Distributed Systems

Security

Lecture 09

Henry Novianus Palit

hnpalit@petra.ac.id

Security Threats, Policies, Mechanisms *(1)*

- Security in a computer system is strongly related to the notion of *dependability*
  - A dependable computer system is one that we justifiably trust to deliver its services
  - Dependability includes *availability*, *reliability*, *safety*, and *maintainability*; to put our trust in a computer system, *confidentiality* and *integrity* should be taken into account
- Confidentiality: the property of a computer system whereby its information is disclosed only to authorized parties
- Integrity: the characteristic that alterations to system's assets can be made only in an authorized way

Security Threats, Policies, Mechanisms *(2)*

- Another way of looking at security is that we attempt to <u>protect the service and data it offers against **security threats**</u>:
  - Unauthorized information disclosure (*confidentiality*)
  - Unauthorized information modification (*integrity*)
  - Unauthorized denial of use (*availability*)
- A <u>**security policy** describes</u> precisely <u>which actions the entities in a system</u> (i.e., users, services, data, machines, etc.) <u>are allowed to take and which ones are prohibited</u>

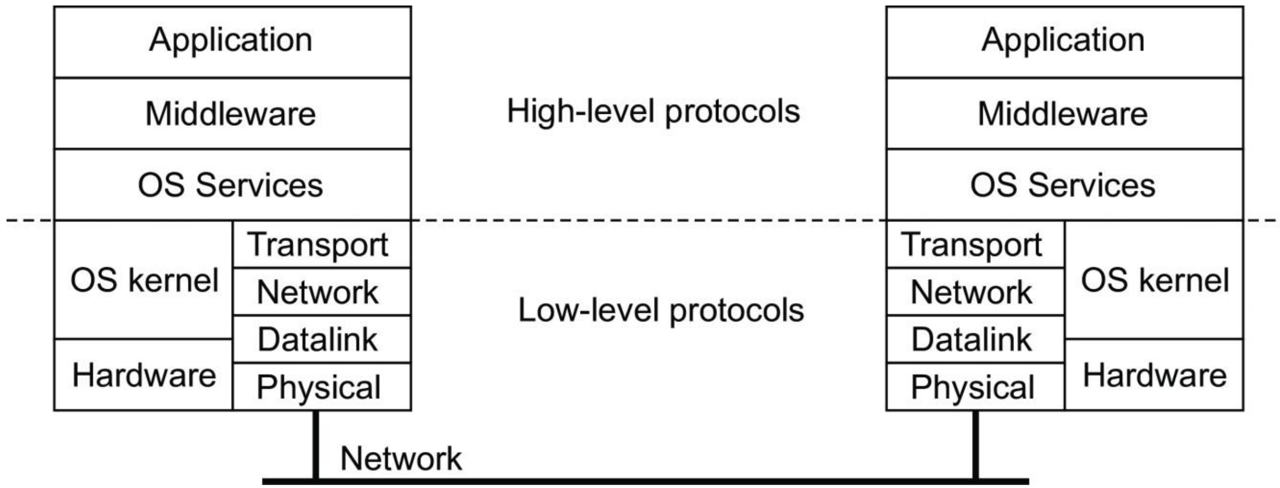
Security Threats, Policies, Mechanisms *(3)*

- Afterwards, we can concentrate on the **security mechanisms** by which a policy can be enforced:
  - *Encryption*: transform data to something an attacker cannot understand, or that can be checked for modifications
  - *Authentication*: verify a claimed identity
  - *Authorization*: check an authenticated entity whether it has the proper rights to access resources
  - *Monitoring and auditing*: (continuously) trace access to resources

Security Principles

- *Fail-safe defaults*: defaults should already <u>provide good protection</u>
  - Infamous example: the default “(admin,admin)” for edge devices
- *Open design*: <u>do not apply security by obscurity</u> $\rightarrow$ every aspect of a distributed system is <u>open for review</u>
- *Separation of privilege*: ensure that <u>critical aspects</u> of a system <u>can never be fully controlled by just a single entity</u>
- *Least privilege*: a process should <u>operate with the fewest possible privileges</u>
- *Least common mechanism*: if multiple components require the same mechanism, then they should <u>all be offered the same implementation of that mechanism</u>

Layering of Security Mechanisms



- Common security solutions: network (VPN), transport (TLS), OS service (SSH), middleware (Kerberos)
- We are increasingly seeing end-to-end security, meaning that mechanisms are implemented at the level of applications
- **Trusted Computing Base**: The set of all security mechanisms in a (distributed) computer system that are necessary and sufficient to enforce a security policy

Privacy

- *Privacy* and *confidentiality* are closely related, yet are different
  - Privacy can be invaded, whereas confidentiality can be breached → ensuring confidentiality is not enough to guarantee privacy
- The right to privacy is about "a right to *appropriate* flow of personal information"
  - Control who gets to see what, when, and how → a person should be able to stop and revoke a flow of personal information
- General Data Protection Regulation (GDPR) is a comprehensive set of regulations aiming to protect personal data

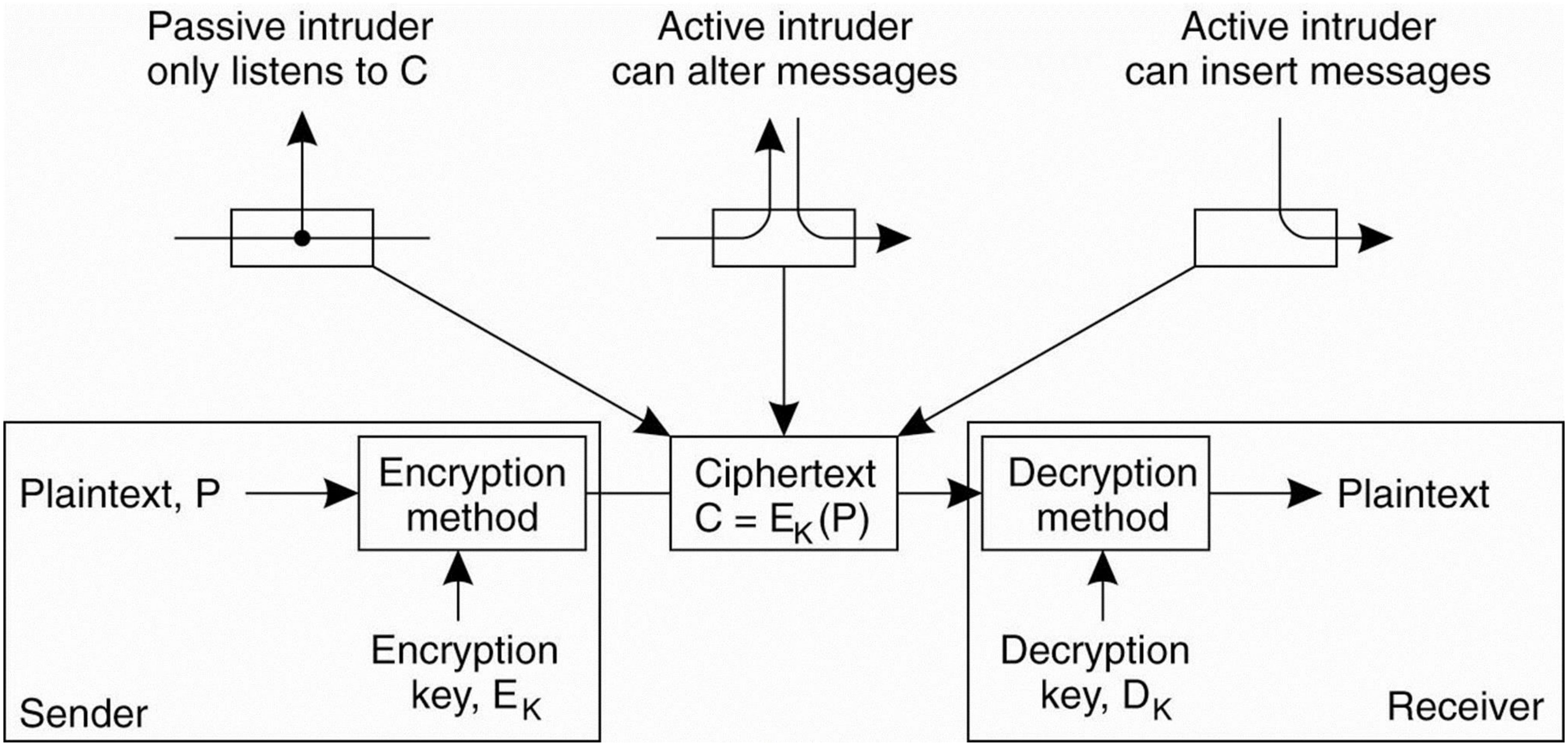
GDPR: Database Perspective

GDPR regulation	Impact on database systems	
	Attributes	**Actions**
Collect data for explicit purposes	Purpose	Metadata indexing
Do not store data indefinitely	TTL	Timely deletion
Inform customers about GDPR metadata associated with their data	Purpose, TTL, Origin, Sharing	Metadata indexing
Allow customers to access their data	Person id	Metadata indexing
Allow customers to erase their data	TTL	Timely deletion
Do not use data for objected reasons	Objections	Metadata indexing
Allow customers to withdraw from algorithmic decision-making	Automated decisions	Metadata indexing
Safeguard and restrict access to data		Access control
Do not grant unlimited access to data		Access control
Audit operations on personal data	Audit trail	Monitor and log
Implement appropriate data security		Encryption
Share audit trails from affected systems	Audit trail	Monitor and log

Cryptography *(1)*

- The use of cryptographic techniques is fundamental to security in distributed systems
- Consider a sender *S* wanting to transmit message *m* to a receiver *R*
  - The sender *S* first **encrypts** it into an unintelligible message *m'* and subsequently sends it to *R*
  - The receiver *R*, in turn, must **decrypt** the received message into its original form *m*
- Encryption and decryption are accomplished by using cryptographic methods parameterized by keys
  - The original form of the message is called the **plaintext** (P)
  - The encrypted form is referred to as the **ciphertext** (C)

Cryptography *(2)*



Intruders and eavesdroppers in communication

Cryptography *(3)*

- Encryption and decryption are … (cont'd)
  - $C = E_K(P) \rightarrow$ ciphertext C is obtained by <u>encrypting the plaintext</u> P <u>using key</u> E_K
  - $P = D_K(C) \rightarrow$ <u>decryption of the ciphertext</u> C <u>using key</u> D_K, resulting in the plaintext P
- There are <u>three different attacks that we need to protect against</u> (for which encryption helps)
  - An intruder may <u>intercept (eavesdrop) the message</u> without either the sender or receiver being aware
  - An intruder may <u>modify the message</u>
  - An intruder may <u>insert encrypted messages</u> into the communication system, attempting to make *R* believe these messages came from *S*

Cryptography *(4)*

- If the transmitted message has been encrypted in such a way that it cannot be easily decrypted without having the proper key, <u>interception is useless: the intruder will see only unintelligible data</u>
- <u>Modifying ciphertext</u> that has been properly encrypted is much more difficult (than modifying plaintext) because <u>the intruder will first have to decrypt the message before he can meaningfully modify it</u>; he will also have to properly encrypt it again, or else, the receiver may notice that the message has been tampered with
- Encryption can also <u>protect against inserting messages</u>

Cryptography *(5)*

- Different cryptographic systems:
  - **Symmetric cryptosystem**: the same key is used to encrypt and decrypt a message

    $$P = D_K\left(E_K(P)\right) \text{ and } \mathrm{D_K} = \mathrm{E_K}$$

    - Also referred to as secret-key or shared-key systems
  - **Asymmetric cryptosystem**: the keys for encryption and decryption are different, but together form a unique pair

    $$P = D_K\left(E_K(P)\right) \text{ and } \mathrm{D_K} \neq \mathrm{E_K}$$

    - One of the keys is kept private, the other is made public
    - Also referred to as public-key systems
  - **Homomorphic encryption**: Mathematical operations on plaintext can be performed on the corresponding ciphertext; if *x* and *y* are two numbers, then

    $$E_K(x) \star E_K(y) = E_K(x \star y)$$

Hash Functions *(1)*

- One final application of cryptography in distributed systems is the use of **hash functions**
  - A hash function *H* takes a message m of arbitrary length as input and produces a bit string h having a fixed length as output $\mathrm{h} = H(\mathrm{m})$ with length of h fixed
  - A hash h is somewhat comparable to the extra bits that are appended to a message in communication systems to allow for error detection, such as cyclic-redundancy check (CRC)
- Essential properties of hash functions:
  - **One-way functions**: it is computationally infeasible to find the input m that corresponds to a known output h
  - **Weak collision resistance**: it is computationally infeasible to find another different input m′ ≠ m, such that *H*(m) = *H*(m′)

Hash Functions *(2)*

- Essential properties of hash functions: (cont'd)
  - **Strong collision resistance**: it is computationally infeasible to find any two different input values m and m', such that $H(\text{m}) = H(\text{m}')$
- Similar properties must apply to any encryption function *E* and the keys that are used
  - For any encryption function E_K, it should be computationally infeasible to find the key E_K when given the plaintext P and associated ciphertext $\mathrm{C} = E_K(\mathrm{P})$
  - Given a plaintext P and a key E_K, it should be effectively impossible to find another key $\mathrm{E}_{\mathrm{K}'}$ such that $E_K(\mathrm{P}) = E_{K'}(\mathrm{P})$

Hash Functions *(3)*

- Example: **digital signature**
  - Alice computes a digest from m; encrypts the digest with her private key; encrypted digest is sent along with m to Bob:

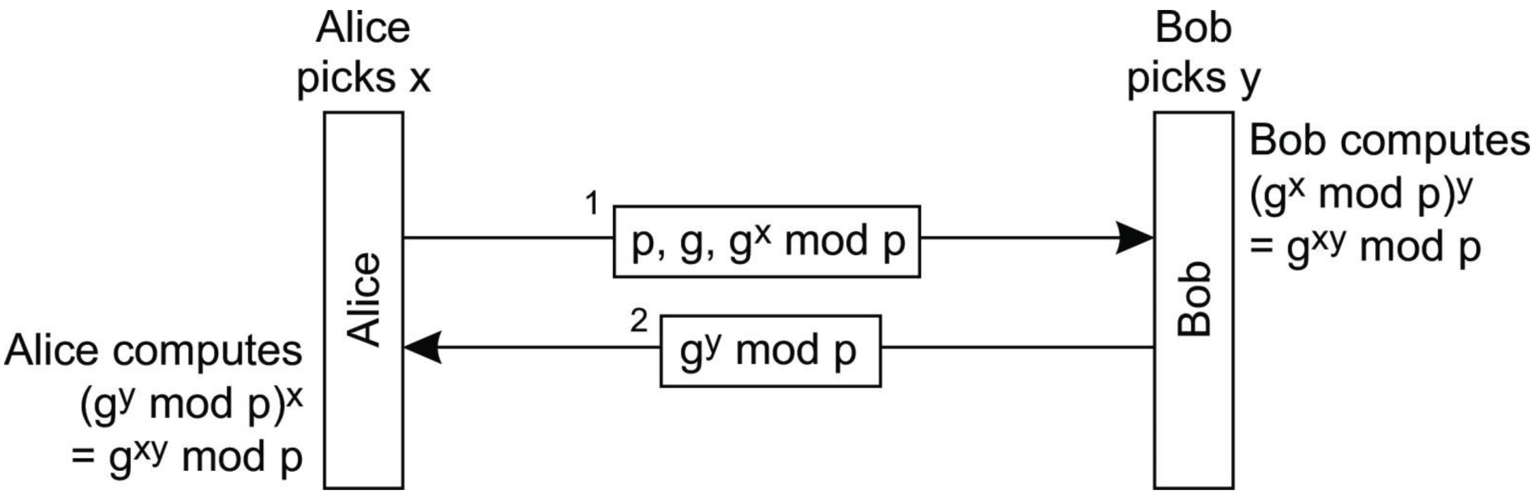
    Alice: *send*[m, sig] with sig = $SK_A(H(\text{m}))$

  - Bob decrypts digest with Alice's public key; separately calculates the message digest; if both match, Bob knows the message has been signed by Alice:

    Bob: *receive*[m, sig], compute h' = H(m) and verify h' = $PK_A(\text{sig})$

Key Management *(1)*

- How do Alice and Bob get the correct (often shared) keys so that they can set up secure channels?
- Diffie-Hellman key exchange
  - Assume two large, nonsecret numbers *p* and *g* (with specific mathematical properties)
  - Large numbers *x* and *y* are kept secret

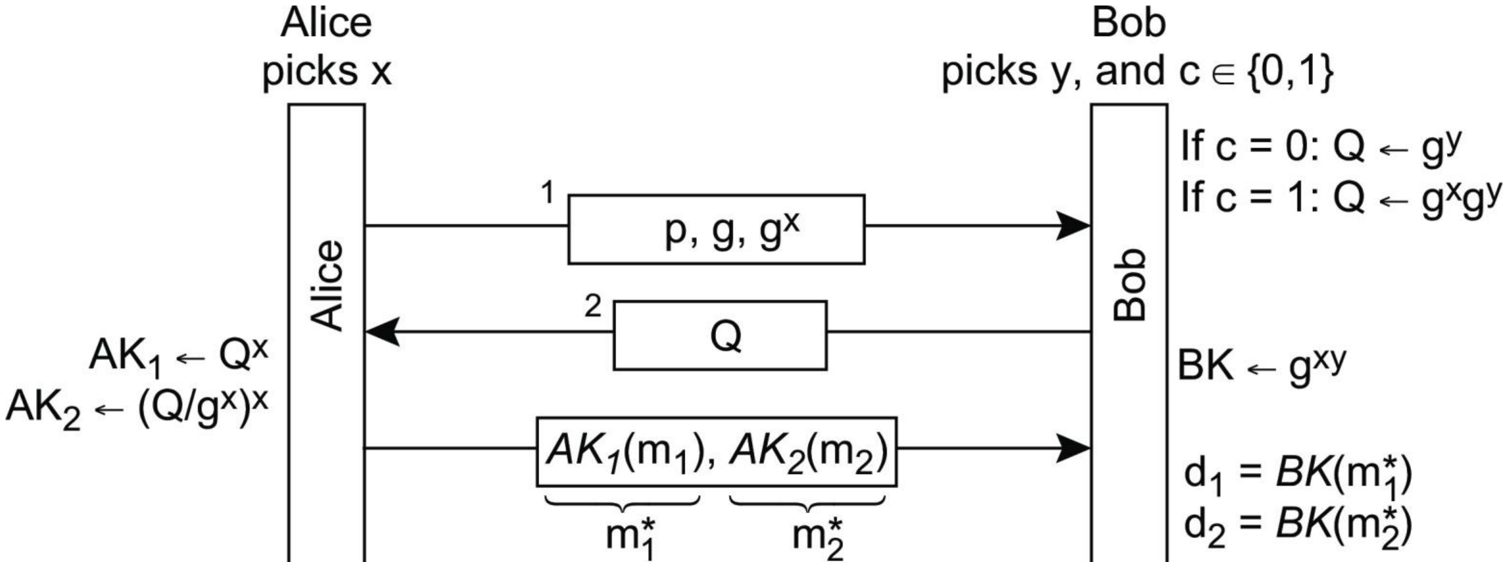


Key Management *(2)*

- DH key exchange example: <u>multiparty computation</u>
  - Can we protect private data while computing statistics? Who has the highest salary without revealing salaries? Can we compute the number of votes cast for a specific candidate without revealing who voted for whom?
  - Alice has <u>*n* secret messages</u> $m_1, ..., m_n$. Bob is <u>interested (and allowed) to know only message</u> m_i. Which message he wants to know should be kept secret to Alice; all messages $m_j \neq m_i$ should be kept secret to Bob.
  - <u>Solution</u>: Bob generates a number Q that Alice, in turn, uses to generate *n* different encryption keys $PK_1, ..., PK_n$ to get $m_i^* = PK_i(m_i)$. Bob uses Q to generate a decryption key SK_i that matches only PK_i. When Bob receives $m_1^*, ..., m_n^*$ he can decrypt only m_i^*. Doing $SK_i(m_j^*)$, with $i \neq j$, will fail.

Key Management *(3)*

- DH key exchange example: MPC (cont'd)
  - 1-out-of-2 oblivious transfer (all operations are modulo p)

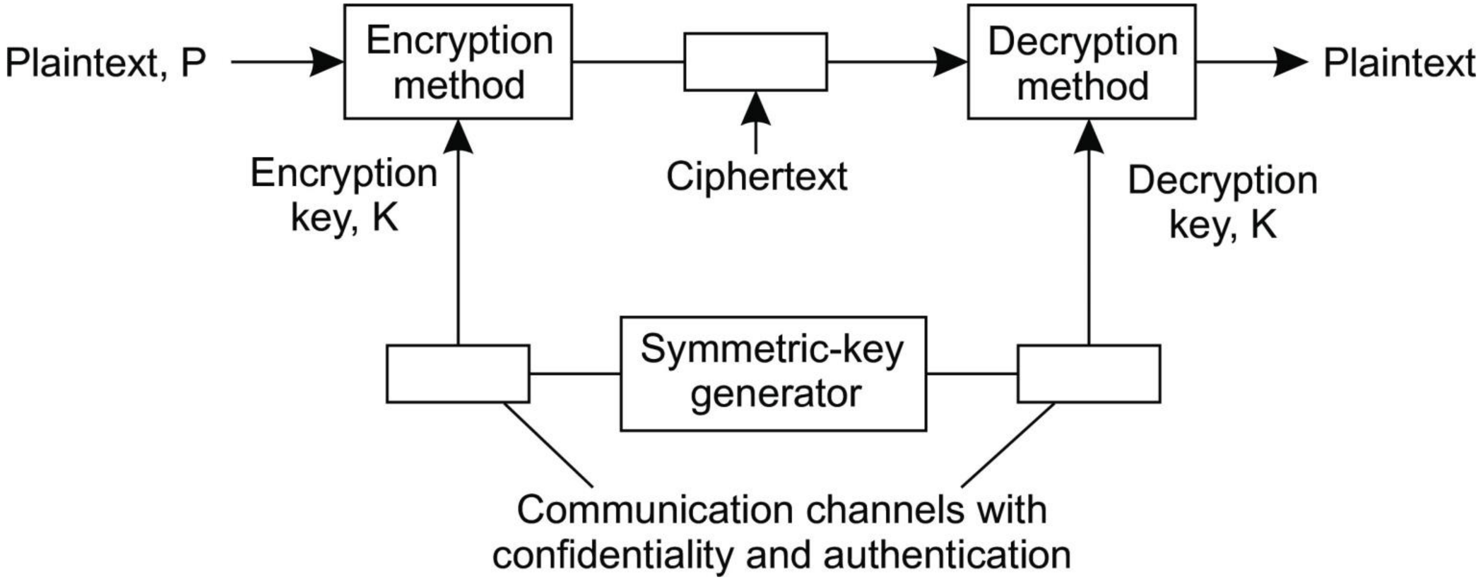


Analysis:

- $c = 0 \Rightarrow Q = g^{y}$, $AK_1 = BK = g^{xy}$, $AK_2 = g^{xy-x^2}$
- $c = 1 \Rightarrow Q = g^{x+y}$, $AK_1 = g^{x^2+xy}$, $AK_2 = BK = g^{xy}$

Key Management *(4)*

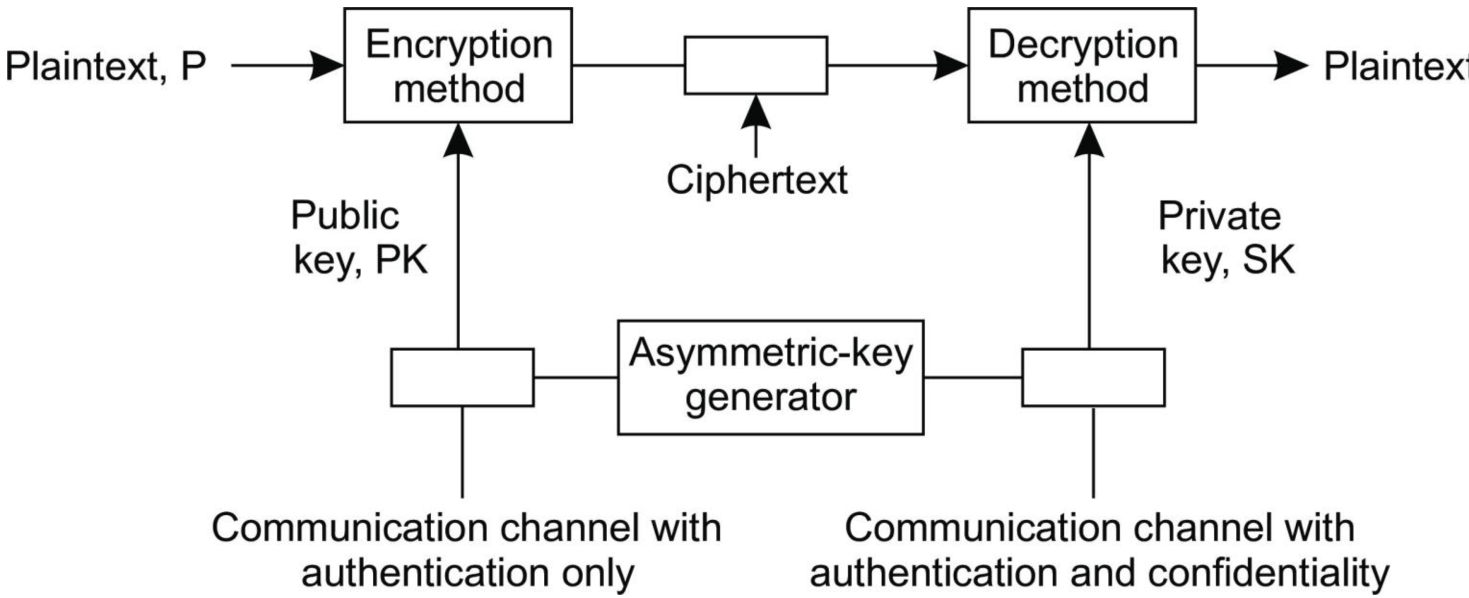
- <u>Symmetric-key distribution</u>



  - In general, we will <u>need a secure channel</u> to distribute the secret key to the communicating parties

Key Management *(5)*

- <u>Public-key distribution</u>



- No need for a secure channel, but you do <u>need to know that the key is authentic</u> ⇒ have the public key be <u>signed by a certification authority</u>
  - Note: we do need to <u>trust that authority</u>, or otherwise make sure that its signature can be verified as well

Authentication *(1)*

- Essence: Verifying the claimed identity of a person, a software component, a device, and so on
- Means of authentication:
  - Based on what a client knows, such as a *password* or a *personal identification number*
  - Based on what a client has, such as an *ID card*, *cell phone*, or *software token*
  - Based on what a client is, i.e., static biometrics such as a *fingerprint* or *facial characteristics*
  - Based on what a client does, i.e., dynamic biometrics such as *voice patterns* or *typing patterns*

Authentication *(2)*

- Authentication and message integrity cannot do without each other
  - Consider a system that supports authentication but no mechanisms to ensure message integrity. Bob may know for sure that Alice sent m, but how useful is that if he doesn't know that m may have been modified?
  - Consider a system that guarantees message integrity, but does not provide authentication. Can Bob be happy with a guaranteed unmodified message that states he just won $1,000,000?
- To ensure integrity of the exchanged data messages after authentication has taken place, it is common practice to use secret-key cryptography by means of session keys

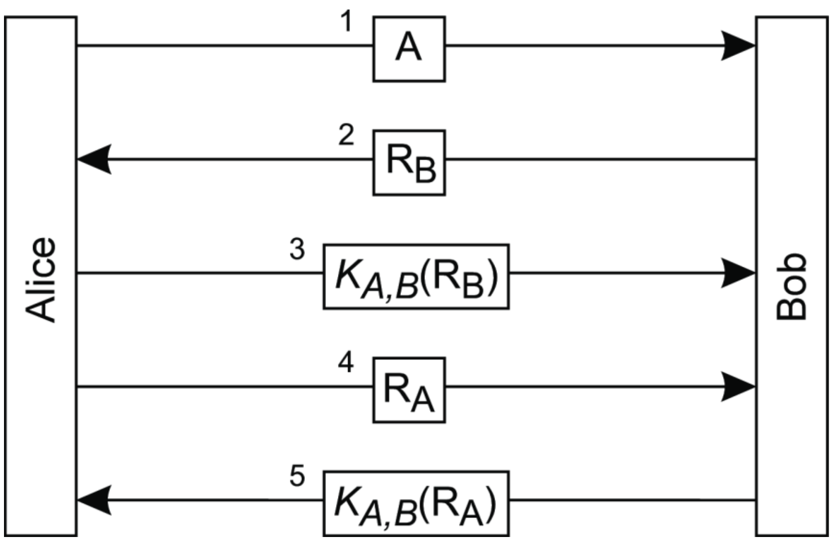
Authentication *(3)*

- A **session key** is a shared (secret) key that is used to encrypt messages for integrity and possibly also confidentiality
  - Such a key is generally used only for as long as the channel exists

Authentication based on a shared secret key

- Alice and Bob are abbreviated by A and B, respectively, and their shared key is denoted as $K_{A,B}$
- The protocol takes a common approach whereby one party challenges the other to a response that can be correct only if the other knows the shared secret key
  - Such solutions are also known as **challenge-response protocols**

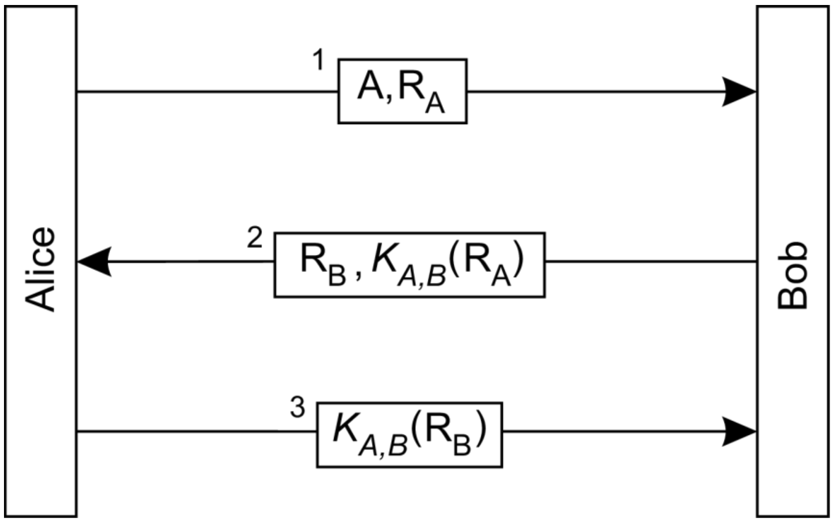
Authentication *(4)*



1. Alice sends her identity
2. Bob sends a challenge (note: could take the form of a random number / **nonce**)
3. Alice encrypts the challenge with the secret key and returns the encrypted challenge
4. Alice sends a challenge
5. Bob responds to by returning the encrypted challenge

Decrypting the message at step 3, <u>Bob can verify that he is indeed talking to Alice</u> (who else could have encrypted the challenge with the secret key). However, <u>Alice has not verified Bob</u>. Therefore, <u>steps 4 and 5 are required</u>.

Authentication *(5)*

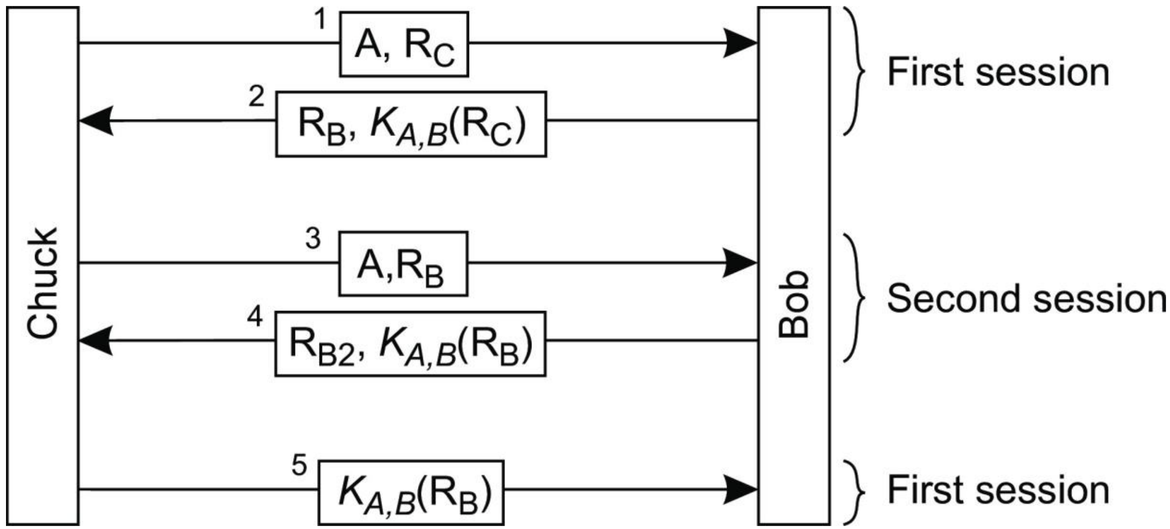


Optimization: <u>three steps, instead of five</u>

1. Alice sends a challenge along with her identity
2. Bob returns his response to that challenge, along with his own challenge
3. Alice returns her response to Bob's challenge

Unfortunately, this protocol **no longer works**. It <u>can easily be defeated by</u> what is known as <u>a **reflection attack**</u> (explained in the next slide).

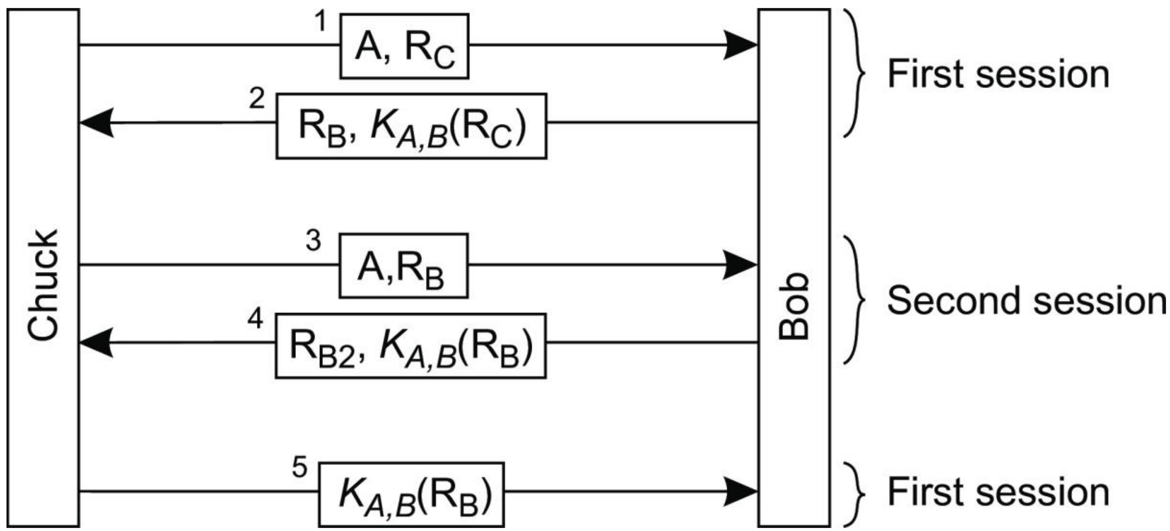
Authentication *(6)*



Consider an intruder called Chuck, denoted as C in our protocols:

1. Chuck starts out by sending a message containing Alice's identity A, along with a challenge R_C
2. Bob returns his challenge R_B and the response to R_C in a single message; at that point, Chuck would need to prove he knows the secret key

Authentication *(7)*



Consider an intruder called Chuck, denoted as C in our protocols:

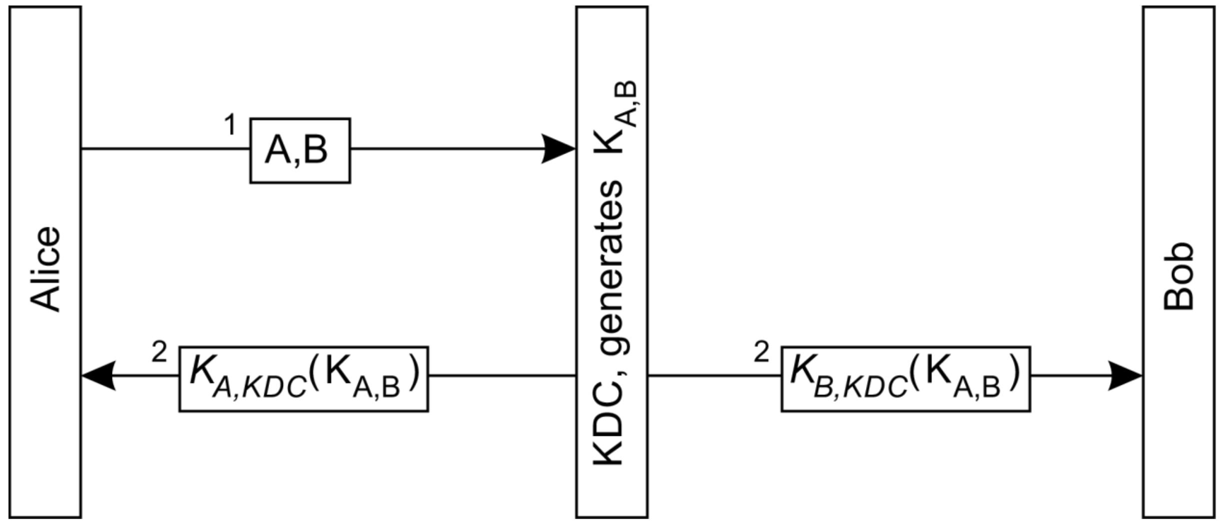
3. Chuck <u>sets up a second session</u> to let Bob encrypt for him
4. Bob <u>returns the response to his own challenge</u>
5. Chuck finishes setting up the first session by returning the response received from Bob

Authentication *(8)*

Authentication using a key distribution center

- One of the problems with using a shared secret key for authentication is scalability
  - If a distributed system contains N hosts, and each is required to share a secret key with each of the other $N - 1$ hosts, the system as a whole needs to manage $N(N - 1)/2$ keys
- An alternative is to use a centralized approach by means of a **Key Distribution Center** (KDC)
  - KDC shares a secret key with each of the hosts, but no pair of hosts is required to have a shared secret key as well
  - Using a KDC, the system requires to manage N keys only

Authentication *(9)*

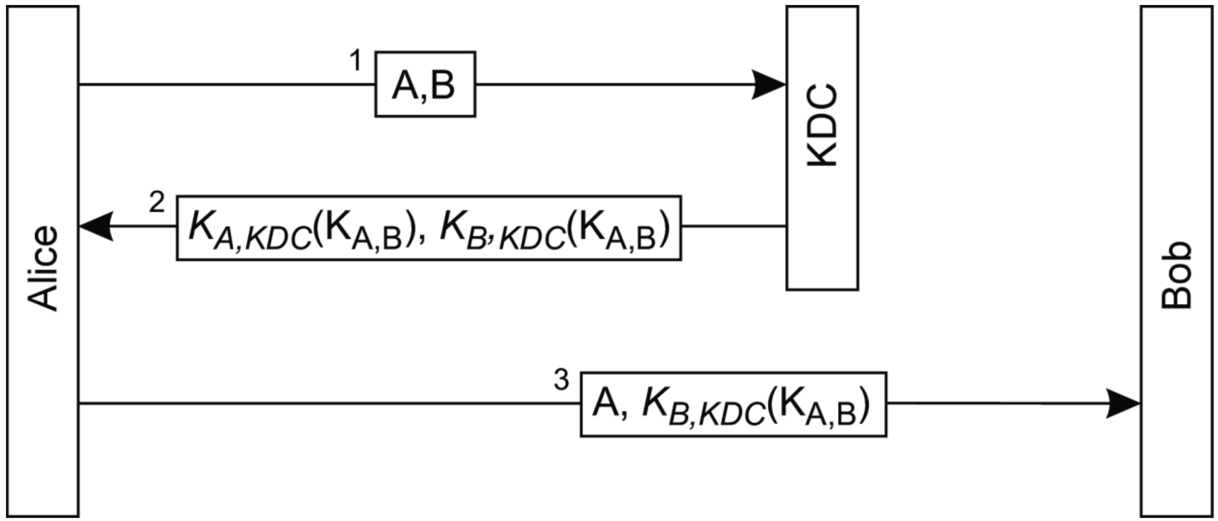


- If Alice wants to <u>set up a secure channel</u> with Bob, she can do so <u>with the help of a (trusted) KDC</u>
  1. Alice sends a message to KDC stating that she wants to talk to Bob
  2. KDC returns a message containing a shared secret key $K_{A,B}$ to Alice and Bob, encrypted with the secret key that KDC shares with Alice and Bob, respectively

Authentication *(10)*

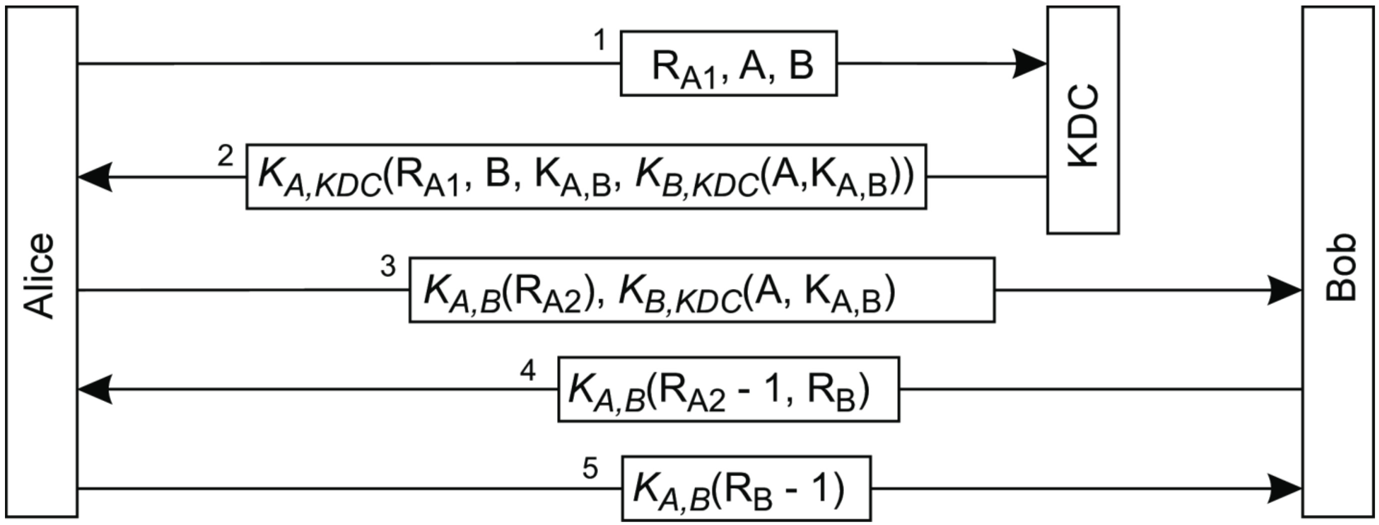
- The main drawback is that
  - Alice may start setting up a secure channel with Bob even before Bob had received the shared key from KDC
  - KDC is required to get Bob into the loop by passing him the key
- The problems can be circumvented if KDC just passes $K_{B,KDC}(K_{A,B})$ back to Alice, and lets her take care of connecting to Bob
  - The message $K_{B,KDC}(K_{A,B})$ is also known as a **ticket**
  - It is Alice's job to pass this ticket to Bob
  - Bob is still the only one who can make sensible use of the ticket, as he is the only one (besides KDC) who knows how to decrypt the information it contains

Authentication *(11)*



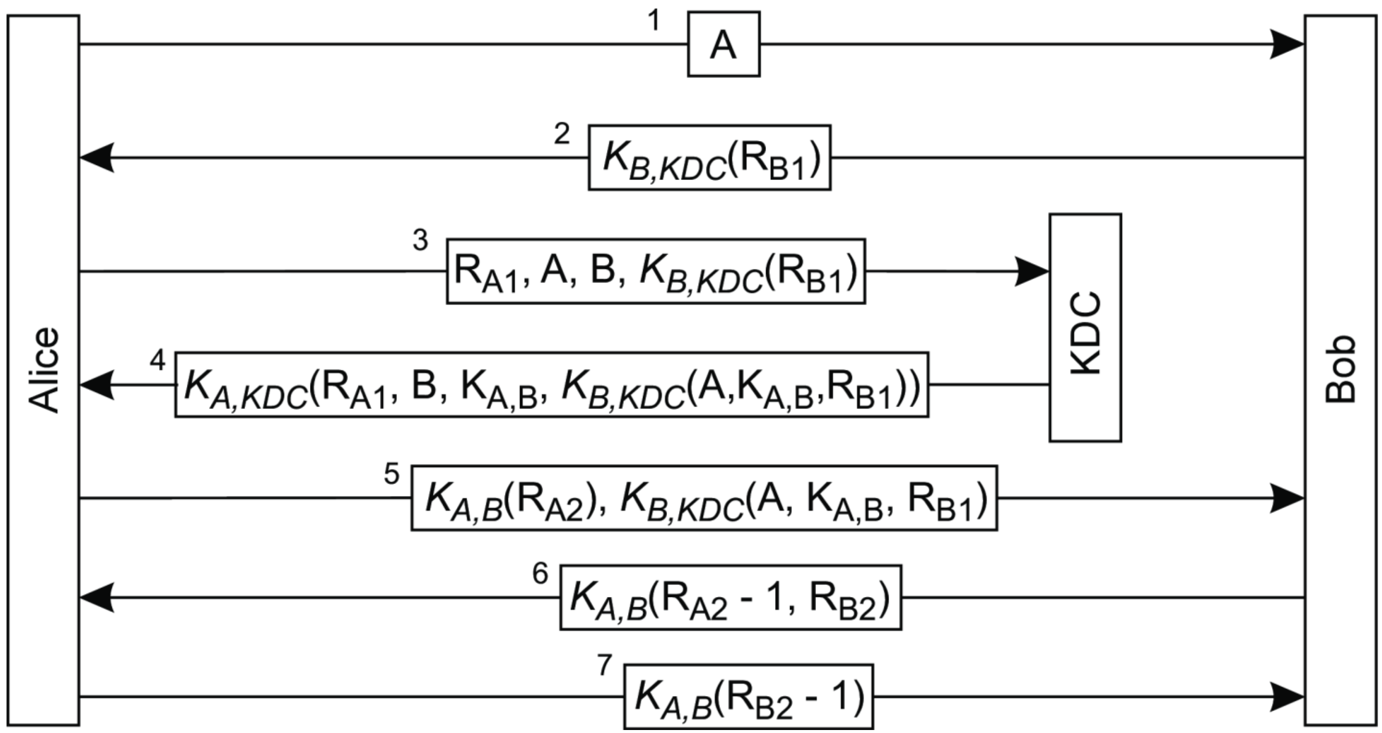
- The protocol shown above is actually <u>a variant of</u> a well-known example of an authentication protocol using a KDC, known as the **<u>Needham-Schroeder authentication protocol</u>** or **<u>multiway challenge-response protocol</u>** (presented in the next slide)

Authentication *(12)*



- The Needham-Schroeder protocol <u>still has the weak point</u> that if Chuck ever got a hold of an old key, he could <u>replay message 3</u> and get Bob to set up a channel
- The solution is <u>to incorporate a nonce that has to come from Bob</u> (presented in the next slide)

Authentication *(13)*

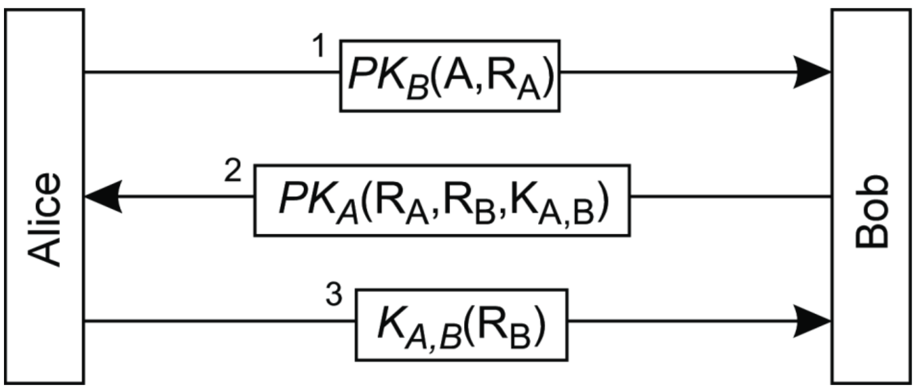


Protection against malicious reuse of a previously generated session key in the Needham Schroeder protocol

Authentication *(14)*

Authentication using public-key cryptography

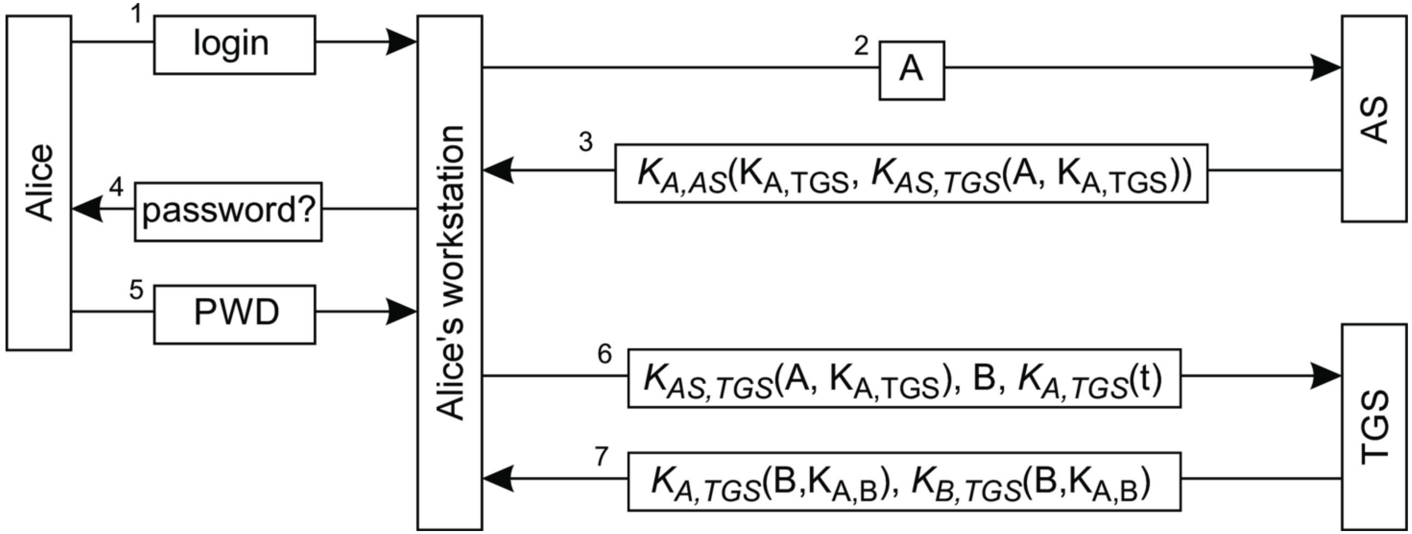
- A typical authentication protocol based on public-key cryptography is shown below



1. Alice sends a challenge R_A to Bob with his public key
2. Bob returns the decrypted challenge, his own challenge R_B, and a generated session key – encrypted with her public key
3. Alice returns the decrypted challenge – encrypted with the session key generated by Bob

Authentication *(15)*

Example: Kerberos



1,2 Alice types in her login name.

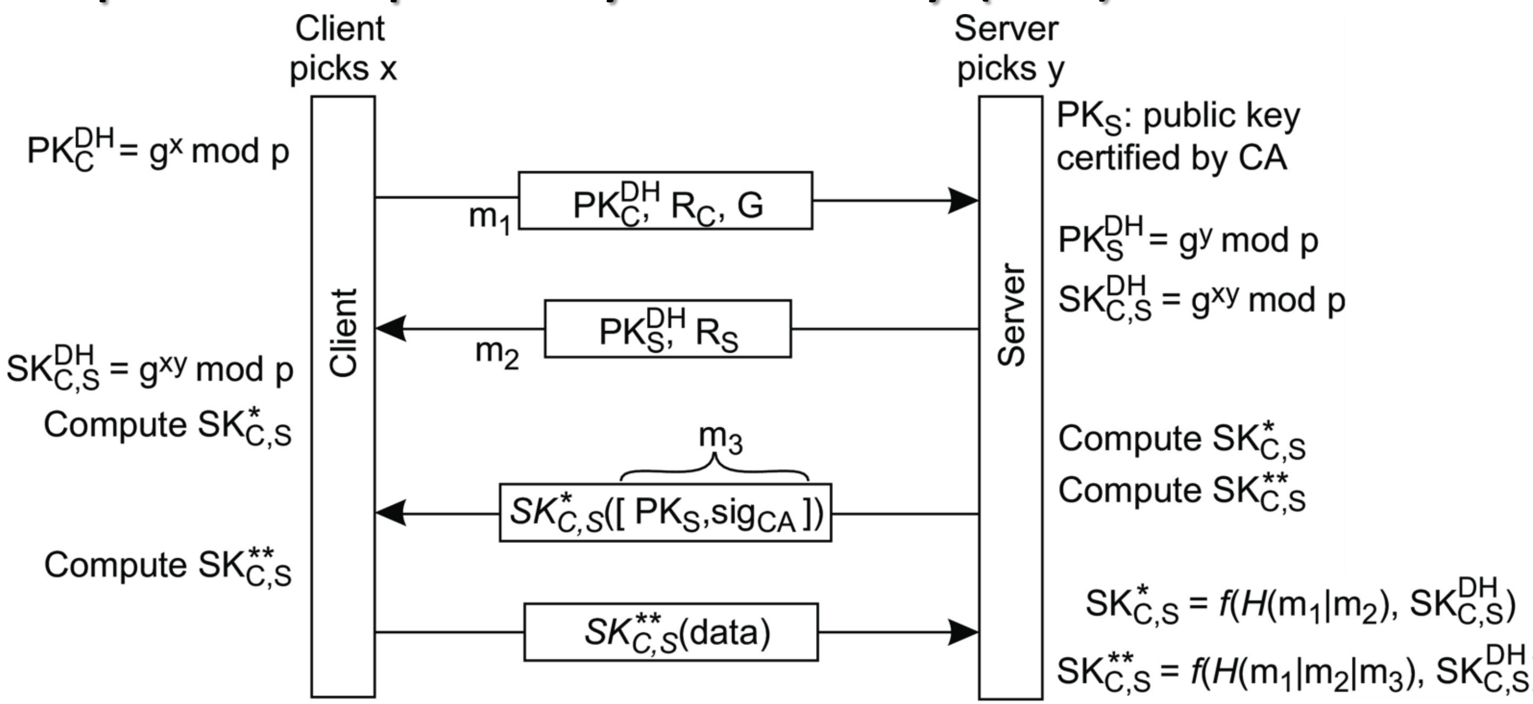
3 The **Authentication Service** (AS) returns a ticket $K_{AS,TGS}(A, K_{A,TGS})$ that she can use with the **Ticket Granting Service** (TGS).

4,5 To be able to decrypt the message, Alice must type in her password. She is then logged in. Using the AS in this way, we have a **single sign-on** system.

6,7 Alice wants to talk to Bob, and requests the TGS for a session key.

Authentication *(16)*

Example: Transport Layer Security (TLS)



- G denotes a specific set of parameter settings, called a **group** (e.g., values for p and g)
- The client uses a **nonce** R_C ; the server uses R_S
- $H(m_1|m_2)$ denotes the hash over the concatenation of m_1 and m_2

Trust *(1)*

- Definition: Trust is the assurance that <u>one entity holds that another will perform particular actions</u> according to a specific expectation
- <u>Important observation</u>:
  - Expectations have been made explicit $\rightarrow$ no need to talk about trust?
  - Example: Consider a Byzantine fault-tolerant process group of size *n*
    - Specification: the group can tolerate that at most $k \leq (n - 1)/3$ processes go rogue
    - Realization: for example PBFT (Practical Byzantine FT)
    - Consequence: if more than *k* processes fail, all bets are simply off
    - Consequence: it's not about trust, it's all about meeting specifications
  - Observation: if a process group often does not meet its specifications, one may start to doubt its reliability, but this is something else than (dis)trusting the system

Trust *(2)*

Sybil Attack

- Essence: Create multiple identities, but owned by one entity
- In the case of a P2P network:

```
H = set of honest nodes
S = set of Sybil nodes
A = Attacker node
d = minimal fraction of Sybil nodes needed for an attack

while True:
    s = A.createNode()          # create a Sybil node
    S.add(s)                    # add it to the set S

    h = random.choice(H)        # pick an arbitrary honest node
    s.connectTo(h)              # connect the new sybil node to h

    if len(S) / len(H) > d:     # enough sybil nodes for...
        A.attack()              # ...an attack
```

Trust *(3)*

- In the case of a Web-of-trust:
  - Endorse a public key without an out-of-band check
  - Bob checks with $k > 1$ others that they have endorsed Alice's key
  - Alice creates $k > 1$ identities each stating her key is valid

Eclipse Attack

- Essence: Try to isolate a node from the network
- Example: A hub attack in the case of a gossip-based service. In this case, when exchanging links to other peers, a colluding node returns links only to other colluders.
- Affected node: has links only to colluders
- General solution: use a centralized CA

Trust *(4)*

Preventing Sybil Attacks: blockchain solutions

- Essence: Creating an identity comes at a cost
- In the case of permissionless blockchains:
  - **Proof-of-Work**: Let validators run a computational race. This approach requires considerable computational resources.
  - **Proof-of-Stake**: Pick a validator as a function of the number of tokens it owns. This approach requires risking loss of tokens.

Preventing Sybil Attacks: decentralized accounting

- Simple example:
  - Each node P maintains a list of nodes interested in doing work for P: the **choice set** of P is denoted as $choice(P)$.
  - Selecting $Q \in choice(P)$ depends on Q's work for others (i.e., its **reputation**).

Trust *(5)*

- Simple example: (cont'd)
  - *P* maintains a (**subjective**) view on reputations. Of course, *P* knows precisely what it has done for others, and what others have done for *P*.
  - *P* can compute a **capacity** ($cap(Q)$):

    $$cap(Q) = max\{MF(Q, P) - MF(P, Q), 0\}$$

    with $MF(P, Q)$ the amount of work (maximum flow) that *P* has, or could have contributed to work done for *Q*, including the work done by others.
  - When $cap(Q)$ is positive, *P* "owes" *Q* some work to do. These capacities are then used in maximum-flow computations that result in reputation scores for each node.
- Essence: Keep track of work that nodes do for each other
  - Assume *R* directly contributed 3 units of work for *Q*, and *R* had processed 7 units for $P \rightarrow P$ may have contributed 3 units of work for *Q*, through *R*
  - Reasoning: *R* may never have been able to work for *Q*, if it had not worked for *P*

Trust *(6)*

- How Sybil attacks are prevented:
  - Let $Q \in choice(P)$ create n Sybil nodes $Q_1^*, \ldots, Q_n^*$; $Q = Q_0^*$
  - For work by Q_i^* for Q_j^* to increase cap(Q_i^*):
    1. Q_j^* needs to have worked for some node R
    2. R needs to have worked for P

    In other words: Q can successfully attack only if it had worked for honest nodes. Also, honest nodes have to work for Q: the total capacity $Tcap(Q)$ of the Sybils must grow, with

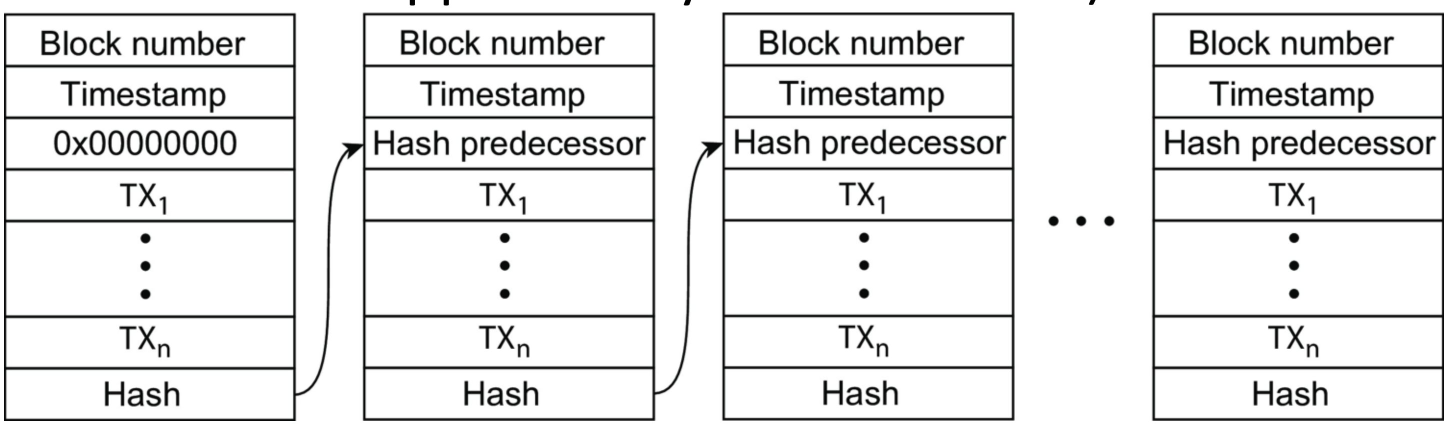
$$Tcap(Q) = \sum_{k=0}^{n} cap(Q_k^*)$$

  - Assume that P works 1 unit for $Q_i^* \rightarrow MF(P, Q_i^*)$ increases by 1 unit $\rightarrow$ cap(Q_i^*) drops by 1 unit, and so does $Tcap(Q)$
  - As soon as $Tcap(Q)$ drops to 0, P will look at other nodes
  - Having the Sybils perform work for each other does not help: if Q_i^* performs a unit of work for Q_j^*, then $cap(Q_i^*)$ goes up by 1 unit, yet $cap(Q_j^*)$ goes down by 1, leaving $Tcap(Q)$ unaffected

Trust *(7)*

Trusting a system: Blockchains

- Essence: Need to know for sure that the info in a blockchain has not been tampered with $\rightarrow$ data integrity assurance
- Solution: Make sure that no change can go unnoticed (recall: a blockchain is an append-only data structure)

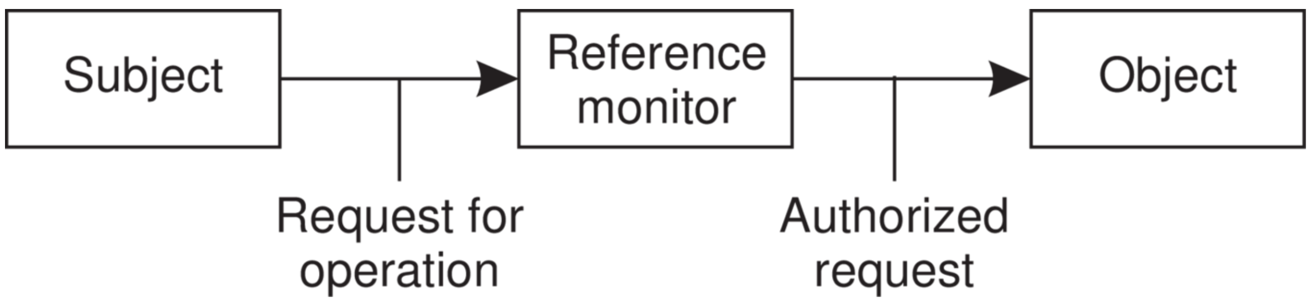


- Observation: Any change of block B_k, will affect its hash value, and thus that of B_{k+1}, which would then also need to be changed, in turn affecting the hash value of B_{k+2}, and so on

Authorization *(1)*

Access Control: General Model

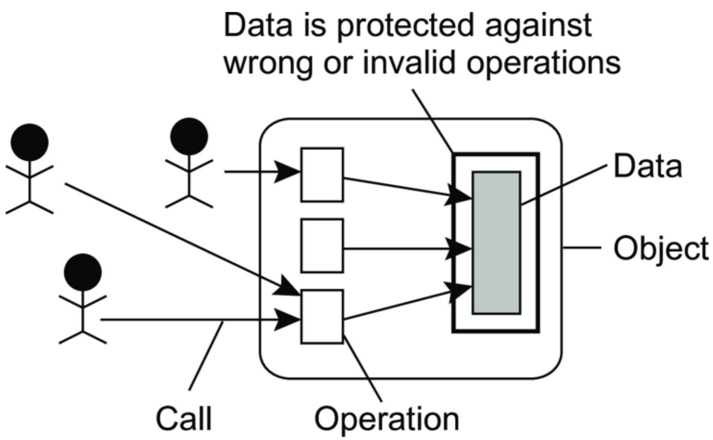
- Essence: Making sure that authenticated entities <u>have only access to specific resources</u>



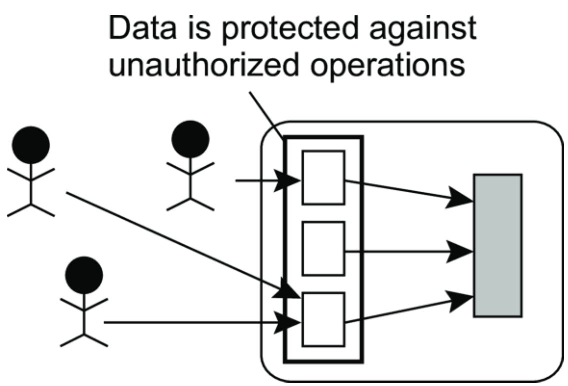
- Observation: The reference monitor <u>needs to be tamperproof</u>: it is generally implemented under full control of the operating system, or a secure server

Authorization *(2)*

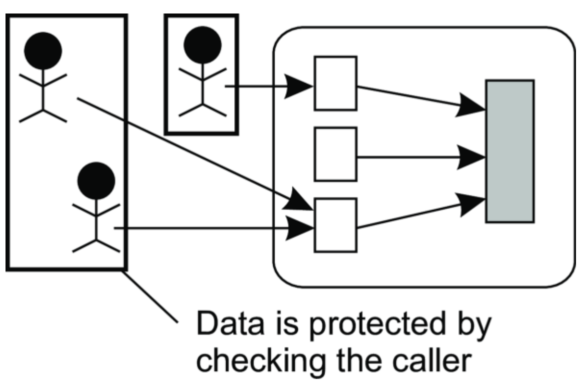
- Protection ...



... against invalid operations



... against unauthorized access



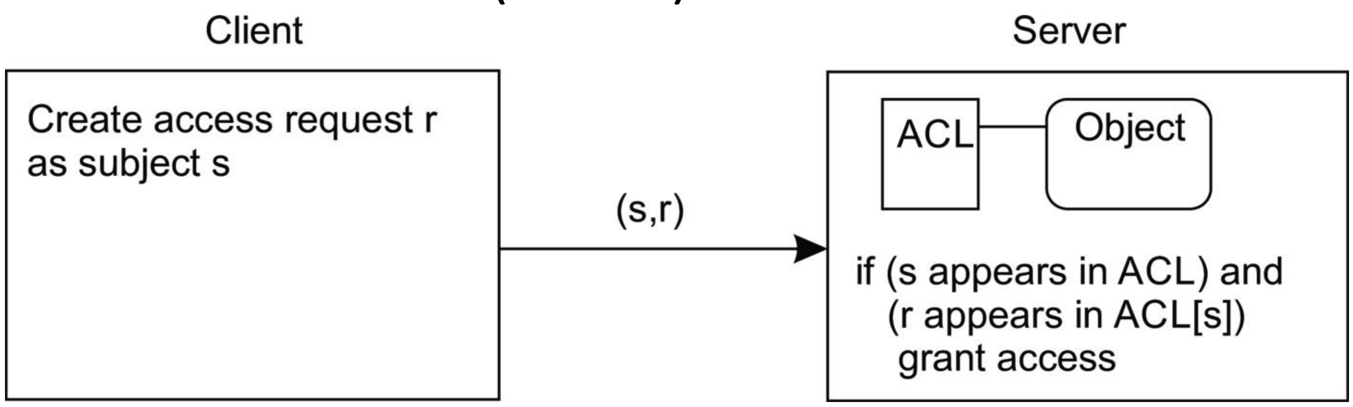
... against unauthorized invokers

Authorization *(3)*

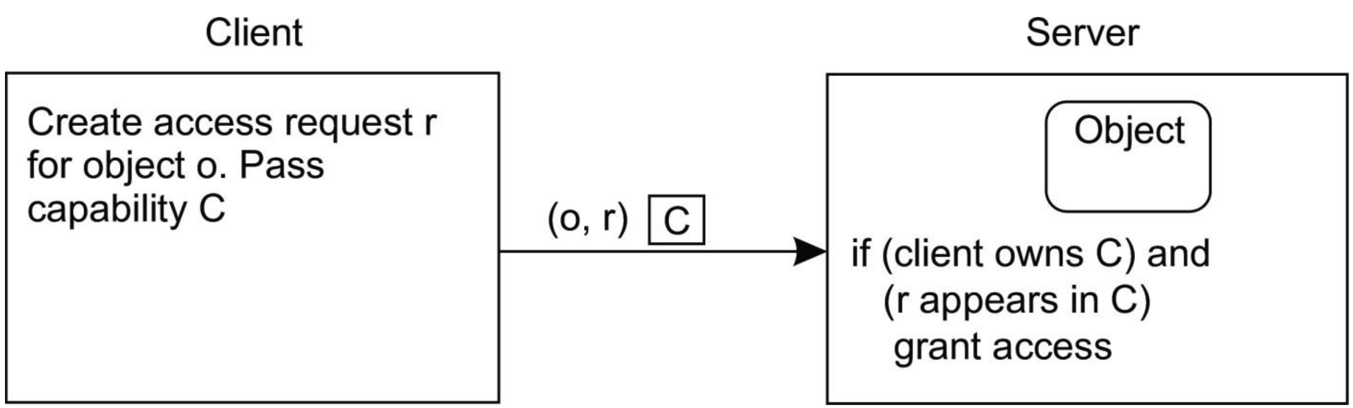
- Access control policies:
  1. <u>Mandatory access control (MAC)</u>: A central administration defines who gets access to what
  2. <u>Discretionary access control (DAC)</u>: The owner of an object can change access rights, but also who may have access to that object
  3. <u>Role-based access control (RBAC)</u>: Users are not authorized based on their identity, but based on the role they have within an organization
  4. <u>Attribute-based access control (ABAC)</u>: Attributes of users and of objects they want to access are considered for deciding on a specific access rule
- Access control matrix:
  - Construct a matrix in which $M[s, o]$ describes <u>the access rights subject *s* has with respect to object *o*</u>
  - Impractical (since many entries in the matrix will be empty), so <u>use **access control lists** or **capabilities**</u>

Authorization *(4)*

- Access control matrix: (cont'd)



Access Control List



Capabilities

Authorization *(5)*

Attribute-Based Access Control

- Distinguish <u>different classes of attributes</u>:
  - <u>User attributes</u>: name, data of birth, current roles, home address, department, qualifiers obtained, contract status, etc. May also depend on role (e.g., teacher or student).
  - <u>Object attributes</u>: anything – creator, last-modified time, version number, file type, file size, but also information related to its content.
  - <u>Environmental attributes</u>: describe the current state of the system, e.g., date and time, current workload, maintenance status, storage properties, available services, etc.
  - <u>Connection attributes</u>: provide information on the current session, e.g., IP address, session duration, available bandwidth and latency estimates, type and strength of security used.
  - <u>Administrative attributes</u>: reflect global policies, e.g., minimal security settings, general access regulations, and maximum session durations.

Authorization *(6)*

- Example: The Policy Machine
  - Essence: A server maintains sets of (*attribute*,*value*) pairs, distinguishing users, applications, operations, and objects. At the core, we formulate access control rules.
  - Access control rules:
    - **Assignment**: A user u can be assigned to an attribute ua: $u \rightarrow ua$. An object to an attribute: $o \rightarrow oa$; an attribute to an attribute: $ua_1 \rightarrow ua_2$ (meaning that if $u \rightarrow ua_1$, then $u \rightarrow ua_2$). Leads to rules like *allowed*(ua, ops, oa): users assigned to ua are allowed to execute operations in ops on objects assigned to oa.
    - **Prohibition**: Explicitly state what is not allowed, such as *denied*(u, ops, os). Also: *denied*(u, ops, $\neg os$), meaning denial when u wants to perform an operation assigned to ops on an object not in os.
    - **Obligation**: Automated action upon an event, such as denying copying of information:

      when u reads $f \in fs$ then *denied*(u, {*write*}, $\neg fs$)

Authorization *(7)*

Delegation

- What's <u>the issue</u>?

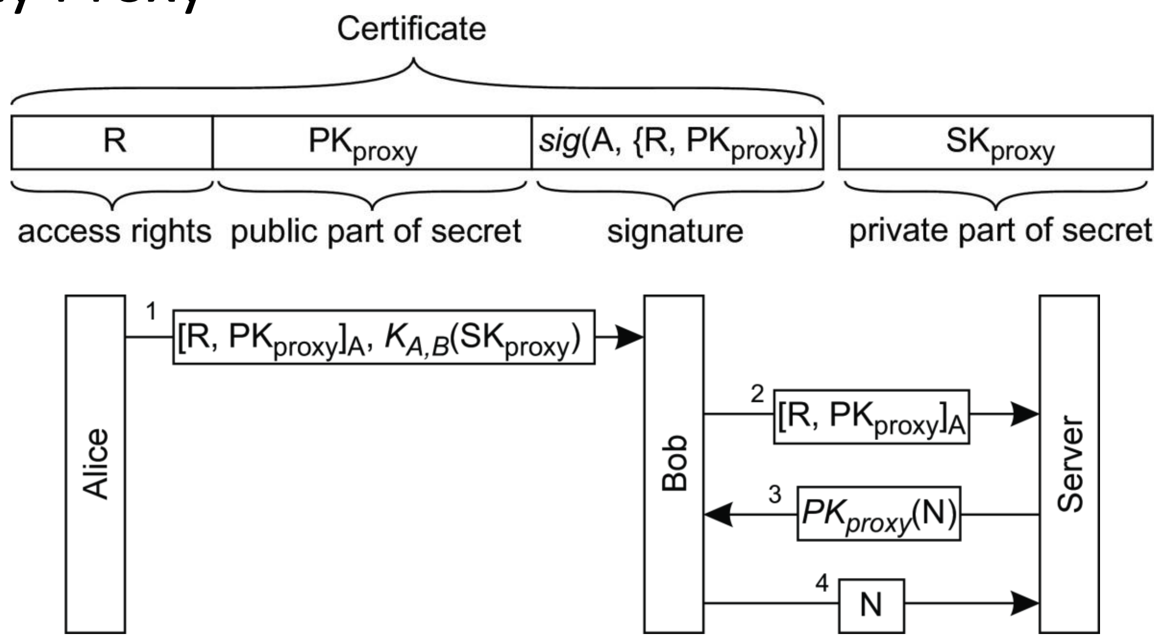
  Alice makes use of an e-mail service provider who stores her mailbox. She is required to log in to the provider to access her mail. Alice wants to use her own local mail client. How to allow that mail client to act on behalf of Alice? How to <u>delegate Alice's access rights</u> to her mail client?

- <u>Observation</u>

  It is <u>not a good idea to hand over all user credentials</u> to an application: why would the application or the machine be trusted? $\rightarrow$ <u>use a security proxy</u>.

Authorization *(8)*

- Security Proxy



<u>How it works</u>

1. Alice passes some rights R to Bob, together with a secret key SK_{proxy}
2. When Bob wants to exercise his rights, he passes the certificate
3. The server wants Bob to prove he knows the secret key
4. Bob proves he does, and thus that Alice had delegated R

Authorization *(9)*

- Example: Open Authorization (OAuth)
  - Four different roles
    - Resource owner: typically an end user
    - Client: an application that one would like to act on behalf of the resource owner
    - Resource server: an interface through which a person would normally access the resource
    - Authorization server: an entity handing out certificates to a client on behalf of a resource owner
  - Initial steps
    1. The client application registers itself at the authorization server and receives its own identifier, *cid*
    2. Alice wants to delegate a list *R* of rights → Client: *send* [*cid*, *R*, *H*(*S*)] with a hash of a temporary secret *S*

Authorization *(9)*

- Example: Open Authorization (OAuth) (cont'd)
  - Final steps
    3. Alice is required to log in and confirm delegation *R* to the client
    4. Server sends a temporary **authorization code** *AC* to client
    5. Client requests a final **access token** →

       Client: *sends* [*cid*, *AC*, *S*]

       Sending *S* to the authorization server allows the latter to verify the identity of the client (by computing *H*(*S*))
  - The authorization server has now
    a) verified that Alice wants to delegate access rights to the client, and
    b) verified the identity of the client

    so, it returns an access token *AT* to the client

Authorization *(10)*

Decentralized Authorization

- WAVE (and keeping it very simple)
  - Essence: Alice delegates rights to Bob, Bob delegates some of those rights to Chuck
    - When Chuck wants to exercise his rights, there should be no need for Alice or Bob to be online
    - No one but Alice, Bob, and Chuck need to be aware of the delegation
  - Essentials:
    - Alice delegates right *R* to Bob, for which he creates a keypair (PK_B^R, SK_B^R):

$$A \text{ sends: } PK_B^R(\underbrace{[R|SK_A^R]}_{m_1})$$

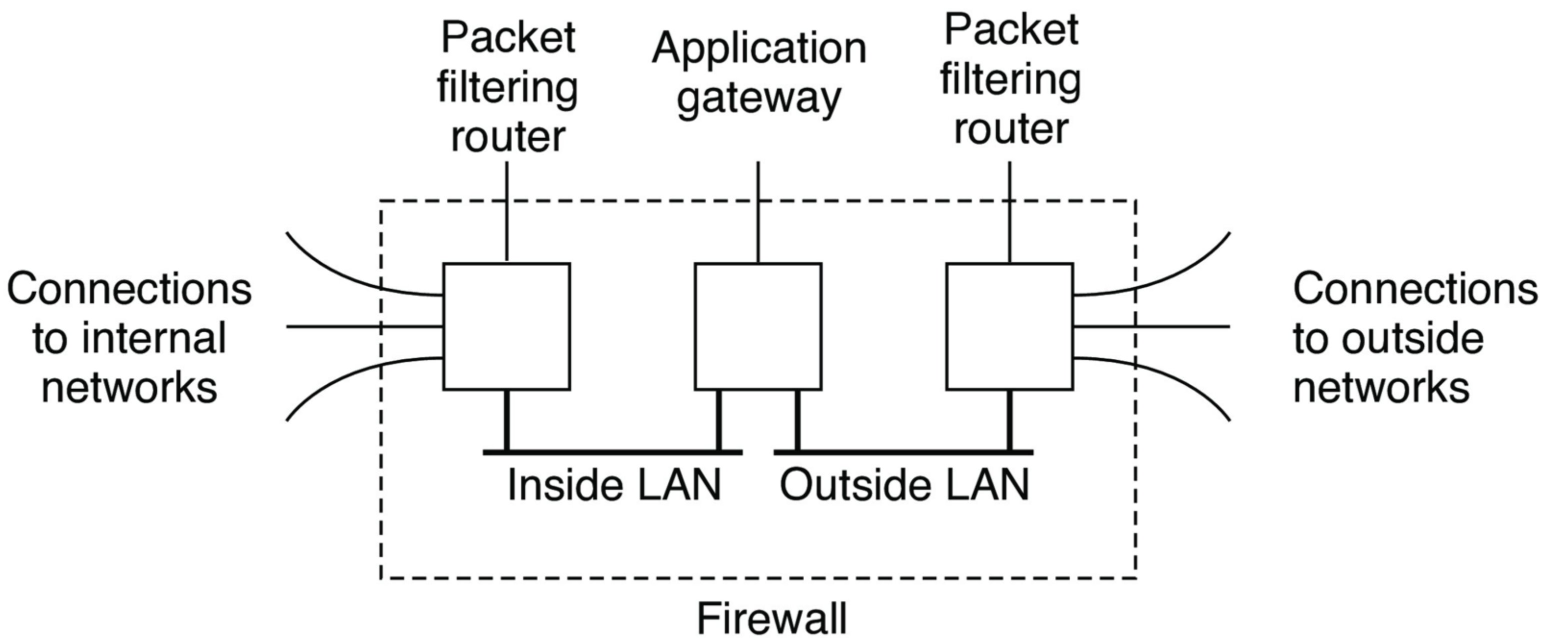
    - Bob delegates parts of those rights *R'* to Chuck, assuming he is allowed to do so:

$$B \text{ sends: } PK_C^{R'}(\underbrace{[R'|m_1|SK_B^R]}_{m_2})$$

Monitoring *(1)*

Firewalls

- Essence: Simply prevent anything nasty coming in, but also prevent unwanted outbound traffic



Monitoring *(2)*

- Different types of firewalls:
  - Packet-filtering gateway: operates as a router and makes filters packets based on source and destination address
  - Application-level gateway: inspects the content of an incoming or outgoing message (e.g., gateways filtering spam e-mail)
  - Proxy gateway: works as a front end to an application, filtering like an application-level gateway (e.g., Web proxies)

Intrusion Detection Systems

- Two flavors:
  - Signature-based (SIDS): matches against patterns of known network-level intrusions. Problematic when series of packets need to be matched, or when new attacks take place.
  - Anomaly-based (AIDS): assumes that we can model or extract typical behavior to subsequently detect nontypical, or anomalous behavior. Relies heavily on modern artificial-intelligence technologies.

Monitoring *(3)*

- Using sensors: Key idea is to manage false and true positives (FP/TP) as well as false and true negatives (FN/TN). Maximize **accuracy** and **precision** $\rightarrow$

Accuracy: $\frac{TP+TN}{TP+TN+FP+FN}$

Precision: $\frac{TP}{TP+FP}$