Chapter 7

Authentication and Public Key Infrastructure (PKI)

Key management

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. Here we assume that the key gives us the true identity. So in our consideration we assume that we already have authentication and the key has been assigned to the user. In this chapter we discuss the following concepts: **Key exchange:** Session vs. interchange keys; Classical, public key methods; Key generation. **Cryptographic key infrastructure:** Certificates. **Key storage:** Key escrow; Key revocation.

Notation

- **$X \rightarrow Y : \{ Z \| W \} k_{X,Y}$**: X sends Y the message produced by concatenating Z and W enciphered by key $k_{X,Y}$, which is shared by users X and Y.
- **$A \rightarrow T : \{ Z \} k_A \| \{ W \} k_{A,T}$:** A sends T a message with the concatenation of Z enciphered using k_A, A's key, and W enciphered using $k_{A,T}$, key shared by A and T.
- **$rand_1$, $rand_2$:** nonces (nonrepeating random numbers)

Session, Interchange Keys

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. For e.g. A wants to send a message m to B, assume public key encryption, here A generates a random cryptographic key $k_{session}$ and uses it to encipher m that is to be used for this message only called a session key. Now A enciphers $k_{session}$ with B's public key k_B (k_B enciphers all session keys A uses to communicate with B called an interchange key). Finally A sends $\{m\} k_{session} \| \{k_{session}\} k_B$.

Benefits

It limits amount of traffic enciphered with single key, here standard practice is to decrease the amount of traffic an attacker can obtain.

Prevents some attacks: E.g.- 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.

Key Exchange

Key exchange is any method in cryptography by which cryptographic keys are exchanged between users, allowing use of a cryptographic algorithm. The **key exchange problem** is how to exchange whatever keys or other information needed so that no one else can obtain a copy. Traditionally, this required trusted couriers (with or without briefcases handcuffed to their wrists), or diplomatic bags, or some other secure channel. With the advent of public key / private key cipher algorithms, the encrypting key could be made public, since (at least for high quality algorithms) no one without the decrypting key could decrypt the message.

In principle, then, the only remaining problem was to be sure (or at least confident) that a public key actually belonged to its supposed owner. Because it is possible to 'spoof' another's identity in any of several ways, this is not a trivial or easily solved problem, particularly when the two users involved have never met and know nothing about each other.

The goal of key exchange is to enable A to B, and vice versa communication secret, using a shared cryptographic key. Solutions to this problem must meet the following criteria.

1. The key that A and B are to share cannot be transmitted in the clear. Either it must be enciphered when sent, or A and B must derive it without an exchange of data from which the key can be derived. (A and B can exchange data, but a third party cannot derive the key from the data exchanged.)
2. A and B may decide to trust a third party (called "C" here).
3. The cryptosystems and protocols are publicly known. The only secret data is to be the cryptographic keys involved.

Classical Key Exchange

Suppose Alice(A) and Bob(B) wish to communicate. If they share a common key, they can use a classical cryptosystem. But how do they agree on a common key? If A sends one to B, Eve the eavesdropper will see it and be able to read the traffic between them. In this context A can't send the key to be shared to B in the clear. To avoid this bootstrapping problem, classical protocols rely on a trusted third party, Cathy(C). A and C share a secret key, $k_{A,}$ and B and C share a (different) secret key, k_B. The goal is to provide a secret key, k_S that A and B share. The following simple protocol provides a starting point

Simple Protocol

To avoid bootstrap problem we can use the following simple protocol.

1. $A \rightarrow C$: {request for session key to B} k_A.
2. $C \rightarrow A$: $\{ k_{session} \}k_A \,\|\, \{ k_{session} \}k_B$.
3. $A \rightarrow B$: $\{ k_{session} \}k_B$.

B now deciphers the message and uses $k_{session}$ to communicate with A.

Problems: This protocol is the basis for many more sophisticated protocols. However, B does not know to whom he is talking. This problem leads us to **replay attack**: Eve records message from A to B, later replays it; B may think he's talking to A, but he isn't and **session key reuse**: Eve replays message from A to B, so B re-uses session key. So the protocols must provide authentication and defense against replay attack. The following algorithm provides solution to the above problem.

Needham-Schroeder Protocol

1. $A \rightarrow C : \{ A \,\|\, B \,\|\, rand_1 \}$

   Alice sends a message to the server identifying herself and Bob, telling the server she wants to communicate with Bob.
2. $C \rightarrow A : \{ A \,\|\, B \,\|\, rand_1 \,\|\, k_{session} \,\|\{A \,\|\, k_{session}\}\, k_B \}\, k_A$

   The server (Cathy) generates $k_{session}$ and sends back to Alice a copy encrypted under k_B for Alice to forward to Bob and also a copy for Alice. Since Alice may be requesting keys for several different people, the nonce assures Alice that the message is fresh and that the server (Cathy) is replying to that particular message and the inclusion of Bob's name tells Alice who she is to share this key with.
3. $A \rightarrow B : \{ A \,\|\, k_{session} \}\, k_B$

Alice forwards the key to Bob who can decrypt it with the key he shares with the server (Cathy), thus authenticating the data.

4. B $\rightarrow$ A : { rand$_2$ } k$_{session}$

   Bob sends Alice a nonce encrypted under k$_{session}$ to show that he has the key.

5. A $\rightarrow$ B : { rand$_2$ – 1 }k$_{session}$

   Alice performs a simple operation on the nonce, re-encrypts it and sends it back verifying that she is still alive and that she holds the key.

Argument: A talking to B

Second message: This is the response to the first message (since rand$_1$ in second message is same as of in rand$_1$ first message) enciphered using key only A and C knows.

Third message: A knows only B can read it since only B can derive session key from message and any messages enciphered with that key are from B.

Argument: B talking to A

Third message: This message contains A (name) and session key provided by C that is enciphered using key only B and C know.

Fourth message: Uses session key to determine if it is replay from Eve the eavesdropper. 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 is 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), and encipher the result and return it to Bob. Then Bob will be sure he is talking to Alice.

Denning-Sacco Modification

Needham-Schroeder protocol assumes all keys are secret but suppose that if Eve can obtain session key. How does that affect protocol? In this context Eve knows $k_{session}$. So we have situation where B thinks that A has sent the message as seen from below:

1. Eve $\rightarrow$ B : { A || $k_{session}$ } k_B
2. B $\rightarrow$ A : { $rand_2$ } $k_{session}$ [intercepted by Eve]
3. Eve $\rightarrow$ B : { $rand_2 - 1$ }$k_{session}$.

Solution: In protocol (Needham-Schroeder), Eve impersonates A. So we have replay in third step (first in above). For this solution can be use of **time stamp T** to detect replay.

Needham-Schroeder with Denning-Sacco Modification

Denning and Sacco suggest using timestamps to enable B to detect replay.

1. A $\rightarrow$C : { A || B || $rand_1$ }
2. C $\rightarrow$A : { A || B || $rand_1$ || $k_{session}$ || {A || T || $k_{session}$} k_B } k_A
3. A $\rightarrow$B : {A || T || $k_{session}$} k_B
4. B $\rightarrow$A : { $rand_2$ } $k_{session}$
5. A $\rightarrow$B : { $rand_2 - 1$ }$k_{session}$

Where, T is a timestamp. When B gets the message in step 3, he rejects it if the timestamp is too old (too old being determined from the system in use). This modification requires synchronized clocks. The weakness with this solution is a party with a slow clock is vulnerable to a replay attack adds that a party with a fast clock is also vulnerable, and simply resetting the clock does not eliminate the vulnerability.

Otway-Rees Protocol

This protocol corrects problem of Eve replaying the third message in the protocol and does not use timestamps so it is not vulnerable to the problems that Denning-Sacco modification has. It uses integer **num** to associate all messages with particular exchange. The following are the steps in the protocol.

1. A $\rightarrow$B : num || A || B || { $rand_1$ || num || A || B }k_A
2. B $\rightarrow$C : num || A || B || { $rand_1$ || num || A || B }k_A || {$rand_2$ || num || A || B }k_B
3. C $\rightarrow$B : num || { $rand_1$ || $k_{session}$ }k_A || { $rand_2$ || $k_{session}$ }k_B

4. B $\rightarrow$A : num || { $rand_1$ || $k_{session}$ }k_A

The purpose of the integer **num** is to associate all messages with a particular exchange.

Argument: A talking to B

Fourth message: When Alice receives the fourth message from Bob, she checks that the num agrees with the num in the first message that she sent to Bob. If so, she knows that this is part of the exchange. She also trusts that Cathy generated the session key because only Cathy and Alice know k_{Alice}, and the random number $rand_1$ agrees with what Alice put in the enciphered portion of the message. Combining these factors, Alice is now convinced that she is talking to Bob.

Argument: B talking to A

Third message: When Bob receives the message from Cathy, he determines that the num corresponds to the one he received from Alice and sent to Cathy. He deciphers that portion of the message enciphered with his key, and checks that $rand_2$ is what he sent. He then knows that Cathy sent the reply, and that it applies to the exchange with Alice.

Suppose Eve gets old $k_{session}$, message in third step num || {$rand_1$ || $k_{session}$}k_A || {$rand_2$ || $k_{session}$}k_B. Eve forwards appropriate part to A

- A has no ongoing key exchange with B: num matches nothing, so is rejected
- A has ongoing key exchange with B: num does not match, so is again rejected
  - If replay is for the current key exchange, and Eve sent the relevant part before B did, Eve could simply listen to traffic; no replay involved.

Kerberos

It is a secret key based service for providing authentication in a network. It is based on Needham-Schroeder with Denning-Sacco modification. It makes use of a trusted third party, termed a key distribution center (KDC), which consists of two logically separate parts: an Authentication Server (AS) and a Ticket Granting Server (TGS). Kerberos works on the basis of "tickets" which serve to prove the identity of users.

The KDC maintains a database of secret keys; each entity on the network — whether a client or a server — shares a secret key known only to itself and to the KDC. Knowledge of this key serves to prove an entity's identity. For communication between two entities, the KDC generates a session key which they can use to secure their interactions

Here once authenticator authenticates the user, the ticket must be used by the client to request the service from the server.

Idea

- User u authenticates to Kerberos server and obtains ticket $T_{u,TGS}$ for ticket granting service (TGS).
- For using service s by u: User sends authenticator A_u, ticket $T_{u,TGS}$ to TGS asking for ticket for service. TGS sends ticket $T_{u,s}$ to user and user sends Au, $T_{u,s}$ to server as request to use s.

Ticket

It is the credential saying issuer has identified ticket requester. Example ticket issued to user u for service s $\mathbf{T_{u,s} = s \parallel \{ u \parallel u\text{'s address} \parallel \text{valid time} \parallel k_{u,s} \} k_s}$, where: $k_{u,s}$ is session key for user and service; Valid time is interval for which ticket valid; u's address may be IP address or something else.

Authenticator

It is the system containing identity of sender of ticket that is used to confirm sender is entity to which ticket was issued. Example: authenticator user u generates for service s
$\mathbf{A_{u,s} = \{ u \parallel \text{generation time} \parallel k_t \} k_{u,s}}$, where: k_t is alternate session key; Generation time is when authenticator generated.

Protocol

1. user $\rightarrow$ C: {user || TGS}
2. C $\rightarrow$ user: $\{k_{u,TGS}\}k_u \parallel T_{u,TGS}$
3. user $\rightarrow$ TGS: service || $A_{u,TGS} \parallel T_{u,TGS}$
4. TGS $\rightarrow$ user: user || $\{k_{u,s}\}k_{u,TGS} \parallel T_{u,s}$
5. user $\rightarrow$ service: $A_{u,s} \parallel T_{u,s}$
6. service $\rightarrow$ user: $(t + 1)\ k_{u,s}$

Analysis

- First two steps get user ticket to use TGS. Here user u can obtain session key only if u knows key shared with C.
- Next four steps show how u gets and uses ticket for service s
  - Service s validates request by checking sender (using $A_{u,s}$) is same as entity ticket issued to.
  - Step 6 optional; used when u requests confirmation

Problems

Kerberos relies on clocks being synchronized to prevent replay attacks. If the clocks are not synchronized, and if old tickets and authenticators are not cached, replay is possible.

The tickets have some fixed fields so a dictionary attack can be used to determine keys shared by services or users and the ticket-granting service or the authentication service.

Public Key Cryptographic Key Exchange

In this approach interchange keys are known as of e_A, e_B A and B's public keys known to all and d_A, d_B A and B's private keys known only to owner. The simple protocol with $k_{session}$ as desired session key is $A \rightarrow B$: $\{k_{session}\}e_B$.

Problem and Solution

It is vulnerable to forgery or replay because e_B known to anyone, B has no assurance that A sent message. A simple fix uses A's private key, where $k_{session}$ is desired session key and $A \rightarrow B$: $A \| \{\{k_{session}\}d_A\}e_B$.

Notes:

A can include message enciphered with $k_{session}$. The above solution assumes B has A's public key, and vice versa. If not, each must get it from public server. If keys not bound to identity of owner, attacker Eve can launch a man-in-the-middle attack. Solution to this (binding identity to keys) discussed later as public key infrastructure (PKI).

Man-in-the-Middle Attack

1. $A \rightarrow P$: { send me B's public key } [intercepted by Eve]
2. $Eve \rightarrow P$: { send me B's public key }
3. $P \rightarrow Eve : e_B$
4. $Eve \rightarrow A : e_{Eve}$
5. $A \rightarrow B$: { $k_{session}$ } e_{Eve} [intercepted by Eve]
6. $Eve \rightarrow B$: { $k_{session}$ }e_B

Key Generation

The secrecy that cryptosystems provide resides in the selection of the cryptographic key. If an attacker can determine someone else's key, the attacker can read all traffic enciphered using that key or can use that key to impersonate its owner. Hence, generating keys that are difficult to guess or to determine from available information is critical.

Goal: generate keys that are difficult to guess

Problem statement: given a set of K potential keys, choose one randomly. This is equivalent to selecting a random number between 0 and K–1 inclusive.

Why is this hard: generating random numbers is hard since numbers are usually pseudo-random, that is, generated by an algorithm.

What is "Random"?

A sequence of random numbers is a sequence of numbers n_1, n_2, ... such that for any positive integer k, an observer cannot predict n_k even if n_1, ..., n_{k-1} are known.

Best physical source of randomness: Random pulses, Electromagnetic phenomena, Characteristics of computing environment like disk latency, Ambient background noise.

What is "Pseudorandom"?

A sequence of pseudorandom numbers is a sequence of numbers intended to simulate a sequence of cryptographically random numbers but generated by an algorithm.

Very difficult to do this well

Linear congruential generators [$n_k = (an_{k-1} + b) \bmod n$] this is broken; Polynomial congruential generators [$n_k = (a_j n_{k-1}^j + \ldots + a_1 n_{k-1} + a_0) \bmod n$] broken too. Here, “broken” means next number in sequence can be determined.

Best Pseudorandom Numbers

A strong mixing function is a function of two or more inputs that produces an output each bit of which depends on some nonlinear function of all the bits of the input. Examples: DES, MD5, SHA-1.
E.g.: On a UNIX system, the status of the processes is highly variable. An attacker is unlikely to reproduce the state at a future time. So the command **(date ; ps gaux) | md5** would produce acceptable pseudorandom data. In this command, ps gaux lists all information about all processes on the system.

Cryptographic Key Infrastructure

Because classical 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. 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 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.

Certificates

A certificate is a token that binds an identity to a cryptographic key. When B wants to communicate with A, B obtains A's certificate C_A. for e.g. Create token (message) containing: Identity of principal (here, A), Corresponding public key, Timestamp (when issued), and Other information (perhaps identity of signer) signed by trusted authority (here, C) as $C_A = \{ e_A \| A \| T \}d_C$.

Use

B gets A’s certificate: If B knows C’s public key, B can decipher the certificate and see when was certificate issued? Is the principal A? Now B has A’s public key.

Problem: B needs C’s public key to validate certificate. Two approaches deal with this problem. The first, by Merkle, eliminates Cs signature; the second structures certificates into signature chains.

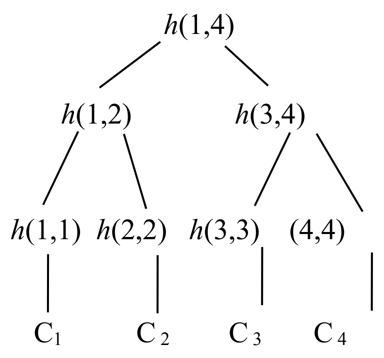
Merkle’s Tree Scheme

This scheme keeps certificates in a file and changing any certificate changes the file. This reduces the problem of substituting faked certificates to a data integrity problem. Cryptographic hash functions create checksums that reveal changes to files.

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

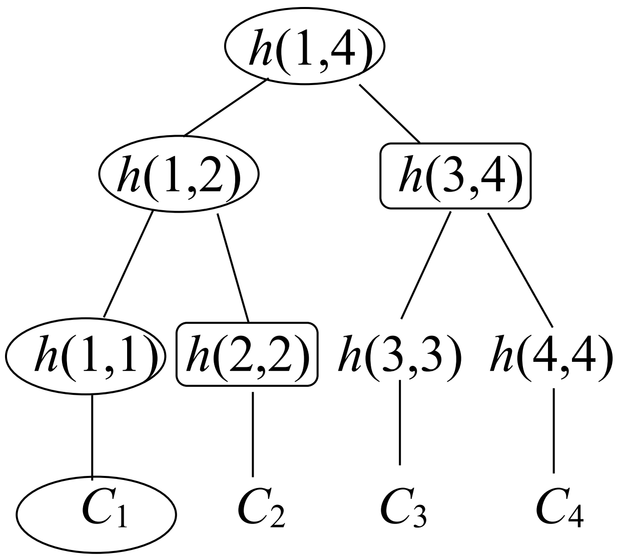
$$h(i, j) = f(C_i, C_j) \qquad \text{if } i \geq j,$$

$$h(i, j) = f(h(i, \lfloor (i+j)/2 \rfloor), h(\lfloor (i+j)/2 \rfloor +1, j)) \qquad \text{if } i < j,$$



Example:

- ❑ Construct Merkle hash tree by computing hashes recursively
    - ❖ *h* is hash function
    - ❖ *Ci* is certificate *i*
- ❑ Root hash (*h*(1,4) in example) is published and is known to all
    - ❖ Root hash is signed by the certificate authority to ensure the value’s integrity



Validation

To validate C_1:

Compute h(1, 1), obtain h(2, 2)

Compute h(1, 2), obtain h(3, 4)

Compute h(1,4) and compare to known h(1, 4)

- Need to know siblings of nodes on path from C1 to the root
  - The proof from CA consists of these hashes (in rectangles on the left)

Problem

In this scheme file must be available for validation otherwise, can't recompute hash at root of tree. Not practical if there are too many certificates and users and users and certificates distributed over widely separated systems.

Issuers

Certification Authority (CA): 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.

Assume that X<<Y>> represents the certificate that X generated for the subject Y (X is the CA that issued the certificate). 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.

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>>

Storing and Recovering 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 classical cryptosystems) and private keys (for public key cryptosystems) must have their confidentiality protected as well.

Storing Keys

In case of multi-user or networked systems: attackers may defeat access control mechanisms. Even the encipherment of file containing key does not help since the attacker can monitor keystrokes to decipher files as key will be resident in memory that attacker may be able to read. One of the solutions can be the use of physical devices like "smart card" where key never enters system. In this approach the card can be stolen, so use of two devices that combine bits to make single key can be used.

Key Revocation

If the certificate is invalidated before expiration i.e. the key is invalid, then it may be due to compromised key or may be due to change in circumstance (e.g., someone leaving company).

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.

CRLs (Certificate Revocation Lists)

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. Under X.509, only the issuer of a certificate can revoke it.

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.