

# Authentication II

Computer and Network Security

Emilio Coppa

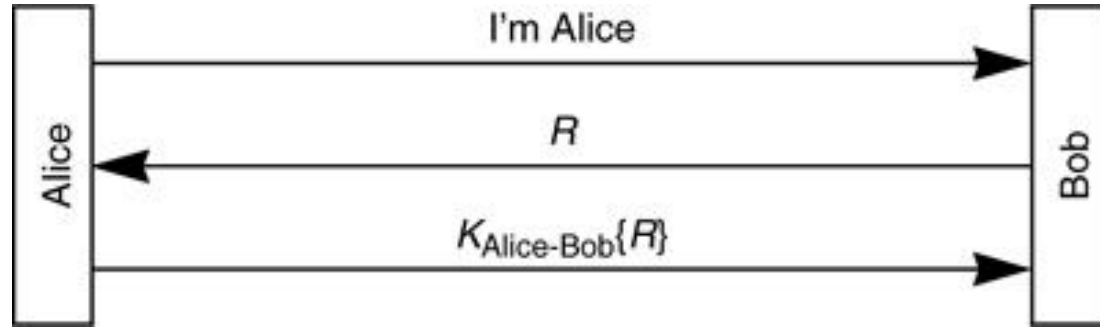
# **Authentication protocols based on a shared secret**

# How to perform authentication?

We can design different protocols. To make them stronger we can use different techniques:

- **timestamps**: prevent attackers from reusing old messages; require clock synchronization among entities
- **nonce (or challenge)**: random numbers, never re-used (**N**umber used only **ONCE**), used to “challenge” someone to encrypt/decrypt something. In some protocols, nonce could be not random but a sequence number.

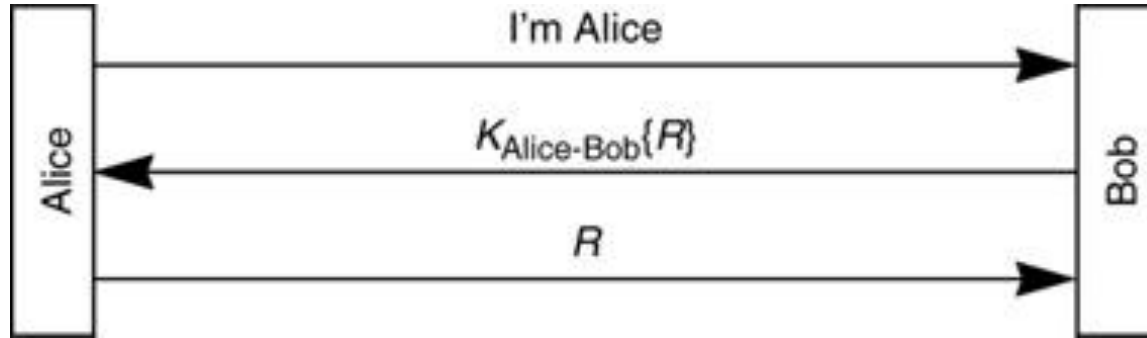
# Challenge/response based on shared secret



- $R$  is a nonce with limited lifetime
- function to encrypt it can be an hash function
- Bob authenticates Alice, but not the opposite, hence it is not mutual.
- if  $K_{AB}$  is derived from a password, then an attacker can still guess the password
- offline password guessing: a valid pair is known to the attacker
- if Bob (e.g., the server) is compromised then Alice is in trouble
- ... still an improvement w.r.t. sending password in clear (or hash of password)

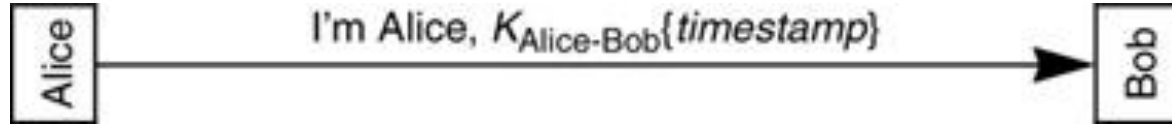
**Problem:** attacker may impersonate Bob and challenge Alice with  $R$

## Challenge/response based on shared secret (2)



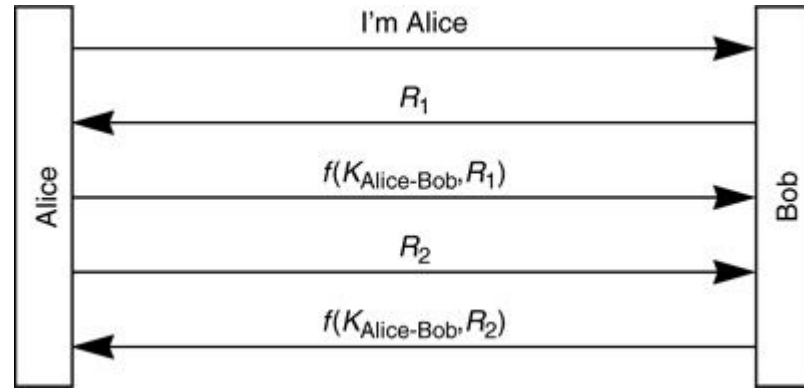
- similar to previous protocol
- requires that encryption is reversible (decryption)

# Use of a timestamp to limit replay attacks



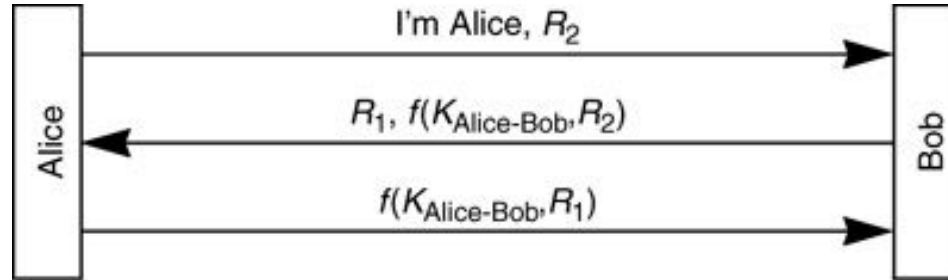
- Bob and Alice have reasonably synchronized clocks (**can be a weakness: we need a protocol to set securely the correct time**)
- Alice encrypts the current time, Bob decrypts the result and makes sure the result is acceptable (i.e., within an acceptable clock skew)
- efficient, no intermediate states
- if Bob remembers timestamps until they expire, then no replaying attacks
- if multiple servers with same secret  $K$ , since timestamp is not tied to Bob then Alice could send  $K\{\text{Bob}|\text{timestamp}\}$  to prevent problems
- no mutual authentication

# Mutual authentication



- double challenge/response authentication protocol
- number of messages can be reduced
- Alice and Bob should keep track of nonce over time to avoid replay attacks

# Mutual authentication (optimized)

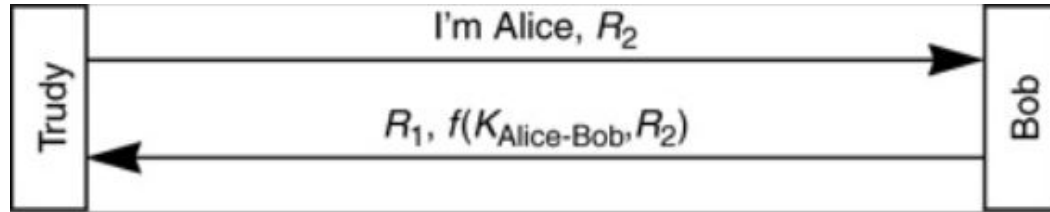


- fewer messages, hence more efficient
- weak to a reflection attack
- weak to offline password guessing



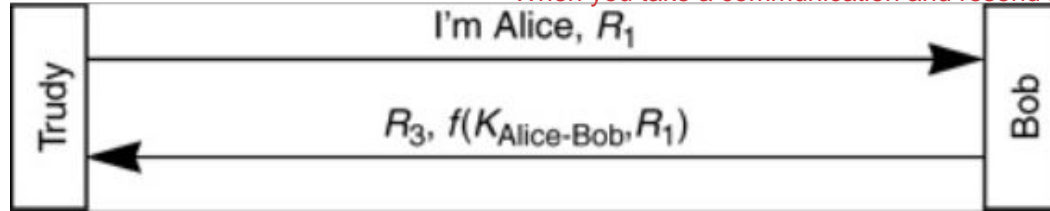
# Mutual authentication (optimized): reflection attack

Session #1:



When you take a communication and resend it in another communication session

Session #2:

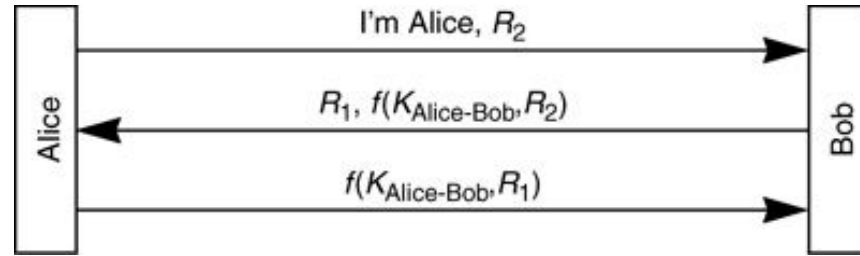


- Trudy wants to impersonate Alice to Bob
- two sessions, the 2<sup>nd</sup> will be incomplete but will allow Trudy to complete the 1<sup>st</sup> one

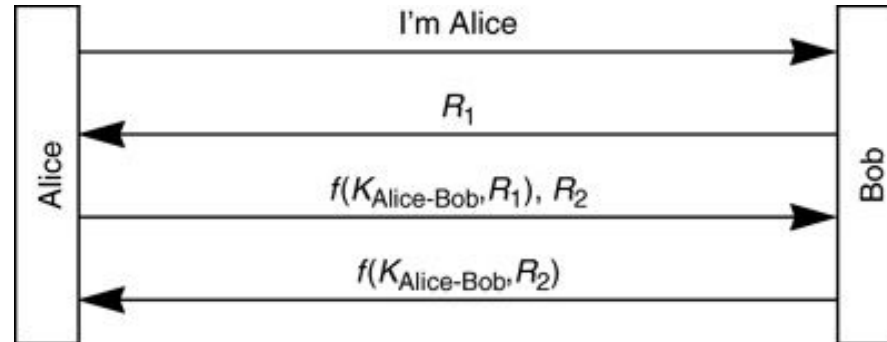
# How to prevent reflection attacks

- **use different keys:**  $K_{AB}$  vs  $-K_{AB}$  vs or  $K_{AB} + 1$
- **different challenges**, i.e. challenge from initiator different from challenge from responder:
  - e.g., even number at initiator's side, odd number at responder's side

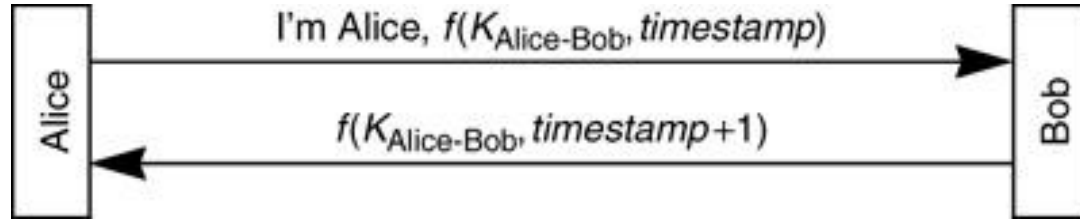
# Mutual authentication (optimized): offline password guessing



Attacker can impersonate Alice, send challenge to Bob, get response and then proceed working offline trying to guess the right password. Notice that inefficient mutual authentication protocol was not affected by this. We can mitigate this by adding a message in the protocol:



# Mutual authentication with timestamps



- crucial to alter the timestamp in the response from Bob
- clock synchronization is crucial

# Practical problem of sharing a secret with each entity

In practice, in a large network, it is not possible to share a secret pairwise:

**Number of Keys:** if  $n$  users in a network then

- we need  $\frac{n \cdot (n - 1)}{2}$  keys.
- each user has to store  $n-1$  keys, which is reasonable
- adding one user to the network, we have to transfer securely  $n$  keys (hard!)

# Practical problem of sharing a secret with each entity (2)

Possible alternative approaches:

- share a secret with a centralized trusted party
- use public-key schemes

**Authentication protocols  
based on a trusted party**

# Trusted Party

## Scenario:

- A and B share a secret key with C ( $K_{AC}$  or  $K_{\text{Alice}}$  and  $K_{BC}$  or  $K_{\text{Bob}}$ )
- C is also called **authentication server** or **Key Distribution Center** (KDC)
- A and B might not be human user but also entity of the system (e.g., printer, databases etc.)

## Goals:

- authenticate A (or A and B)
- [optional] decide on a session key  $K_{AB}$  to be used between A and B for short time, where  $K_{AB}$  is known only to A and B, randomly chosen and never used in the past communications



# Attacker's capabilities

Trudy (attacker) can:

- be a legitimate user of the system (share a key with C)
- sniff and spoof messages
- concurrently run more than one session with A, B and C
- different execution of the protocol can be done interleaved
- T is able to convince A and/or B to start a new session with T
- might know old session keys

# Attacker's limits

Trudy:

- is NOT able to guess random numbers chosen by A or B
- does not know keys  $K_{AC}$  and  $K_{BC}$  (in general does not know secret keys of other users)
- is not able to decode in a short time messages encrypted with unknown keys

# Authentication with a trusted party

1. A sends to C:

$A, B$

2. C chooses  $K$  (secret shared key) and sends to A:

$K_{AC}(K)$  and  $K_{BC}(K)$

3. A decodes and computes  $K$  and sends to B:

$C, A, K_{BC}(K)$

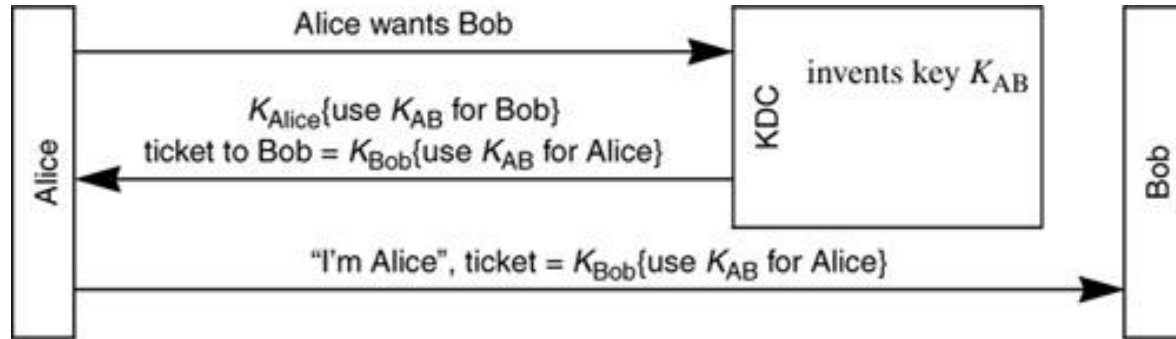
4. B decodes  $K_{BC}(K)$  finds  $K$  and sends to A:

$K(\text{Hello A, this is B})$

In several protocols, the message generated by C for Bob (containing the session key) but sent to Alice is called a “**ticket**”.

# Authentication with a trusted party (2)

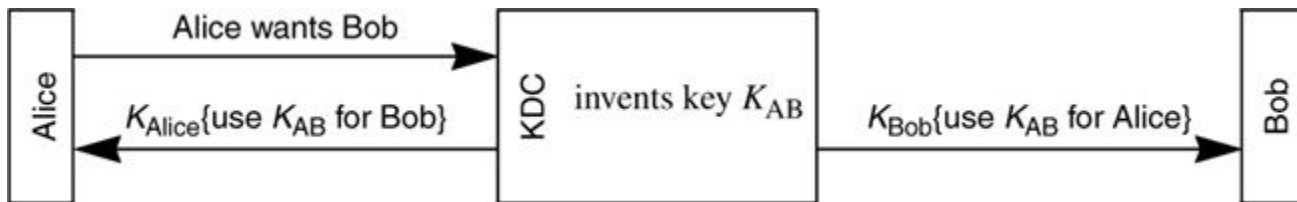
Protocol:



- in several protocols, the message generated by KDC for Bob (containing the session key) but sent to Alice is called a **“ticket”**
- this protocol is weak against a MITM attack

# Authentication with a trusted party

A variant of the previous protocol:



- KDC needs to communicate with both Alice and Bob
- Attacker may easily trick KDC to send messages to Bob: DoS? Unlikely, but still a concern
- **Problem:** This requires synchronization: Alice may send something encrypted to Bob, before Bob has received the shared key

# MITM attack: authentication with a trusted party

1. A sends to T (instead of A sends to C) :

A, B

2. immediately T sends to C:

A, T

3. C chooses K and sends to A:

$K_{AC}(K)$  and  $K_{TC}(K)$

4. A sends to B:

C, A,  $K_{TC}(K)$

5. T intercepts the message sends to A:

$K(\text{Hello A, this is B})$

**Problem:** Alice believes to have a shared key with Bob, instead the key is with Trudy.

# Authentication with a trusted party (revised)

1. A sends to C:

$A, K_{AC}(B)$

2. C chooses  $K$  (secret shared key) and sends to A:

$K_{AC}(K)$  and  $K_{BC}(K)$

3. A decodes and computes  $K$  and sends to B:

$C, A, K_{BC}(K)$

4. B decodes  $K_{BC}(K)$  finds  $K$  and sends to A:

$K(\text{Hello A, this is B})$

Trudy now does not know  
that A wants to talk to B.

Is this enough to make the protocol secure?

# MITM attack: authentication with a trusted party (revised)

1. A sends to T (instead of A sends to C) :

$A, K_{AC}(B)$

2. T intercept the message and replays old message:

$A, K_{AC}(T)$

3. C chooses K and sends to A:

$K_{AC}(K)$  and  $K_{TC}(K)$

4. A sends to B:

$C, A, K_{TC}(K)$

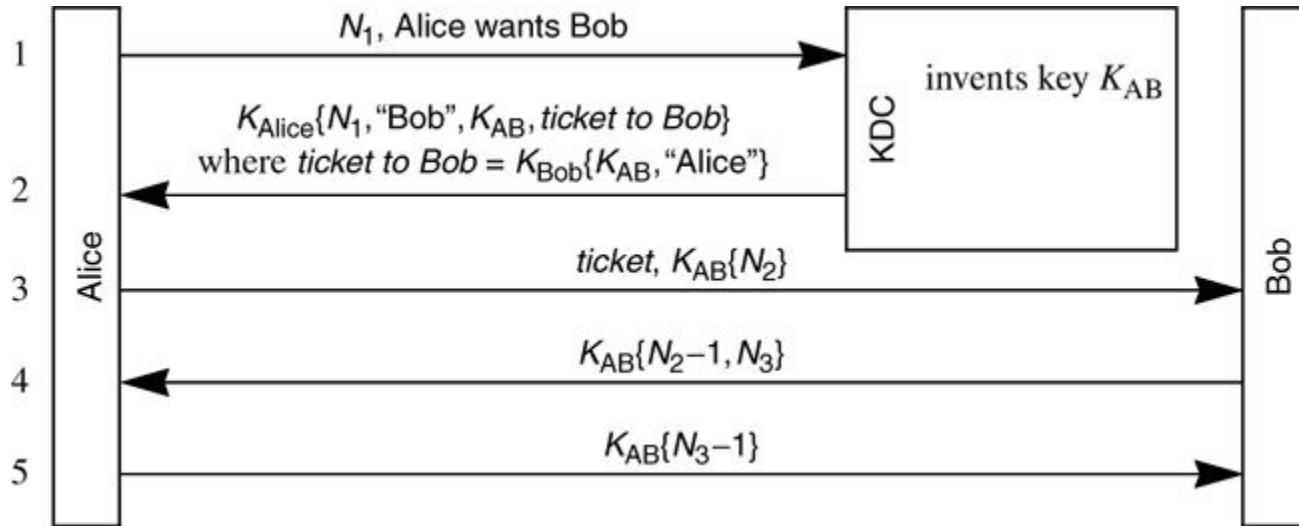
5. T intercepts the message, understand that A is trying to talk to B and sends to A:

$K(\text{Hello A, this is B})$

**Problem:** Alice believes to have a shared key with Bob, instead the key is with Trudy. The only difference now is that Trudy has to wait the end of the attack to know the identity B.



# Needham–Schroeder (NS) symmetric protocol



**Remark.** KDC does not communicate with Bob

# NS replay attack (Denning and Sacco)

Bob is not communicating with KDC and does not know when the shared key has been generated whether KDC is still active. Suppose that an old session key  $K'_{AB}$  has been compromised then:

1. A chooses  $N$  (nonce) and sends to C:  $A, B, N_1$
2. C chooses  $K$  and sends to A:  $K_{AC}(N_1, B, K_{AB}, K_{BC}(K_{AB}, A))$
3. A decodes, checks  $N_1$  and  $B$  and sends to B:  $K_{BC}(K_{AB}, A). K_{AB}(N_2)$
4. T blocks the message for B, replays (as A) to B:  $K_{BC}(K'_{AB}, A), K'_{AB}(N''_2)$
5. B decodes, chooses nonce  $N_2$  and sends to T (not to A):  $K'_{AB}(N''_2 - 1, N_3)$
6. T sends to B:  $K'_{AB}(N_3 - 1)$

# NS replay attack (2)

Hence, to carry out the attack, two sessions are needed:

- session #1:
  - Trudy records  $K_{BC}(K'_{AB}, A)$
  - Trudy (later) is able to recover  $K'_{AB}$
- session #2:
  - Trudy intercepts messages from A to B
  - Trudy reuse old  $K_{BC}(K'_{AB}, A)$
  - Bob is convinced to have a new shared key with Alice, which however it is known to Trudy

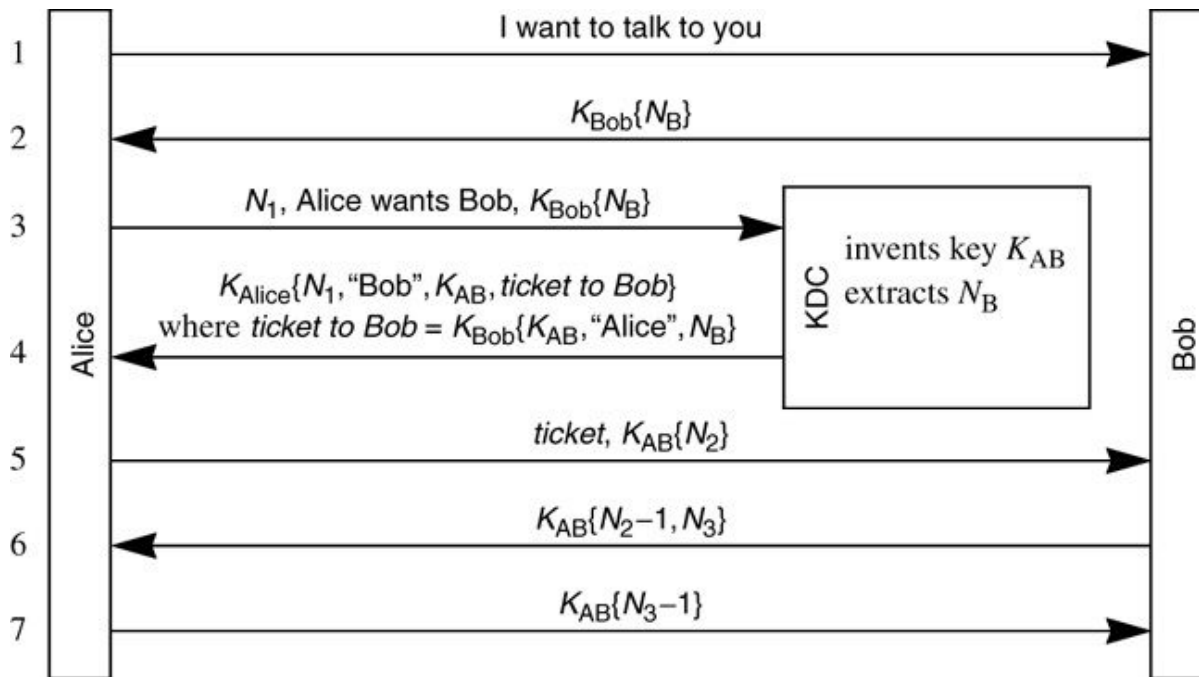
# How to prevent replay attacks?

Mitigations to fix replay attacks can be based on:

- **timestamps**: a message is valid only in a small time window
- **sequence numbers**: A and B remember sequence number of exchanged messages to avoid replay attacks in which the attacker sends old messages
- **nonce**: used in some protocols, but not easy mitigation (NS is already using nonce...)

# Needham-Schroeder symmetric protocol (fixed)

Mitigation based on a nonce.



Last message could be safely removed

# Needham-Schroeder symmetric protocol (fixed variant)

Kerberos is a well-known protocol inspired by NS where KDC exchanges messages with both A and B and uses a timestamp to prevent replay attacks:

1. A chooses N and sends to B:

$A, N$

2. B chooses  $N'$  and sends to C:

$B, N', K_{BC}(N, A, t)$

3. C sends to A:

$K_{AC}(B, N, K_{AB}, t), \text{ticket} = K_{BC}(A, K_{AB}, t), N'$

4. A sends to B:

$K_{BC}(A, K_{AB}, t), K_{AB}(N')$

**Standard: Kerberos v4/v5**

# Challenge-response Symmetric Key

What do we learn from previous attacks?

- **Timestamps** are valid only in a small time window
- **Sequence numbers** attached to messages are useful (to avoid replication attacks)
- **Nonce**: we should carefully use them (and we require good random number generators)



# Kerberos

- Kerberos provides authentication in distributed systems
  - Guarantees safe access to network resources (e.g. printer, databases etc.)
  - There is a central authority that allows to reduce the number of passwords that users must memorize
- Reference:
  - proposed by MIT <http://web.mit.edu/kerberos/www/dialogue.html>
  - free download in US and Canada (after 2000, in most locations)
  - widespread use (most operating systems)

# Kerberos

- **Scenario:** A needs to access service provided by B
  - Authentication of A
  - Optional: authentication of B
  - Optional: decide session keys for secret communication and/or authentication
- C is trusted server (authority that shares keys with A and B)
- **Idea:** use ticket to access services; tickets are valid in a given time window

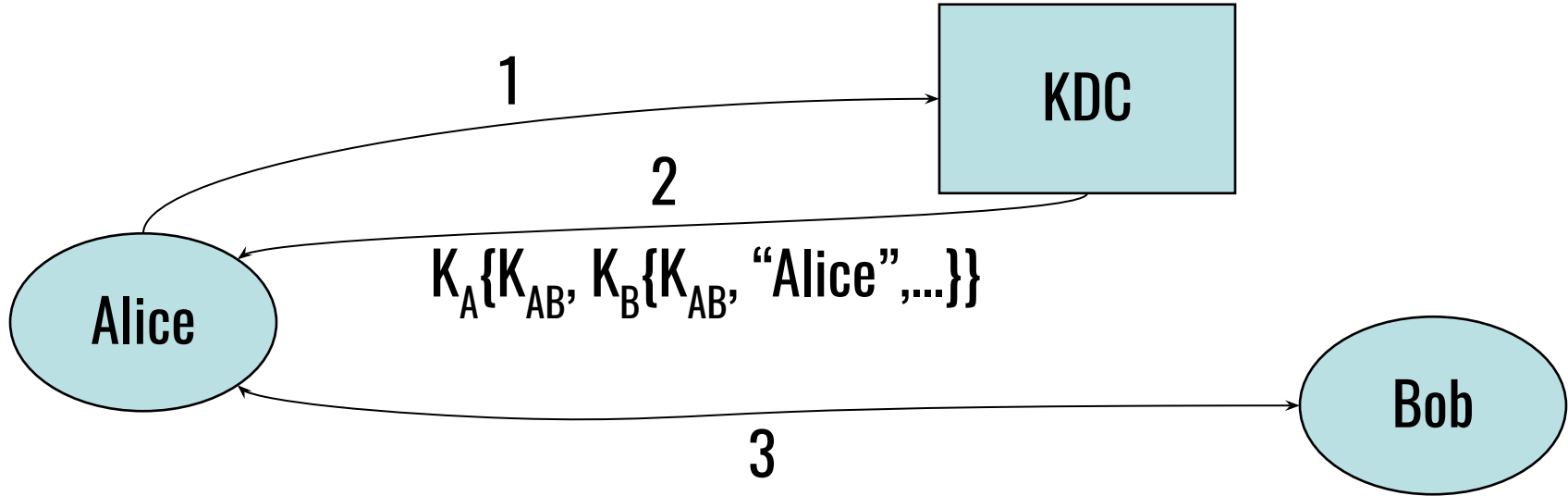
# Kerberos

- KDC (Key Distribution Center) is the server (both trusted and physically safe)
- Messages are safe with respect to cryptographic attacks and data integrity
- Kerberos provides security for applications like
  - telnet
  - rtools (rlogin, rcp, rsh)
  - network file systems (NFS/AFS)
  - e-mail
  - etc.

# Preliminaries

- Each user (also named as principal) has a master secret key with KDC
  - for human users master secret key is derived from password
  - for system resources, keys are defined while configuring the application
- Each principal is registered by the KDC
- All master keys are stored in the KDC database, **encrypted with the KDC master key**

# Tickets, Alice, Bob, KDC



**Ticket of Alice for Bob:**  $K_B\{K_{AB}, \text{"Alice"}, \dots\}$ ,

$K_A$  master key of Alice,  $K_B$  master key of Bob,

$K_{AB}$  session key to be used by A and B

**only Bob is able to decode and checks the message**

# Tickets

- a ticket is encrypted with the secret key associated with the service
- ticket basically contains
  - sessionkey
  - username
  - client network address
  - servicename
  - lifetime
  - timestamp

# Kerberos: simplified version (some fixes later)

A asks for a ticket TicketB for B:

1. A sends to C: A, B, N (N nonce),  $t_A$  (timestamp)
2. C sends to A: TicketB,  $K_{AC}(K_{AB}, N, L, B)$   
(where: TicketB =  $K_{BC}(K_{AB}, A, L, t_A)$ , L is “lifetime of ticket”)
3. [A checks N and knows ticket lifetime]  
A sends to B: TicketB,  $K_{AB}(A, t_A)$  [authenticator]
4. [B checks that A's identity in TicketB and in authenticator are the same, time validity of ticket]  
B sends to A:  $K_{AB}(t_A)$  [in this way shows knowledge of  $t_A$ ]

# Session key and Ticket-granting Ticket (TGT)

- Messages between host and KDC should be protected using the master key (derived from user's password)
- For each request to the KDC:
  - user must type the password each time, or
  - user's password is temporarily stored (to avoid the user the need of retyping)

**all above solutions are inadequate!**



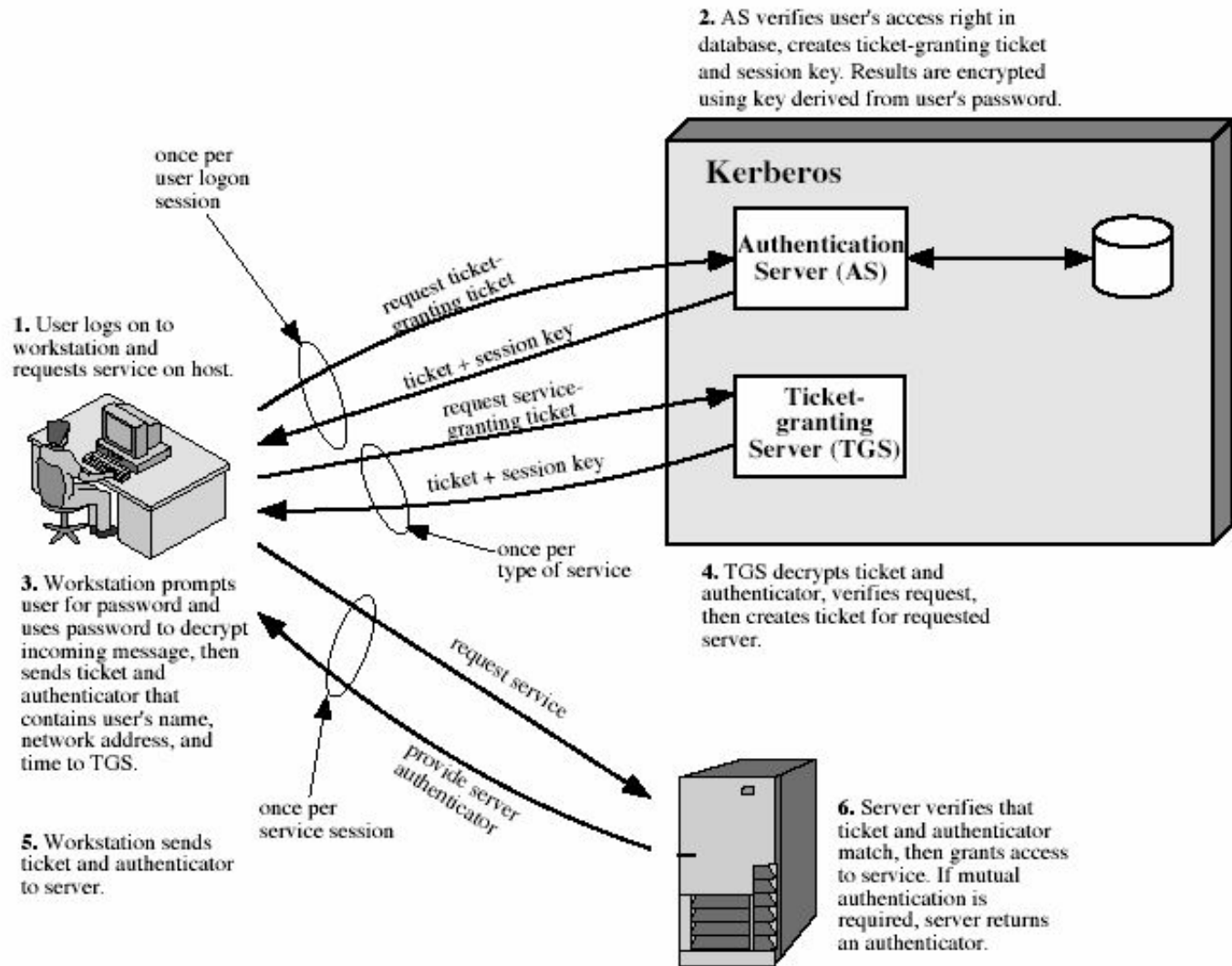
# Session key and Ticket-granting Ticket (TGT)

Proposed solution to reduce # of times user types the password and/or master key:

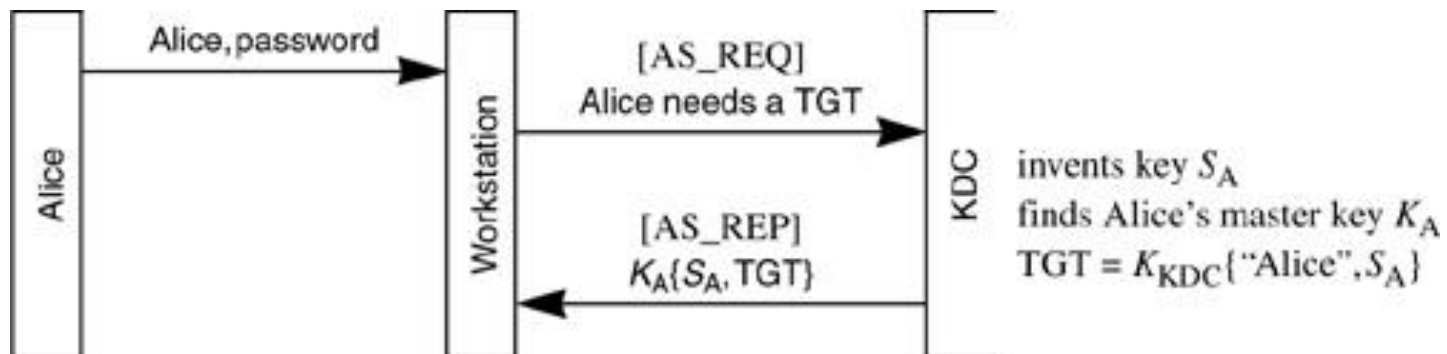
- at initial login a session key  $S_A$  is derived for Alice by KDC
- $S_A$  has a fixed lifetime (e.g., 1 day, 4 hours)
- KDC gives Alice a TGT that includes session key  $S_A$  and other useful information to identify Alice (encrypted with KDC's master key)

# Session key and Ticket-granting Ticket (TGT)

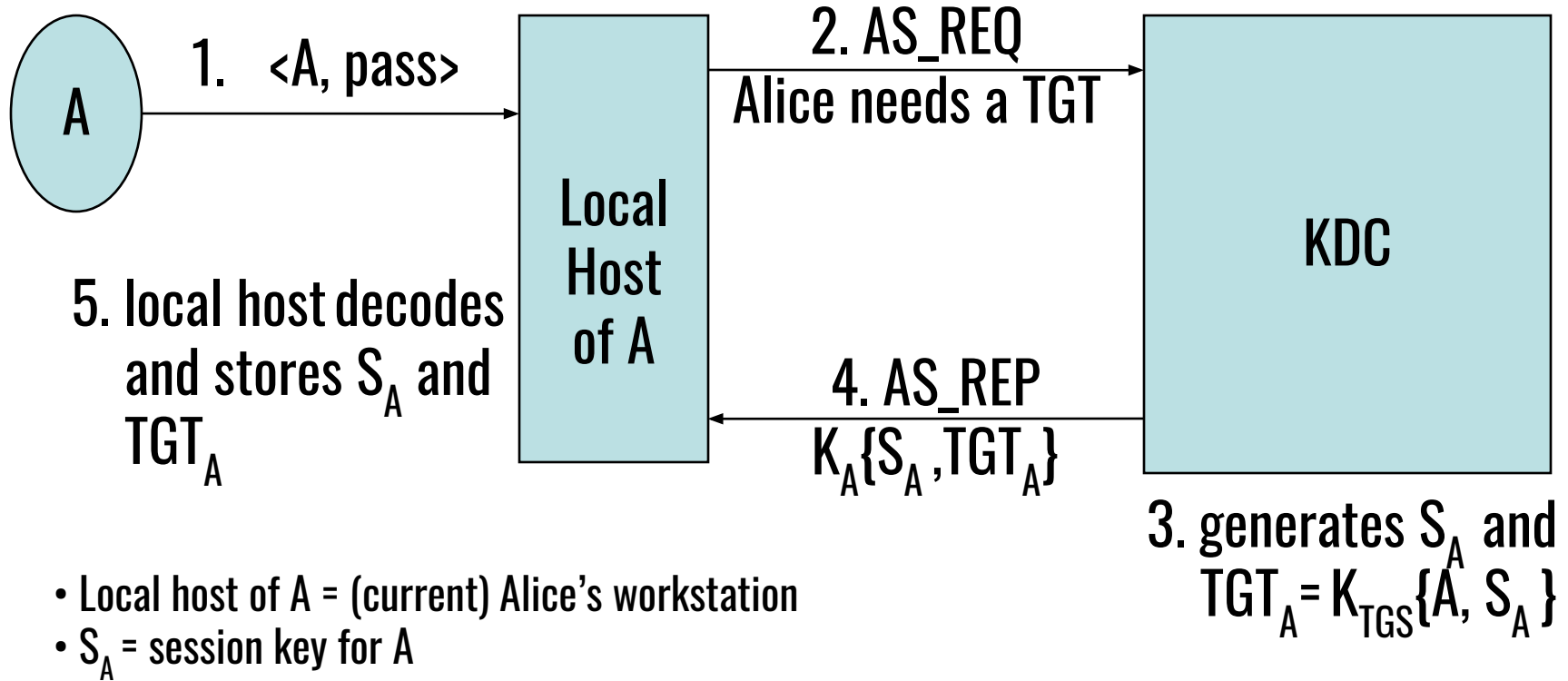
- Subsequent requests from Alice to KDC use TGT in the initial message
- Subsequent tickets provided by KDC for accessing server V are decoded using  $K_{VC}$
- User provides password only once
- No password is stored



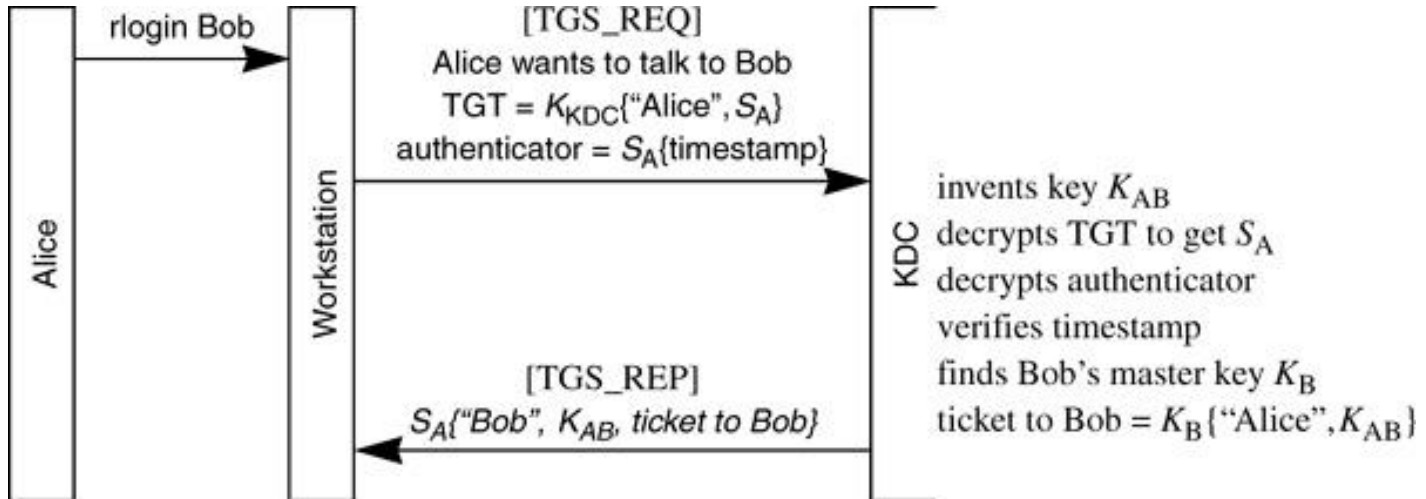
# Login



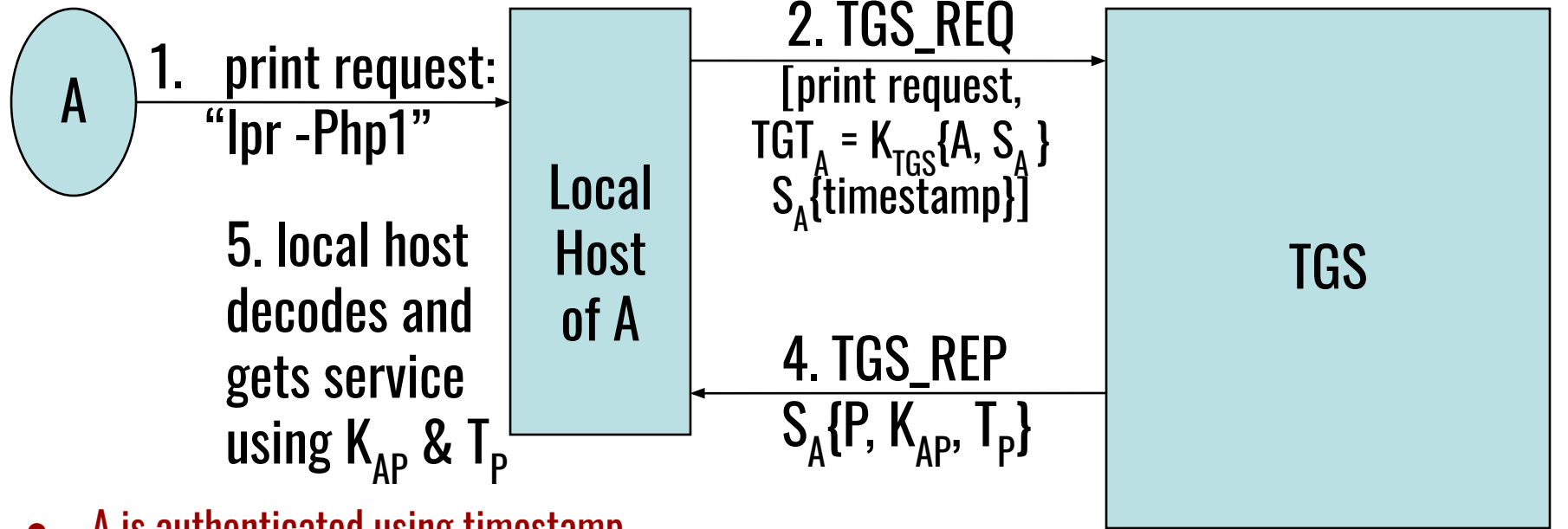
# Login (simplified)



# Ticket request

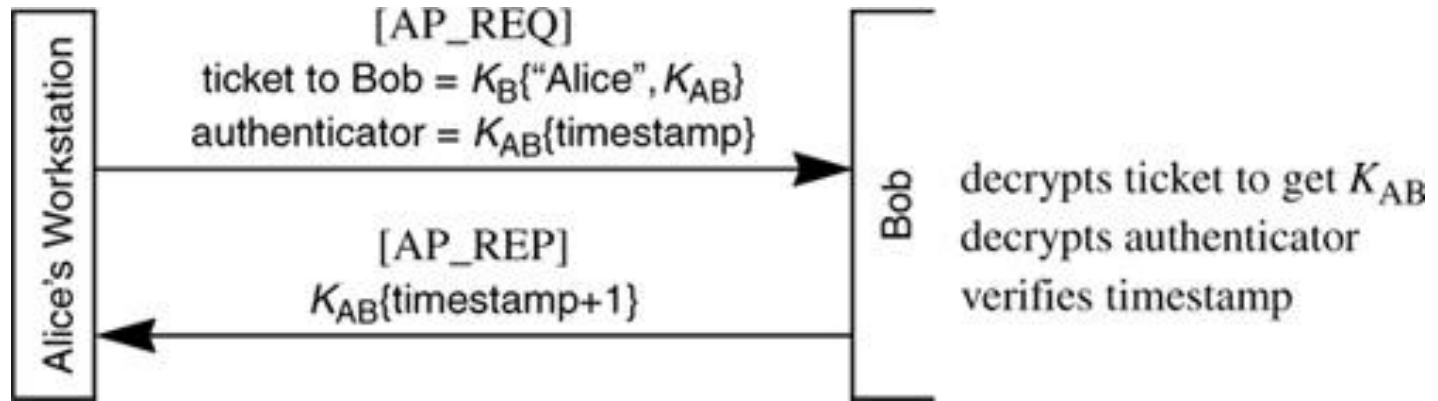


# Ticket request (simplified)



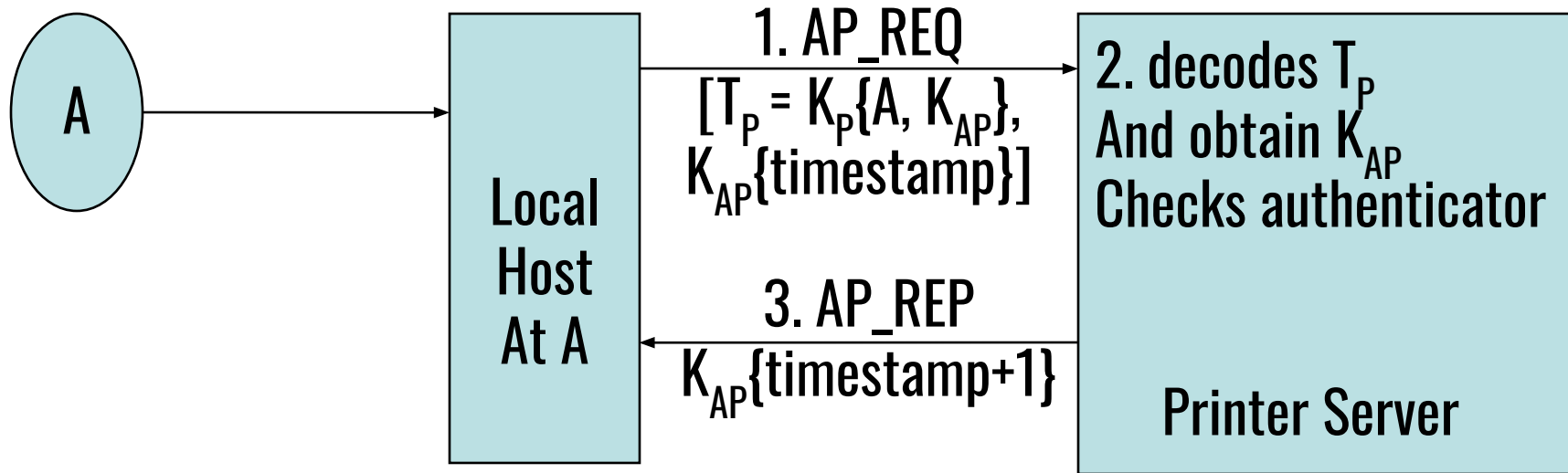
- A is authenticated using timestamp
- P = printer

# Use of ticket





# Use of Ticket for printer P



- printer request is managed by A's local host
- there is mutual authentication using timestamp

# Authentication and time synchronization

- Authenticator:  $K_x\{\text{timestamp}\}$ 
  - $K_x$  is a session key
- Global Synchronous Clock is required
- Authenticator is used to avoid
  - replay of old messages sent to the same server by the adversary (old messages are eliminated)
  - replay to a server (when there are many servers)
  - Authenticator DOES NOT guarantee data integrity (a MAC is required)
- **Vulnerability:** many instances of same server all using same master key. Replay attack!

# KDC and TGS

- KDC and TGS are similar (the same?) why do we need two different entity?
  - Historical reasons
  - One KDC can serve different systems (1 KDC many TGS)
- multiple copies of KDC, sharing same KDC master key - availability and performance
- Consistency issues in KDC databases (master-slaves)
  - A single KDC stores information concerning principal (safer)
  - Periodically upload information to other KDC

# Kerberos - Performance

- KDC stores only TGT and tickets
- Most work is on client
- KDC is involved only at login to provide TGT
- KDC uses only permanent information

# Message types

- **AS\_REQ**
  - Used when asking for the initial TGT.
- **AS\_REPLY** (also **TGS\_REP**)
  - Used to return a ticket, either a TGT or a ticket to some other principal.
- **AP\_REQ** (also **TGS\_REQ**)
  - Used to talk to another principal (or the TGS) using a ticket (or a TGT).
- **AP\_REQ\_MUTUAL**
  - This was intended to be used to talk to another principal and request mutual authentication. In fact, it is never used; instead, applications know whether mutual authentication is expected.
- **AS\_ERR**
  - Used for the KDC to report why it can't return a ticket or TGT in response to AS\_REQ or TGS\_REQ.
- **PRIV**
  - This is a message that carries encrypted integrity-protected application data.
- **SAFE**
  - This is a message that carries integrity-protected application data.
- **AP\_ERR**
  - Used by an application to report why authentication failed.

# Ticket (Alice, Bob)

**It is encrypted with Bob's key:**

- Alice's name, instance and realm
- Alice's Network Layer address
- session key for Alice, Bob
- ticket lifetime, units of 5 minutes
- KDC's timestamp when ticket made
- Bob's name and instance
- pad of 0s to make ticket length multiple of eight octets

# Authenticator

**It is encrypted with session key**

- Alice's name, instance and realm
- checksum
- 5-millisecond timestamp
- timestamp (time in seconds)
- pad of 0s to make authenticator multiple of eight octets

# Kerberos: Realms

- In very large systems security and performance issues suggest to use not only a domain but more (i.e., several KDC):

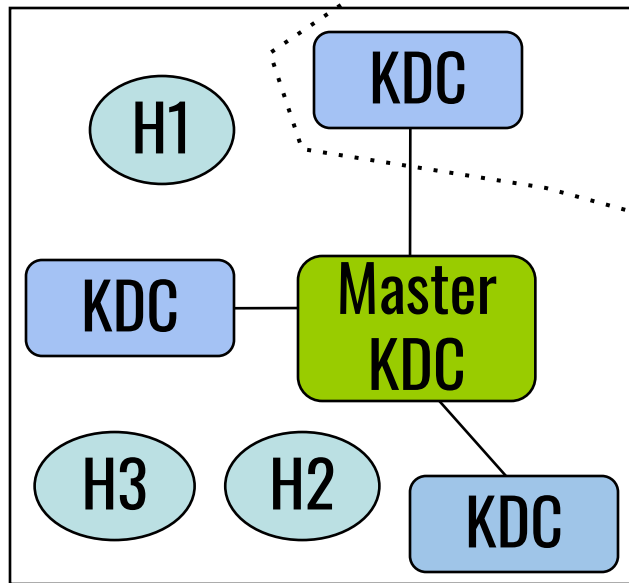
## REALM

- a single realm in real-world: unlikely since this would mean that all organizations have to trust a single entity...
- each realm has a different master KDC
- all KDCs share the same KDC master key
- two KDCs in different realms have different databases of users

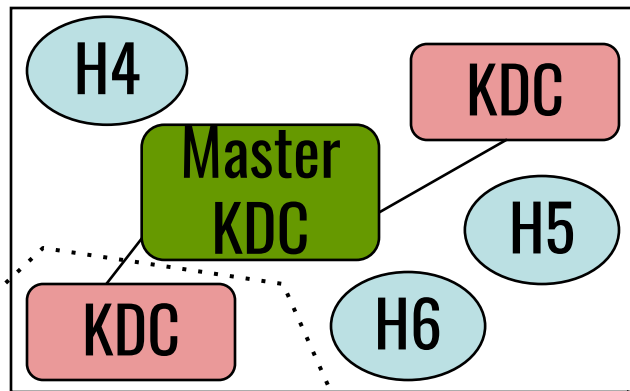


# Kerberos V. 4: Realms

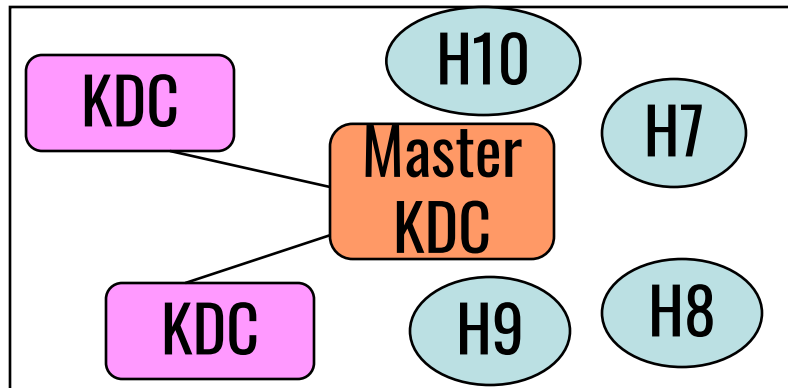
Realm #1



Realm #2

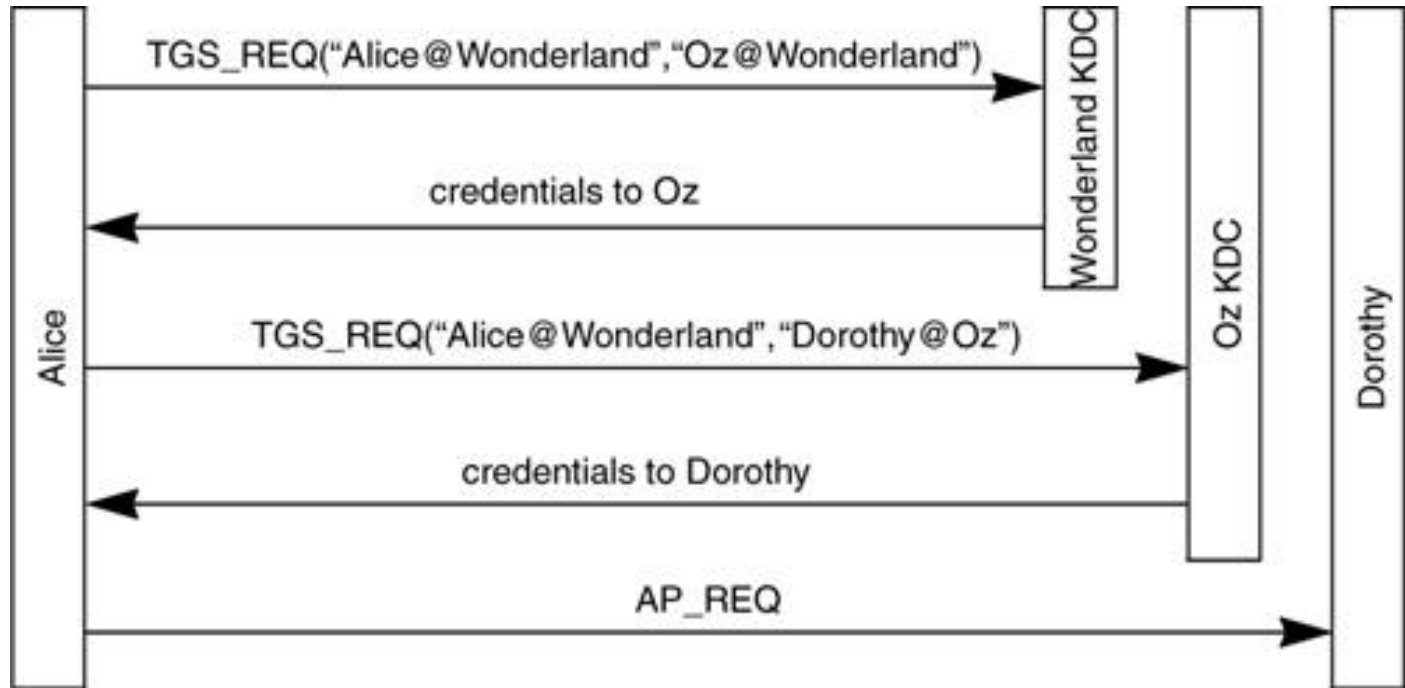


Realm #3

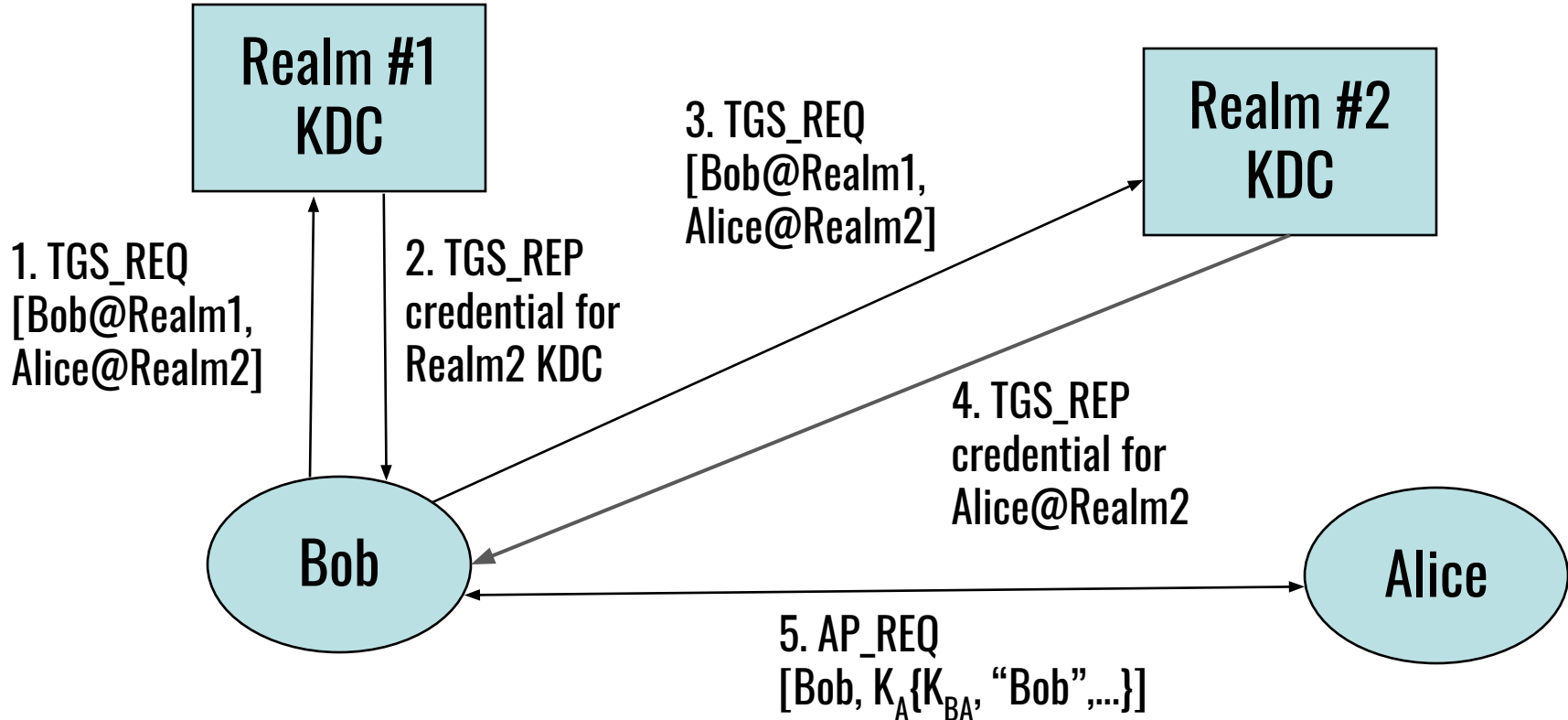


Authentication may be allowed only between realms that have an agreement.

# Authentication between realms



# Authentication between realms



## Other features

- **key version numbers**
  - for supporting changes of master keys while keeping existing tickets valid
- **encryption for privacy and integrity**
  - DES + Propagating/Plaintext Cipher Block Chaining (PCBC)
  - problem: in PCBC two adjacent blocks can be swapped
- **encryption for integrity only: custom checksum algorithm**

# Kerberos: version 5 (RFC 4120)

- Same philosophy
- Major changes
- Integrity of messages, authentication using nonce (not only timestamps)
- Flexible encoding: many optional fields,
  - allows future extensions
  - overhead
- Major extensions to the functionality
- Delegation of rights: Alice allows Bob to access:
  - her resources for a specified amount of time
  - a specific subset of her resources
- Renewable tickets: tickets can be used for long time
- More encryption methods (Kerberos designed for DES)
- Hierarchy of realms

# Additional details on Kerberos v4/v5

- **RFC 4120:** [\[URL\]](#)
- **Chapter 13.** Charlie Kaufman, Radia Perlman, and Mike Speciner. Network Security - Private Communication in a Public World. Prentice Hall.
- **Chapter 14.** Charlie Kaufman, Radia Perlman, and Mike Speciner. Network Security - Private Communication in a Public World. Prentice Hall.

# Credits

These slides are based on material from:

- Slides of Prof. D'Amore from CNS 2019-2020
- Christof Paar and Jan Pelzl. Understanding Cryptography: A Textbook for Students and Practitioners. Springer. <http://www.crypto-textbook.com/>
- Charlie Kaufman, Radia Perlman, and Mike Speciner. Network Security - Private Communication in a Public World. Prentice Hall.
- Wikipedia (english version)