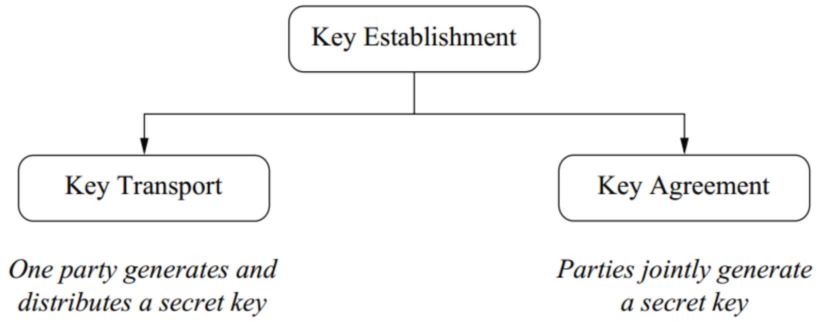
Chapter 8- Key Establishment

In this lecture, we will learn how to use **symmetric** and **asymmetric** cryptosystems for:

1- **Distribute** keys among remote parities.
2- **Establish** keys between two parties**.**

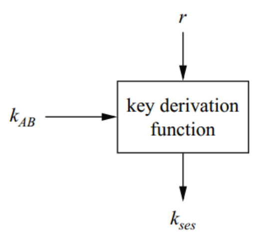
- Key **establishment** deals with establishing a shared secret between two or more parties.
- It is methods can be classified into ***key transport*** and ***key agreement***.



Key Freshness and Key Derivation

In many (but not all) security systems it is desirable to use cryptographic keys whichare only valid for a **limited** time, e.g., for one Internet connection.

- Such keys arecalled ***session keys***.
- We use an already established secret keyto *derive* fresh session keys.
- The principal idea is to use a key derivation function (KDF).



- Typically, a non-secret parameter *r* is processed together with the joint secret k_{AB} between the users Alice and Bob.
- Derivation function can be **encryption** function such as AES, or **hashing** as HMAC.

Key Derivation with Nonces

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

Or

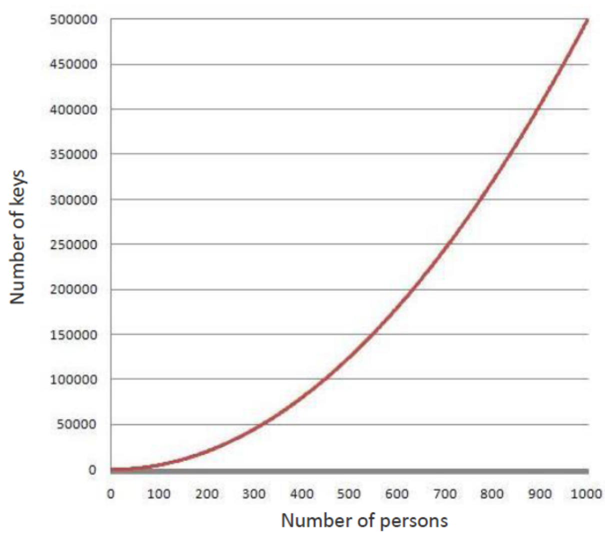
$$k_{ses} = HMAC_{k_{AB}}(r)$$

The n^2 Key Distribution Problem

If we have n users then:

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

Example. A mid-size company with 750 employees wants to set up secure email communication with symmetric keys. For this purpose, $750\times749/2 = 280,875$ symmetric key pairs must be generated, and $750\times749 = 561,750$ keys must be distributed via secure channels.

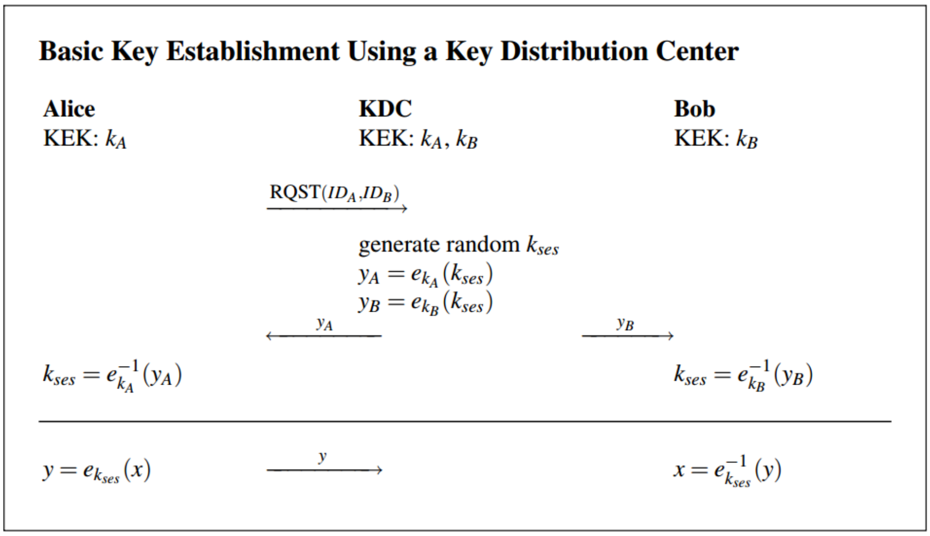


Key Establishment Using Symmetric-Key Techniques

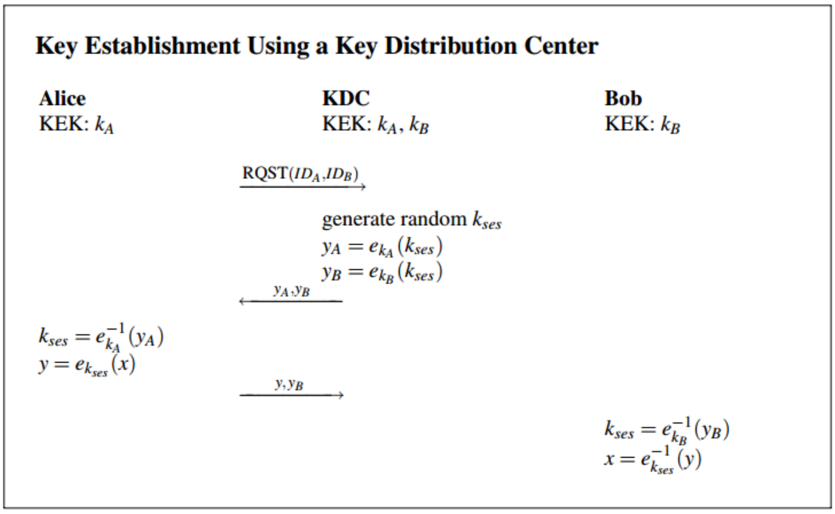
- Symmetric ciphers can be used to establish *secret (session)* keys.
- The protocols introduced in the following all perform **key transport** and **<u>not</u> key agreement**.

Key Establishment with a Key Distribution Center (KDC)

- KDC is a **server** that is fully **trusted** by all users and that shares a secret key with each user.
- This key, which is named the *Key Encryption Key* (**KEK**), is used to securely transmit session keys to users.



- The KEKs *kA* and *kB* are long-term keys that do not change.
- The session key *kses* is session key that changes frequently, ideally for every communication session.
- It is easy to modify the above protocol such that we **save one** communication session.



- Both of the KDC-based protocols have the advantage that there are only *n* long term symmetric key pairs in the system.
- The *n* long-term KEKS only need to be stored by the KDC, while each user only stores his or her own KEK.

Security of KDC

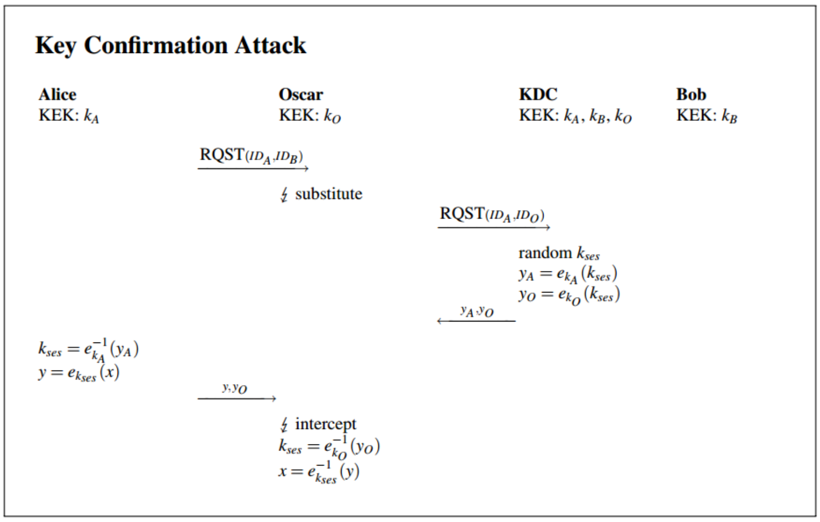
KDC suffers from two attacks: **replay attack** and **key confirmation attack**.

Replay attack:

If Oscar gets hold of a previous session key, he can impersonate the KDC and resend old messages yA and yB to Alice and Bob.

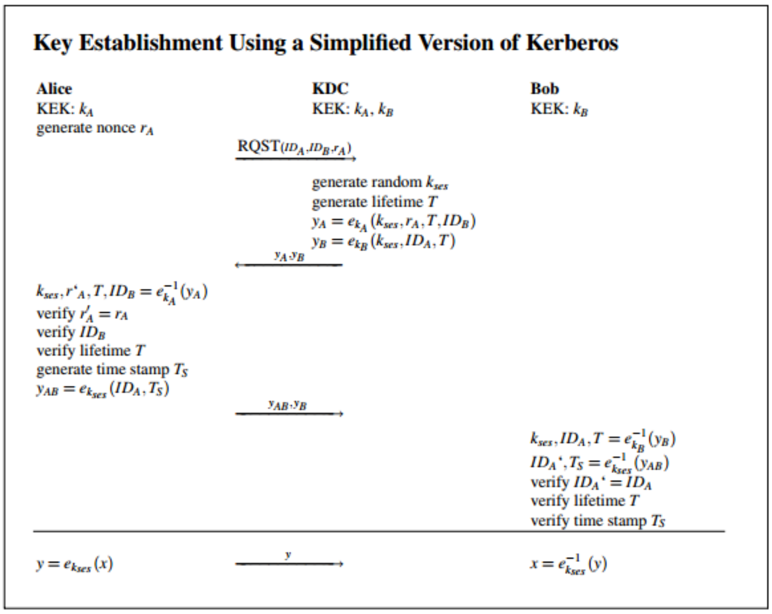
Key confirmation attack

By changing the session-request message Oscar can trick the KDC and Alice to set up session between him and Alice as opposed to between Alice and Bob.



Kerberos

- A more advanced protocol that protects against both replay and key confirmation attacks is Kerberos.
- It is, in fact, more than a mere key distribution protocol; itsmain purpose is to provide user **authentication** in computer networks.



- In the beginning, Alice sends a random nonce r_A to the KDC.
- This can be considered as a *challenge* because she challenges the KDC to encrypt it with their joint KEK kA.
- If the returned challenge rA matches the sent one, Alice is assured that the message yA was actually sent by the KDC.
- This method to authenticate users is known as *challenge-response protocol* and is widely used, e.g., for authentication of smart cards.

Problems with Symmetric-Key Distribution

1- Communication requirements
2- Secure channel during initialization
3- Single point of failure
4- No perfect forward secrecy

- If any of the KEKs becomes compromised, e.g., through a hacker or Trojan software running on a user's computer, the consequences are serious.
- For instance, if Oscar got a hold of Alice's KEK kA, he can recover the session key from all messages yA that the KDC sends out.
- A cryptographic protocol has *perfect forward secrecy* (PFS) if the compromise of long-term keys does not allow an attacker to obtain past session keys.
- The main mechanism to assure PFS is to employ **public-key techniques**.

Key Establishment Using Asymmetric Techniques

- Public key cryptosystems can be used for both **key transport** (such as encrypt key by RSA) and **key agreement**.

Diffie–Hellman Key Exchange

- The *Diffie–Hellman key exchange (DHKE)*, proposed by Whitfield Diffie and Martin Hellman in 1976, was the first asymmetric scheme published in the open literature.
- It provides a practical solution to the key distribution problem, i.e., it enables two parties to derive a common secret key by communicating over an insecure channel.
- The DHKE is a based on the *discrete logarithm* problem.
- This fundamental key agreement technique is implemented in many open and commercial cryptographic protocols like Secure Shell (SSH), Transport Layer Security (TLS), and Internet Protocol Security (IPSec).
- The basic idea behind the DHKE is that exponentiation in Z^*_p, p prime, is a one-way function and that exponentiation is commutative, i.e.,

$$k = (\alpha^x)^y \equiv (\alpha^y)^x \bmod p$$

Diffie–Hellman Set-up

1. Choose a large prime p.
2. Choose an integer $\alpha \in \{2, 3, \ldots, p-2\}$.
3. Publish p and α.

If Alice and Bob both know the public parameters p and α computed in the set-up phase, they can generate a joint secret key k with the following key-exchange protocol:

Diffie–Hellman Key Exchange

Alice		**Bob**
choose $a = k_{pr,A} \in \{2, \ldots, p-2\}$ compute $A = k_{pub,A} \equiv \alpha^a \bmod p$		choose $b = k_{pr,B} \in \{2, \ldots, p-2\}$ compute $B = k_{pub,B} \equiv \alpha^b \bmod p$
	$\xrightarrow{k_{pub,A}=A}$	
	$\xleftarrow{k_{pub,B}=B}$	
$k_{AB} = k_{pub,B}^{k_{pr,A}} \equiv B^a \bmod p$		$k_{AB} = k_{pub,A}^{k_{pr,B}} \equiv A^b \bmod p$

Example 8.1. The Diffie–Hellman domain parameters are $p = 29$ and $\alpha = 2$. The protocol proceeds as follows:

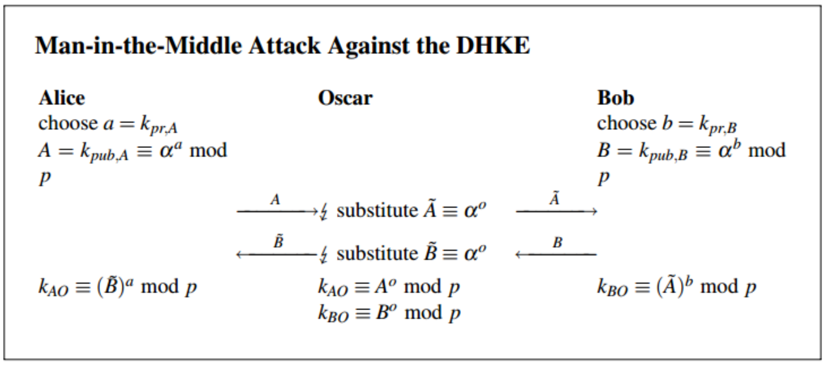
Alice		**Bob**
choose $a = k_{pr,A} = 5$ $A = k_{pub,A} = 2^5 \equiv 3 \bmod 29$		choose $b = k_{pr,B} = 12$ $B = k_{pub,B} = 2^{12} \equiv 7 \bmod 29$
	$\xrightarrow{A=3}$	
	$\xleftarrow{B=7}$	
$k_{AB} = B^a \equiv 7^5 = 16 \bmod 29$		$k_{AB} = A^b = 3^{12} \equiv 16 \bmod 29$

As one can see, both parties compute the value $k_{AB} = 16$, which can be used as a joint secret, e.g., as a session key for symmetric encryption.

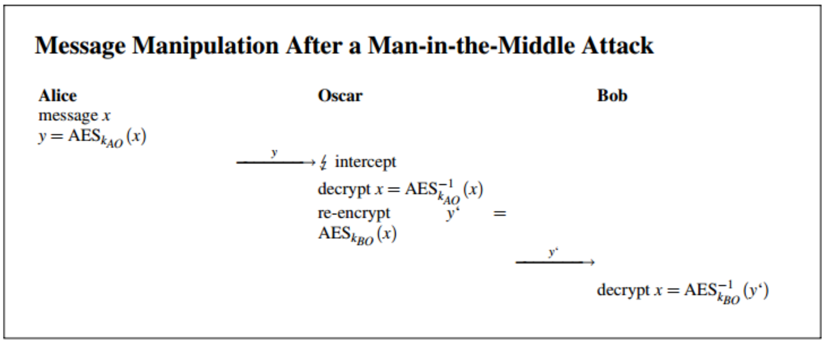
- The session key k_{AB} that is being computed in the protocol has the same bit length as p.
- If we want to use it as a symmetric key for algorithms such as AES, we can simply take the 128 most significant bits.
- Alternatively, a hash function is sometimes applied to kAB and the output is then used as a symmetric key.

Man-in-the-Middle Attack (MIN)

The underlying idea of the MIM attack is that Oscar replaces both Alice'sand Bob's public key by his own. The attack is shown here:



- However, neither Alice nor Bob is aware of the fact that they share a key with Oscar and not with each other!
- Oscar has much control over encrypted traffic between Alice and Bob.
- As an example, here is how he can read encrypted messages in a way that goes unnoticed by Alice and Bob:



Certificates

- The underlying problem of the man-in-the-middle attack is that public keys are not authenticated.
- Certificate is a mechanism to address the problem of key authentication.
- The idea behind certificates is to use digital signature.

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

- The idea is that the receiver of a certificate verifies the signature prior to using the public key.
- The signatures for certificates are provided by a mutually trusted third party.
- This party is called the *Certification Authority* commonly abbreviated as *CA*.
- It is the task of the CA to generate and issue certificates for all users in the system.

Certificate Generation with User-Provided Keys

Alice		CA
generate $k_{pr,A}, k_{pub,A}$		
	$\xrightarrow{RQST(k_{pub,A}, ID_A)}$	
		verify ID_A $s_A = sig_{k_{pr,CA}}(k_{pub,A}, ID_A)$ $Cert_A = [(k_{pub,A}, ID_A), s_A]$
	$\xleftarrow{Cert_A}$	

In practice it is often advantageous that the CA not only **signs** the public keys but also **generates** the public–private key pairs for each user.

Certificate Generation with CA-Generated Keys

Alice		CA
request certificate	$\xrightarrow{RQST(ID_A)}$	
		verify ID_A generate $k_{pr,A}, k_{pub,A}$ $s_A = sig_{k_{pr,CA}}(k_{pub,A}, ID_A)$ $Cert_A = [(k_{pub,A}, ID_A), s_A]$
	$\xleftarrow{Cert_A, k_{pr,A}}$	

- Let's have a look at the DHKE which is protected with certificates:

Diffie–Hellman Key Exchange with Certificates

Alice		Bob
$a = k_{pr,A}$ $A = k_{pub,A} \equiv \alpha^a \bmod p$ $Cert_A = [(A, ID_A), s_A]$		$b = k_{pr,B}$ $B = k_{pub,B} \equiv \alpha^B \bmod p$ $Cert_B = [(B, ID_B), s_B]$
	$\xrightarrow{Cert_A}$	
	$\xleftarrow{Cert_B}$	
verify certificate: $ver_{k_{pub,CA}}(Cert_B)$ compute session key: $k_{AB} \equiv B^a \bmod p$		verify certificate: $ver_{k_{pub,CA}}(Cert_A)$ compute session key: $k_{AB} \equiv A^b \bmod p$