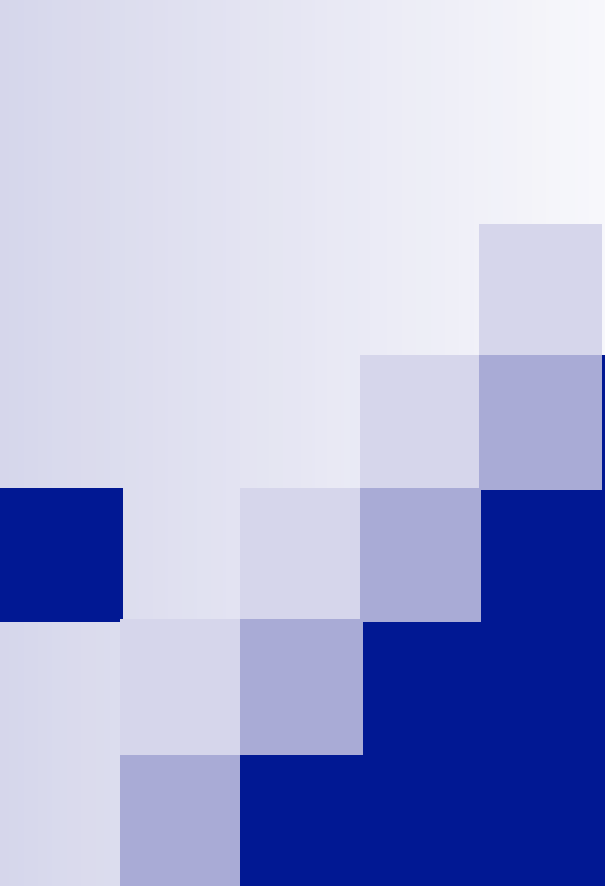




Distributed Systems

Lecture 13

Professor Louise E. Moser
Spring 2014

Security

- Security comprises:
 - **Confidentiality:** No unauthorized disclosure of information
 - **Integrity:** No accidental or malicious alteration of information
- To provide security, encrypt and decrypt messages using keys
- Two different approaches and kinds of keys
 - **Secret key:** Use a single shared secret key to encrypt and decrypt messages exchanged between Alice and Bob
 - **Public/private key:** Each party has his/her own public key and private key
 - Alice encrypts a message m that she sends to Bob with Bob's public key K_B^+
 - Bob decrypts an encrypted message that he receives from Alice with Bob's private key K_B^-

Security

■ Several problems with keys

- **Keys wear out:** The more data that are encrypted with a single key, the easier it is to discover the key => *don't use keys too often*
- **Compromised keys:** If a key is compromised, you can't use it again => *don't use the same key for all communication*
- **Long-lived keys:** Untrusted entities might play along and behave properly for a while, but not forever
=> *don't use the same key forever; make keys disposable*
- **Message replay:** Using the same key for more than one session allows old messages to be inserted into the current session
=> *use a different key for each session/connection*

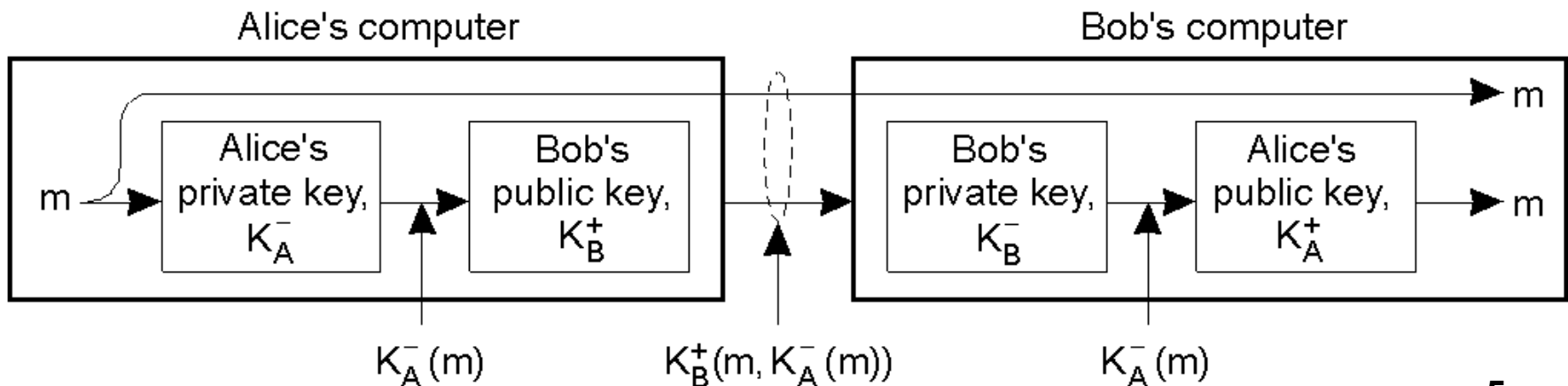
- Don't use valuable and expensive (encryption/decryption) keys for all communication, but only for **authentication**

Authentication

- In addition to security, we need:
 - **Authentication:** The receiver can verify the claimed identity of the sender
 - **Non-repudiation:** The sender cannot later deny that he/she sent the message
- To provide authentication and non-repudiation, have the sender sign all messages he/she transmits using a **digital signature** such that:
 - The signature can be verified, and
 - The message and the signature are uniquely associated

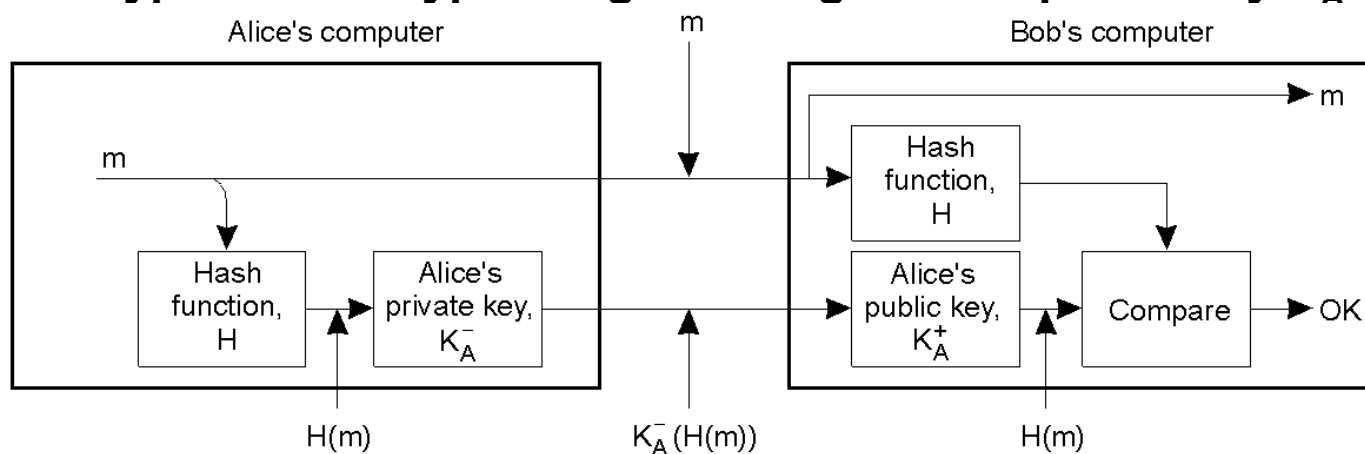
Digital Signatures

- Encrypting a message using public-key cryptography to provide both **authentication** and **security**
 - Alice signs the message using her private key K_A^-
 - Alice encrypts the message using Bob's public key K_B^+
 - Bob decrypts the message using his secret key K_B^-
 - Bob decrypts Alice's signature using Alice's public key K_A^+



Digital Signatures

- To improve performance, don't mix authentication and security
 - Instead, have the sender **transmit the message in the clear**, and **sign a message digest** using his/her digital signature
 - Alice takes a digest of message m using a hash function H , such as MD5
 - Alice signs the message digest using her private key K_A^-
 - Bob takes a digest of message m using the same hash function H
 - Bob decrypts the encrypted digest using Alice's public key K_A^+
- and

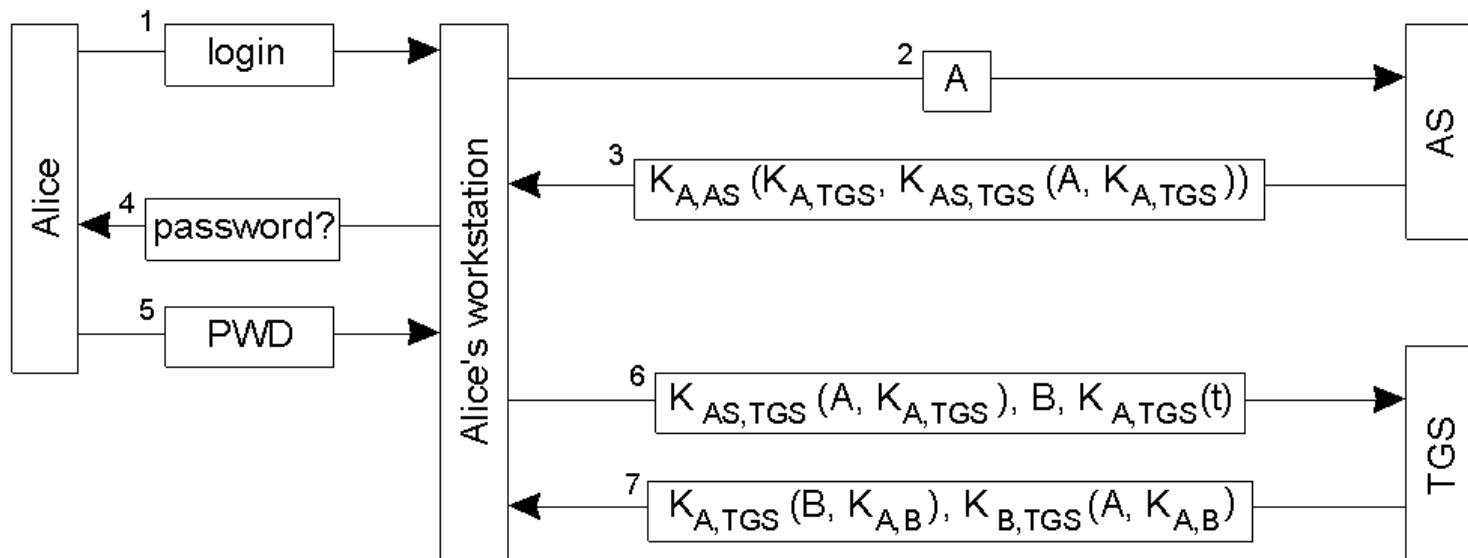


Example: Kerberos

- **Authentication** in **Kerberos** (a security system that helps clients to set up a secure channel with a server, developed at MIT, uses Needham-Schroeder authentication)

(2) Alice's workstation sends Alice's identity **A** to the **Authentication Server (AS)**

(3) AS sends back information to allow Alice to communicate with the **Ticket Generating Service (TGS)**

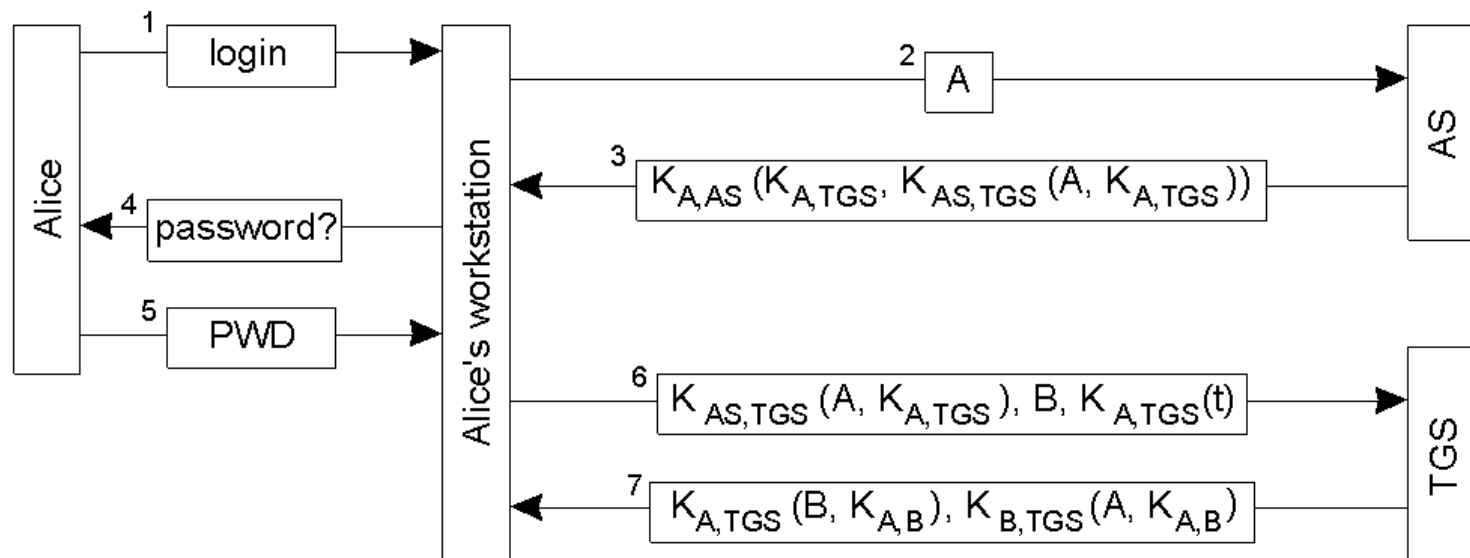


Example: Kerberos (cont.)

(5) Alice's password is converted into a **key $K_{A,TGS}$** which Alice uses to communicate with the TGS

(6) Alice's workstation sends a request to the TGS asking for a session key $K_{A,B}$ to be used for communication with Bob

(7) The TGS sends back the **session key $K_{A,B}$** , encrypted with Alice's key $K_{A,TGS}$ and encrypted with Bob's key $K_{B,TGS}$



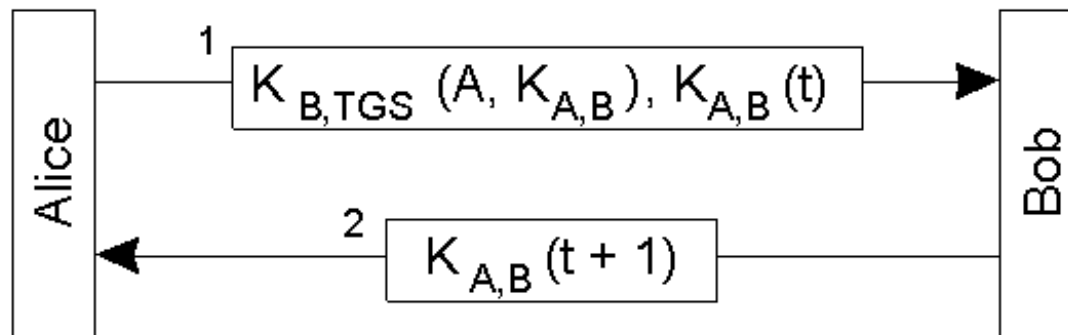
Example: Kerberos (cont.)

■ Setting up a secure channel in Kerberos

(1) Alice sends her identity A and the session key $K_{A,B}$ encrypted with Bob's key $K_{B,TGS}$ and also a nonce t encrypted with the session key $K_{A,B}$

(2) Bob returns the nonce (after adding 1) encrypted with the session key $K_{A,B}$

■ **Note:** Alice obtained $K_{B,TGS}(A, K_{A,B})$ from the TGS (on the previous slide)



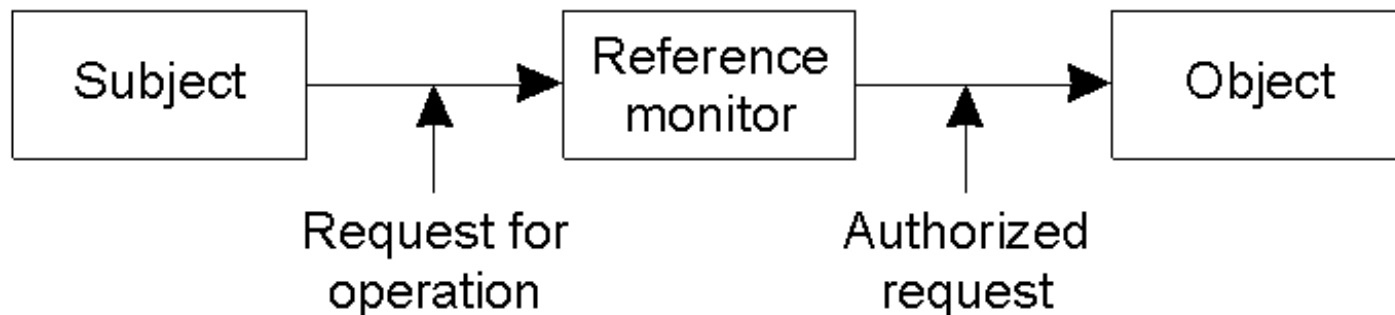


Access Control

- Authorization
- Access Control Matrix
- Protection Domain
- Firewall
- Secure Mobile Code

Authorization

- We need not only authentication but also authorization
 - **Authentication:** Verify that a subject says that he/she is who he/she claims to be, i.e., verify the identity of the subject
 - **Authorization:** Determine whether a subject is permitted to use certain data/operations of an object
- Authorization makes sense only if the requesting subject has been authenticated first



Access Control Matrix

■ Maintain an **Access Control Matrix (ACM)**

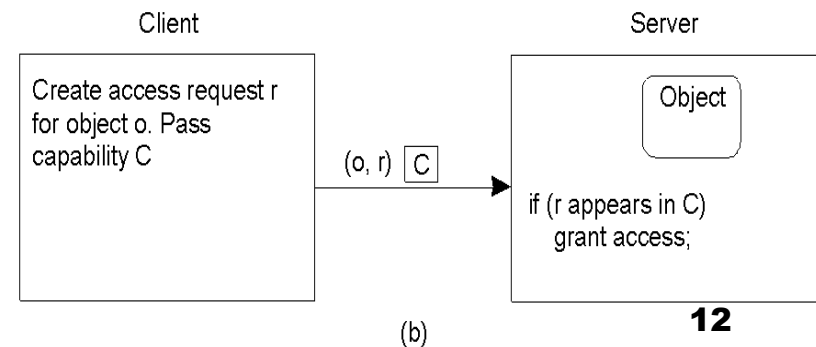
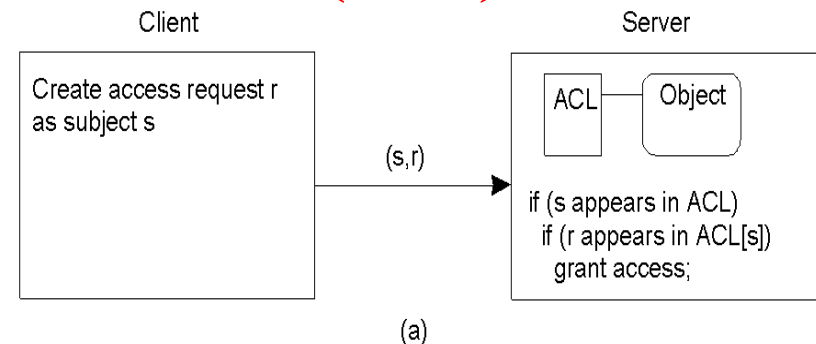
- $ACM[S,O]$ contains the permissible operations that subject S can perform on object O

(a) An object O maintains an **Access Control List (ACL)**

- The column $ACM[*,O]$ describes the permissible operations on object O by each subject (or group of subjects)

(b) A subject S has a **Capability List**

- The row $ACM[S,*]$ describes the permissible operations by subject S on each object (or category of objects)

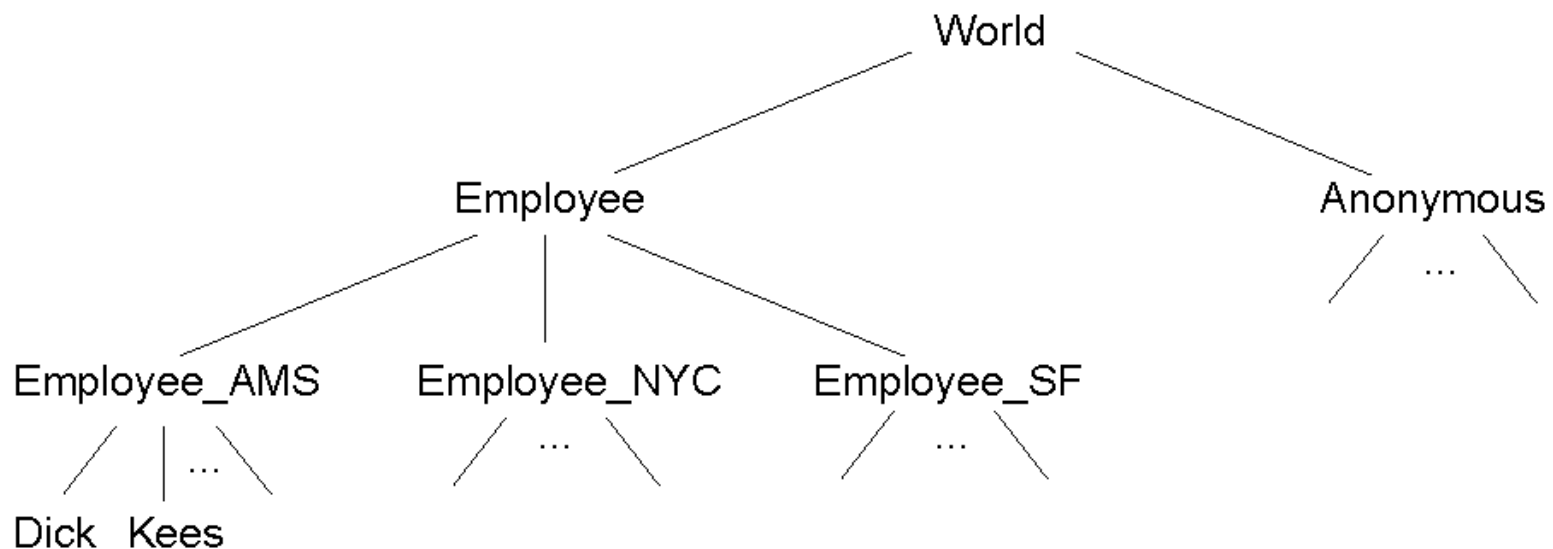


Protection Domain

- **Problem:** Access Control Lists and Capability Lists can be very large
- **Solution:** Reduce the amount of information that needs to be stored using **protection domains**
 - An **(object, access rights)** pair is associated with a protection domain
 - For each access request, first look up the appropriate protection domain
- Common implementations of protection domains
 - **Groups:** Users belong to specific groups; each group has associated access rights
 - **Roles:** Users have different roles; a user's role is determined at login time; a user's role may change

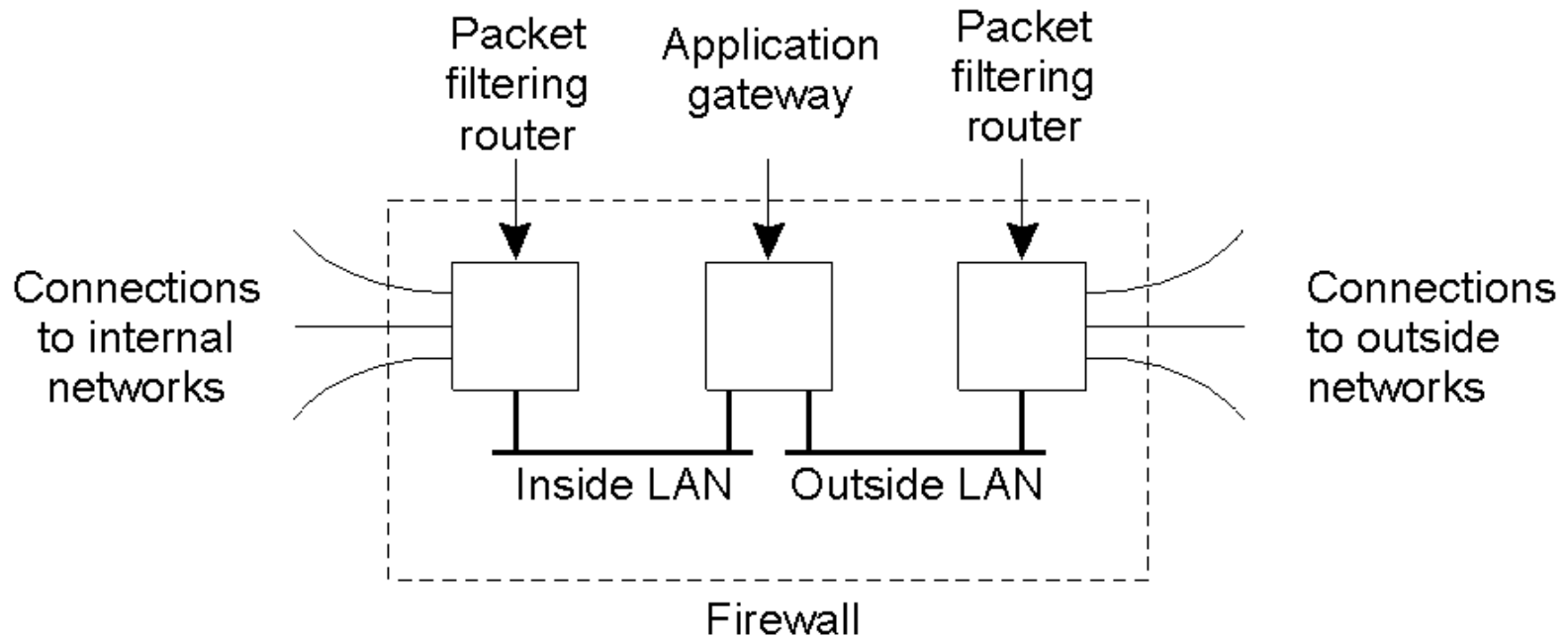
Protection Domain

- The hierarchical organization of protection domains, implemented as **groups** of users



Firewall

- Sometimes it's better to consider requests at the level of **packets**. Packets that do not satisfy certain requirements are simply removed from the channel
- A company is often protected by a **firewall** that implements **access control**



Secure Mobile Code

- **Mobile code**, e.g., Java applets or Java aglets
- If the data are very large, it makes sense to **move the code to the data**, rather than move the data to the code
 - The remote node (host) needs to be protected against malicious mobile code
 - The mobile code needs to be protected against a malicious host

Protecting the Host

- First, distinguish local code from remote code
- Assign **a set of permissions** to mobile code before it is executed and check operations against those permissions
- Assign different sets of permissions to different units of mobile code
- **Authenticate** mobile code using **digital signatures**

Protecting the Host

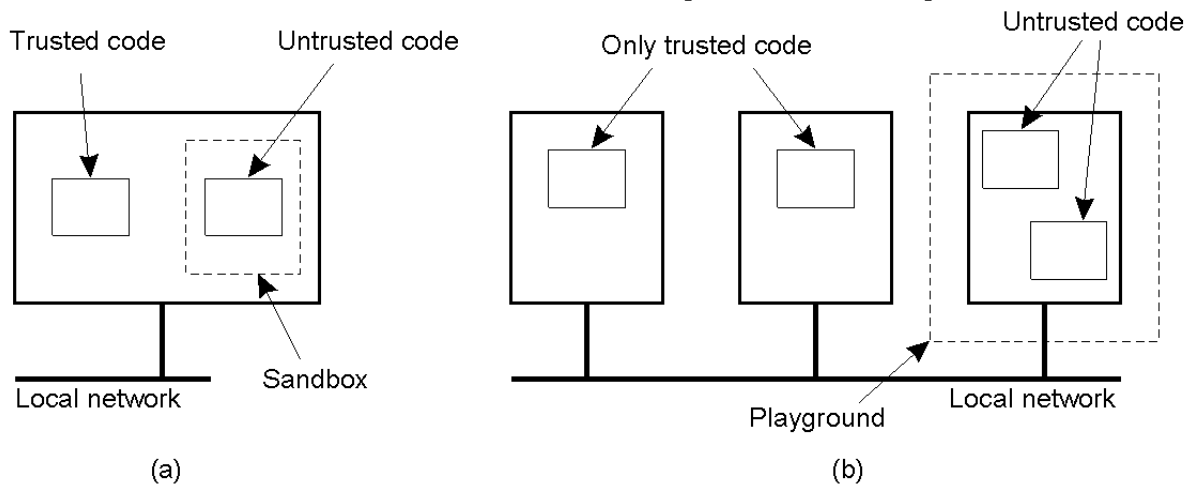
- Enforce a single strict policy, and implement the policy using a few simple mechanisms

(a) **Sandbox model**

- **Policy:** Allow remote code to access only a pre-defined set of resources and services
- **Mechanism:** Check instructions for illegal access to resources and services

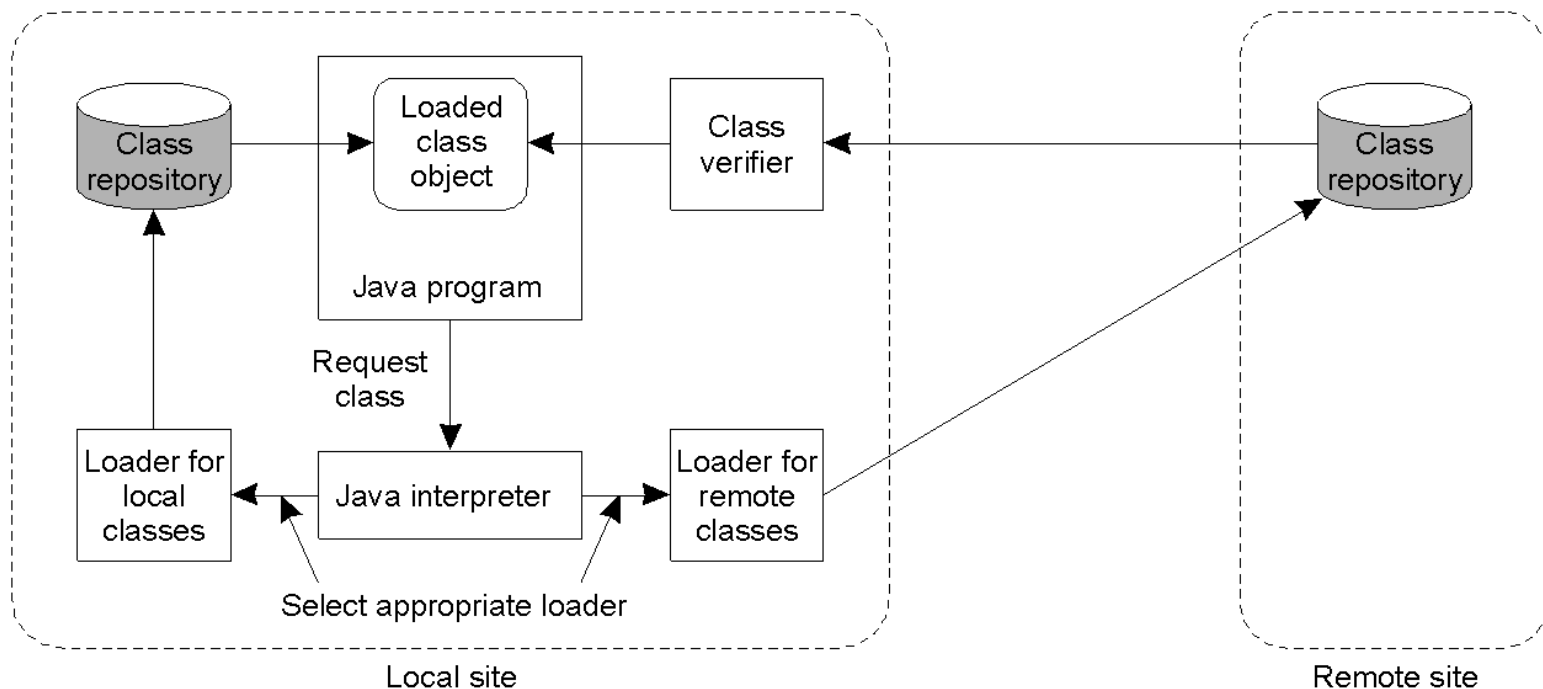
(b) **Playground model**

- **Policy:** Same
- **Mechanism:** Run code on a separate “unprotected” machine



Protecting the Host

- A **Java sandbox** for Java applets, aglets, etc.



Protecting the Mobile Code

- **Example Ajanta:** A mobile code (mobile agent) system that detects whether mobile code has been tampered with, while it is on the move
- Most important, **append-only logs**
 - Data can be only appended, not removed
 - Data has an associated **checksum**
 - Initially, the checksum has the value:
$$C_{\text{init}} = K_{\text{owner}}^+(N) \text{ with nonce } N$$
 - When server S adds data X to the log, the new checksum is calculated as follows:

$$C_{\text{new}} = K_{\text{owner}}^+(C_{\text{old}}, \text{sig}(S, X), S)$$

where $\text{sig}(S, X)$ indicates server S has signed data X

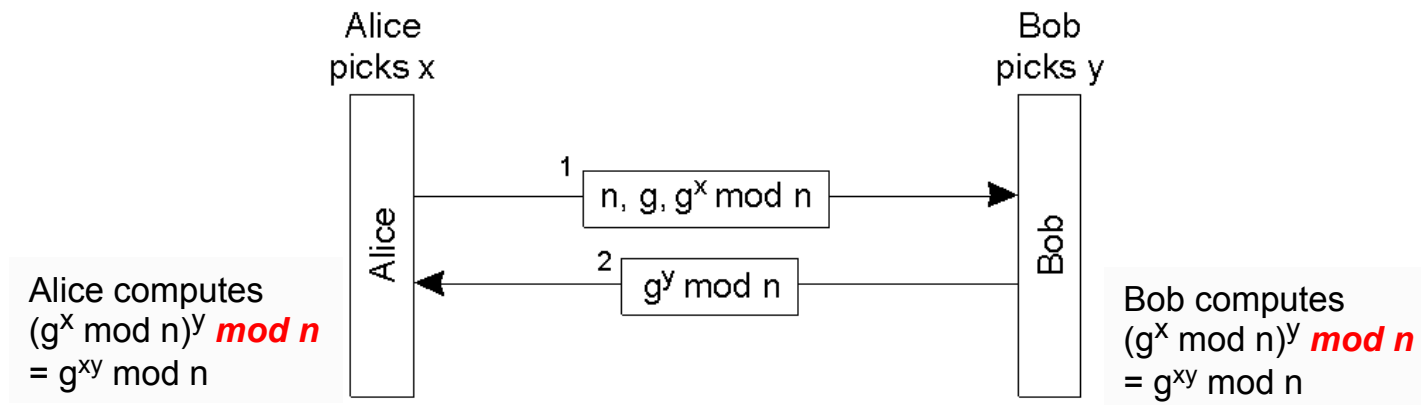


Security Management

- Key establishment
- Key distribution
- Secure group management
- Authorization management

Key Establishment

- **Diffie Hellman Key Exchange:** Construct a shared secret key $K_{A,B}$ in a safe way *without* having to trust a Key Distribution Center
 - Alice and Bob must agree on two large numbers, n and g , which are regarded as public (and which satisfy certain mathematical properties)
 - Alice chooses a large number x , which she keeps private
Bob chooses a large number y , which he keeps private



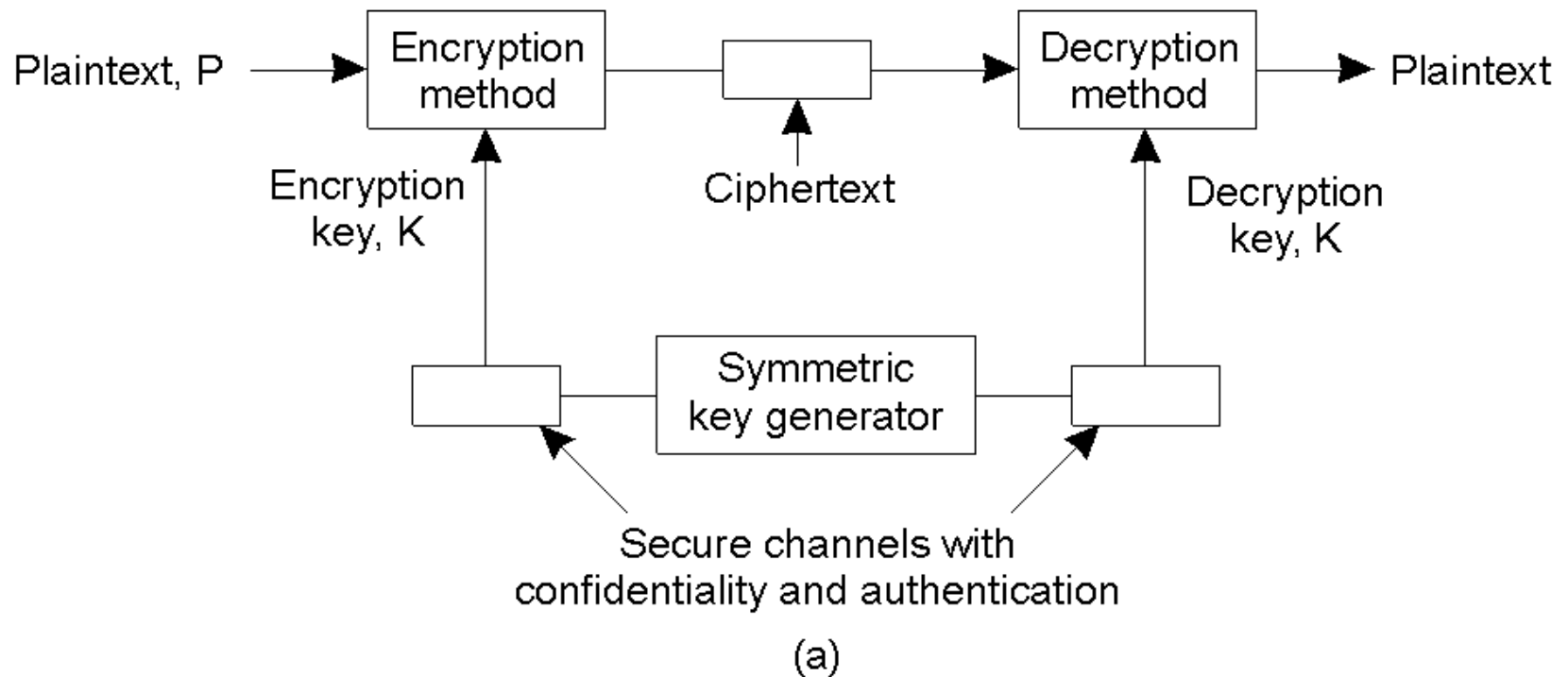
- (1) Alice sends $(n, g, g^x \bmod n)$ to Bob
- (2) Bob sends $(g^y \bmod n)$ to Alice
- (3) Alice computes $K_{A,B} = (g^y \bmod n)^x \bmod n = g^{xy} \bmod n$
- (4) Bob computes $K_{A,B} = (g^x \bmod n)^y \bmod n = g^{xy} \bmod n$

Key Distribution

- If authentication is based on cryptographic protocols and session keys are used to establish secure channels, who is responsible for handing out the keys?
 - (a) **Shared secret keys:** Alice and Bob need to get a shared secret key
 - They can **invent their own shared secret key** as on the previous slide, or
 - They can ask a trusted **Key Distribution Center (KDC)** for a shared secret key as in the previous lecture
 - (b) **Public/private keys:** Alice needs Bob's public key to decrypt (signed) messages from Bob, and Bob needs his private key to encrypt messages that he sends to Alice (and vice versa)
 - But, how does Alice know that it is indeed Bob who owns the public key
 - They can use a trusted **Certification Authority (CA)** to hand out the public/private keys, by putting them in certificates signed by the CA

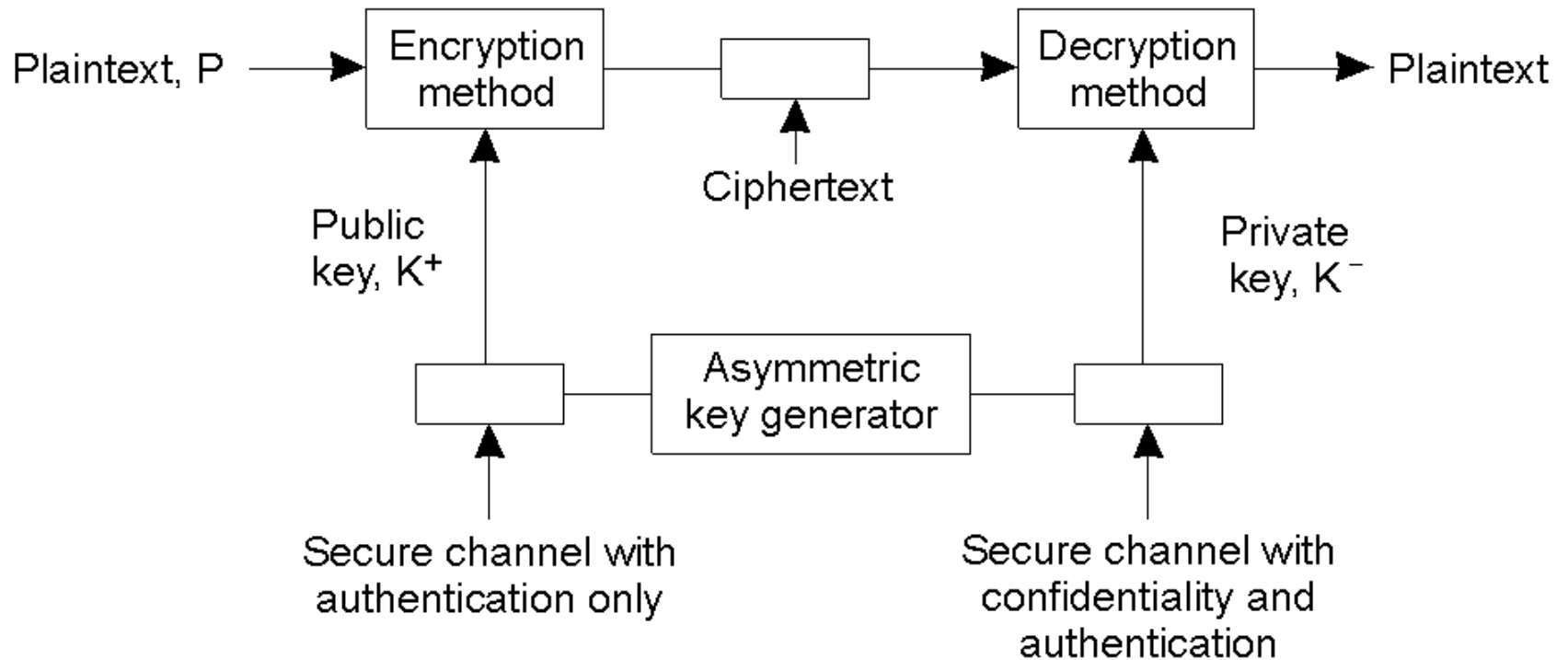
Key Distribution

(a) **Secret key distribution** using a trusted KDC



Key Distribution

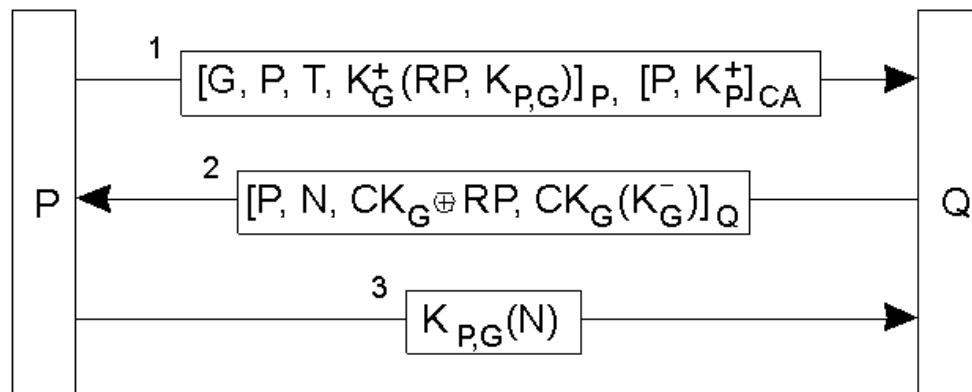
(b) **Public/private key distribution** using a trusted CA



(b)

Secure Group Management

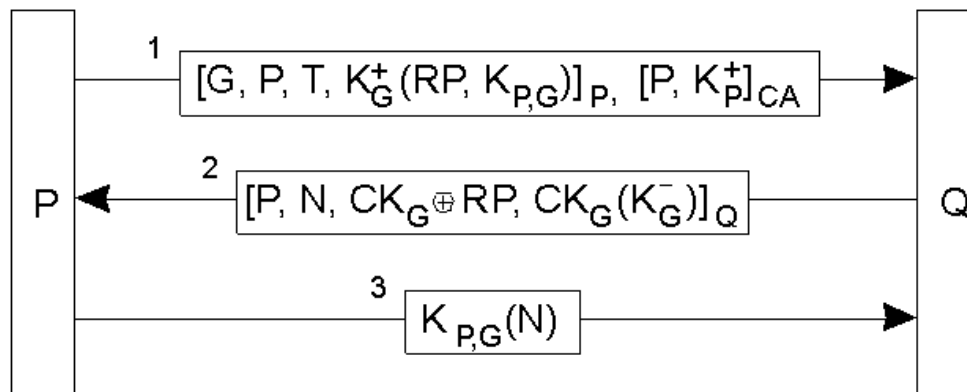
- A group G of processes using a hybrid symmetric/asymmetric scheme
 - A **shared secret key CK_G** to communicate with **group members**
 - A **key pair (K_G^+, K_G^-)** to communicate with **non-group members**
- Process P wants to **join** the group G ; process Q is a member of G
 - (1) P generates a **one-time ReplyPad RP** and a **secret key $K_{P,G}$**
 P sends a **JoinRequest** to Q signed by P (notation: $[]_P$), along with a Certificate containing its **public key K_P^+** (notation: $[]_{CA}$)



Secure Group Management (cont.)

(2) Q authenticates P and checks whether P can be allowed to join G. Q returns a **nonce N**, the **group key CK_G** XORed with RP and the **group private key K_G^-** encrypted with CK_G , signed by Q (notation: $[]_Q$)

(3) P authenticates Q and sends back the nonce N encrypted with $K_{P,G}$, letting Q know that it now has the keys



Authorization Management

- To represent the **access rights** of a user, we use **capabilities**
 - 11111111 represents full access rights, and 00000001 represents very limited access rights
 - The **capability** includes a **random number C**
 - When a restricted capability is **issued**, it is XORed with C and then it is run through a one-way function to generate a **check code**
 - When the restricted capability is **used**, the **owner** of the object verifies the check code before it authorizes access

