Chapter 11
Key Management

VALENTINE: Why then, I would resort to her by night.
DUKE: Ay, but the doors be lock'd and keys kept safe,
That no man hath recourse to her by night.
VALENTINE: What lets but one may enter at her window?
— *The Two Gentlemen of Verona*, III, i, 110–113.

Key management refers to the distribution of cryptographic keys; the mechanisms used to bind an identity to a key; and the generation, maintenance, and revoking of such keys. We assume that identities correctly define principals—that is, a key bound to the identity "Bob" is really Bob's key. Alice did not impersonate Bob's identity to obtain it. Chapter 15, "Representing Identity," discusses the problem of identifiers naming principals; Chapter 13, "Authentication," discusses a principal authenticating herself to a single system. This chapter assumes that authentication has been completed and that identity is assigned. The problem is to propagate that authentication to other principals and systems.

We first discuss authentication and key distribution. Next comes key generation and the binding of an identity to a key using certificates. We then discuss key storage and revocation. We conclude with digital signatures.

A word about notation. The statement

$$X \rightarrow Y : \{m\}k$$

means that entity X sends entity Y a message m enciphered with key k. Subscripts to keys indicate to whom the keys belong, and are written where multiple keys are in use. For example, k_{Alice} and k_{Bob} refer to keys belonging to Alice and Bob, respectively. If Alice and Bob share a key, that key will be written as $k_{Alice,Bob}$ when the sharers are not immediately clear from the context. In general, k represents a secret key (for a symmetric cryptosystem), e a public key, and d a private key (for a public key cryptosystem). If multiple messages are listed sequentially, they are concatenated and sent. The operator $a \parallel b$ means that the bit sequences a and b are concatenated.

11.1 Session and Interchange Keys

We distinguish between a *session key* and an *interchange key* [1949].

> **Definition 11–1.** An *interchange key* is a cryptographic key associated with a principal to a communication. A *session key* is a cryptographic key associated with the communication itself.

This distinction reflects the difference between a communication and a user involved in that communication. Alice has a cryptographic key used specifically to exchange information with Bob. This key does not change over interactions with Bob. However, if Alice communicates twice with Bob (and "communication" can be using, for example, an email or a web browser), she does not want to use the same key to encipher the messages. This limits the amount of data enciphered by a single key and reduces the likelihood of an eavesdropper being able to break the cipher. It also hinders the effectiveness of replay attacks. Instead, she will generate a key for that single session. That key enciphers the data, and it is discarded when the session ends. Hence, the name of the key is a "session key."

Session keys also prevent forward searches (see Section 12.1.1). A forward search attack occurs when the set of plaintext messages is small. The adversary enciphers all plaintexts using the target's public key. When ciphertext is intercepted, it is compared with the precomputed texts. This quickly gives the corresponding plaintext. A randomly generated session key, used once, would prevent this attack. (See Exercise 2 for another approach.)

EXAMPLE: Suppose Alice is a client of Bob's stockbrokering firm. She needs to send Bob one of two messages: BUY or SELL. The attacker, Cathy, enciphers both messages with Bob's public key. When Alice sends her message, Cathy compares it with her messages and sees which one it matches.

An interchange key is associated with a principal. Alice can use the key she shares with Bob to convince Bob that the sender is Alice. She uses this key for all sessions. It changes independently of session initiation and termination.

11.2 Key Exchange

The goal of key exchange is to enable Alice to communicate secretly with Bob, and vice versa, using a shared cryptographic key. Solutions to this problem must meet the following criteria:

- The key that Alice and Bob are to share cannot be transmitted in the clear. Either it must be enciphered when sent, or Alice and Bob must

derive it without an exchange of data from which the key can be derived. Alice and Bob can exchange data, but a third party cannot derive the key from the data exchanged.

- Alice and Bob may decide to trust a third party (called "Cathy" here).
- The cryptosystems and protocols are publicly known. The only secret data is to be the cryptographic keys involved.

Symmetric cryptosystems and public key cryptosystems use different protocols.

11.2.1 Symmetric Cryptographic Key Exchange

Suppose Alice and Bob wish to communicate. If they share a common key, they can use a symmetric cryptosystem. But how do they agree on a common key? If Alice sends one to Bob, Eve the eavesdropper will see it and be able to read the traffic between them.

To avoid this bootstrapping problem, symmetric protocols rely on a trusted third party, Cathy. Alice and Cathy share a secret key, and Bob and Cathy share a (different) secret key. The goal is to provide a secret key that Alice and Bob share. The following simple protocol provides a starting point [1684]:

1. Alice $\rightarrow$ Cathy : {request for session key to Bob}k_{Alice}
2. Cathy $\rightarrow$ Alice : $\{k_{session}\}k_{Alice} \parallel \{k_{session}\}k_{Bob}$
3. Alice $\rightarrow$ Bob : $\{k_{session}\}k_{Bob}$

Bob now deciphers the message and uses $k_{session}$ to communicate with Alice.

This particular protocol is the basis for many more sophisticated protocols. However, it can be compromised. Assume that Alice sends Bob a message (such as "Deposit $500 in Dan's bank account today") enciphered under $k_{session}$. If an adversary "Eve" records the third message in the exchange above, and the message enciphered under $k_{session}$, she can send Bob the message $\{k_{session}\}k_{Bob}$ followed by the message enciphered under $k_{session}$. Bob will not know that this is a repeat of an earlier message.

Avoiding problems such as this replay attack adds considerable complexity. Key exchange protocols typically add, at a minimum, some sort of defense against replay attack. In the process of the exchange, they also may provide authentication.

11.2.1.1 Needham-Schroeder Protocol

One of the best-known symmetric key exchange and authentication protocols is the Needham-Schroeder protocol [1435].

1. Alice $\rightarrow$ Cathy : {Alice $\parallel$ Bob $\parallel$ r_1}
2. Cathy $\rightarrow$ Alice : {Alice $\parallel$ Bob $\parallel$ r_1 $\parallel$ $k_{session}$ $\parallel$ {Alice $\parallel$ $k_{session}\}k_{Bob}\}k_{Alice}$

3. Alice $\rightarrow$ Bob : $\{\text{Alice} \parallel k_{session}\}k_{Bob}$
4. Bob $\rightarrow$ Alice : $\{r_2\}k_{session}$
5. Alice $\rightarrow$ Bob : $\{r_2 - 1\}k_{session}$

In this protocol, r_1 and r_2 are two numbers generated at random, except that they cannot repeat between different protocol exchanges. These numbers are called *nonces*. So if Alice begins the protocol anew, her r_1 in the first exchange will not have been used there before. The basis for the security of this protocol is that both Alice and Bob trust Cathy.

When Bob receives the third message and deciphers it, he sees that the message names Alice. Since he could decipher the message, the message was enciphered using a key he shares only with Cathy. Because he trusts Cathy not to have shared the key k_{Bob} with anyone else, the message must have been enciphered by Cathy. This means that Cathy is vouching that she generated $k_{session}$ so Bob could communicate with Alice. So Bob trusts that Cathy sent the message to Alice, and that Alice forwarded it to him.

However, if Eve recorded the message, she could have replayed it to Bob. In that case, Eve would not have known the session key, so Bob sets out to verify that his unknown recipient does know it. He sends a random message enciphered by $k_{session}$ to Alice. If Eve intercepts the message, she will not know what to return; should she send anything, the odds of her randomly selecting a message that is correct are very low and Bob will detect the attempted replay. But if Alice is indeed initiating the communication, when she gets the message she can decipher it (because she knows $k_{session}$), apply some fixed function to the random data (here, decrement it by 1), encipher the result, and return it to Bob. Then Bob will be sure he is talking to Alice.

Alice needs to convince herself that she is talking to Bob. When she receives the second message from Cathy, she deciphers it and checks that Alice, Bob, and r_1 are present. This tells her that Cathy sent the second message (because it was enciphered with k_{Alice}, which only she and Cathy know) and that it was a response to the first message (because r_1 is in both the first and second messages). She obtains the session key and forwards the rest to Bob. She knows that only Bob has $k_{session}$, because only she and Bob can read the messages containing that key. So when she receives messages enciphered with that key, she will be sure that she is talking to Bob.

The Needham-Schroeder protocol assumes that all cryptographic keys are secure. In practice, session keys will be generated pseudorandomly. Depending on the algorithm used, it may be possible to predict such keys. Denning and Sacco [545] assumed that Eve could obtain a session key and subvert the authentication. After the previous steps, the following exchange takes place:

1. Eve $\rightarrow$ Bob : $\{\text{Alice} \parallel k_{session}\}k_{Bob}$
2. Bob $\rightarrow$ Alice : $\{r_3\}k_{session}$ [intercepted by Eve]
3. Eve $\rightarrow$ Bob : $\{r_3 - 1\}k_{session}$

Now Bob thinks he is talking to Alice. He is really talking to Eve.

5. Alice → Gutenberg : $A_{Alice,Gutenberg} \parallel T_{Alice,Gutenberg}$
6. Gutenberg → Alice : $\{t + 1\}k_{Alice,Gutenberg}$

In these steps, Alice first constructs an authenticator and sends it, with the ticket and the name of the server, to Barnum. Barnum validates the request by comparing the data in the authenticator with the data in the ticket. Because the ticket is enciphered using the key Barnum shares with Cerberus, he knows that it came from a trusted source. He then generates an appropriate session key and sends Alice a ticket to pass on to Gutenberg. Step 5 repeats step 3, except that the name of the service is not given (because Gutenberg is the desired service). Step 6 is optional; Alice may ask that Gutenberg send it to confirm the request. If it is sent, t is the timestamp.

Bellovin and Merritt [165] discuss several potential problems with the Kerberos protocol. In particular, Kerberos relies on clocks being synchronized to prevent replay attacks [1129]. If the clocks are not synchronized, and if old tickets and authenticators are not cached, replay is possible. In Kerberos 5, authenticators are valid for five minutes, so tickets and authenticators can be replayed within that interval. Also, because the tickets have some fixed fields, a dictionary attack can be used to determine keys shared by services or users and the ticket-granting service or the authentication service, much as the WordPerfect cipher was broken (see the end of Section 10.2.2.1). Researchers at Purdue University used this technique to show that the session keys generated by Kerberos 4 were weak; they reported deciphering tickets, and finding session keys, within minutes [579]. Yu, Hartman, and Raeburn [2067] showed a flaw in Kerberos 4 that enabled an attacker to impersonate any principal. As a result, the MIT Kerberos Team announced that Kerberos 4 had reached the end of its life in 2006 [1359].

The Kerberos Version 5 protocol has been formally analyzed [330] and shown to provide the claimed authentication and secrecy properties. It supports intra-organizational communication, called "cross-realm operation" [1442]. To do this, the two realms must share an inter-realm key. Cervesato et al. [369] note that this requires the intermediate ticket-granting servers to be trusted. Sakane et al. [1639] present six requirements for cross-realm operation in large-scale industrial systems that deal with this, and other, threats.

11.2.3 Public Key Cryptographic Key Exchange and Authentication

Conceptually, public key cryptography makes exchanging keys very easy. Alice simply uses Bob's public key to encipher a session key she generates:

Alice → Bob : $\{k_{session}\}e_{Bob}$

where e_{Bob} is Bob's public key. Bob deciphers the message and obtains the session key $k_{session}$. Now he and Alice can communicate securely, using a symmetric cryptosystem.

As attractive as this protocol is, it has a similar flaw to our original symmetric key exchange protocol. Eve can forge such a message. Bob does not know who the message comes from.

One obvious fix is to sign the session key:

$$\text{Alice} \rightarrow \text{Bob} : \{\text{Alice} \parallel \{k_{session}\}d_{Alice}\}e_{Bob}$$

where d_{Alice} is Alice's private key. When Bob gets the message, he uses his private key to decipher the message. He sees the key is from Alice. Bob then uses her public key to obtain the session key. Schneier [1684] points out that Alice could also include a message enciphered with $k_{session}$.

These protocols assume that Alice has Bob's public key e_{Bob}. If not, she must get it from a public server, Peter. With a bit of ingenuity, Eve can arrange to read Bob's messages to Alice, and vice versa.

1. Alice $\rightarrow$ Peter : {send me Bob's public key} [intercepted by Eve]
2. Eve $\rightarrow$ Peter : {send me Bob's public key}
3. Peter $\rightarrow$ Eve : e_{Bob}
4. Eve $\rightarrow$ Alice : e_{Eve}
5. Alice $\rightarrow$ Bob : $\{k_{session}\}e_{Eve}$ [intercepted by Eve]
6. Eve $\rightarrow$ Bob : $\{k_{session}\}e_{Bob}$

Eve now has the session key and can read any traffic between Alice and Bob. This is called a *man-in-the-middle attack* and illustrates the importance of identification and authentication in key exchange protocols. The man-in-the-middle attack works because there is no binding of identity to a public key. When presented with a public key purportedly belonging to Bob, Alice has no way to verify that the public key in fact belongs to Bob. This issue extends beyond key exchange and authentication. To resolve it, we need to look at the management of cryptographic keys.

11.2.3.1 Diffie-Hellman

The Diffie-Hellman scheme [564] was the first public key cryptosystem proposed, and it is still in use today. A pair of users use this algorithm to generate a common key. It is based on the discrete logarithm problem. This problem is to find a value of d such that $n = g^d \bmod p$ for a given n, g, and prime p (see Section 10.3.1). Although solutions are known for small values of p, the difficulty increases exponentially as p increases [1128].

In this cryptosystem, all users share a common modulus p and a g other than 0, 1, or $p-1$. Each user chooses a private key d and computes a public key e. When two users want to communicate, each enciphers the other's public key using their own private key, and uses the result as the shared secret key S.

For example, data gathered from brain signals and galvanic skin responses passed statistical and complexity tests for randomness [1849].

More commonly, biometrics are used to generate cryptographic keys that are tied to individuals. These keys need to be chosen in such a way that an adversary is unlikely to be able to determine them, but must be able to be regenerated consistently.

To generate a key from biometric data, that data is first represented as a bit string, the *feature descriptor*. The feature descriptor is then transformed in some way, such as using error-correcting codes [507], a lattice mapping [2094], or a secret sharing scheme such as Shamir's (see Section 16.3.2) [1372]. The cryptographic key is then generated from this transformed data.

Methods have been developed for generating cryptographic keys from many features, for example faces [396, 1867], handwritten signatures [725], and voice [351, 1372].

Generating cryptographic keys from biometrics requires care in choosing both the biometric and the method for measuring that biometric. If two measurements of the same biometric feature of a principal are made, the variation must be small enough so that the measurements are statistically indistinguishable; similarly, if two measurements of the same biometric feature of two different principals are made, the variation must be large enough so that the measurements are statistically distinguishable. This constrains the choice of features used for biometrics.

If a cryptographic key generated from biometrics is compromised, it must be replaced with a different key. One way to enable this is to introduce random data into the generation, for example by distorting the biometric measurement in some way [729, 1569]. Then, if the key is compromised, the random data is changed.

11.4 Cryptographic Key Infrastructures

Because symmetric cryptosystems use shared keys, it is not possible to bind an identity to a key. Instead, two parties need to agree on a shared key. Section 11.2, "Key Exchange," presents protocols that do this.

Public key cryptosystems use two keys, one of which is to be available to all. The association between the cryptographic key and the principal is critical, because it determines the public key used to encipher messages for secrecy. If the binding is erroneous, someone other than the intended recipient could read the message.

For purposes of this discussion, we assume that the principal is identified by a name of some acceptable sort (Chapter 15, "Representing Identity," discusses this issue in more detail) and has been authenticated to the entity that generates the cryptographic keys. The question is how some (possibly different) principal can bind the public key to the representation of identity.

An obvious idea is for the originator to sign the public key with her private key, but this merely pushes the problem to another level, because the recipient would only know that whoever generated the public key also signed it. No identity is present.

Kohnfelder [1087] suggests creating a message containing a representation of identity, the corresponding public key, and having a trusted authority sign it. A timestamp t is also added:

$$C_{Alice} = \{e_{Alice} \parallel \text{Alice} \parallel t\}d_{Cathy}$$

This type of structure is called a *certificate*.

Definition 11–5. A *certificate* is a token that binds an identity to a cryptographic key.

When Bob wants to communicate with Alice, he obtains Alice's certificate C_{Alice}. Assuming that he knows the trusted authority Cathy's public key, he can decipher the certificate. He first checks the timestamp t to see when the certificate was issued. From this, he can determine whether the certificate is too old to be trusted. The public key in the certificate belongs to the subject named in the certificate, so Bob now has Alice's public key. He knows that Cathy signed the certificate and therefore that Cathy is vouching to some degree that the public key belongs to Alice. If he trusts Cathy to make such a determination, he accepts the public key as valid and belonging to Alice.

One immediate problem is that Bob must know Cathy's public key to validate the certificate. Two approaches deal with this problem. The first, by Merkle, eliminates Cathy's signature; the second structures certificates into signature chains.

11.4.1 Merkle's Tree Authentication Scheme

Merkle [1322] notes that public keys and associated identities can be kept as data in a file. Changing any of these changes the file. This reduces the problem of substituting faked keys or identities to a data integrity problem. Cryptographic hash functions create checksums that reveal changes to files. Merkle uses them to protect the file.

Let Y_i be an identifier and its associated public key, and let $Y_1, \ldots, Y_n$ be stored in a file. Define a function $f : D \times D \rightarrow D$, where D is a set of bit strings. Let $h : \mathbb{N} \times \mathbb{N} \rightarrow D$ be a cryptographic hash function:

$$h(i,j) = \begin{cases} f\left(h\left(i, \left\lfloor \frac{i+j}{2} \right\rfloor\right), h\left(\left\lfloor \frac{i+j}{2} \right\rfloor + 1, j\right)\right) & \text{if } i < j \\ f(Y_i, Y_j) & \text{otherwise} \end{cases}$$

11.4.2 Certificate Signature Chains

The usual form of a certificate is for the issuer to encipher a hash of the identity of the subject (to whom the certificate is issued), the public key, and information such as time of issue or expiration using the issuer's private key. To validate the certificate, a user uses the issuer's public key to decipher the hash and check the data in the certificate. The user trying to validate the certificate must obtain the issuer's public key. If the issuer has a certificate, the user can get that key from the issuer's certificate. This pushes the problem to another level: how can the issuer's certificate be validated?

Two approaches to this problem are to construct a tree-like hierarchy, with the public key of the root known out of band, or to allow an arbitrary arrangement of certifiers and rely on each individual's knowledge of the certifiers. First, we examine X.509, which describes certificates in general. We then look at the PGP certificate structure.

11.4.2.1 X.509: Certificate Signature Chains

The ITU standard X.509 [2176] is the basis for many other protocols. It defines certificate formats and certificate validation in a generic context. Soon after its original issue in 1988, I'Anson and Mitchell [939] found problems with both the protocols and the certificate structure. These problems were corrected in the 1993 version, referred to as X.509v3. Based on experiences using X.509 certificates in privacy-enhanced electronic mail (see Section 12.5.1), other fields were added.

The X.509v3 certificate has the following components [457, 2176]:

1. *Version*. Each successive version of the X.509 certificate has new fields added. If fields 8, 9, and 10 (see below) are present, this field must be 3; if fields 8 and 9 are present, this field is either 2 or 3; and if none of fields 8, 9, and 10 are present, the version number can be 1, 2, or 3.
2. *Serial number*. This must be unique among the certificates issued by this issuer. In other words, the pair (*issuer's Distinguished Name*, *serial number*) must be unique.
3. *Signature algorithm identifier*. This identifies the algorithm, and any parameters, used to sign the certificate.
4. *Issuer's Distinguished Name*. This is a name that uniquely identifies the issuer. See Chapter 15, "Representing Identity," for a discussion.
5. *Validity interval*. This gives the times at which the certificate becomes valid and expires.
6. *Subject's Distinguished Name*. This is a name that uniquely identifies the subject to whom the certificate is issued. See Section 15.5 for a discussion.

7. *Subject's public key information*. This identifies the algorithm, its parameters, and the subject's public key.
8. *Issuer's unique identifier*. Under some circumstances, issuer Distinguished Names may be recycled (for example, when the Distinguished Name refers to a role, or when a company closes and a second company with the same Distinguished Name opens). This field allows the issuer to disambiguate among entities with the same issuer name.
9. *Subject's unique identifier*. This field is like field 8, but for the subject.
10. *Extensions*. These define certain extensions in the areas of key and policy information, certificate path constraints, and issuer and subject information. Each extension is a triplet, the first field being the extension identifier, the second a flag indicating whether the extension is critical or not, and the third being the value.
11. *Signature*. This field identifies the algorithm and parameters used to sign the certificate, followed by the signature (an enciphered hash of fields 1 to 10) itself.

To validate the certificate, the user obtains the issuer's public key for the particular signature algorithm (field 3) and deciphers the signature (field 11). The user then uses the information in the signature field (field 11) to recompute the hash value from the other fields. If it matches the deciphered signature, the signature is valid if the issuer's public key is correct. The user then checks the period of validity (field 5) to ensure that the certificate is current.

Definition 11–6. A *certificate authority* (CA) is an entity that issues certificates.

If all certificates have a common issuer, then the issuer's public key can be distributed out of band. However, this is infeasible. For example, it is highly unlikely that France and the United States could agree on a single issuer for their organizations' and citizens' certificates. This suggests multiple issuers, which complicates the process of validation.

Suppose Alice has a certificate from her local CA, Cathy. She wants to communicate with Bob, whose local CA is Dan. The problem is for Alice and Bob to validate each other's certificates.

Let $X\ll Y\gg$ represent the certificate that the CA X issues for the subject Y. Bob's certificate is Dan≪Bob≫. If Cathy has issued a certificate to Dan, Dan has a certificate Cathy≪Dan≫; similarly, if Dan has issued a certificate to Cathy, Cathy has a certificate Dan≪Cathy≫. In this case, Dan and Cathy are said to be cross-certified.

Definition 11–7. Two CAs are *cross-certified* if each has issued a certificate for the other.

Because Alice has Cathy's (trusted) public key, she can obtain Cathy≪Dan≫ and form the signature chain

Cathy≪Dan≫ Dan≪Bob≫

Because Alice can validate Dan's certificate, she can use the public key in that certificate to validate Bob's certificate. Similarly, Bob can acquire Dan≪Cathy≫ and validate Alice's certificate:

Dan≪Cathy≫ Cathy≪Alice≫

Signature chains can be of arbitrary length. The only requirement is that each certificate can be validated by the one before it in the chain. (X.509 suggests organizing CAs into a hierarchy to minimize the lengths of certificate signature chains, but this is not a requirement.)

Certificates can be revoked or canceled. A list of such certificates enables a user to detect, and reject, invalidated certificates. Section 11.5.2 discusses this.

11.4.2.2 PGP Certificate Signature Chains

PGP is an encipherment program widely used to provide privacy for electronic mail throughout the Internet, and to sign files digitally. It uses a certificate-based key management infrastructure for users' public keys. Its certificates and key management structure differ from X.509's in several ways. Here, we describe OpenPGP's structure [340], but much of this discussion also applies to other versions of PGP.

An OpenPGP certificate is composed of *packets*. A packet is a record with a tag describing its purpose. A certificate contains a public key packet followed by zero or more signature packets. An OpenPGP public key packet has the following structure:

1. *Version*. This is either 3 or 4. Version 3 is compatible with all versions of PGP; Version 4 is not compatible with old (Version 2.6) versions of PGP.
2. *Time of creation*. This specifies when the certificate was created.
3. *Validity period* (Version 3 only). This gives the number of days that the certificate is valid. If it is 0, the certificate does not expire.
4. *Public key algorithm and parameters*. This identifies the algorithm used and gives the parameters for the cryptosystem used. Version 3 packets contain the modulus for RSA (see Section 10.3.2). Version 4 packets contain the parameters appropriate for the cryptosystem used.
5. *Public key*. This gives the public key. Version 3 packets contain the exponent for RSA. Version 4 packets contain the public key for the cryptosystem identified in field 4.

The information in an OpenPGP signature packet is different for the two versions. Version 3 contains the following:

1. *Version*. This is 3.
2. *Signature type*. This describes the specific purpose of the signature and encodes a level of trust (see Section 15.5.3, "Trust"). For example, signature type 0x11 says that the signer has not verified that the public key belongs to the named subject.
3. *Creation time*. This specifies the time at which the fields following were hashed.
4. *Key identifier of the signer*. This specifies the key used to generate the signature.
5. *Public key algorithm*. This identifies the algorithm used to generate the signature.
6. *Hash algorithm*. This identifies the algorithm used to hash the signature before signing.
7. *Part of signed hash value*. After the data is hashed, field 3 is given the time at which the hash was computed, and that field is hashed and appended to the previous hash. The first two bytes are placed into this field. The idea is that the signature can be rejected immediately if the first two bytes hashed during the validation do not match this field.
8. *Signature*. This contains the encipherment of the hash using the signer's private key.

A Version 4 signature packet is considerably more complex, but as a Version 3 signature packet does, it binds a signature to an identifier and data. The interested reader is referred to the OpenPGP specifications [340].

PGP certificates differ from X.509 certificates in several important ways. Unlike X.509, a single key may have multiple signatures. (All Version 4 PGP keys are signed by the owner; this is called *self-signing*.) Also unlike X.509, a notion of "trust" is embedded in each signature, and the signatures for a single key may have different levels of trust. The users of the certificates can determine the level of trust for each signature and act accordingly.

EXAMPLE: Suppose Alice needs to communicate with Bob. She obtains Bob's public key PGP certificate, Ellen,Fred,Giselle,Bob≪Bob≫ (where the X.509 notation is extended in the obvious way). Alice knows none of the signers, so she gets Giselle's PGP certificate, Henry,Irene,Giselle≪Giselle≫, from a certificate server. She knows Henry vaguely, so she obtains his certificate, Ellen,Henry≪Henry≫, and verifies Giselle's certificate. She notes that Henry's signature is at the "casual" trust level, so she decides to look elsewhere for confirmation. She obtains Ellen's certificate, Jack,Ellen≪Ellen≫, and immediately recognizes Jack as her husband. She has his certificate and uses it to validate Ellen's certificate. She notes that his signature is at the "positive" trust level, so she accepts

Ellen's certificate as valid and uses it to validate Bob's. She notes that Ellen signed the certificate with "positive" trust also, so she concludes that the certificate, and the public key it contains, are trustworthy.

In the example above, Alice followed two signature chains:

Henry≪Henry≫ Henry≪Giselle≫ Giselle≪Bob≫

and

Jack≪Ellen≫ Ellen≪Bob≫

The unchecked signatures have been dropped. The trust levels affected how Alice checked the certificate.

A subtle distinction arises here between X.509 and PGP certificates. X.509 certificates include an element of trust, but the trust is not indicated in the certificate. PGP certificates indicate the level of trust, but the same level of trust may have different meanings to different signers. Chapter 15 will examine this issue in considerable detail.

11.4.3 Public Key Infrastructures

The deployment and management of public keys is complex because of the different requirements of various protocols. Several such infrastructures are in place, such as the PGP Certificate Servers and the commercial certificate issuers for web browsers.

Definition 11–8. A *public key infrastructure* (PKI) is an infrastructure that manages public keys and certificate authorities.

Let us examine the Internet X.509 public key infrastructure [457].

11.4.3.1 The Internet X.509 PKI

The Internet X.509 PKI has two basic types of certificates:

- An *end entity certificate* is one issued to entities not authorized to issue certificates.
- A *certificate authority certificate* (called a *CA certificate*) is one issued to a CA. A *self-issued certificate* has the issuer and subject as the same entity. A *self-signed certificate* is a self-issued certificate in which the public key in the certificate can be used to validate the certificate's digital signature; these are useful to provide a public key that begins a certificate chain.

Such a CA is known as a *trust anchor*. A *cross-certificate* is a certificate issued by one CA to another CA, and is intended to describe a trust relationship between the CAs.

When a user wants to obtain a certificate, she first registers with a CA. The CA may delegate the registration task to another entity, called the *registration authority* (RA). In either case, the registering entity is responsible for verifying the identity of the user as required by the CA's policy. The user then initializes her set of keys, obtaining the public key of the CA and generating its public and private keys. The CA then issues the appropriate certificate containing the user's identity and public key, as discussed in Section 11.4.2.1, sends it to the user, and stores it in a certificate repository.

Certificate extensions are either critical or noncritical, as noted earlier. An application supporting Internet certificates must reject a certificate containing an unrecognized critical extension, or one that the application cannot process. The application may ignore any unrecognized noncritical extension, but must process those it recognizes.

All conforming CAs must support the following extensions; they may support others.

- The *authority key identifier* extension, which must be noncritical, identifies the public key that can be used to validate the digital signature of the certificate. This is necessary if the issuer has multiple key pairs used to sign certificates. If the certificate is self-signed, this extension can be omitted; otherwise, it must be present.
- The *subject key identifier* extension, which must be noncritical, contains the same value as the authority key field. If the subject of the certificate is a CA, then this field must be present.
- The *key usage* extension, which should be critical, describes the purposes for which the public key can be used. These purposes include enciphering cryptographic keys (such as session keys), enciphering data, validating digital signatures of certificates, validating digital signatures other than on certificates, signing certificates, and so forth.
- The *basic constraints* extension, which must be critical if the certificate is used to validate the digital signatures of certificates and may be either critical or noncritical otherwise, identifies whether the subject is a CA and, if the public key can be used to verify a certificate's digital signature, the number of intermediate certificates that may follow this one in a certificate chain and that are not self-signed certificates.
- The *certificate policies* extension, when present on an end-entity certificate, says under what policy the certificate is issued, and what the certificate may be used for. When present on a CA certificate, this extension limits the set of policies on any certificate chain that includes this certificate.

The presence of these extensions simplifies processing and eliminates some earlier constraints on the Internet PKI. If the first one were not present, the validator would need to try different keys of the issuing CA to determine whether the certificate was valid. In earlier versions of the Internet PKI, the specific key used to sign the certificate often indicated which policy applied to the certificate. Now, the key identifiers and the certificate policy extensions do this explicitly. The key usage extension makes clear what the public key in the certificate is to be used for. Before, this was either embedded in the issuer's policy, or the public key was assumed to be valid for all purposes. Finally, the basic constraints extension limits the length of the certificate subchain beginning at the certificate and extending to the end point, not including self-signed certificates.

All conforming applications that process these certificates must recognize the following extensions:

- The key usage, certificate policies, and basic constraints extensions.
- The *subject alternative name* extension, which must be critical, provides another name for the subject, such as an email address, an IP address, and so forth. If present, the issuing CA must verify that this is another name for the subject of the certificate.
- The *name constraints* extension is in CA certificates only. It constrains what names are allowed in the subject field and subject alternative name extension of certificates following it in the certificate chain, unless those certificates are self-signed. It does not apply to self-signed certificates.
- The *policy constraints* extension, which must be critical, controls when the policy for the certificate chain containing this certificate must be explicit or when the policy in the issuer of a certificate in the chain can no longer be the same as the policy of the subject, even if the certificate says that it is.
- The *extended key usage* extension allows the issuer to specify uses of the public key beyond those given in the key usage extension, for example using the public key to sign downloadable executable code.
- The *inhibit anyPolicy* extension, which must be critical, enables a wild-card (*anyPolicy*) to match policies only if it occurs in an intermediate self-signed certificate in a certificate chain.

These extensions also add flexibility and control. The subject alternative name allows multiple subject names in a certificate; earlier versions did not allow this. The name constraints, policy constraints, and inhibit anyPolicy extensions control the policies that apply to the use of the certificate and the meaning of the subject names. The extended key usage field allows the public key to be used for purposes beyond the ones identified in the key usage extension.

11.4.3.2 Problems with PKIs

The heart of any PKI is trust. Ultimately, problems with PKIs are problems with the trust reposed in the infrastructure.

Consider the nature of a certificate. The issuer is binding the identity of a subject to a public key, so the issuer claims with some degree of confidence that the identity belongs to the principal claiming that identity. The degree of confidence of the identity depends entirely on the CA or its delegate (usually the registration authority). Section 15.5 explores this issue in depth.

The understanding of the CA's policies is also critical. If an end entity uses a certificate, that entity trusts that the CA is the appropriate CA for the policy that embodies the use of the certificate. Failure to validate this may result in accepting or rejecting a certificate inappropriately.

A common source of confusion is the belief that a certificate embodies authorization of some kind. It does not. An authorization may be associated with an identity, but that is external to the PKI.

Trust in implementation also abounds. For example, the CA's systems containing the private keys used to sign the certificates must protect those keys. If an adversary can obtain those private keys, it can issue certificates in the name of the CA, or revoke existing certificates issued by that CA.

One final, critical assumption is that no two certificates will have the same public (and hence private) key. If Alice discovers Bob's certificate has the same public key as hers, she knows Bob's private key, violating a key assumption in the use of public key cryptosystems. A study of certificates throughout the Internet shows this problem has arisen in practice [1151, 1152].

11.5 Storing and Revoking Keys

Key storage arises when a user needs to protect a cryptographic key in a way other than by remembering it. If the key is public, of course, any certificate-based mechanism will suffice, because the goal is to protect the key's integrity. But secret keys (for symmetric cryptosystems) and private keys (for public key cryptosystems) must have their confidentiality protected as well.

11.5.1 Key Storage

Protecting cryptographic keys sounds simple: just put the key into a file, and use operating system access control mechanisms to protect it. Unfortunately, as discussed in Chapter 24, operating system access control mechanisms can often be evaded or defeated, or may not apply to some users. On a single-user system, this consideration is irrelevant, because no one else will have access to the system while the key is on the system. On a multiuser system, other users have access to the system. On a networked system, an attacker could trick the owner into downloading a program that would send keystrokes and files to the attacker, thereby revealing the confidential cryptographic key. We consider these systems.

On such systems, enciphering the file containing the keys will not work, either. When the user enters the key to decipher the file, the key and the contents

of the file will reside in memory at some point; this is potentially visible to other users on a multiuser system. The keystrokes used to decipher the file could be recorded and replayed at a later date. Either will compromise the key.

A feasible solution is to put the key onto one or more physical devices, such as a special terminal, ROM, or smart card [536, 618, 1265]. The key never enters the computer's memory. Instead, to encipher a message, the user inserts the smart card into a special device that can read from, and write to, the computer. The computer sends it the message to be protected, and the device uses the key on the smart card to encipher the message and send it back to the computer. At no point is the cryptographic key present on the computer.

A variant relies on the observation that if the smart card is stolen, the thief has the cryptographic key. Instead of having it on one card, the key is split over multiple devices (two cards, a card and the physical card reader, and so on.) Now, if a thief steals one of the cards, the stolen card is useless because it does not contain the entire key.

11.5.1.1 Key Escrow

As the previous discussion implies, keys can belong to roles.

EXAMPLE: The UNIX superuser password, like the Windows Administrator password, refers to the role of system administrator. In the absence of other password management techniques (see Chapter 13) all people who take those roles need to know the password.

A reasonable concern is how one recovers the key if it is lost, or if the people who know it are unable or unwilling to reveal it. Three alternatives arise: either the key or the cryptosystem can be weak, or a copy of the key can be placed somewhere.

> **Definition 11–9.** A *key escrow system* is a system in which a third party can recover a cryptographic key.

The contexts in which key escrow arises are business (recovery of backup keys, for example) and law enforcement (recovery of keys used to encipher communications to which an authority requires access, such as enciphered letters or telephone messages). Beth et al. [188] identify five desirable properties or goals.

1. The escrow system should not depend on the encipherment algorithm. The escrow techniques should work regardless of how the messages are enciphered.
2. Privacy protection mechanisms must work from one end to the other and be part of the user interface. This protects the user's privacy unless the escrowed keys are used, and then only those who have the escrowed keys can access the messages.

the related key, the authority must solve an instance of the discrete log problem. Techniques such as this assume that the difficulty of solving a particular problem is relatively constant. With advances in technology, such assumptions must be examined carefully.

Bellare and Rivest [158] have proposed a technique called "translucent cryptography," in which some fraction f of the messages Alice sends to Bob can be read. Their proposal relies on a cryptographic technique called "oblivious transfer," in which a message is received with a given probability [156]. This is not a key escrow system, because the keys are not available, but it does serve the ends of such a system in that the messages can be read with a specified probability. The puzzle is the value to which f must be set.

Identity-based encryption uses as a public key a publicly known identifier, for example an identifier that uniquely names the user. First proposed by Shamir [1724], such a scheme requires a trusted third party to use (or provide to the requester to use) a secret to compute the private keys, because if no secret were used, anyone could derive a private key from a public key. Such a scheme also provides an effective escrow system, because given a message enciphered with a public key, the trusted third party can use the secret to derive the corresponding private key and read the message.

Shamir identified two additional properties that public key cryptosystems must meet in order to be suitable for identity-based encryption:

1. Private keys can be easily computed from public keys and a secret s.
2. It is computationally infeasible to compute a private key from a public key without knowing s.

He then showed that the RSA cryptosystem cannot meet both these conditions at the same time. In 2001, Cocks [430] and Boneh and Franklin [260, 261] independently developed identity-based encryption schemes. Boneh and Franklin's method provides key escrow in the way that Shamir's scheme does, and they further showed how to augment their system to provide a "global escrow" key to decrypt any ciphertext encrypted using the public keys of their system.

11.5.2 Key Revocation

Certificate formats contain a key expiration date. If a key becomes invalid before that date, it must be revoked. Typically, this means that the key is compromised, or that the binding between the subject and the key has changed.

We distinguish this from an expired certificate. An expired certificate has reached a predesignated period after which it is no longer valid. That the lifetime has been exceeded is the only reason. A revoked certificate has been canceled at the request of the owner or issuer for some reason other than expiration.

There are two problems with revoking a public key. The first is to ensure that the revocation is correct—in other words, to ensure that the entity revoking the

key is authorized to do so. The second is to ensure timeliness of the revocation throughout the infrastructure. This second problem depends on reliable and highly connected servers and is a function of the infrastructure as well as of the locations of the certificates and the principals who have copies of those certificates. Ideally, notice of the revocation will be sent to all parties when received, but invariably there will be a time lag.

The Internet X.509 PKI uses lists of certificates.

Definition 11–10. A *certificate revocation list* is a list of certificates that are no longer valid.

A certificate revocation list contains the serial numbers of the revoked certificates and the dates on which they were revoked. It also contains the name of the issuer, the date on which the list was issued, and when the next list is expected to be issued. The issuer also signs the list [457, 2176]. Under X.509, only the issuer of a certificate can revoke it.

To minimize the time lag, the Internet X.509 PKI also supports an online revocation system [1665]. When validating a certificate, the system can use the Online Certificate Status Protocol (OCSP) to determine whether the certificate has been revoked. The request includes the certificate's serial number, the hash of its issuer name, the hash of its issuer's public key, and an identification of the hash algorithm. The server will respond that the certificate is "good," "revoked," or "unknown" (meaning the responder does not know about the certificate being requested). This method is particularly useful when time is critical, for example during stock trades.

PGP allows signers of certificates to revoke their signatures as well as allowing owners of certificates, and their designees, to revoke the entire certificates. The certificate revocation is placed into a PGP packet and is signed just like a regular PGP certificate. A special flag marks it as a revocation message.

Boneh and Franklin [260, 261] point out that identity-based encryption provides a simple key revocation mechanism, provided the lifetime of the key is known when it is generated. Simply add some extra data that depends upon the time—for example, the current year or month—to the public key. Then the corresponding private key is valid only until the extra information expires, for example at the end of the year or the month. This revokes the key at that time by causing it to expire.

11.6 Summary

Cryptographic infrastructure provides the mechanisms needed to use cryptography. The infrastructure sees to the distribution of keys and the security of the procedures and mechanisms implementing cryptographic algorithms and protocols.