources, including computing resources, as efficiently as possible. Explain what role key management can play in the 'greening' of information technology.

19. A 128-bit block cipher can be thought of as a family of 2^{128} different 'codebooks' (see Section 1.4.4), each of which defines how to convert any block of plaintext into a block of ciphertext under one specific key. One way to appear to avoid having to deal with some 'key management' issues for a particular hardware device might be to directly implement the 'codebook' corresponding to a particular key onto the device. Thus the hardware cannot be used with *any* block cipher key, but instead has an implementation of the unique version of the 'block cipher' that arises from one specific key.

(a) In what types of application environment might this be an attractive idea?
(b) What are the disadvantages of such an approach?
(c) Does this approach make 'key management' any easier?

11 Public-Key Management

This chapter continues our investigation of key management by looking at particular issues that relate to public-key management. These issues primarily arise due to the need to provide assurance of purpose of public keys. It is important to state from the outset that this chapter should be regarded as an *extension* of Chapter 10 for public-key cryptography, not a *replacement*. Most of the key management issues discussed in Chapter 10 are also relevant to the management of key pairs in public-key cryptography.

The term *public-key infrastructure* (PKI) is often associated with key management systems for supporting public-key cryptography. We avoid it for several reasons:

1. The term 'PKI' is often used in confusing ways. In particular, it is often incorrectly used to refer to public-key cryptography itself, rather than the supporting key management system.
2. The notion of a PKI is strongly associated with a key management system that supports public-key certificates. While this is the most common approach to designing a public-key management system, it is not the only option. We will consider alternative approaches in Section 11.4.
3. The attention paid to the concept of a PKI rather deflects from the fact that all cryptosystems require key management systems to support them. We do not often hear the term *symmetric key infrastructure* (SKI), yet key management support for symmetric cryptography is just as essential as it is for public-key cryptography.

At the end of this chapter you should be able to:

- Explain the purpose of a public-key certificate.
- Describe the main phases in the lifecycle of a public-key certificate.
- Discuss a number of different techniques for implementing the different phases in the public-key certificate lifecycle.
- Compare several different certificate-based public-key management models.
- Be aware of alternative approaches to certificate-based public-key management.

11.1 Certification of public keys

Recall from our discussion in Section 10.1.3 that the main challenge for the management of public keys is providing assurance of purpose of public keys. In this section we introduce the most popular mechanism for providing this assurance of purpose, the *public-key certificate*.

11.1.1 Motivation for public-key certificates

We begin by recalling why we need assurance of purpose of public keys, since this is of crucial importance in public-key management.

A SCENARIO

Suppose that Bob receives a digitally signed message that claims to have been signed by Alice and that Bob wants to verify the digital signature. As we know from Chapter 7, this requires Bob to have access to Alice's verification key. Suppose that Bob is presented with a key (we do not concern ourselves with how this is done) that is alleged to be Alice's verification key. Bob uses this key to 'verify' the digital signature and it appears to be correct. What guarantees does Bob have that this is a valid digital signature by Alice on the message?

As is often the case in security analysis, the best way of approaching this question is to consider what might have gone wrong. Here are some questions that Bob would be strongly advised to consider, especially if the digital signature is on an important message:

Does the verification key actually belong to Alice? This is the big question. If an attacker is able to persuade Bob (incorrectly) that their verification key belongs to Alice, then the fact that the signature verification is successful will suggest to Bob that Alice signed the contract, when in fact it might have been signed by the attacker.

Could Alice deny that this is her verification key? Even if Bob does have Alice's correct verification key, Alice could deny that it belonged to her. If Alice denies signing the message and denies that the verification key belongs to her, then the fact that the signature verifies correctly is of little use to Bob, since he cannot prove who the signer was.

Is the verification key valid? Recall from Section 10.2 that cryptographic keys have finite lifetimes. It is possible that, even if Alice did use this verification key once, it is no longer a valid verification key for Alice since it has expired. Alice might have (naughtily!) signed the message with an expired key, knowing that the digital signature would not be legally accepted because she did not sign it with a signature key that was valid at the time of signing.

Is the verification key being used appropriately? It is generally regarded as good practice that cryptographic keys should have specifically designated uses. For example, in order to enforce the principle of key separation that we discussed in Section 10.6.1 it might be wise to have different RSA key pairs for encryption and digital signatures. Even more fine-grained usage policies are possible. For example, a particular digital signature key pair might only be authorised for use on messages that relate to transactions worth less than a particular limit (beyond which a different key pair needs to be used, perhaps consisting of longer keys). If Alice has used the signature key inappropriately (in our first example this might be by using an RSA private key designated for decryption use only, in the second case by using a signature key to sign a transaction above the limit) then even if the verification key confirms the result to be 'cryptographically' verified, the signature may not be valid in any legal sense.

PROVIDING ASSURANCE OF PURPOSE

The above scenario is pessimistic in its outlook, but it is important to observe that if we can provide assurance of purpose of verification keys then all of Bob's concerns should be put to rest. In particular we need to:

1. provide a 'strong association' between a public key and the *owner* of that key (the entity whose identity is linked to the public key);
2. provide a 'strong association' between a public key and any other relevant data (such as expiry dates and usage restrictions).

Once again, we emphasise that these issues are not unique to public keys but, as discussed in Section 10.1.3, are often provided implicitly for secret keys. Because public keys are usually publicly available, assurance of purpose must be provided explicitly.

PROVIDING A POINT OF TRUST

The concept of the provision of 'trust' will be a central theme in this chapter. This is because public-key cryptography lends itself to use in relatively open environments where common points of trust are not always present by default. This is in contrast to symmetric key cryptography, which we saw in Chapter 10 typically requires explicit points of trust to be deployed within a key management system, such as a trusted key centre.

The problem in designing any public-key management system is that we need to find a source for the provision of the 'strong association' between a public-key value and its related data. In public-key management systems this is often provided by introducing points of trust in the form of trusted third parties who 'vouch' for this association.

USING A TRUSTED DIRECTORY

Perhaps the crudest approach to providing assurance of purpose for public keys is to use a trusted 'directory', which lists all public keys next to their related data (including the name of the owner). Anyone requiring assurance of purpose of a public key, simply looks it up in the trusted directory. This is analogous to the idea of a telephone directory for telephone numbers.

While this approach may suffice for some applications of public-key cryptography, there are several significant problems:

Universality. The directory has to be trusted by all users of the public-key management system.

Availability. The directory has to be online and available at all times to users of the public-key management system.

Accuracy. The directory needs to be maintained accurately and protected from unauthorised modification.

In truly open application environments, such a trusted directory might potentially need to account for public keys associated with public-key owners throughout the world. Establishing a trusted directory that everyone trusts, is always online, and is always accurate, is likely to be impossible.

However, this basic idea does provide the required assurance of purpose. A more practical solution would be to provide assurance of purpose by distributing the functionality of the trusted directory in some manner. This motivates the notion of a public-key certificate, which we now discuss.

11.1.2 Public-key certificates

A *public-key certificate* is data that binds a public key to data relating to the assurance of purpose of that public key. It can be thought of as a trusted directory entry in a sort of distributed database.

CONTENTS OF A PUBLIC-KEY CERTIFICATE

A public-key certificate contains four essential pieces of information:

Name of owner. The name of the owner of the public key. This owner could be a person, a device, or even a role within an organisation. The format of this name will depend upon the application, but it should be a unique identity that identifies the owner within the environment in which the public key will be employed.

Public-key value. The public key itself. This is often accompanied by an identifier of the cryptographic algorithm with which the public key is intended for use.

Validity time period. This identifies the date and time from which the public key is valid and, more importantly, the date and time of its expiry.

Signature. The creator of the public-key certificate digitally signs all the data that forms the public-key certificate, including the name of owner, public-key value and validity time period. This digital signature not only binds all this data together, but is also the guarantee that the creator of the certificate believes that all the data is correct. This provides the 'strong association' that we referred to in Section 11.1.1.

Most public-key certificates contain much more information than this, with the precise contents being dictated by the certificate format that is chosen for use in

Table 11.1: Fields of an X.509 Version 3 public-key certificate

Field	Description
Version	Specifies the X.509 version being used (in this case V3)
Serial Number	Unique identifier for the certificate
Signature	Digital signature algorithm used to sign the certificate
Issuer	Name of the creator of the certificate
Validity	Dates and times between which the certificate is valid
Subject	Name of the owner of the certificate
Public-Key Info.	Public-key value; Identifier of public-key algorithm
Issuer ID	Optional identifier for certificate creator
Subject ID	Optional identifier for certificate owner
Extensions	A range of optional fields that include:
	Key identifier (in case owner owns several public keys);
	Key usage (specifies usage restrictions);
	Location of revocation information;
	Identifier of policy relating to certificate;
	Alternative names for owner.

the public-key management system. The most well known public-key certificate format is probably X.509 Version 3. The entries (or *fields*) of an X.509 Version 3 public-key certificate are shown in Table 11.1. The public-key certificate itself consists of all the information in Table 11.1 plus a digital signature on the contents, signed by the certificate creator.

INTERPRETING A PUBLIC-KEY CERTIFICATE

It is important to recognise that a public-key certificate binds the assurance-of-purpose data relating to a public key to the public-key value, but does *nothing more than this*. In particular:

A public-key certificate cannot be used to encrypt messages or verify digital signatures. A public-key certificate is simply a statement that the public key contained in it belongs to the named owner and has the properties specified in the certificate. Of course, once the certificate has been checked, the public key can be extracted from the certificate and then used for its specified purpose.

A public-key certificate is not a proof of identity. A public-key certificate can be made available to anyone who needs to use the public key contained in it, so presenting a public-key certificate is not a proof of identity. In order to identify someone using their public-key certificate it is necessary to obtain evidence that they know the private key that corresponds to the public key in the public-key certificate. This technique is commonly used in entity authentication protocols based on public-key cryptography. We saw an example in Section 9.4.2 when we studied the STS protocol.

PUBLIC-KEY CERTIFICATE CREATORS

It should be clear that the creator of a public-key certificate plays an extremely important role since this creator, by signing the certificate, is providing the guarantee that all the data relating to the public key (including the name of the owner) is correct.

A creator of a public-key certificate is referred to as a *certificate authority* (CA). The certificate authority normally plays three important roles:

Certificate creation. The CA takes responsibility for ensuring that the information in a public-key certificate is correct before creating and signing the public-key certificate, and then issuing it to the owner.

Certificate revocation. The CA is responsible for revoking the certificate in the event that it becomes invalid (see Section 11.2.3).

Certificate trust anchor. The CA acts as the point of trust for any party relying on the correctness of the information contained in the public-key certificate. To fulfil this role, the CA will need to actively maintain its profile as a trusted organisation. It may also need to enter into relationships with other organisations in order to facilitate wider recognition of this trust (see Section 11.3.3).

We will shortly discuss all of these roles in more detail.

RELYING ON A PUBLIC-KEY CERTIFICATE

Recall from Section 11.1.1 that the motivation for public-key certificates is to provide assurance of purpose of public keys. We thus need to establish exactly how the use of a public-key certificate provides this assurance.

There are three things that someone who wishes to rely on a public-key certificate (whom we will call a *relying party*) needs to be able to do in order to obtain assurance of purpose of the public key:

Trust the CA. The relying party needs to be able to trust (directly or indirectly) the CA to have performed its job correctly when creating the certificate. We will discuss exactly why a relying party might come to trust a CA in Section 11.3.

Verify the signature on the certificate. The relying party needs to have access to the verification key of the CA in order to verify the CA's digital signature on the public-key certificate. If the relying party does not verify this signature then they have no guarantee that the contents of the public-key certificate are correct. Of course, this transfers the problem to one of providing assurance of purpose of the CA's verification key. However, just as we saw for symmetric keys in Section 10.4.1, transferring a key management problem 'higher up a chain' allows for a more scalable solution. We will discuss this issue in more detail in Section 11.3.

Check the fields. The relying party needs to check all the fields in the public-key certificate. In particular, they must check the name of the owner and that the public-key certificate is valid. If the relying party does not check these fields then they have no guarantee that the public key in the certificate is valid for the application for which they intend to use it.

DIGITAL CERTIFICATES

It is worth noting that the principle of having some specific data digitally signed by a trusted third party can have other applications. Public-key certificates represent a special class of *digital certificates*. An example of another type of digital certificate is an *attribute certificate*, which can be used to provide a strong association between a specific attribute and an identity, such as:

- the identity specified is a member of the access control group *administrators*;
- the identity specified is over the age of 18.

Attribute certificates might contain several fields that are similar to a public-key certificate (for example, owner name, creator name, validity period) but would not contain a public-key value. Like a public-key certificate, the data that they contain is digitally signed by the creator in order to vouch for its accuracy.

11.2 The certificate lifecycle

Many of the details of the phases of the key lifecycle (Figure 10.1) that we discussed in Chapter 10 are just as valid for private keys as they are for symmetric keys. However, there are several important differences with respect to the lifecycle of public keys. In this section we will consider these lifecycle differences for a public-key certificate, which is essentially the embodiment of a public key.

11.2.1 Differences in the certificate lifecycle

We now recall the main phases of the key lifecycle from Figure 10.1 and comment on where differences lie:

Key generation. This is one of the phases which differs significantly. We have already observed in Section 10.3.4 that the generation of a public-key pair is an algorithm-specific, and often technically complex, operation. Having done this, creation of a public-key certificate is even harder from a process perspective, since it involves determining the validity of information relating to the public key. We will discuss this in more detail in Section 11.2.2.

Key establishment. Private key establishment is potentially easier than symmetric key establishment since the private key only needs to be established by one entity. Indeed, this entity could even generate the private key themselves (we discuss the pros and cons of this approach in Section 11.2.2). If another entity generates the private key then private key establishment may involve the private key being distributed to the owner using a secure channel of some sort, such as physical distribution of a smart card on which the private key is installed.

Public-key certificate establishment is not a sensitive operation, since the public-key certificate does not need to be kept secret. Most techniques can either be described as:

Pushing a public-key certificate, meaning that the owner of the public-key certificate provides the certificate whenever it is required by a relying party (for example, in the STS Protocol of Figure 9.12, both Alice and Bob pushed their public-key certificates to one another).

Pulling a public-key certificate, meaning that relying parties must retrieve public-key certificates from some sort of repository when they first need them. One potential advantage of pulling public-key certificates is that they could be pulled from a trusted database that only contains valid public-key certificates.

Key storage, backup, archival. We have discussed these processes from the perspective of private keys in Section 10.5. They are all less-sensitive operations when applied to public-key certificates.

Key usage. The principle of key separation, discussed in Section 10.6.1, applies equally to public-key pairs. Many public-key certificate formats, such as the X.509 Version 3 certificate format depicted in Table 11.1, include fields for specifying key usage.

Key change. This is the other phase of the key lifecycle that differs significantly for public-key pairs. We identified why this is the case in Section 10.6.2 and we will discuss potential techniques for facilitating key change in Section 11.2.3.

Key destruction. Destruction of private keys is covered by Section 10.6.4. Destruction of public-key certificates is less sensitive, and may not even be required.

The remainder of this section discusses key pair generation and key pair change, the two phases in the key lifecycle for which specific issues arise for the management of key pairs.

11.2.2 Certificate creation

We now discuss the creation of public-key certificates.

LOCATION OF KEY PAIR AND CERTIFICATE CREATION

It is important to be aware of the fact that we are dealing with two separate processes here:

- generation of the public-key pair itself;
- creation of the public-key certificate.

Key pair generation can be performed either by the owner of the public-key pair or a trusted third party (who may or may not be the CA). The choice of location for this operation results in different certificate creation scenarios:

Trusted third party generation. In this scenario, a trusted third party (which could be the CA) generates the public-key pair. If this trusted third party is not the CA then they must contact the CA to arrange for certificate creation. The advantages of this approach are that:

- the trusted third party may be better placed than the owner to conduct the relatively complex operations involved in generation of the public-key pair (see Section 10.3.4);
- the key pair generation process does not require the owner to do anything.

The possible disadvantages are that:

- the owner needs to trust the third party to securely distribute the private key to the owner; the only exception to this is if the private key is managed on behalf of the owner by the trusted third party, in which case processes must exist for securely managing ‘access’ to the private key when the owner needs to use it.
- the owner needs to trust the third party to destroy the private key after it has been distributed to the owner; an exception to this would be if the third party provides a backup and recovery service for the owner (see Section 10.5.5).

This scenario lends itself most naturally to closed environments where a trusted third party with the extra responsibilities outlined above can be established.

Combined generation. In this scenario, the owner of the key pair generates the public-key pair. The owner then submits the public key to a CA for generation of the public-key certificate. The main advantages of this approach are that:

- the owner is in full control of the key pair generation process;
- the private key can be locally generated and stored, without any need for it to be distributed.

The possible disadvantages of this approach are that:

- the owner is required to have the ability to generate key pairs;
- the owner may need to demonstrate to the CA that the owner knows the private key that corresponds to the public key submitted to the CA for certification (we discuss this shortly).

This scenario is most suitable for open environments where owners wish to control the key pair generation process themselves.

Self-certification. In this scenario, the owner of the key pair generates the key pair and certifies the public key themselves. This approach is certainly simple. However, it might seem a strange option, since a public-key certificate generated by a CA provides 'independent' assurance of purpose of a public key, whereas self-certification requires relying parties to trust in the assurance of purpose provided by the owner of the public key. However, if relying parties trust the owner then this scenario may be justifiable. Examples of situations where this might be the case are:

- the owner is a CA; it is not uncommon for CAs to self-certify their own public-keys, which is an issue that we will discuss in a moment;
- all relying parties have an established relationship with the owner and hence trust the owner's certification; for example, a small organisation using a self-certified public key to encrypt content on an internal website.

REGISTRATION OF PUBLIC KEYS

If either trusted third-party generation or combined generation of a public-key pair is undertaken then the owner of the public-key pair must engage in a *registration* process with the CA before a public-key certificate can be issued. This is when the owner presents their credentials to the CA for checking. These credentials not only provide a means of authenticating the owner, but also provide information that will be included in some of the fields of the public-key certificate. Registration is arguably the most vital stage in the process of generating a public-key certificate. It is also a process that varies greatly between different applications.

It is worth re-emphasising that the requirements for a registration process are not unique to public keys. It is also extremely important that a symmetric key is issued to the correct entities and that associated information (such as intended purpose of the key, expiry date and usage restrictions) is linked by some means to the key value. However, as we argued in Section 5.1.1 and Section 10.1.3, 'registration' of symmetric keys tends to be implicitly provided by the supporting key management system. It is important that registration is explicit for public keys, particularly in the case of combined generation.

In many application environments a separate entity known as a *Registration Authority* (RA) performs this operation. The roles of RA and CA can be separated for several reasons:

- Registration involves a distinct set of procedures that generally require an amount of human intervention, whereas certificate creation and issuance can be automated.
- Checking the credentials of a public-key certificate applicant is often the most complex part of the certificate creation process. Centralised checking of credentials represents a likely major bottleneck in the process, particularly for large organisations. There is thus a strong argument for distributing the registration activities across a number of local RAs, which perform the checking and then

report the results centrally. On the other hand, the security-critical processes associated with a CA, such as public-key pair generation and certificate signing, are probably best done within one well-defined business unit.

Whether the CA and RA roles are incorporated as one, or kept entirely separate, there remains an important problem to address: what credentials should be presented to the RA during registration?

The answer to this is, of course, application dependent. It is worth noting that many CAs issue different types of public-key certificate (sometimes referred to as *levels* of public-key certificate) depending upon the thoroughness of the registration process. Public-key certificates of different levels may then be used in different types of application. These certificates might have quite different properties. For example, the liability that the CA accepts responsibility for (with respect to any relying parties) might vary for different levels of public-key certificate. We now give some examples of credentials:

- A very low level of public-key certificate might simply require a valid email address to be presented at registration. The registration process might include checking that the applicant can receive email at that address. This level of credential is often enough for public-key certificates that can be obtained online at no cost.
- Registration for public-key certificates for use in a closed environment, such as an organisation's internal business environment, might involve presentation of an employee number and a valid internal email address.
- Commercial public-key certificates for businesses trading over the Internet might require a check of the validity of a domain name and the confirmation that the applicant business is legally registered as a limited company.
- Public-key certificates for incorporation into a national identity card scheme require a registration process that unambiguously identifies a citizen. This can be surprisingly complex to implement. Credentials might include birth certificates, passports, domestic utility statements, etc.

PROOF OF POSSESSION

If a public key and its certificate are created using combined generation then, strictly speaking, it is possible for an attacker to attempt to register a public key for which they do not know the corresponding private key. Such an 'attack' on a verification key for a digital signature scheme might work as follows:

1. The attacker obtains a copy of Alice's verification key. This is a public piece of information, so the attacker can easily obtain this.
2. The attacker presents Alice's verification key to an RA, along with the attacker's legitimate credentials.
3. The RA verifies the credentials and instructs the associated CA to issue a public-key certificate in the name of the attacker for the presented verification key.
4. The CA issues the public-key certificate for the verification key to the attacker.

The attacker now has a public-key certificate issued in their name for a verification key for which they do not know the corresponding signature key. At first glance this might not seem a very useful outcome for the attacker. However, a problem arises if Alice now digitally signs a message with her signature key, since the attacker will be able to persuade relying parties that this is actually the attacker's digital signature on the message. This is because the attacker's name is on a public-key certificate containing a verification key that successfully verifies the digital signature on the message.

This attack can be prevented if the CA conducts a simple check that the public-key certificate applicant knows the corresponding private key. This type of check is often referred to as *proof of possession* (of the corresponding private key). If the public key is an encryption key then one possible proof of possession is as follows:

1. The RA encrypts a test message using the public key and sends it to the certificate applicant, along with a request for the applicant to decrypt the resulting ciphertext.
2. If the applicant is genuine, they decrypt the ciphertext using the private key and return the plaintext test message to the RA. An applicant who does not know the corresponding private key will not be able to perform the decryption to obtain the test message.

It should be noted that proof of possession checks are only required in applications where the outlined 'attack' is deemed to be meaningful. Proof of possession does require a small overhead, so once again we encounter a potential tradeoff between the extra security gained by conducting the check versus the efficiency gained by omitting to do so.

GENERATING CA PUBLIC-KEY PAIRS

Public-key certificates involve a CA digitally signing the owner's public key together with related data. This in turn requires the CA to possess a public-key pair. This raises the interesting question of how assurance of purpose of the CA's verification key will be provided.

The most natural solution is to create a public-key certificate for the CA's public key. But who will sign the public-key certificate of the CA? This is an absolutely crucial question, since any compromise or inaccuracy of this public-key certificate may compromise all public-key certificates signed by the CA. The two most common methods of certifying the CA's verification key are:

Use a higher-level CA. If the CA is part of a *chain* of CAs (we discuss this in Section 11.3.3) then the CA may choose to have their public key certified by another CA. Of course, this does not address the question of who certifies the public key of the higher-level CA.

Self-certification. A top-level CA probably has no choice other than self-certification. It may suffice that this process involves making the public key available in high-profile media, such as daily newspapers. There is a strong case for arguing that a top-level CA's business model involves them being in such a position of trust that they have no motivation for providing incorrect information about their public keys. Hence publication of the information on their own website may suffice.

Note that wide distribution of the public key (certificate) of a CA is also extremely important, since all relying parties of the public-key certificates signed by the CA require this information. As an example, CAs who certify public keys that are used in web-based commercial applications need to have their public-key certificates incorporated into leading web browsers, or have them certified by a higher-level CA who has done this.

11.2.3 Key pair change

The second phase of the key lifecycle that is significantly different for public keys is key change.

REVOCATION OF PUBLIC-KEY CERTIFICATES

We explained in Section 10.6.2 that the main reason why key change is challenging for public keys is because it is almost impossible (and in many cases undesirable) to control who has access to a public key. This makes withdrawing an existing public key very difficult. This process is normally referred to as *revoking* the public key, since it involves 'calling back' information that has been released into the public domain and is now no longer valid. In contrast, establishing a new public key is relatively easy. Thus our discussion of key change for public keys will focus on public-key revocation.

We observe that it does not suffice just to establish a new public key, since we cannot always determine who has access to the old public key and hence we cannot guarantee that all holders of the old public key will realise that a new public key has been issued.

Revoking a public key essentially means revoking the public-key certificate. With this in mind, it is worth observing that there may be situations where a public-key certificate needs to be revoked and then a new public-key certificate created for the *same* public-key value. We will assume that revocation of a public-key certificate only takes place prior to its expiry date. A public-key certificate should not be relied on by any relying parties if its expiry date has been exceeded.

REVOCATION TECHNIQUES

Revocation of public-key certificates can only realistically be approached in one of three ways:

Blacklisting. This involves maintaining a database that contains serial numbers of public-key certificates that have been revoked. This type of database is often referred to as a *certificate revocation list* (or CRL). These CRLs need to be maintained carefully, normally by the CA who is responsible for issuing the certificates, with clear indications of how often they are updated. The CRLs need to be digitally signed by the CA and made available to relying parties.

Whitelisting. This involves maintaining a database that contains serial numbers of public-key certificates that are valid. This database can then be queried by a relying party to find out if a public-key certificate is valid. An example is the *Online Certificate Status Protocol* (OCSP), which has been standardised as RFC 2560. This is particularly useful for applications that require real-time information about the revocation status of a public-key certificate.

Rapid expiration. This removes the need for revocation by allocating very short lifetimes to public-key certificates. This, of course, comes at the cost of requiring certificates to be reissued on a regular basis.

Blacklisting is a common technique when real-time revocation information is not required. There are many different ways of implementing the blacklisting concept, often involving networks of distributed CRLs rather than one central CRL. The main problem with blacklisting is one of synchronisation. In particular, there are likely to be:

- reporting delays between the time that a public-key certificate should be revoked (for example, the time of a private key compromise) and the CA being informed;
- CRL issuing delays between the time that the CA is informed of the revocation of a public-key certificate and the time that the next version of the CRL is signed and made publicly available.

Thus, in theory, a relying party could rely on a public-key certificate in the gap period between the time the public-key certificate should have been revoked and the publication time of the updated CRL. This is an issue that must be 'managed' through suitable processes and procedures. For example:

- The CA should inform all relying parties of the update frequency of CRLs.
- The CA should clarify who is responsible for any damage incurred from misuse of a public key in such a gap period. It might be reasonable to address this by:
  - the CA accepting limited liability during gap periods;
  - relying parties accepting full liability if they fail to check the latest CRL before relying on a public-key certificate.

The means of conveying this information to relying parties is through publication of the key management policies and practices of a CA (see Section 10.7.1). The relevant documents for a CA are often referred to as *certificate policy statements* and *certificate practice statements*. They not only clarify the issues just discussed, but also the wider key management issues relating to public keys that the CA certifies.

11.3 Public-key management models

In this section we consider different public-key management models. We begin by discussing the issue of trusting CAs, particularly techniques for joining CA domains. We then examine the relationship between a relying party and a CA, and use this to define several different management models.

11.3.1 Choosing a CA

In a closed environment, the choice of who will play the role of a CA may be straightforward, since central administrative functions within an organisation are well placed to serve such a role. Choosing an organisation to play the role of a CA in an open environment is less straightforward. Currently, most CAs serving open environments are commercial organisations who have made it their business to be 'trusted' to play the role of a CA.

While CAs serving open environments can be regulated to an extent by commercial pressure (if they fail to offer attractive services or experience reputational damage then they are likely to suffer financially), the importance of their role may demand tighter regulation of their practices. Options for this include:

Licensing. This approach requires CAs to obtain a government license before they can operate. Government, thus, ultimately provides the assurance that a CA conforms to minimum standards.

Self-regulation. This approach requires CAs to form an industry group and set their own minimum operational standards through the establishment of best practices.

In the UK, licensing was considered in the 1990s but was met with considerable objections from industry. Currently the self-regulation approach is being adopted.

11.3.2 Public-key certificate management models

The owner of a public-key certificate has, by necessity, placed some trust in the CA who issued the certificate. This may be because the owner belongs to the same organisation as the CA (typically in closed environments) or because the owner and the CA have a direct business relationship (typically in open environments).

However, the same cannot necessarily be said for a relying party. Indeed, the relationship between a relying party and the public-key certificate owner's CA defines a number of distinct public-key certificate management models, which we now review.

CA-FREE CERTIFICATION MODEL

The *CA-free certification model* is depicted in Figure 11.1 and applies when there is no CA involved. In the CA-free certification model, the owner generates a key pair

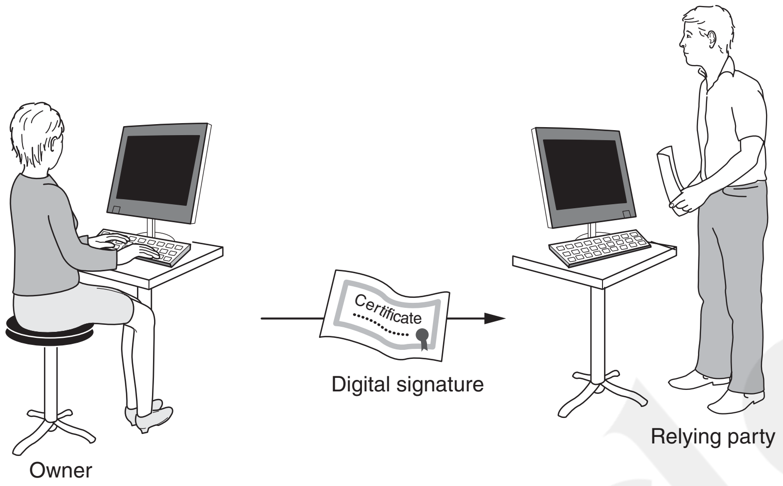

Figure 11.1. CA-free certification model

and then either self-certifies the public key or does not use a public-key certificate. Any relying party obtains the (self-certified) public key directly from the owner. For example, the owner could include their public key in an email signature or write it on a business card. The relying party then has to make an independent decision as to whether they trust the owner or not. The relying party thus carries all the risk in this model. A variation of this idea is the *web of trust* model, which we discuss in Section 11.4.1.

REPUTATION-BASED CERTIFICATION MODEL

The *reputation-based certification model* is depicted in Figure 11.2 and applies when the owner has obtained a public-key certificate from a CA, but the relying party has no relationship with this CA. Even if the relying party obtains the verification key of the CA, which enables them to verify the public-key certificate, because they do not have any relationship with the CA itself they do not by default gain assurance of purpose from verification of the certificate. They are left to *choose* whether to trust that the CA has done its job correctly and hence that the information in the public-key certificate is correct. In the worst case, they may not trust the CA at all, in which case they have no reason to trust any information affirmed by the CA.

The only assurance that they might gain is through the *reputation* of the CA that signed the public-key certificate. If the relying party has some trust in the reputation of the CA, for example, it is a well-known organisation or trust service provider, then the relying party might be willing to accept the information in the public-key certificate.

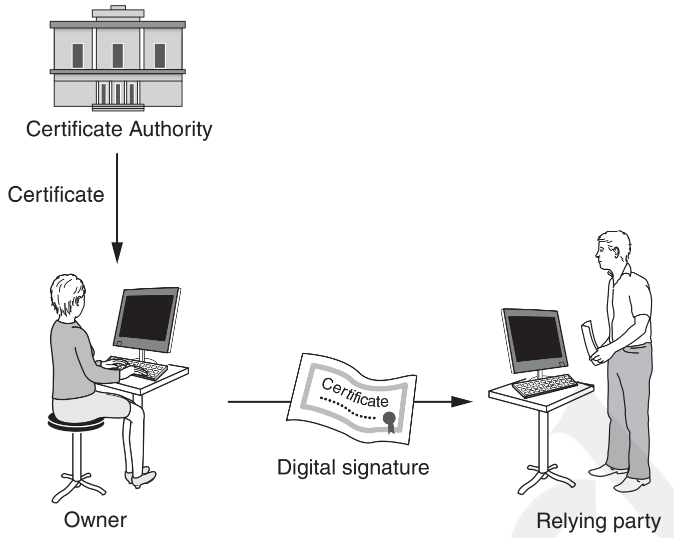

Figure 11.2. Reputation-based certification model

CLOSED CERTIFICATION MODEL

The *closed certification model* is depicted in Figure 11.3 and applies when the relying party has a relationship with the owner's CA. The closed certification model is the most 'natural' certification model, but is only really applicable to closed environments where a single CA oversees the management of all public-key

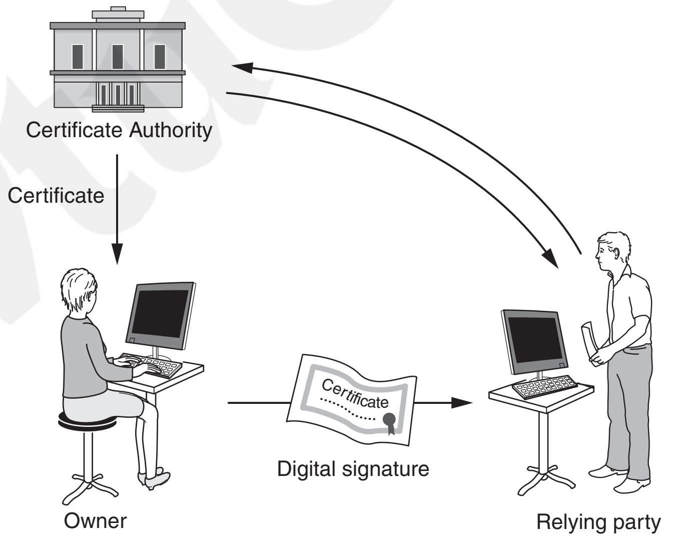

Figure 11.3. Closed certification model

certificates. In the closed certification model, the more onerous issues concerning public-key management, such as those relating to liability and revocation, are more straightforward to solve than for the other models. This is because public-key certificate owners and relying parties are all governed by the same certificate management policies and practices.

CONNECTED CERTIFICATION MODEL

The *connected certification model* is depicted in Figure 11.4 and applies when the relying party has a relationship with a trusted third party, which in turn has a relationship with the owner's CA. The trusted third party that the relying party has a relationship with could be another CA. In Figure 11.4 we describe it as a *validation authority* because its role is to assist the relying party to validate the information in the owner's public-key certificate. Strictly speaking, this validation authority may not necessarily be a CA.

We do not further specify the nature of the relationship between the owner's CA and the relying party's validation authority, since there are many different ways in which this could manifest itself. For example, the CA and validation authority could both be members of a federation of organisations who have agreed to cooperate in the validation of public-key certificates and have signed up to common certificate management policies and practices. The important issue is that, because the relying party has a relationship with the validation authority, the relying party essentially delegates the task of verifying the public-key certificate

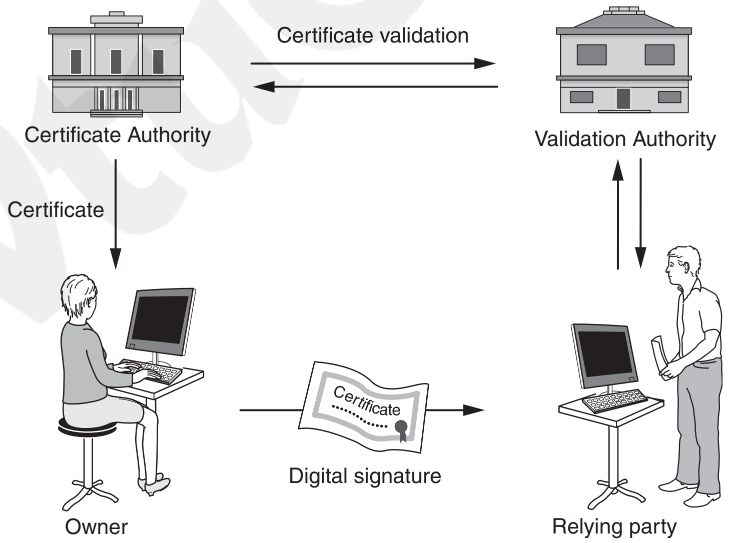

Figure 11.4. Connected certification model

to the validation authority. The validation authority can then do this through its relationship with the owner's CA.

The connected certification model is a pragmatic 'stretching' of the closed certification model, in order to allow public-key certificates to be managed in environments that are either:

open, in the sense that owners and relying parties are not governed by any single management entity;
distributed, in the case of a closed environment that is distributed, for example, a large organisation with different branches or regional offices.

11.3.3 Joining CA domains

The connected certification model is of particular interest since it allows public-key certificates to be used in environments where the owner and relying party do not have trust relationships with the same CA. We will now assume that both the owner Alice and relying party Bob have relationships with their own CAs, which we label CA1 and CA2, respectively (hence for simplicity we now assume that the validation authority in Figure 11.4 is a CA). We now consider the nature of the relationship between CA1 and CA2. In particular, we will look at techniques for 'joining' their respective *CA domains* and allowing certificates issued by CA1 to be 'trusted' by relying parties who have trust relationships with CA2.

CROSS-CERTIFICATION

The first technique for joining two CA domains is to use *cross-certification*, whereby each CA certifies the other CA's public key. This idea is depicted in Figure 11.5. Cross-certification implements a *transitive* trust relationship. By cross-certifying, relying party Bob of CA2, who wishes to trust a public-key certificate issued to Alice by CA1, can do so by means of the following argument:

1. I (Bob) trust CA2 (because I have a business relationship with CA2);
2. CA2 trusts CA1 (because they have agreed to cross-certify one another);

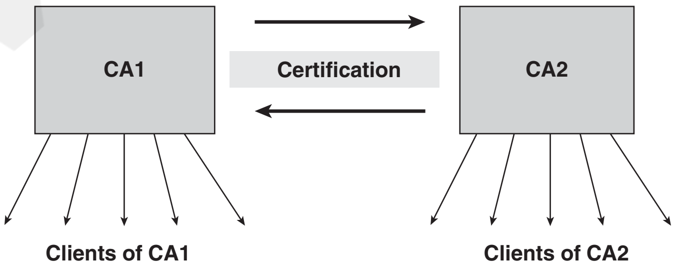

Figure 11.5. Cross-certification

3. CA1 has vouched for the information in Alice's public-key certificate (because CA1 generated and signed it);
4. Therefore, I (Bob) trust the information in Alice's public-key certificate.

CERTIFICATION HIERARCHIES

The second technique, which we also encountered in Section 11.2.2, is to use a *certification hierarchy* consisting of different levels of CA. A higher-level CA, which both CA1 and CA2 trust, can then be used to 'transfer' trust from one CA domain to the other. The simple certification hierarchy in Figure 11.6 uses a higher-level CA (termed the *root CA*) to issue public-key certificates for both CA1 and CA2. Relying party Bob of CA2, who wishes to trust a public-key certificate issued to Alice by CA1, can do so by means of the following argument:

1. I (Bob) trust CA2 (because I have a business relationship with CA2);
2. CA2 trusts root CA (because CA2 has a business relationship with root CA);
3. Root CA has vouched for the information in CA1's public-key certificate (because root CA generated and signed it);
4. CA1 has vouched for the information in Alice's public-key certificate (because CA1 generated and signed it).
5. Therefore, I (Bob) trust the information in Alice's public-key certificate.

CERTIFICATE CHAINS

The joining of CA domains makes verification of public certificates a potentially complex process. In particular, it results in the creation of *certificate chains*, which consist of a series of public-key certificates that must all be verified in order to have trust in the end public-key certificate. Just as an example to illustrate this complexity, consider the apparently simple CA topology in Figure 11.7. In this

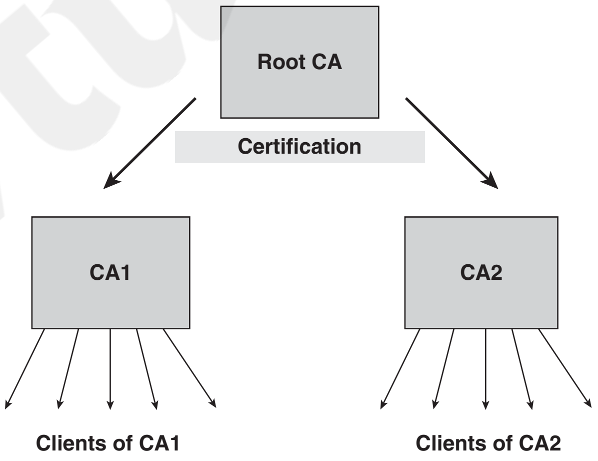

Figure 11.6. Certification hierarchy

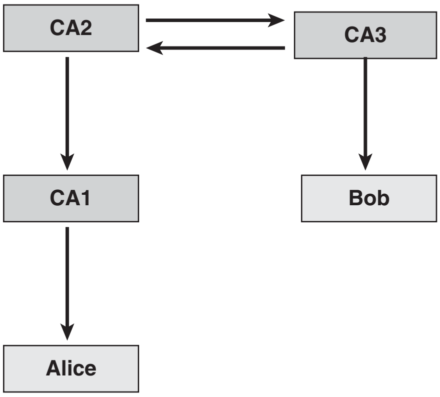

Figure 11.7. A simple CA topology

topology Alice has a relationship with CA1, which is a low-level CA whose root CA is CA2. Bob has a relationship with CA3, which has cross-certified with CA2. Now suppose that Bob wishes to verify Alice's public-key certificate. To do so, he will need to verify a certificate chain that consists of the three public-key certificates shown in Table 11.2.

In other words, Bob first verifies Alice's public-key certificate, which is signed by CA1. Then Bob needs to verify CA1's public-key certificate, which is signed by CA2. Finally, Bob verifies CA2's public-key certificate, which is signed by CA3. The certificate chain ends at CA3, since this is the CA that Bob has a relationship with and thus we assume that he trusts signatures by CA3.

Indeed, to properly verify the above certificate chain, for *each* of these public-key certificates Bob should:

1. verify the signature on the public-key certificate;
2. check all the fields in the public-key certificate;
3. check whether the public-key certificate has been revoked.

Table 11.2: Example of a certificate chain

Certificate	Containing public key of	Certified by
1	Alice	CA1
2	CA1	CA2
3	CA2	CA3

JOINING CA DOMAINS IN PRACTICE

While these techniques for joining CA domains appear to work in theory, it is less clear how well they work in practice. Issues such as liability start to become extremely complex when CA domains are joined in such ways. Our discussion of certificate chains has shown that even verification of a public-key certificate chain can be a complex process.

One of the highest profile examples of the connected certification model in practice is the *web-based certification model* implemented by major web browser manufacturers. This can be thought of as a fairly 'flat' certification hierarchy, where commercial CAs have their root certificates embedded into a web browser. Rather than 'cross-certifying' amongst these root CAs, the browser manufacturer should ensure that CAs whose root certificates they accept have met certain business practice standards. This allows relying parties to gain some assurance of purpose of public-key certificates issued by CAs that they do not have direct business relationships with. It also implements the reputation-based model, since even relying parties who have no relationship with any CA can still gain some degree of trust in a public-key certificate, so long as they trust the web browser manufacturer to have vetted CAs to an appropriate level.

One of the problems with the web-based certification model is that the trust linkage between the root CAs is not particularly 'tight'. Arguably, a more serious problem is that relying parties are left to conduct a significant portion of the certificate chain verification process, since a web browser cannot automatically check that all the fields of each certificate are what the relying party is expecting. Relying parties cannot always be trusted to understand the importance of conducting these checks. Indeed, even informed relying parties may choose, for convenience, to omit these checks.

The connected certification model is probably most effective when the CAs involved have strong relationships, such as when they operate in what we referred to in Section 11.3.2 as a distributed environment. Some examples of this type of environment arise in the financial and government sectors.

11.4 Alternative approaches

As we have seen from the discussion in this chapter, there are many complicated issues to resolve when trying to implement a certificate-based approach to public-key management. There are a number of alternative approaches that attempt to resolve these by avoiding the use of public-key certificates. We now discuss two such approaches.

Note that the use of public-key certificates is more common than either of these alternative approaches. However, consideration of these approaches not only indicates that certificates are not the only option for public-key management, but also helps to place the challenges of public-key certificate management in context.

11.4.1 Webs of trust

In the CA-free certification model of Section 11.3.2, we noted that public keys could be made available directly by owners to relying parties without the use of a CA. The problem with this approach is that the relying party is left with no trust anchor other than the owner themselves.

A stronger assurance can be provided if a *web of trust* is implemented. Suppose that Alice wishes to directly provide relying parties with her public key. The idea of a web of trust involves other public-key certificate owner's acting as 'lightweight CAs' by digitally signing Alice's public key. Alice gradually develops a *key ring*, which consists of her public key plus a series of digital signatures by other owners attesting to the fact that the public-key value is indeed Alice's.

These other owners are, of course, not acting as formal CAs, and the relying party may have no more of a relationship with any of these other owners than with Alice herself. Nonetheless, as Alice builds up her key ring there are two potentially positive impacts for relying parties:

1. A relying party sees that a number of other owner's have been willing to sign Alice's public key. This is at least *some* evidence that the public key may indeed belong to Alice.
2. There is an increasing chance (as the key ring size increases) that one of the other owners is someone that the relying party knows and trusts. If this is the case then the relying party might use a transitive trust argument to gain some assurance about Alice's public key.

Webs of trust clearly have limitations. However, they represent a lightweight and scalable means of providing some assurance of purpose of public keys in open environments, where other solutions are not possible.

However, the extent to which webs of trust make a real impact is unclear since, for the types of open applications in which they make most sense, relying parties are often likely to choose to simply trust the owner (in many cases they may already have an established trust relationship).

11.4.2 Identity-based public-key cryptography

Recall that the main purpose of a public-key certificate is to bind an identity to a public-key value. Thus one way of eliminating the need for public-key certificates is to build this binding directly into the public keys themselves.

THE IDEA BEHIND IDPKC

One way in which this binding could be built in is if the public-key value can be uniquely derived from the identity. And one way in which *this* can be done is to make the public-key value and the identity *the same value*. This is the motivation behind *identity-based public-key cryptography* (IDPKC).

A significant difference between IDPKC and certificate-based approaches to management of conventional public-key cryptography is that IDPKC requires a trusted third party to be involved in private-key generation. We will refer to this trusted third party as a *trusted key centre* (TKC), since its main role is the generation and distribution of private keys. The basic idea behind IDPKC is:

- A public-key owner's 'identity' *is* their public key. There is a publicly known rule that converts the owner's 'identity' into a string of bits, and then some publicly known rule that converts that string of bits into a public key.
- The public-key owner's private key can be calculated from their public key only by the TKC, who has some additional secret information.

In this way public-key certificates are not required, since the linkage between the owner's identity and the public key is by means of the publicly known rules. Despite the fact that public keys are easily determined by anyone, private keys are only computable by the TKC.

A MODEL FOR IDPKC ENCRYPTION

Figure 11.8 shows the process behind using IDPKC to encrypt a message from Alice to Bob. The model consists of the following stages:

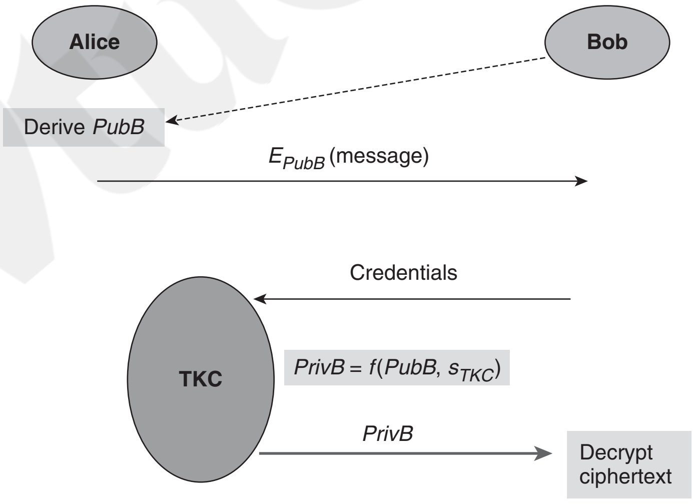

Figure 11.8. The IDPKC process

Encryption. Alice derives Bob's public key *PubB* from Bob's identity using the publicly known rules. Alice then encrypts her message using *PubB* and sends the resulting ciphertext to Bob.

Identification. Bob identifies himself to the TKC by presenting appropriate credentials and requests the private key *PrivB* that corresponds to *PubB*.

Private key derivation. If the TKC accepts Bob's credentials then the TKC derives *PrivB* from *PubB* and a system secret value s_{TKC}, known only by the TKC.

Private-key distribution. The TKC sends *PrivB* to Bob using a secure channel.

Decryption. Bob decrypts the ciphertext using *PrivB*.

One of the most interesting aspects of this IDPKC encryption model is that encryption can occur *before* private-key derivation and distribution. In other words, it is possible to send an encrypted message to someone who has not yet established a private decryption key. Indeed, they may not even be aware of the possibility of receiving encrypted messages! This is quite an unexpected property and one that certainly does not hold for conventional public-key cryptography.

It should also be noted that once a user has obtained their private key, there is no need for them to repeat the middle three stages until either *PubB* or the system secret s_{TKC} change. We return to this issue shortly.

IDPKC ENCRYPTION ALGORITHMS

The most important issue regarding algorithms for IDPKC is that *conventional public-key cryptosystems cannot be used for IDPKC*. There are two principal reasons for this:

1. In conventional public-key algorithms, such as RSA, it is not possible for *any* value to be a public key. Rather, a public key is a value that satisfies certain specific mathematical properties. Given an arbitrary numerical identity of a public-key owner, it is unlikely that this corresponds to a valid public key (it *might*, but this would be lucky, rather than expected).
2. Conventional public-key algorithms do not feature a system secret s_{TKC} that can be used to 'unlock' each private key from the corresponding public key.

For these reasons, IDPKC requires the design of different encryption algorithms that are explicitly designed for the IDPKC setting. Several such algorithms exist, but we will not discuss them in any further detail.

PRACTICAL ISSUES WITH IDPKC

While IDPKC directly solves some of the problems associated with certificate-based public-key cryptography, it results in some new issues. These include:

The need for an online, centrally trusted TKC. There is no getting around this requirement, which immediately restricts IDPKC to applications where the existence of such a trusted entity is acceptable and practical. Note in particular that:

- the TKC should be *online*, since it could be called upon at any time to establish private keys;
- only the TKC can derive the private keys in the system, hence it provides a source of key escrow, which can either be seen as desirable or undesirable (see Section 10.5.5);
- the TKC requires secure channels with all private key owners, with the similar advantages and disadvantages as discussed for trusted third-party generation of key pairs in Section 11.2.2, except that the TKC cannot 'destroy' the private keys as it always has the ability to generate them.

Nonetheless, there are many potential applications where the existence of such a TKC is reasonable, especially in closed environments.

Revocation issues. One of the problems with tying a public-key value to an identity is the impact of revocation of a public key, should this be required. If the public key has to change, we are essentially requiring the owner's 'identity' to change, which clearly might not be practical. An obvious solution to this is to introduce a temporal element into the owner's 'identity', perhaps featuring a time period for which the public key is valid. For example, Bob's public key might become *PubB3rdApril* for any encrypted messages intended for Bob on 3rd April, but change to *PubB4thApril* the next day. The cost of this is a requirement for Bob to obtain a new private key for each new day.

Multiple applications. Another issue is that it is no longer immediately obvious how to separate different applications. In conventional public-key cryptography, one owner can possess different public keys for different applications. By tying an identity to a public-key value, this is not immediately possible. One solution is again to introduce variety into the encryption key using, say, *PubBBank* as Bob's public key for his online banking application. However, it should be pointed out that IDPKC offers a potential advantage in this area since it is possible for the *same* public key *PubB* to be used across multiple applications, so long as different system secrets s_{TKC} are used to generate different private keys.

It is important to recognise that these issues are *different* issues to those of certificate-based public-key cryptography. This means that IDPKC presents an interesting alternative to conventional public-key cryptography, which may suit certain application environments better, but may not be suitable for others. For example, IDPKC may be attractive for low-bandwidth applications because there is no need to transfer public-key certificates, but will be unattractive for any application that has no central point of trust that can play the role of the TKC.

It is also important to realise that some of the challenges of certificate-based public-key cryptography remain. One example is that the need to register for public-key certificates is now simply translated into a need to register in order to obtain private keys.

MORE GENERAL NOTIONS OF IDPKC

The idea behind IDPKC is both compelling and intriguing, since it represents a quite different approach to implementing public-key cryptography. In fact, it is even more interesting than it first appears, since there is no need to restrict public keys to being associated with identities. They could take the form of almost any string of data.

One of the most promising extensions of the IDPKC idea is to associate public keys in an IDPKC system with *decryption policies*. The idea involves only a slight modification of the process described in Figure 11.8:

Encryption. Alice derives a public key *PubPolicy* based on a specific decryption policy using publicly known rules. For example, this policy could be *Qualified radiographer working in a UK hospital*. Alice then encrypts her message (say, a health record) using *PubPolicy* and (continuing our example) stores it on a medical database, along with an explanation of the decryption policy.

Identification. Qualified UK radiographer Bob, who wishes to access the health record, identifies himself to the TKC by presenting appropriate medical credentials and requests the private key *PrivPolicy* that corresponds to *PubPolicy*.

Private-key derivation. If the TKC accepts Bob's credentials then the TKC derives *PrivPolicy* from *PubPolicy* and a system secret value s_{TKC}, known only by the TKC.

Private-key distribution. The TKC sends *PrivPolicy* to Bob using a secure channel.

Decryption. Bob decrypts the ciphertext using *PrivPolicy*.

This example illustrates the power of being able to encrypt before the recipient obtains the private key since, in this example, Alice does not necessarily even know the recipient. This idea is being instantiated by *attribute-based* encryption schemes, which model decryption policies in terms of a set of *attributes* that a recipient must possess before the TKC will release the corresponding private key to them.

IDPKC IN PRACTICE

While IDPKC presents an interesting alternative framework for managing public-key cryptography, it is still early in its development. Cryptographic algorithms for implementing IDPKC are relatively new, although several are now well respected. Some commercial applications, including email security products (see Section 12.7.2), have already implemented IDPKC. While we have primarily discussed IDPKC in order to illustrate that there are public-key management alternatives to public-key certificates, it also represents a cryptographic primitive that is likely to be used more often in future applications.