Key Distribution
(CS-452)

Week 5

Symmetric Distribution using Needham–Schroeder Symmetric Key Protocol

Key Distribution

- For symmetric encryption to work, the two parties must share a common key and that key must be protected from access by others.
- Frequent key changes are usually desirable to limit the amount of data compromised if an attacker learns the key.
- Key distribution: refers to the means of delivering a key to two parties who wish to exchange data, without allowing others to see the key
- Often secure systems fail due to a break in the key distribution scheme

Key Distribution

- For two parties A and B, key distribution can be achieved in a number of ways:

Key Distribution

- For two parties A and B, key distribution can be achieved in a number of ways:
  1. A can select key and physically deliver to B
  2. A third party can select & physically deliver key to A & B
  3. If A and B have communicated previously, A can transmit the new key to B, encrypted using the old key.

Key Distribution

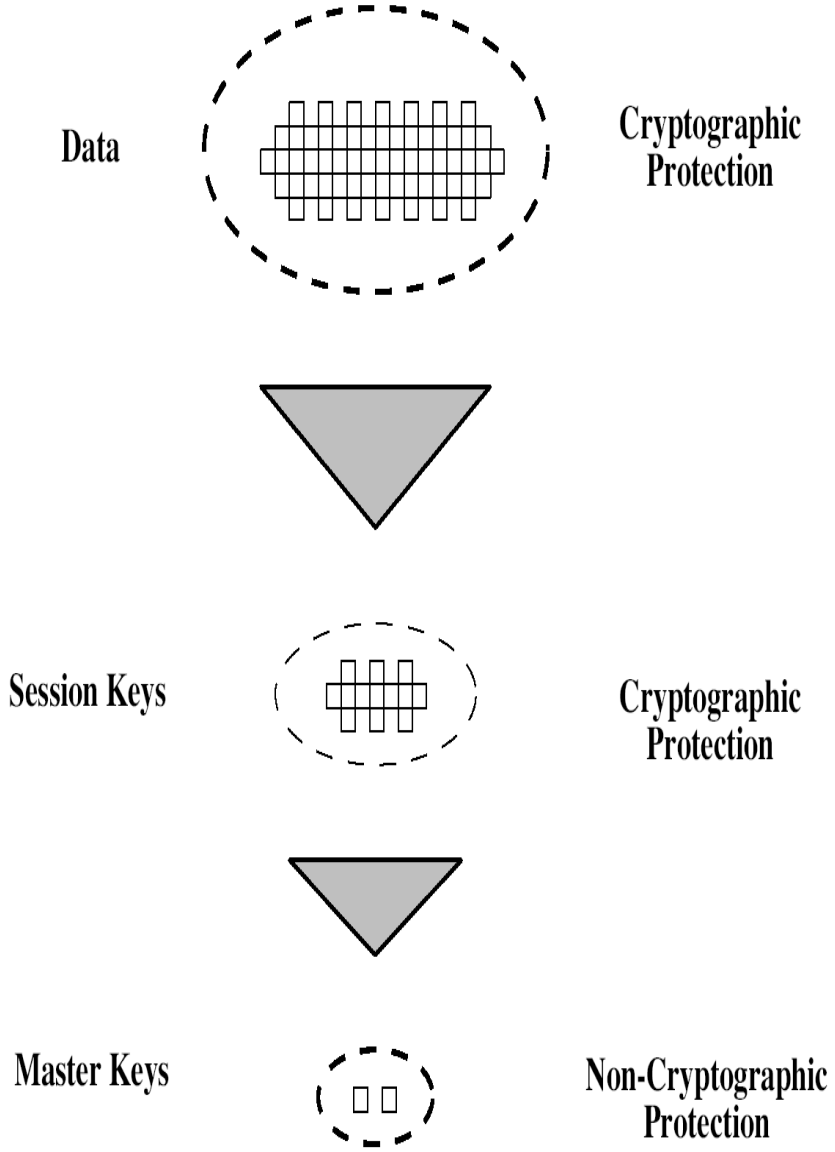
- For two parties A and B, key distribution can be achieved in a number of ways:
  1. A can select key and physically deliver to B
  2. A third party can select & physically deliver key to A & B
  3. If A and B have communicated previously, A can transmit the new key to B, encrypted using the old key.
     - If an attacker succeeds in getting one key, then all subsequent keys will be revealed

Key Distribution

- For two parties A and B, key distribution can be achieved in a number of ways:
  4. If A & B have secure communications with a third party C, C can deliver a key on the encrypted links to A and B
     - A key distribution center is responsible for distributing keys to pairs of users.
     - Each user must share a unique key with the key distribution center for purpose of key distribution.

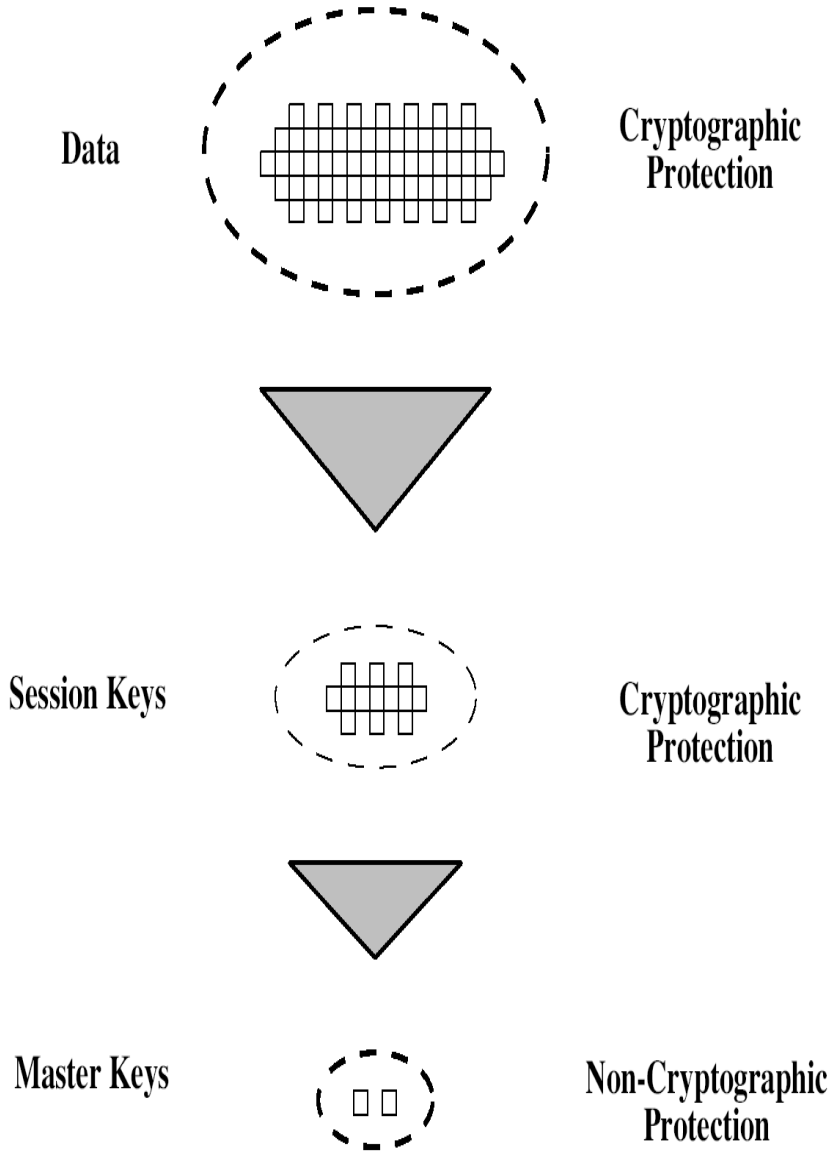
Key Hierarchy

- The use of a key distribution center is based on the use of a hierarchy of keys.
- Master key
  - Used to encrypt session keys
  - Shared by user & key distribution center
- Session key
  - Temporary key
  - Used for encryption of data between users



Key Hierarchy

- The use of a key distribution center is based on the use of a hierarchy of keys.
- Master key
  - Used to encrypt session keys
  - Shared by user & key distribution center
- Session key
  - Temporary key
  - Used for encryption of data between users

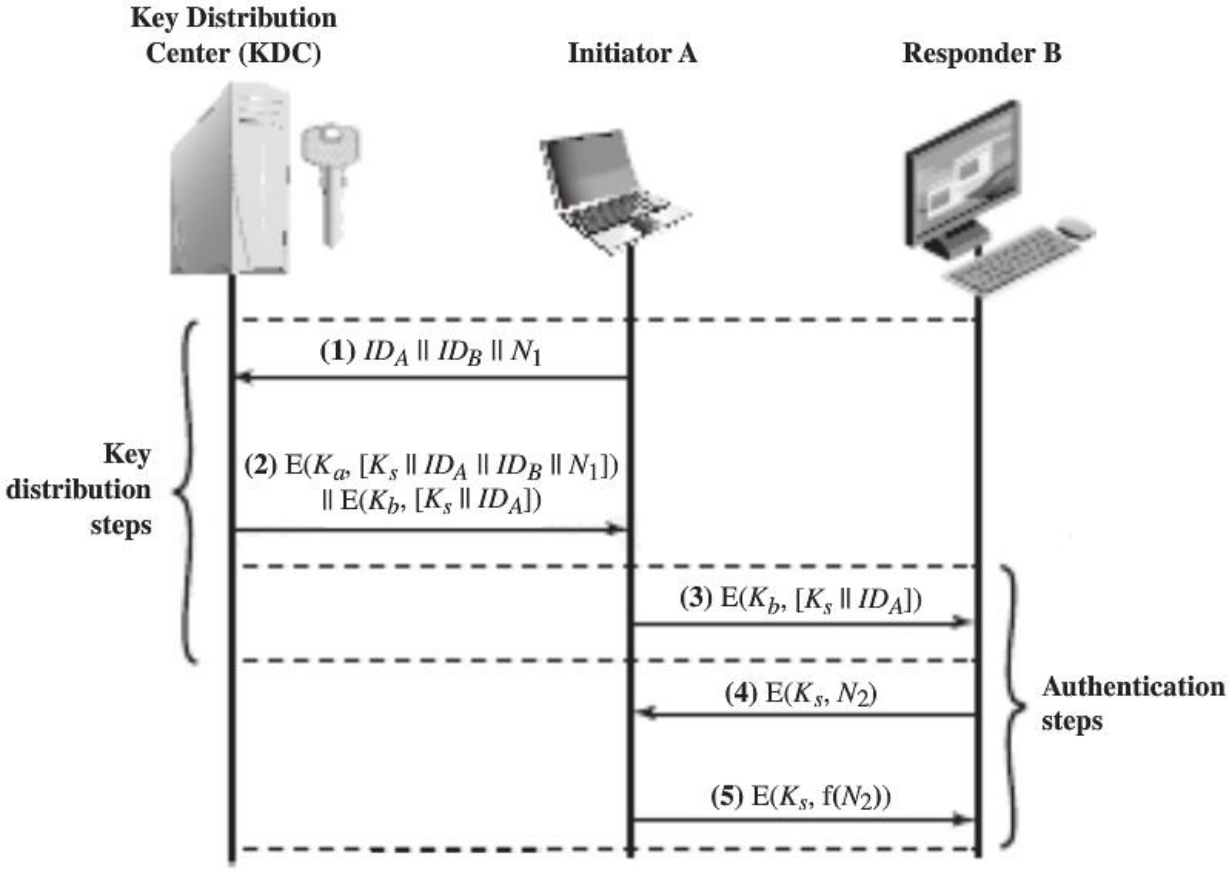


Symmetric Needham-Shroeder Key Distribution Protocol

- We will begin by studying the Needham–Schroeder Symmetric Key Protocol
- Securely distributes session keys to the two communicating parties
- Based on the concept of the symmetric key hierarchy
- Forms the basis of the widely-used Kerberos protocol

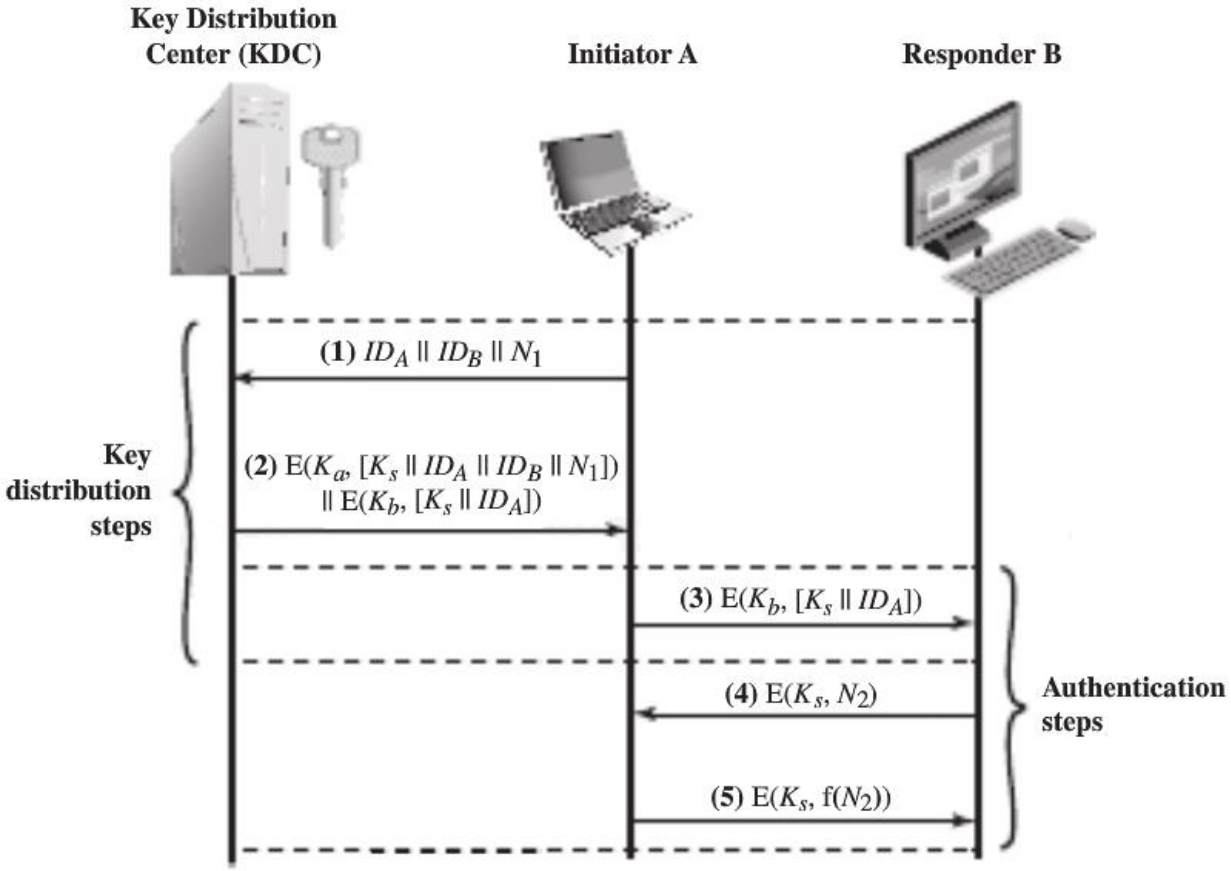
Symmetric Needham-Shroeder Key Distribution

- A wishes to establish a logical connection with B and requires a one-time session key to protect the data transmitted over the connection
- A shares the master key K_a with the KDC
- B shares the master key K_b with the KDC



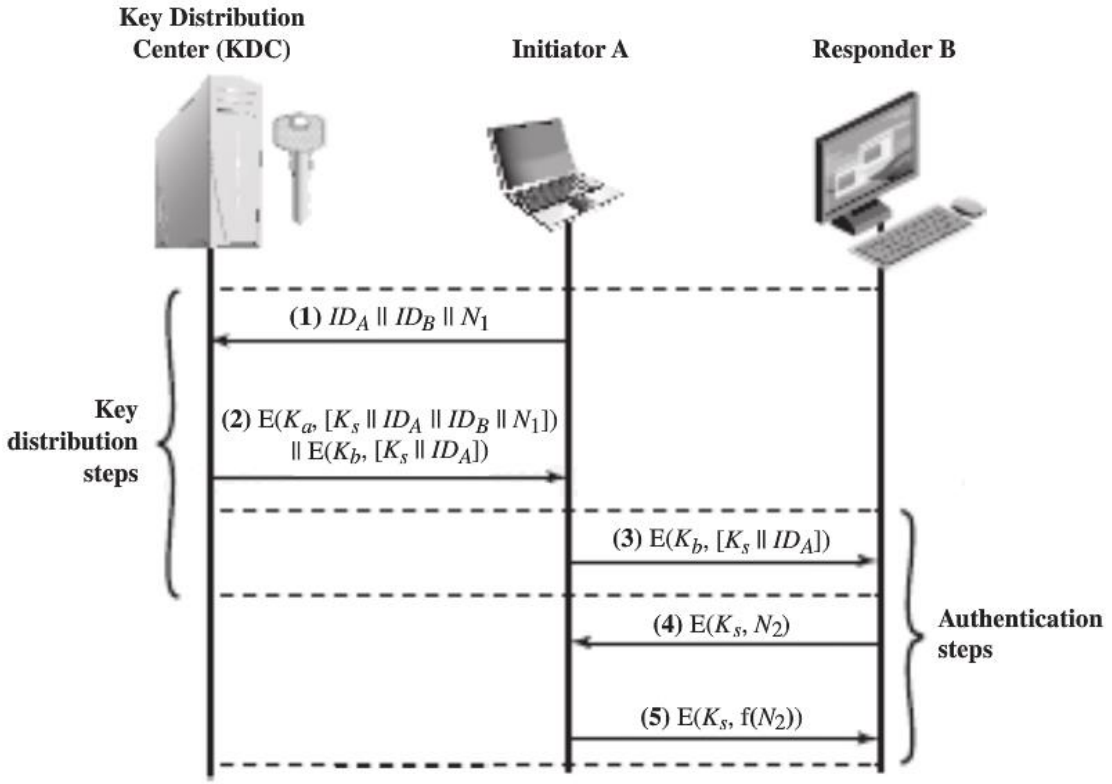
Symmetric Needham-Shroeder Key Distribution

- Msg1: A issues a request to the KDC for a session key to protect a connection to B. The message includes the identity of A and B, and a unique identifier, N1 (nonce).
- Nonce: a random number that is used to demonstrate the freshness of a session – prevent replay attack



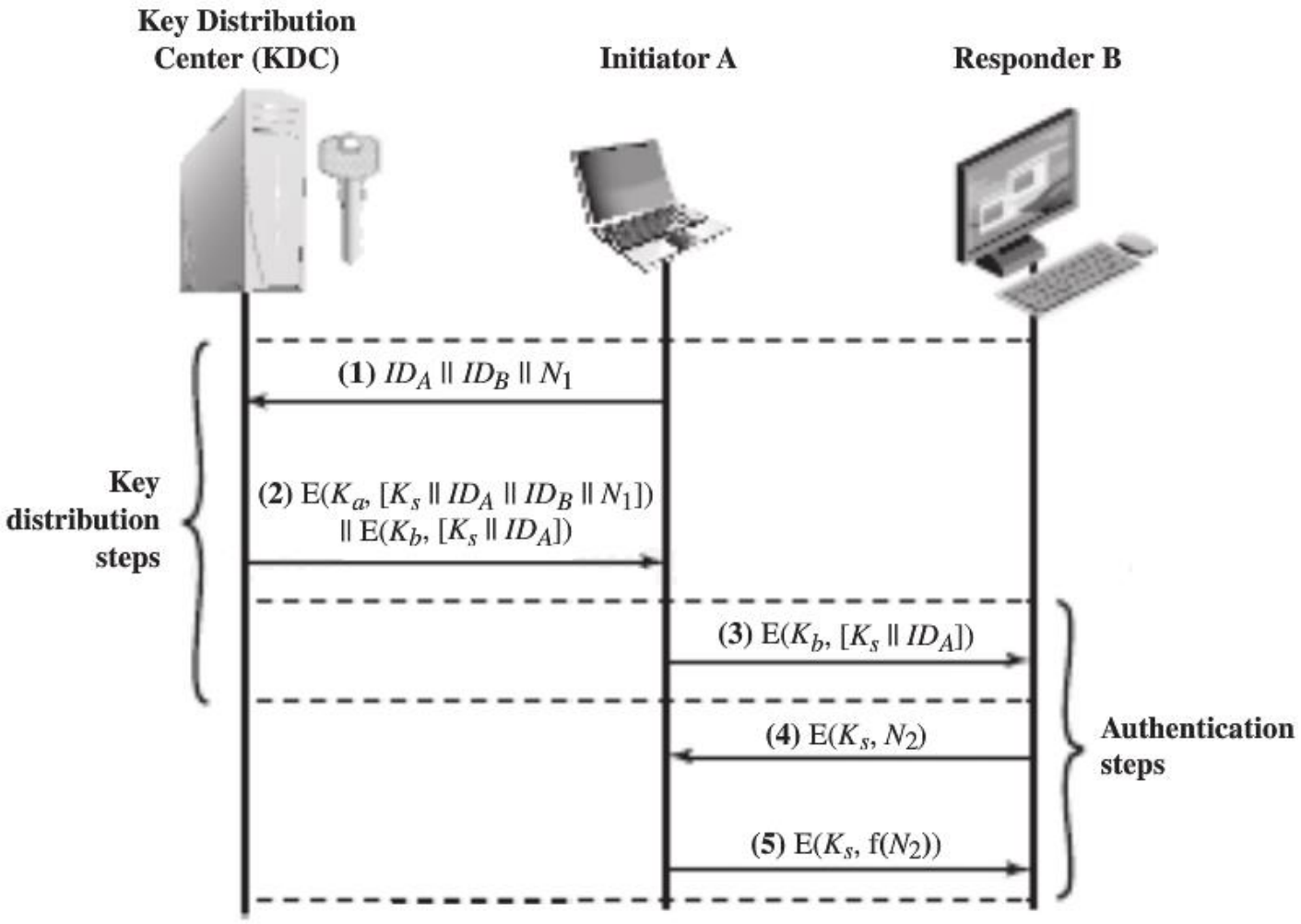
Symmetric Needham-Shroeder Key Distribution

- Msg2: The KDC responds with a message encrypted using K_a
  1. The one-time session key K_s
  2. The original request message and the nonce
  3. Two items for B, encrypted using K_b: the one-time session key K_s and an identity of A, ID_A.



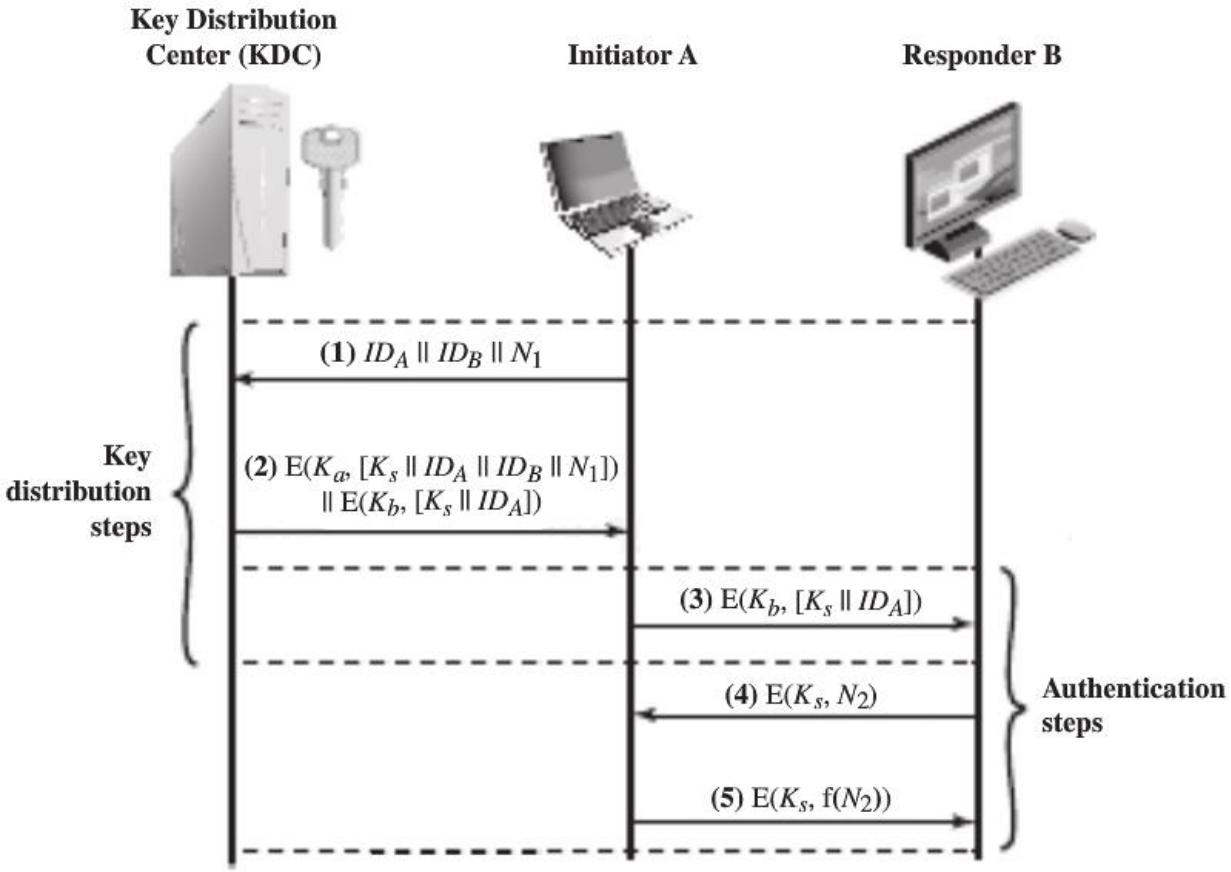
Symmetric Needham-Shroeder Key Distribution

- Msg3: A stores the session key for use in the upcoming session and forward to B the information that originated at the KDC for B, E(Kb,[Ks,IDA]).



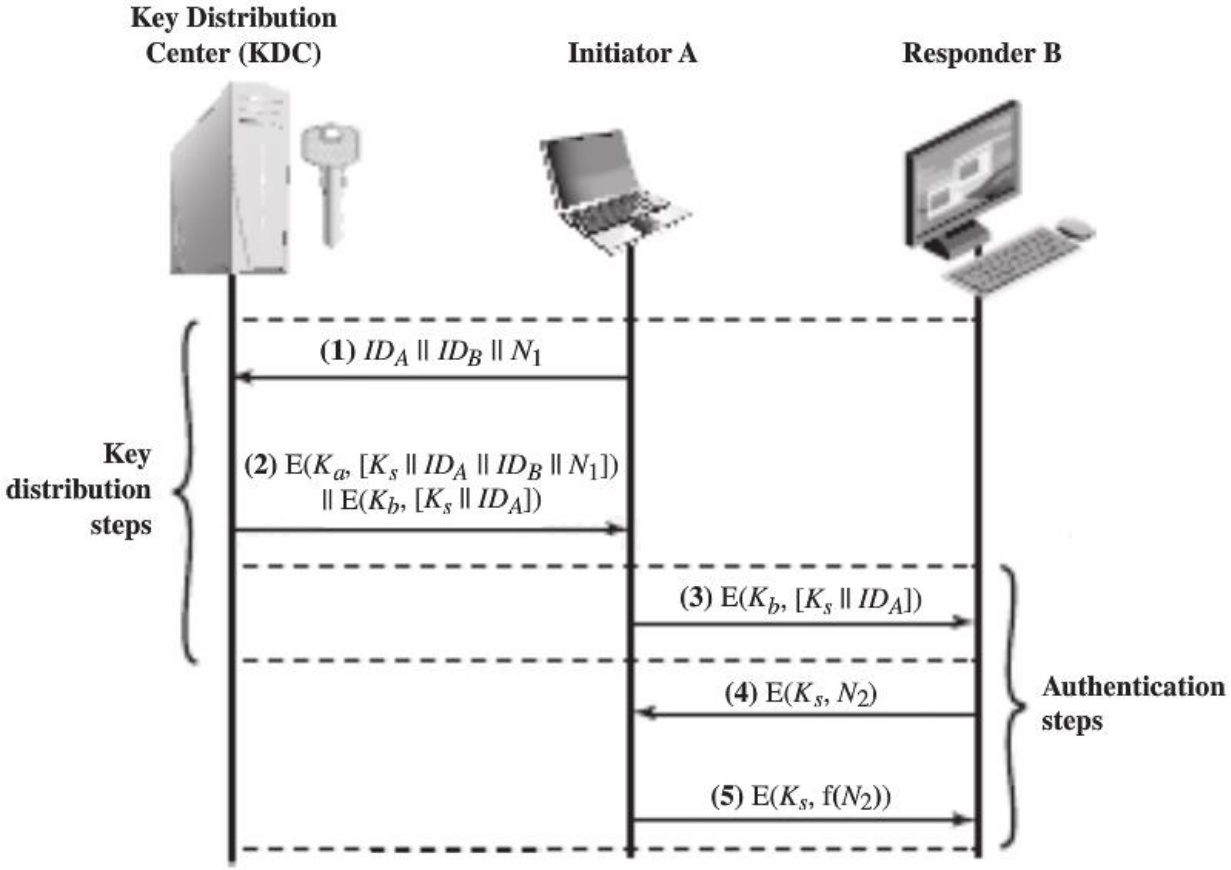
Symmetric Needham-Shroeder Key Distribution

- Now, a session key has been securely delivered to A and B.
- Msg4: B sends a nonce N_2 to A using the newly minted session key.
- Msg5: Also using K_s, A responds with $f(N_2)$, where f is a function that perform some transformation on N_2



Symmetric Needham-Shroeder Key Distribution

- Now, a session key has been securely delivered to A and B.
- Msg4: B sends a nonce N_2 to A using the newly minted session key.
- Msg5: Also using K_s, A responds with $f(N_2)$, where f is a function that perform some transformation on N_2



Attack: Symmetric Needham-Schroeder Protocol

3. A -> B: $E_{Kb}[Ks||ID_A]$
4. B -> A: $E_{Ks}[N_2]$
5. A -> B: $E_{Ks}[f(N_2)]$

- Suppose that an attacker X has been able to compromise an old session key.

Attack: Symmetric Needham-Schroeder Protocol

3. A -> B: $E_{Kb}[K_s||ID_A]$

4. B -> A: $E_{Ks}[N_2]$

5. A -> B: $E_{Ks}[\mathrm{f}(N_2)]$

- Suppose that an attacker X has been able to compromise an old session key.
- X can impersonate A and trick B into using the old key by simply replaying step 3.
- Unless B remembers indefinitely all previous session keys used with A, B will be unable to determine that this is a replay.
- X then intercepts the step 4 and sends bogus messages to B that appear to B to come from A using an authenticated session key.

Attack: Symmetric Needham-Schroeder Protocol

- Use a timestamp T that assures A and B that the session key has only just been generated.
- Revised protocol:

1. A->KDC: $ID_A \| ID_B \| N_1$

2. KDC -> A: $E_{Ka}[Ks \| ID_B \| N_1 \| E_{Kb}[Ks \| ID_A \| T]]$

3. A -> B: $E_{Kb}[Ks \| ID_A \| T]$

4. B -> A: $E_{Ks}[N_2]$

5. A -> B: $E_{Ks}[f(N_2)]$

Timestamp

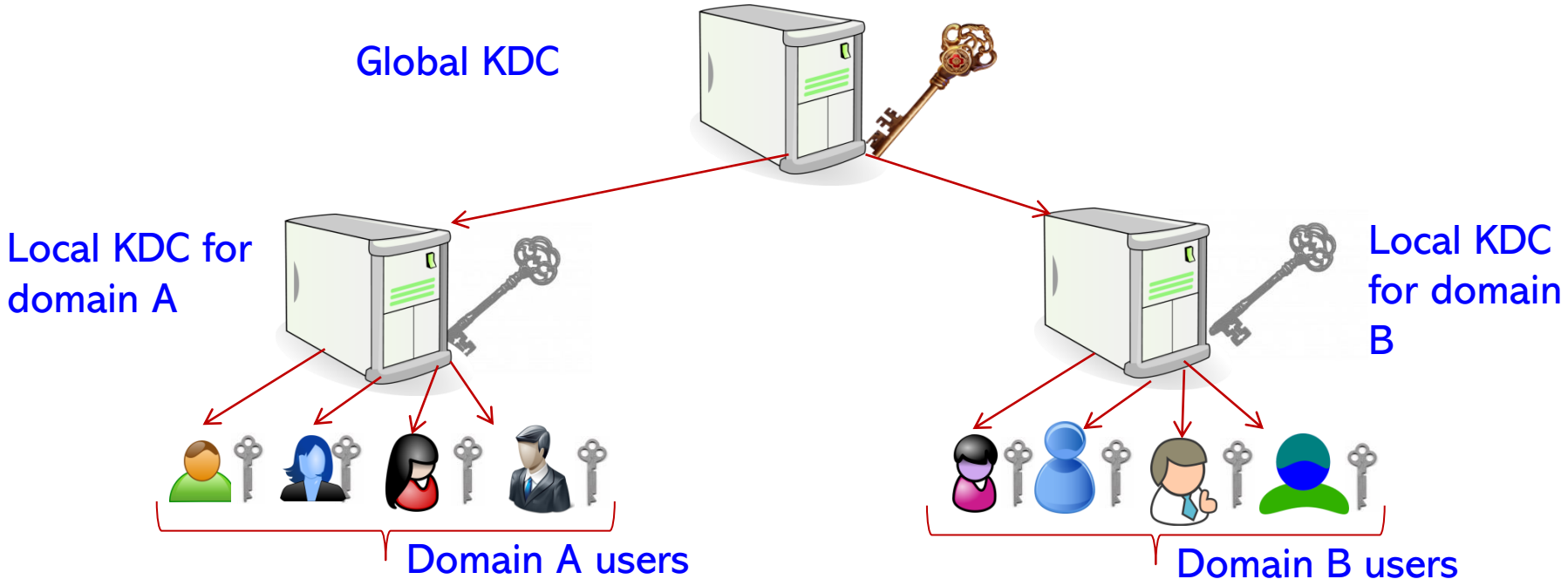
- Principals can verify the timeliness by checking:

$$|\text{Clock} - T| < \Delta t_1 + \Delta t_2$$

  - Δt_1: The estimated normal discrepancy between the KDC's clock and the local clock (principals' clock)
  - Δt_2: The expected network delay time
- Need to synchronize clock

Hierarchical Key Control

- It is not necessary to limit the key distribution function to a single KDC – for large networks, a hierarchy of KDCs can be established
  - E.g. local KDCs, each responsible for a small domain
  - If two entities are in different domains, then local KDCs can communicate through a global KDC.



Kerberos

Kerberos

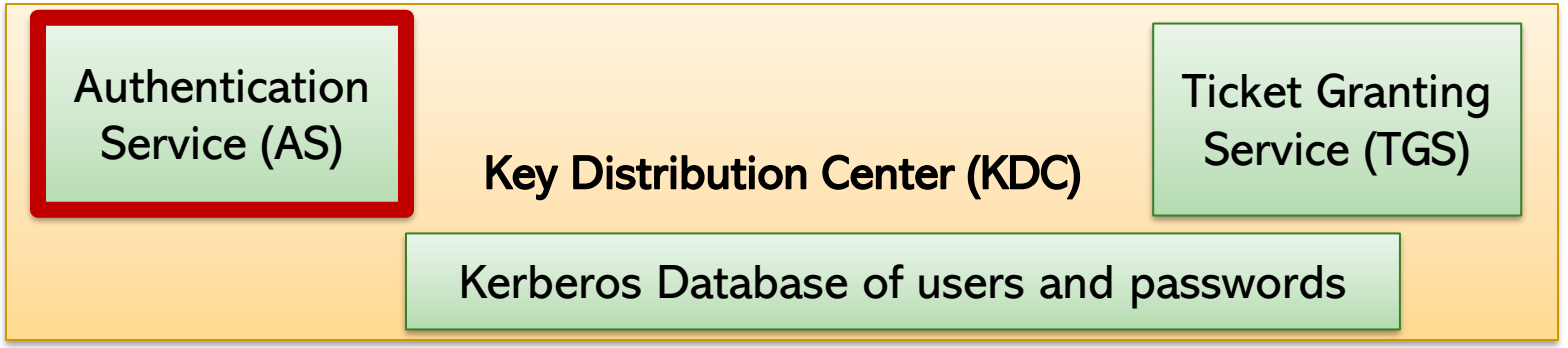
- Authentication protocol: widely used on Windows and Unix-based networks
- Enables authenticated users to securely and seamlessly use multiple network services with a single logon (e.g., email, printing, company portal, etc)
- Based on the concept of KDC
- Based completely on symmetric key cryptographic techniques

Kerberos Components

- Kerberos Client: a program that runs on the client system and interacts with the Kerberos infrastructure on behalf of the user.

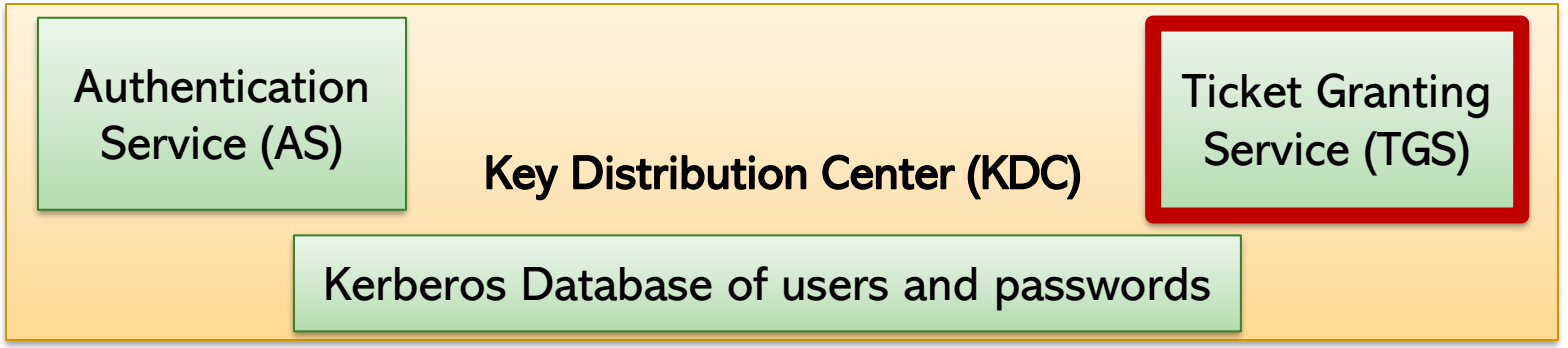
Kerberos Components

- Key Distribution Center: consists of
  - 1. Authentication Service (AS): authenticates users when they initially attempt to login (using username and password)



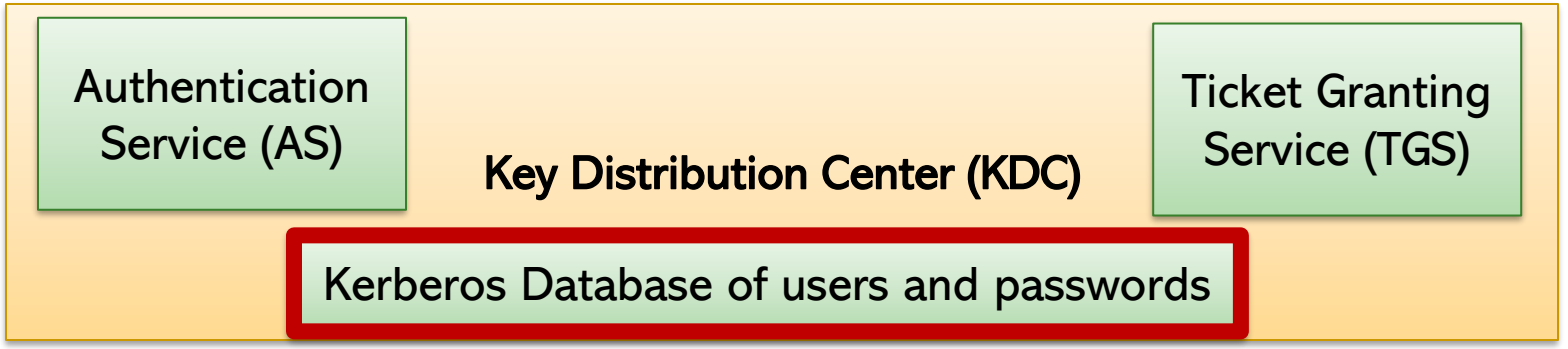
Kerberos Components

- Key Distribution Center: consists of
  - 2. Ticket Granting Service (TGS): grants users tickets for accessing a specific Kerberos-enabled service



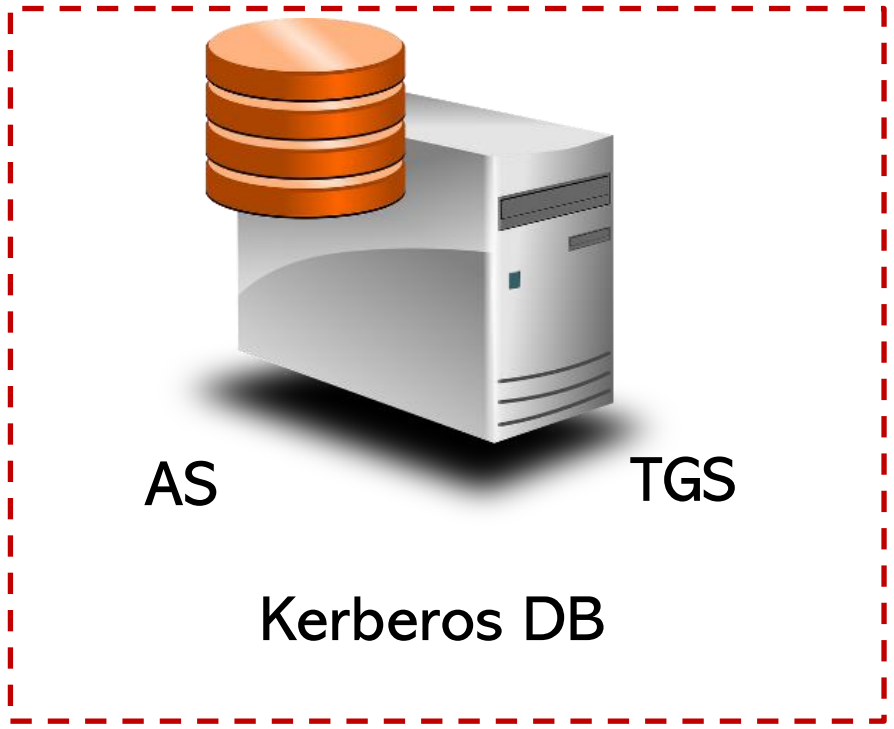
Kerberos Components

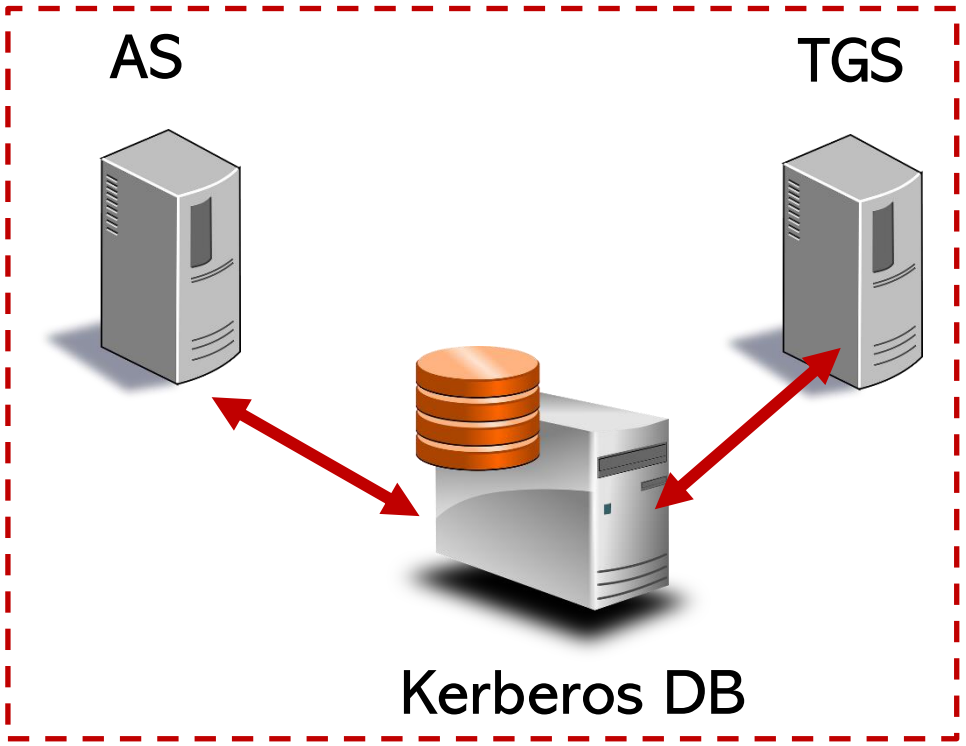
- Key Distribution Center:
  - …also, hosts the database of IDs and passwords of authorized users



Kerberos Components

- Key Distribution Center: the different components can be hosted on either the same physical server, or multiple physical systems – implementations can vary





Kerberos Components

- Service Server: Provides a specific type of service.

Service Server

Kerberos Components

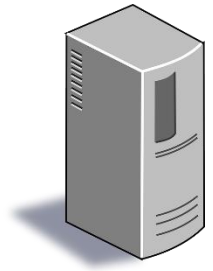
- Let's take a look at the basic Kerberos flow!

Kerberos Basic Idea

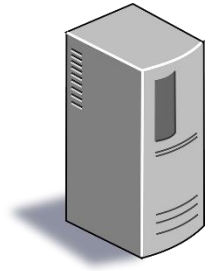
1. Ticket Granting Ticket (TGT) Request

KDC

Authentication Service (AS)



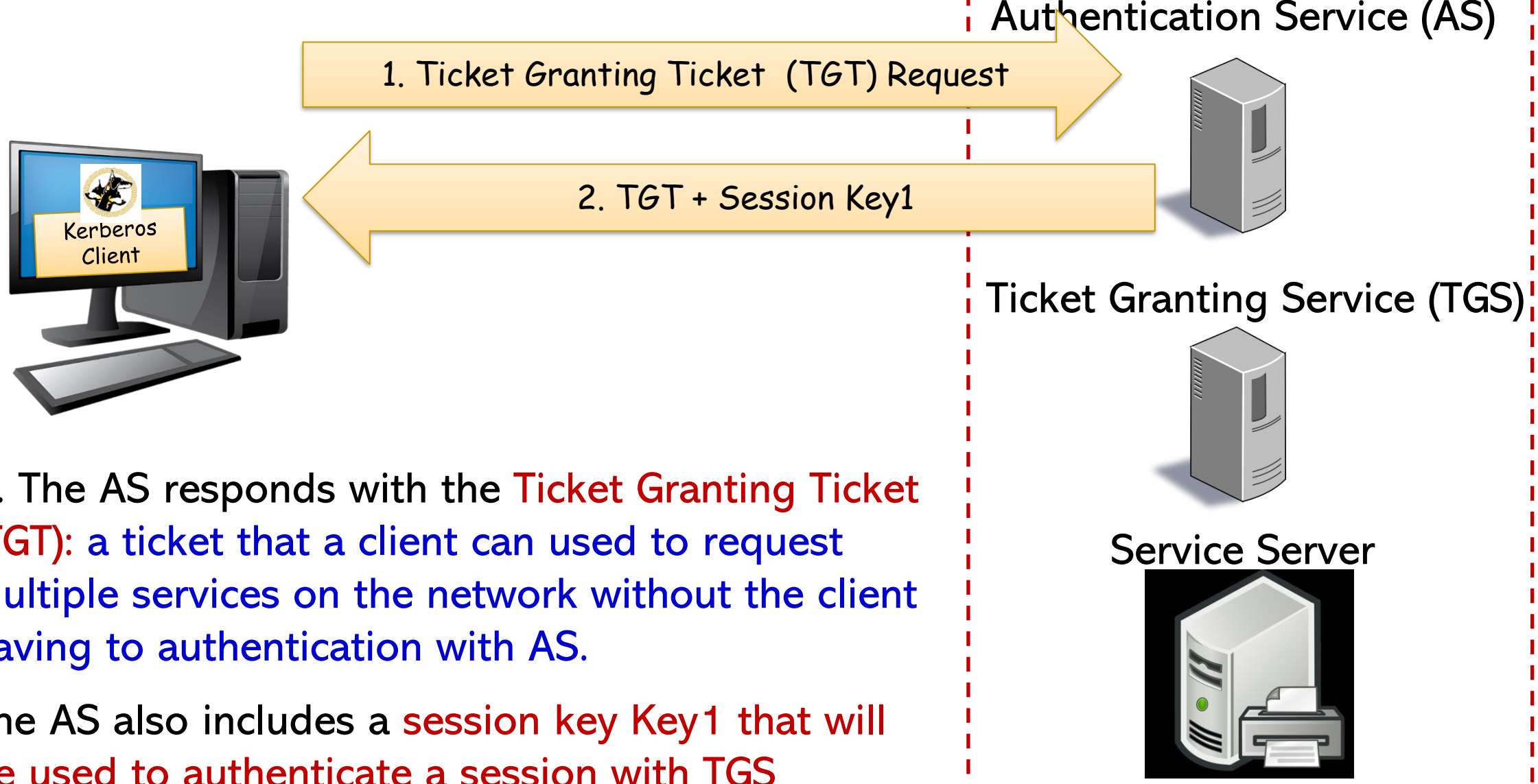
Ticket Granting Service (TGS)



Service Server

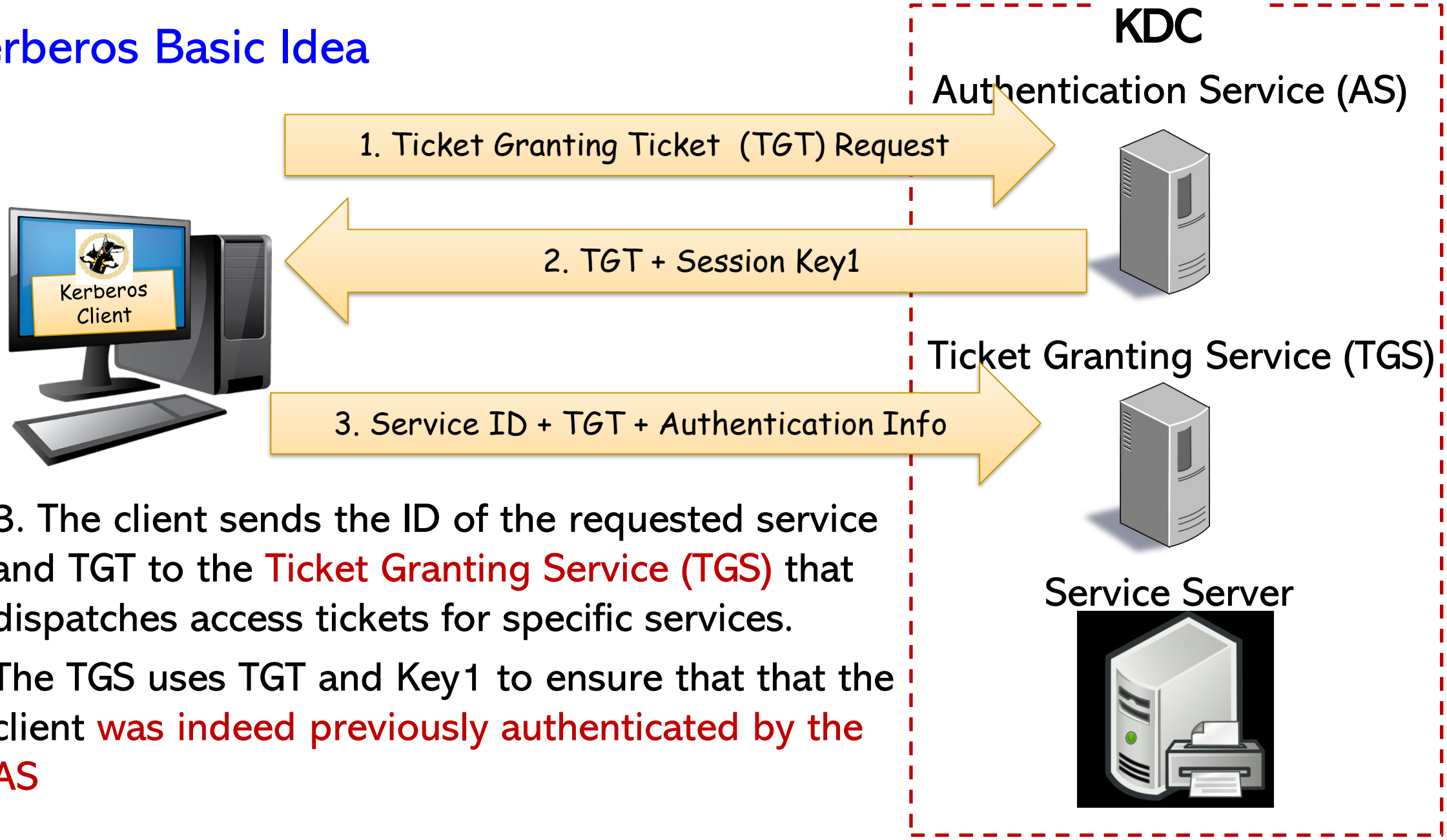
- 1. The client wishing to receive a network service authenticates with the AS

Kerberos Basic Idea



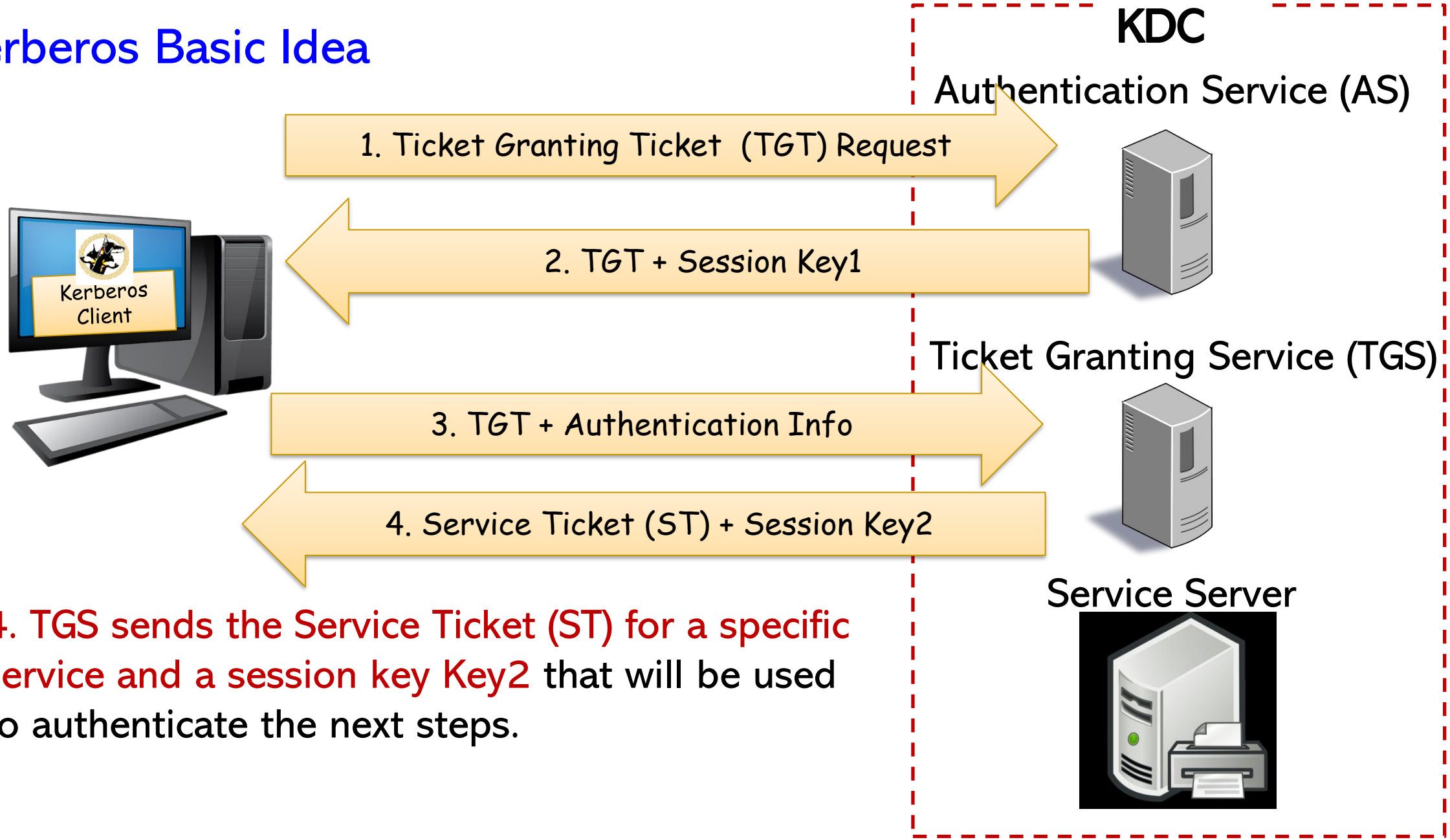
- 2. The AS responds with the Ticket Granting Ticket (TGT): a ticket that a client can used to request multiple services on the network without the client having to authentication with AS.
- The AS also includes a session key Key1 that will be used to authenticate a session with TGS

Kerberos Basic Idea



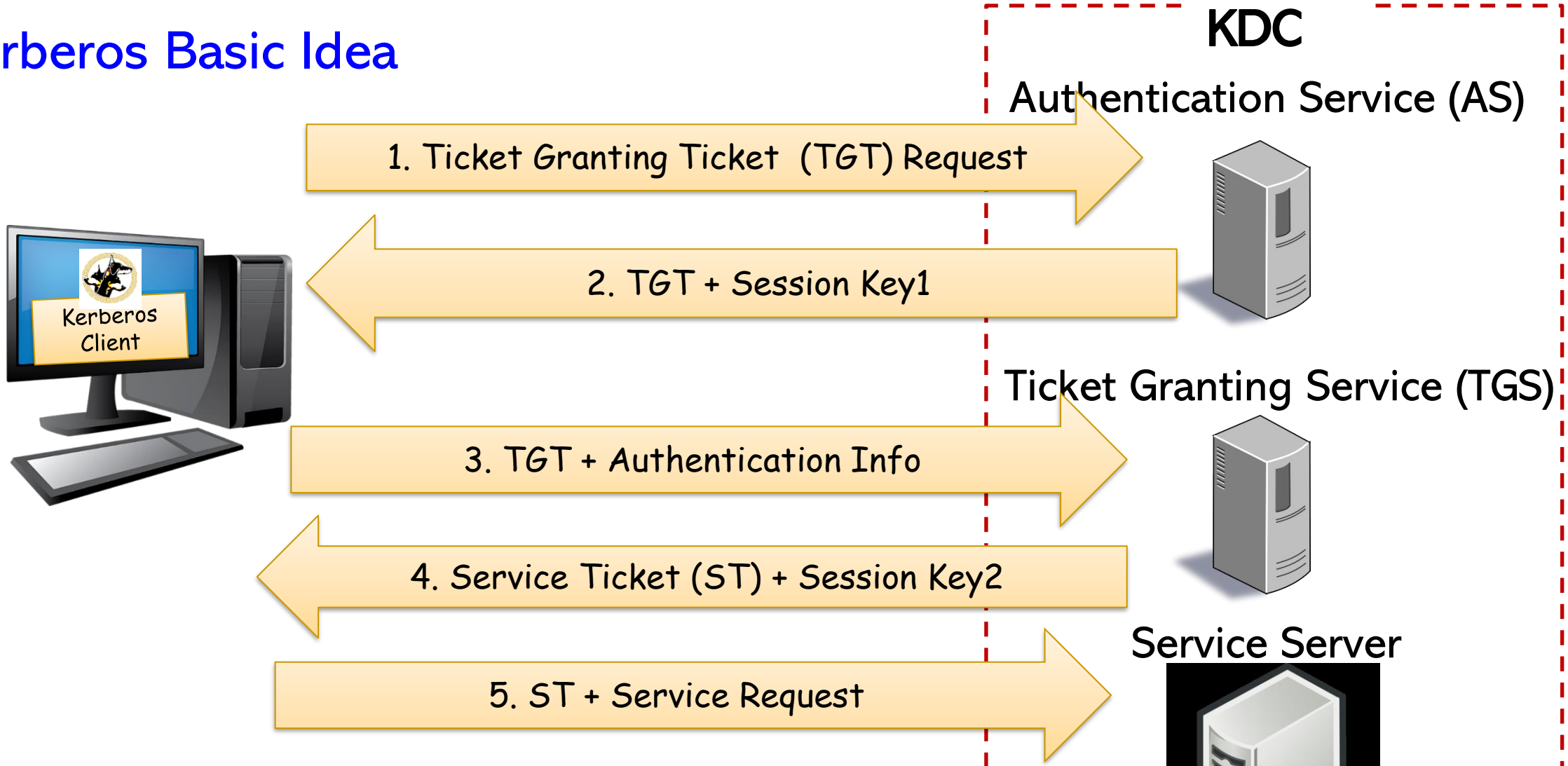
- 3. The client sends the ID of the requested service and TGT to the Ticket Granting Service (TGS) that dispatches access tickets for specific services.
- The TGS uses TGT and Key1 to ensure that that the client was indeed previously authenticated by the AS

Kerberos Basic Idea



- 4. TGS sends the Service Ticket (ST) for a specific service and a session key Key2 that will be used to authenticate the next steps.

Kerberos Basic Idea



- 5. The client presents the ST to the Service Server (SS) to request a specific service. The ST is authenticated using Key Key2.

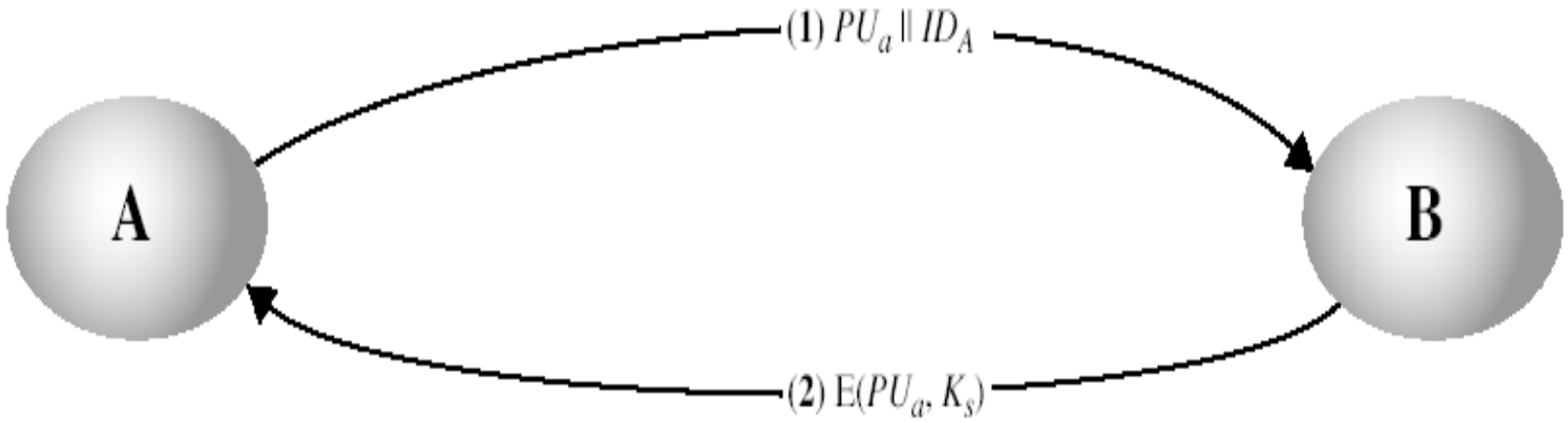
Distribution of Secret Keys Using Public-Key Cryptography

Distribution of Secret Keys Using Public-Key Cryptography

- Use previous methods to obtain public-key
- Can use for secrecy or authentication
- But public-key algorithms are slow
- So usually want to use symmetric-key encryption to protect message contents - need to distribute the session key
- Public-key cryptography can be used to distribute the session keys.

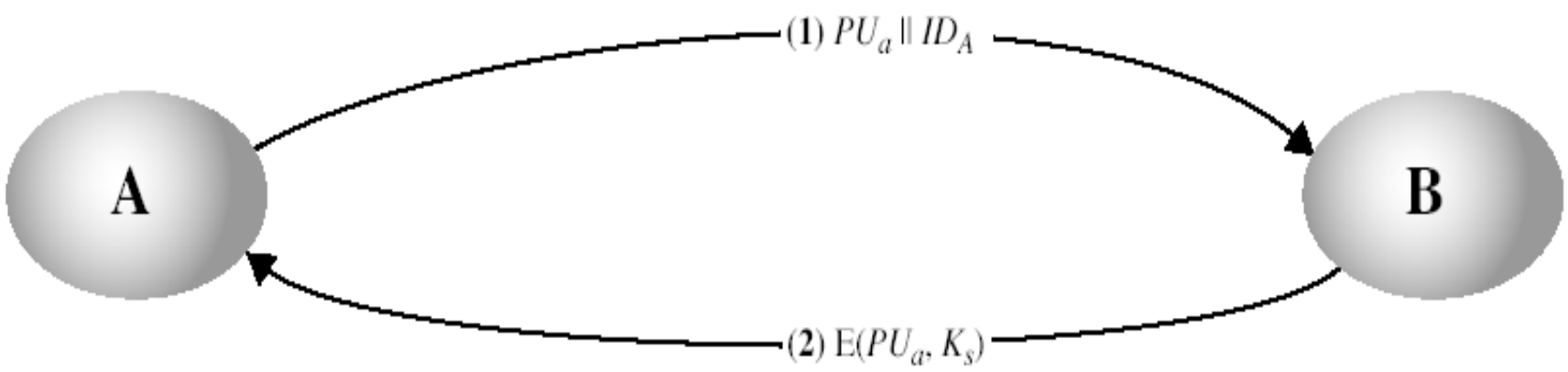
Simple Secret Key Distribution

- Proposed by Merkle in 1979
  1. A generates a new temporary public key pair
  2. A sends B the public key and their identity
  3. B generates a session key K, sends it to A encrypted using the supplied public key
  4. A decrypts the session key



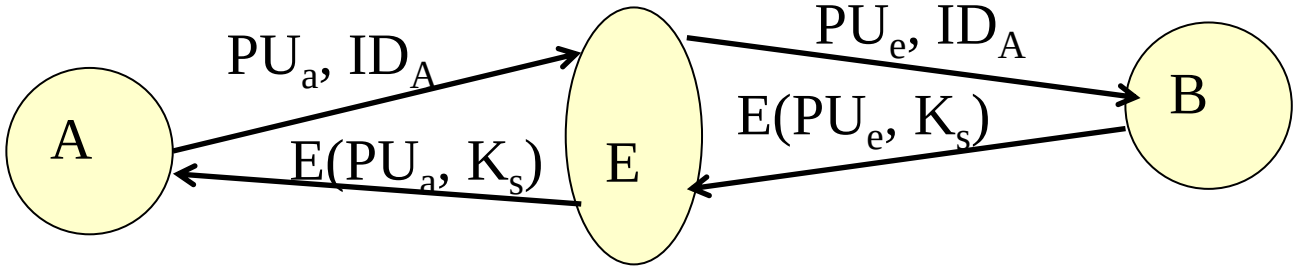
Simple Secret Key Distribution

- Proposed by Merkle in 1979
  5. A discards the public key pair and B discards A' public key
  6. At the completion of the exchange, A and B discard the session key



Simple Secret Key Distribution: Attack

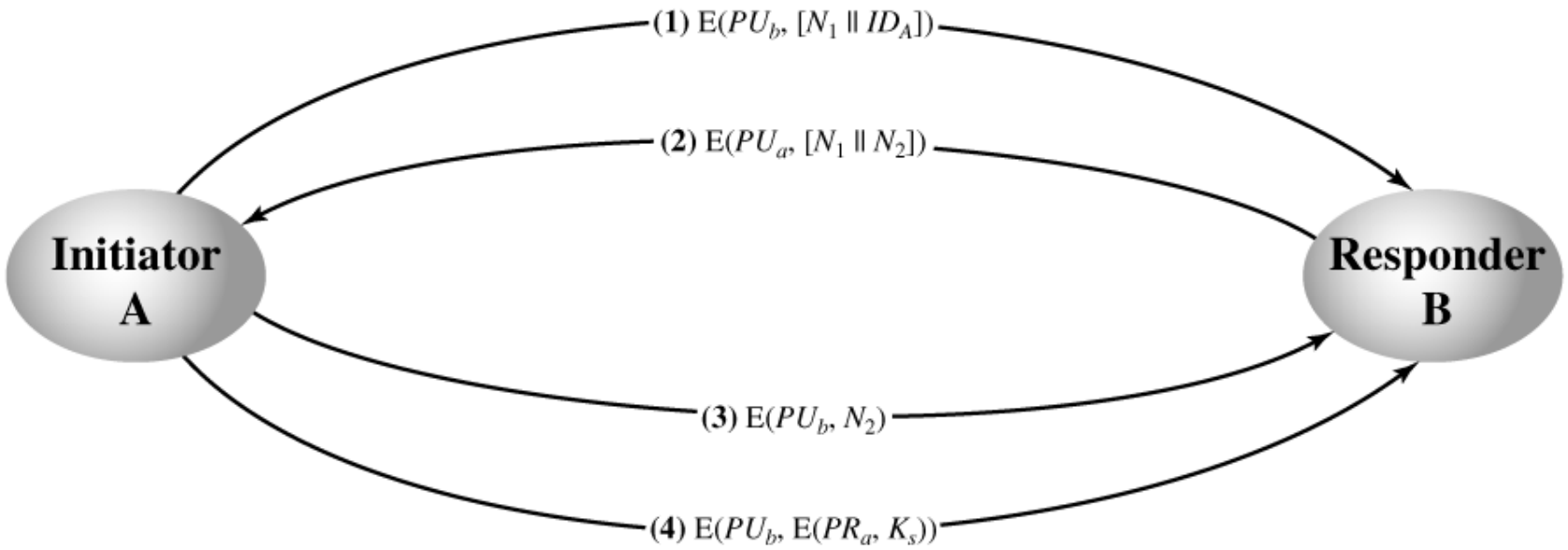
- Problem: man-in-the-middle attack – an adversary can intercept messages and then replay the intercepted message or send another message.



- A transmits a message intended for B consisting of PU_a and A's identifier ID_A
- E intercepts the message, creates its own public/private key pair $\{PU_e, PR_e\}$ and transmits $PU_e \| ID_A$ to B
- B generates a secret key K_s and transmits $E(PU_e, K_s)$
- E intercepts the message, and learns Ks by computing $D(PR_e, E(PU_e, K_s))$
- E transmits $E(PU_a, K_s)$ to A
- Both A and B know K_s and are unaware that K_s has been revealed to E.

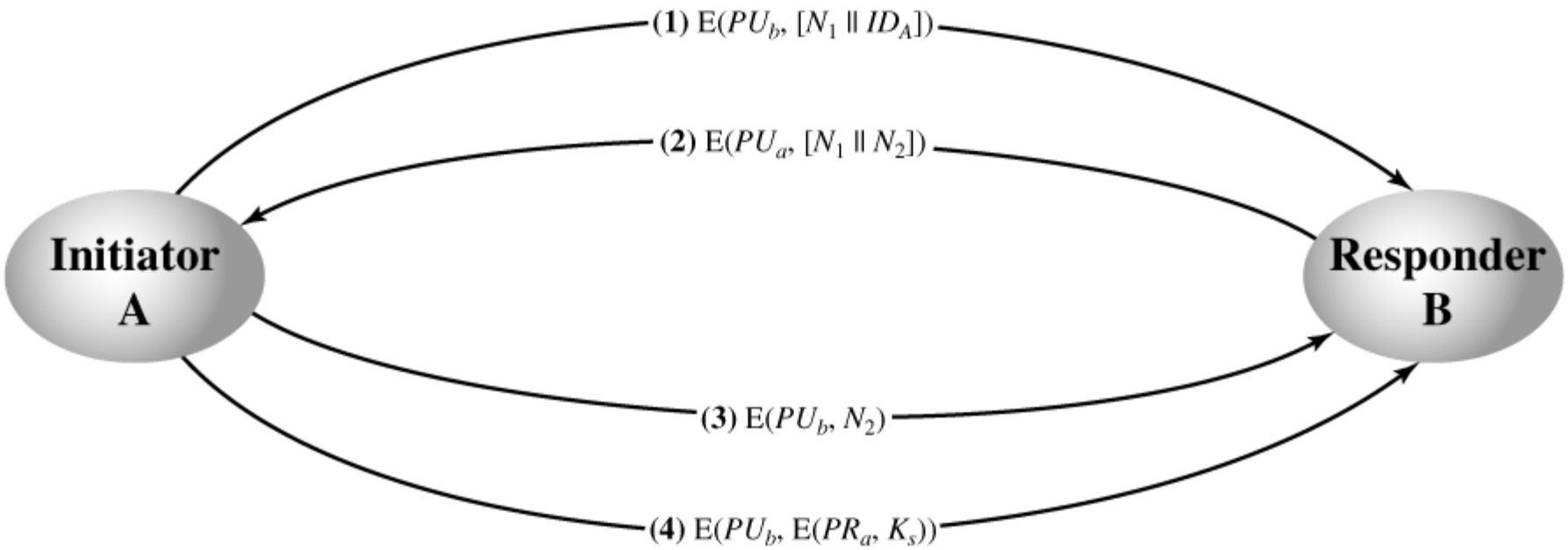
Simple Secret Key Distribution: Solution

1. A uses B's public key to encrypt a message to B containing ID_A and a nonce N_1
2. B sends a message to A encrypted with PU_a and containing A's nonce N_1 and a new nonce generated by B (N_2). Because only B could have decrypted message 1, N_1 in message 2 assures A that the correspondent is B



Simple Secret Key Distribution: Solution

3. A returns N_2 encrypted using B's public key, to assure B that its correspondent is A
4. A selects a secret key K_s and sends $M = E(PU_b, E(PR_a,K_s))$ to B. Encryption with PU_b ensures that only B can read it; encryption with PR_a ensures that only A can send it
5. B computes $D(PU_a,D(PR_b,M))$ to recover the secret key



Hybrid Key Distribution

- Session keys may change frequently.
- **Problem:** Distribution of session keys by public-key encryption could degrade overall system performance because of the relatively high computational load of public-key encryption/decryption

Hybrid Key Distribution

- Session keys may change frequently.
- Distribution of session keys by public-key encryption could degrade overall system performance because of the relatively high computational load of public-key encryption/decryption

- **Solution:** Hybrid key distribution
  - Retain the use of a key distribution center (KDC) that shares a secret master key with each user and distributes secret session keys encrypted with the master key.
  - A public-key is used to distribute master keys - master keys are usually updated occasionally

Symmetric Key Exchange using Diffie-Hellman

Diffie-Hellman Key Exchange

- The first public-key algorithm proposed by Diffie & Hellman in 1976
- The purpose is to enable two users to securely exchange a key that can then be used for subsequent encryption of messages.
- Used in a number of commercial products

Diffie-Hellman Key Exchange

- First, we will review some background regarding the Euler Totient function

Euler Totient Function $\varnothing(n)$

- Euler Totient Function $\varnothing(n)$: the number of positive integers less than n and relatively prime to n.
  - m is a relatively prime to n if $\gcd(m,n)=1$
  - $\varnothing(37)$

Euler Totient Function $\varnothing(n)$

- Euler Totient Function $\varnothing(n)$: the number of positive integers less than n and relatively prime to n.
  - m is a relatively prime to n if $\gcd(m,n)=1$
  - $\varnothing(37) = 36$: all integers from 1 through 36 are relatively prime to 37
  - $\varnothing(35)$

Euler Totient Function ø(n)

- Euler Totient Function ø(n): the number of positive integers less than n and relatively prime to n.
  - m is a relatively prime to n if gcd(m,n)=1
  - ø(37) = 36: all integers from 1 through 36 are relatively prime to 37.
  - ø(35) = 24:
    - 1, 2, 3, 4, 6, 8, 9, 11, 12, 13, 16, 17, 18, 19, 22, 23, 24, 26, 27, 29, 31, 32, 33, 34.
  - For a prime number p, ø(p)

Euler Totient Function ø(n)

- Euler Totient Function ø(n): the number of positive integers less than n and relatively prime to n.
  - m is a relatively prime to n if gcd(m,n)=1
  - ø(37) = 36: all integers from 1 through 36 are relatively prime to 37.
  - ø(35) = 24:
    - 1, 2, 3, 4, 6, 8, 9, 11, 12, 13, 16, 17, 18, 19, 22, 23, 24, 26, 27, 29, 31, 32, 33, 34.
  - For a prime number p, ø(p) = p-1

Euler Totient Function $\varnothing(n)$

- Two prime numbers p and q with $p \neq q$, then

$$\varnothing(pq) = \varnothing(p)*\varnothing(q) = (p-1)*(q-1)$$

  - The set of integers less than pq is $\{1, 2, \ldots, pq-1\}$

Euler Totient Function ø(n)

- Two prime numbers p and q with $p \neq q$, then

$$ø(pq) = ø(p)*ø(q) = (p-1)*(q-1)$$

  - The set of integers less than n = pq is {1, 2, …, pq-1}
  - The integers in this set that are not relatively prime to n: {p, 2p, …., (q-1)p} and {q, 2q, …, (p-1)q}

Euler Totient Function ø(n)

- Two prime numbers p and q with $p \neq q$, then

$$ø(pq) = ø(p)*ø(q) = (p-1)*(q-1)$$

  - The set of integers less than n = pq is {1, 2, …, pq-1}
  - The integers in this set that are not relatively prime to n: {p, 2p, …., (q-1)p} and {q, 2q, …, (p-1)q}
  - Example: n = 21
    - pq = 3 * 7
    - p – 1 = 3 – 1 = 2; q – 1 = 7 – 1 = 6
    - Set of integers less than 21 {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}
    - {p…(q-1)p} =

Euler Totient Function ø(n)

- Two prime numbers p and q with $p \neq q$, then

$$ø(pq) = ø(p)*ø(q) = (p-1)*(q-1)$$

  - The set of integers less than n = pq is {1, 2, …, pq-1}
  - The integers in this set that are not relatively prime to n: {p, 2p, …., (q-1)p} and {q, 2q, …, (p-1)q}
  - Example: n = 21
    - pq = 3 * 7
    - p – 1 = 3 – 1 = 2; q – 1 = 7 – 1 = 6
    - Set of integers less than 21 {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}
    - {p…(q-1)p} = {1*3, 2*3, 3*3, 4*3, 5*3, 6*3}
      = {3, 6, 9, 12, 15, 18}

Euler Totient Function ø(n)

- Two prime numbers p and q with $p \neq q$, then

$$ø(pq) = ø(p)*ø(q) = (p-1)*(q-1)$$

  - The set of integers less than n = pq is {1, 2, …, pq-1}
  - The integers in this set that are not relatively prime to n:{p, 2p, …., (q-1)p} and {q, 2q, …, (p-1)q}
  - Example: n = 21
    - pq = 3 * 7
    - p – 1 = 3 – 1 = 2; q – 1 = 7 – 1 = 6
    - Set of integers less than 21 {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}
    - {p…(q-1)p} = {3*1, 3*2, 3*3, 3*4, 3*5, 3*6} = {3, 6, 9, 12, 15, 18}
    - {q…(p-1)q} = {7*1, 7*2} = {7, 14}

Euler Totient Function ø(n)

- Two prime numbers p and q with $p \neq q$, then

$$ø(pq) = ø(p)*ø(q) = (p-1)*(q-1)$$

  - The set of integers less than n = pq is {1, 2, …, pq-1}
  - The integers in this set that are not relatively prime to n: {p, 2p, …., (q-1)p} and {q, 2q, …, (p-1)q}
  - Example: n = 21
    - pq = 3 * 7
    - p – 1 = 3 – 1 = 2; q – 1 = 7 – 1 = 6
    - Set of integers less than 21 {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}
    - How many numbers less than 21 are relatively prime to 21?

Euler Totient Function ø(n)

- Two prime numbers p and q with $p \neq q$, then

$$ø(pq) = ø(p)*ø(q) = (p-1)*(q-1)$$

  - The set of integers less than pq is {1, 2, …, pq-1}
  - The integers in this set that are not relatively prime to p*q:{p, 2p, …., (q-1)p} and {q, 2q, …, (p-1)q}

$$ø(pq) = (pq - 1) – [(q-1) + (p-1)]$$
$$= pq – p – q +1$$
$$= (p-1) * (q-1)$$
$$= ø(p)*ø(q)$$

  - E.g. ø(21) = (3–1)*(7–1) = 2*6 = 12

Euler Totient Function ⌀(n)

- Generalized formula:

If the prime factorization of n is given by

$n = p_1^{e1} * ... * p_n^{en}$, then

$$\phi(n) = n * (1 - 1/p_1) * ... * (1 - 1/p_n).$$

Example: $\phi(64)$:

Prime factorization: $64 = 2^6$

$\phi(64) = 64 * (1 - 1/2) = 32$

Diffie-Hellman Key Exchange

- An integer a is a primitive root of a prime number q if $a \bmod q$, $a^2 \bmod q$, …, $a^{q-1} \bmod q$ are distinct and consist of the integers from 1 through $q-1$ in some permutation.
- There are two publicly known numbers:
  - A prime number q
  - An integer a that is the primitive root of q
- Each user generates his/her key
  - Chooses a private key (number): $x < q$
  - Computes their public key: $y = a^x \bmod q$
- Each user keeps the x value private and makes the y value available publicly

Diffie-Hellman Key Exchange

Property of modular arithmetic

$$[(a \bmod n) * (b \bmod n)] \bmod n = (a*b) \bmod n$$

- Shared session key for users A & B is K_{AB}:

$$K_{AB} = a^{x_A.x_B} \bmod q$$
$$= y_A^{x_B} \bmod q \quad \text{(which B can compute)}$$
$$= y_B^{x_A} \bmod q \quad \text{(which A can compute)}$$

- K_{AB} is used as session key in private-key encryption scheme between A and B
- Question: How to prove that $y_A^{x_B} \bmod q = y_B^{x_A} \bmod q$

Diffie-Hellman Key Exchange

Property of modular arithmetic

$$[(a \bmod n) * (b \bmod n)] \bmod n = (a*b) \bmod n$$

- Question: How to prove that $y_A^{x_B} \bmod q = y_B^{x_A} \bmod q$

$$y_A^{x_B} \bmod q$$

$$= (a^{x_A} \bmod q)^{x_B} \bmod q$$

$$= [(a^{x_A} \bmod q)* \ldots *(a^{x_A} \bmod q)] \bmod q$$

(x_B terms)

$$= a^{x_A x_B} \bmod q = a^{x_B x_A} \bmod q$$

$$= [(a^{x_B} \bmod q)* \ldots *(a^{x_B} \bmod q)] \bmod q$$

(x_A terms)

$$= y_B^{x_A} \bmod q$$

Because $y_A^{x_B} \bmod q = a^{x_A . x_B} \bmod q$

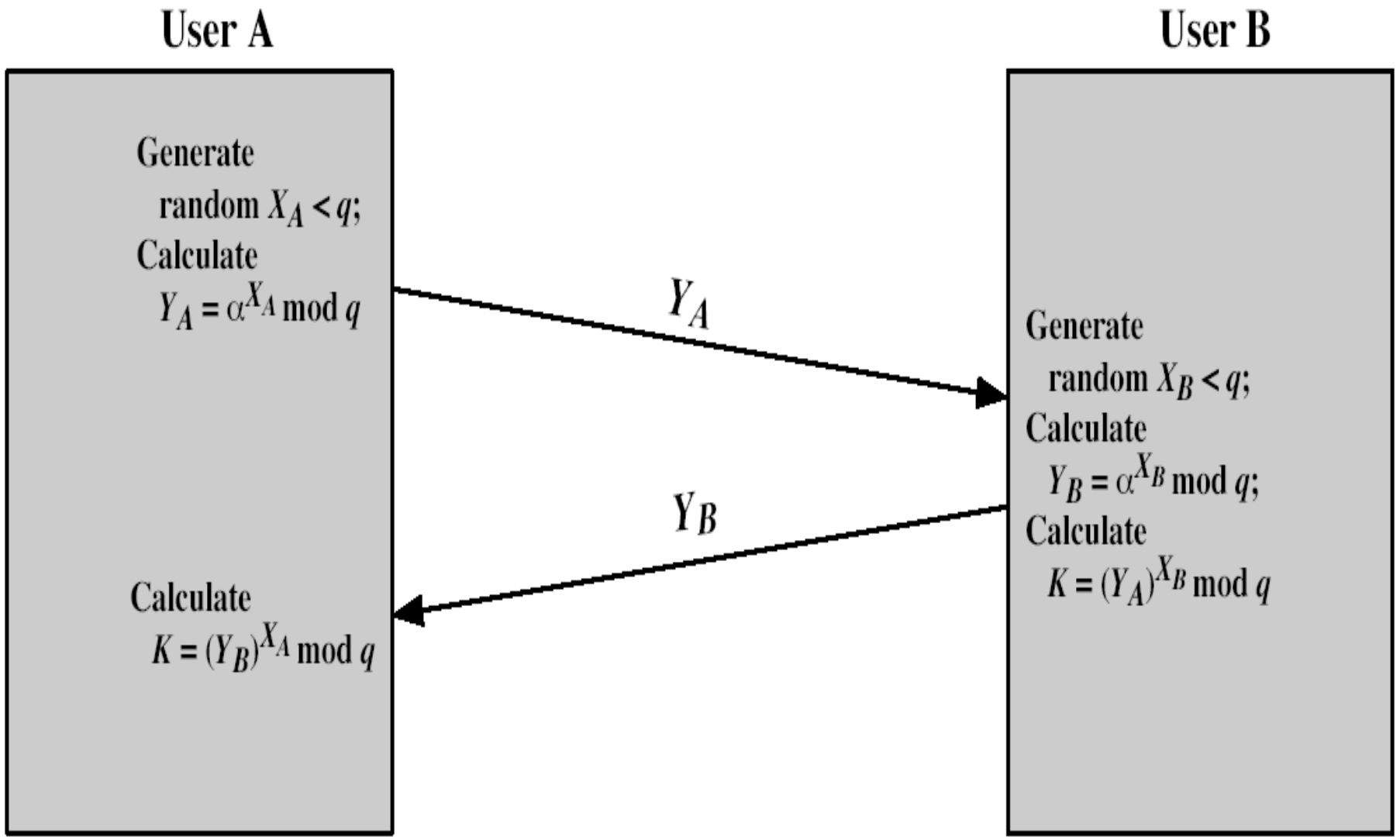
Law of mods:

$$[(a \bmod n) * (b \bmod n)] \bmod n = (a*b) \bmod n$$

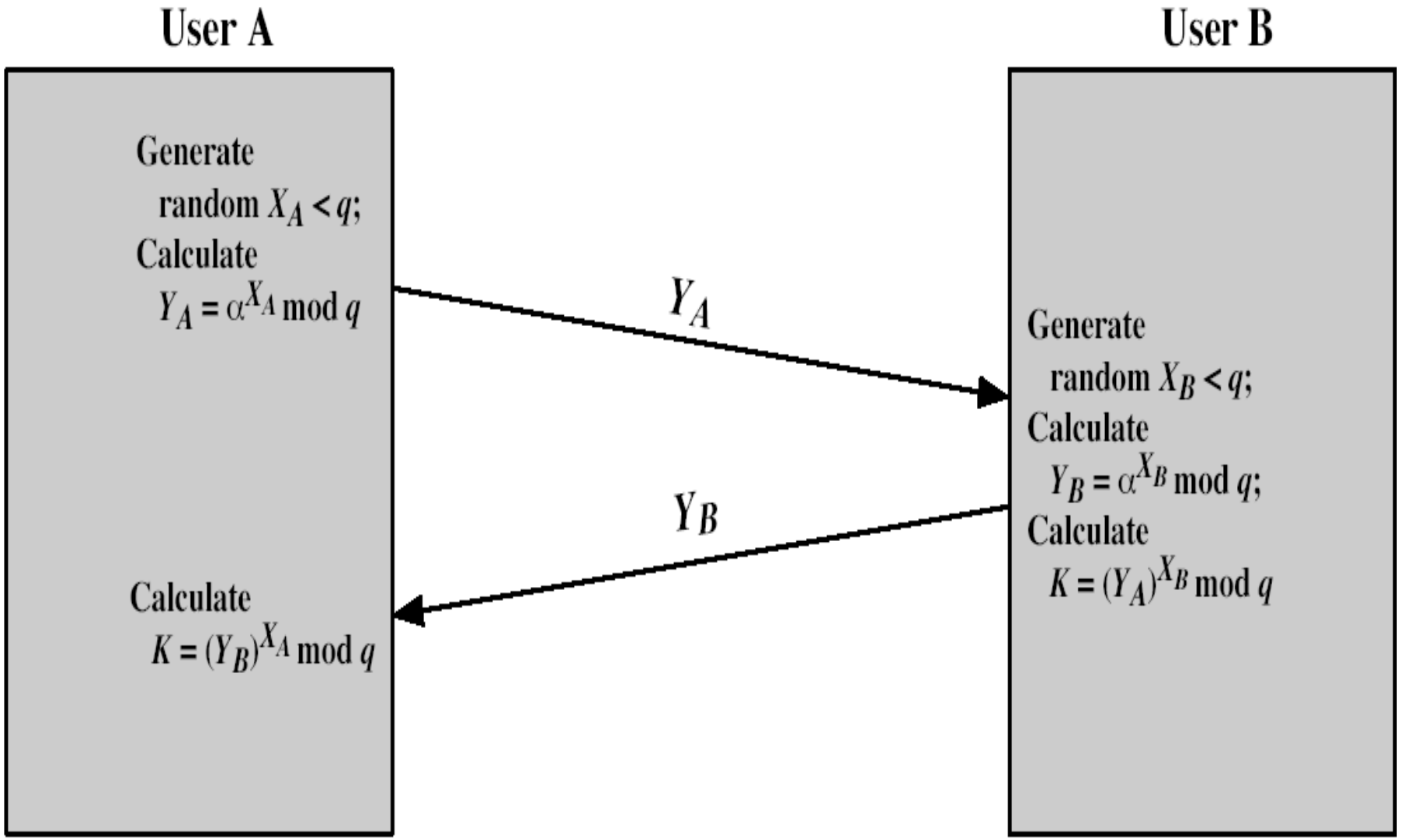
Diffie-Hellman Example

- Users A & B who wish to swap keys:
- Agree on prime $q=353$ and $a=3$
- Select random private keys:
  - A chooses $x_A=97$, B chooses $x_B=233$
- Compute respective public keys:
  - $y_A=3^{97} \bmod 353 = 40$ (A)
  - $y_B=3^{233} \bmod 353 = 248$ (B)
- Compute shared session key as:
  - $K_{AB}= y_B^{x_A} \bmod 353 = 248^{97} \bmod 353 = 160$ (A)
  - $K_{AB}= y_A^{x_B} \bmod 353 = 40^{233} \bmod 353 = 160$ (B)

Key Exchange Protocols

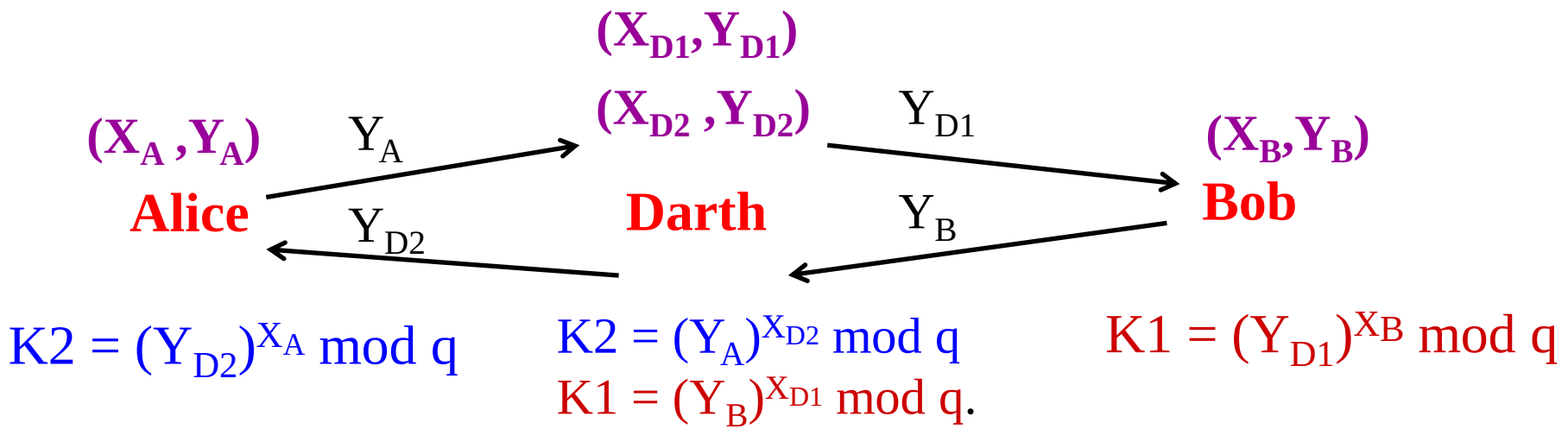


Man-in-the-Middle Attack



Man-in-the-Middle Attack

- Man-in-the-Middle-Attack



- Now, Alice and Bob think that they share a secret key, but instead Bob and Darth share secret key K1. Darth and Alice share secret key K2.

Ensuring the Authenticity of Public Keys in Public Distribution Schemes

Public Key Distribution

- To use public key encryption, communicators need to exchange their public keys
- A secure public key exchange scheme must ensure the authenticity of the public keys – ensuring that the claimant is indeed the rightful owner of the public key
  - Example: when Alice receives public key PU_B claiming to be from Bob, she needs to be sure that PU_B actually belongs to Bob (and e.g., not to Darth)
- We will study three public distribution approaches:
  - Public Announcement
  - Public Key Authority
  - Public Key Certificates

Public Announcement

- Users distribute public keys to recipients or broadcast to community at large

Public Announcement

- Users distribute public keys to recipients or broadcast to community at large

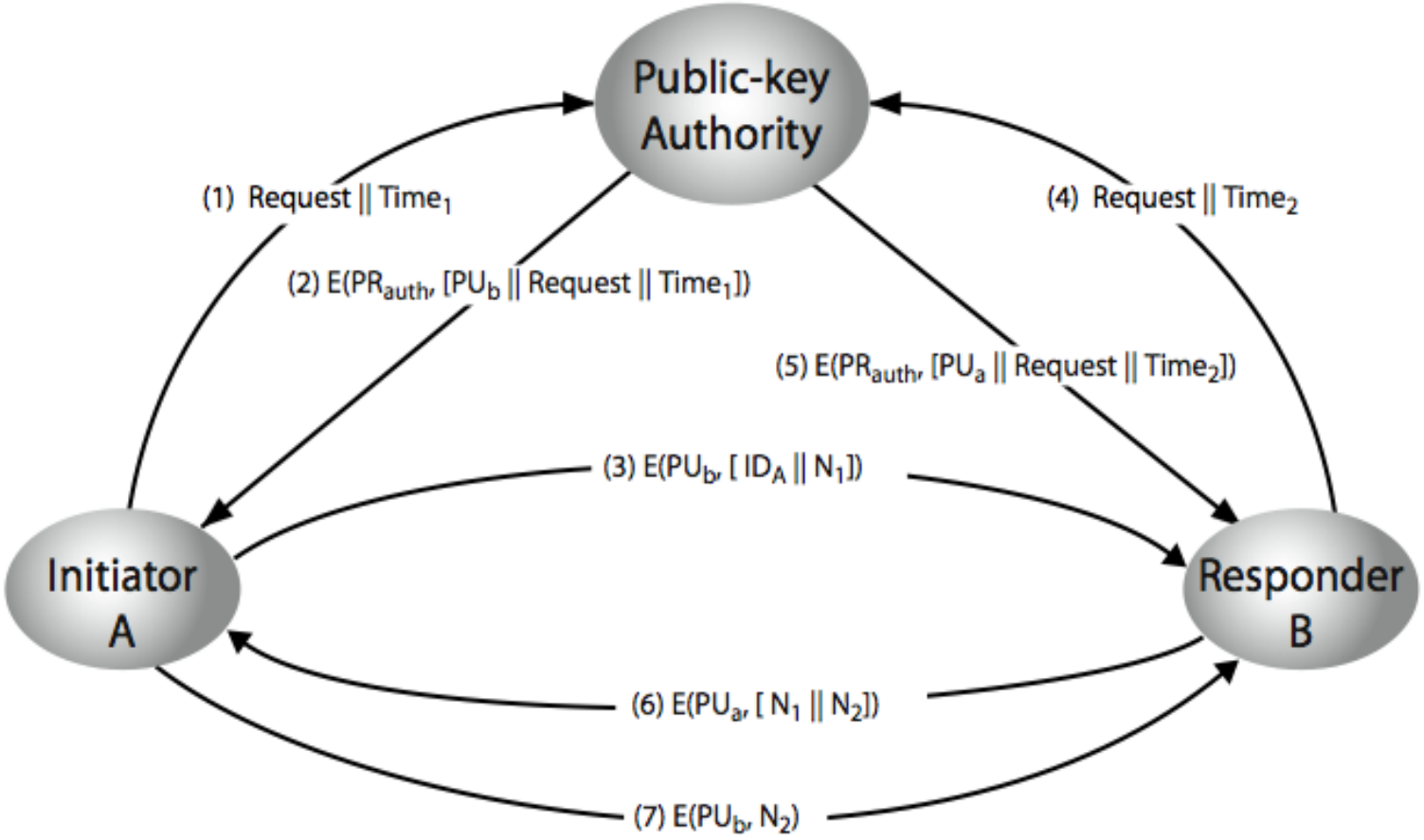
- Major weakness: forgery
  - Anyone can create a key claiming to be someone else and broadcast it
  - Until forgery is discovered can masquerade as claimed user

Public-Key Authority

- A central authority maintains a dynamic directory of public keys of all participants {name, public-key}.
- Each participant registers a public key with the directory authority. Registration would have to be in person or by some form of secure communication.
- Requires users to know public key for the directory. Only the authority knows the corresponding private key.
- Users interact with directory to obtain any desired public key securely.

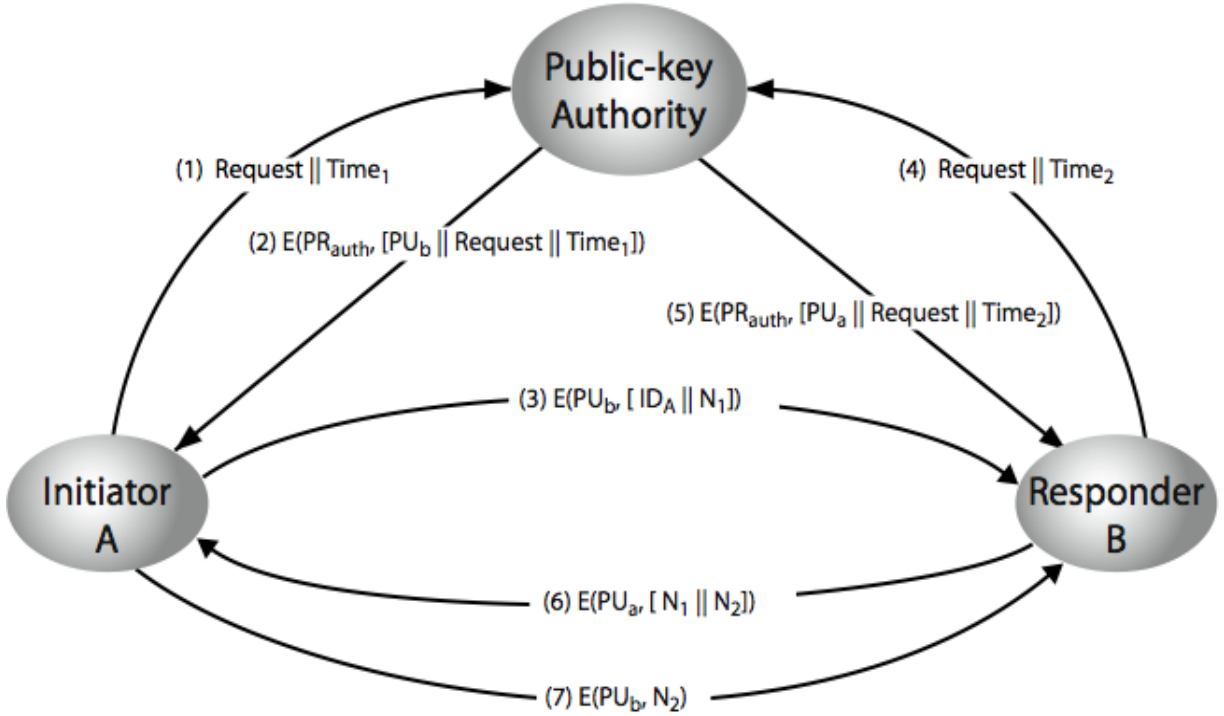
Public-Key Authority

1. A sends a timestamped mesg to the public-key authority containing a request for the public key of B.



Public-Key Authority

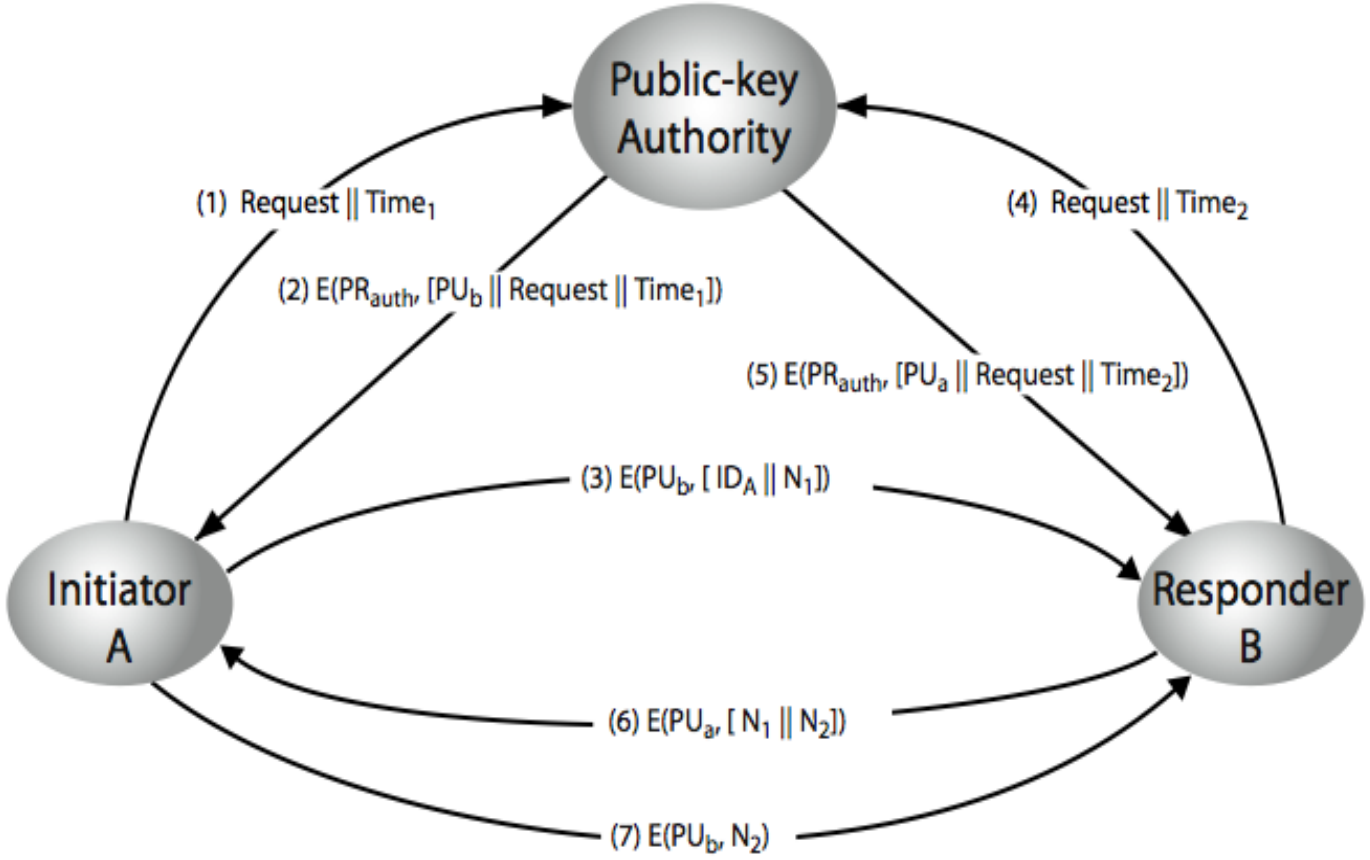
2. The authority responds with a mesg that is encrypted using the authority's private key.
   - B's public key
   - The original request: match with the request
   - The original timestamp: A can determine that this is not an old message from the authority.



Public-Key Authority

3. A stores B's public key and uses it to encrypt a msg. to B containing an identifier of A and a nonce N1.

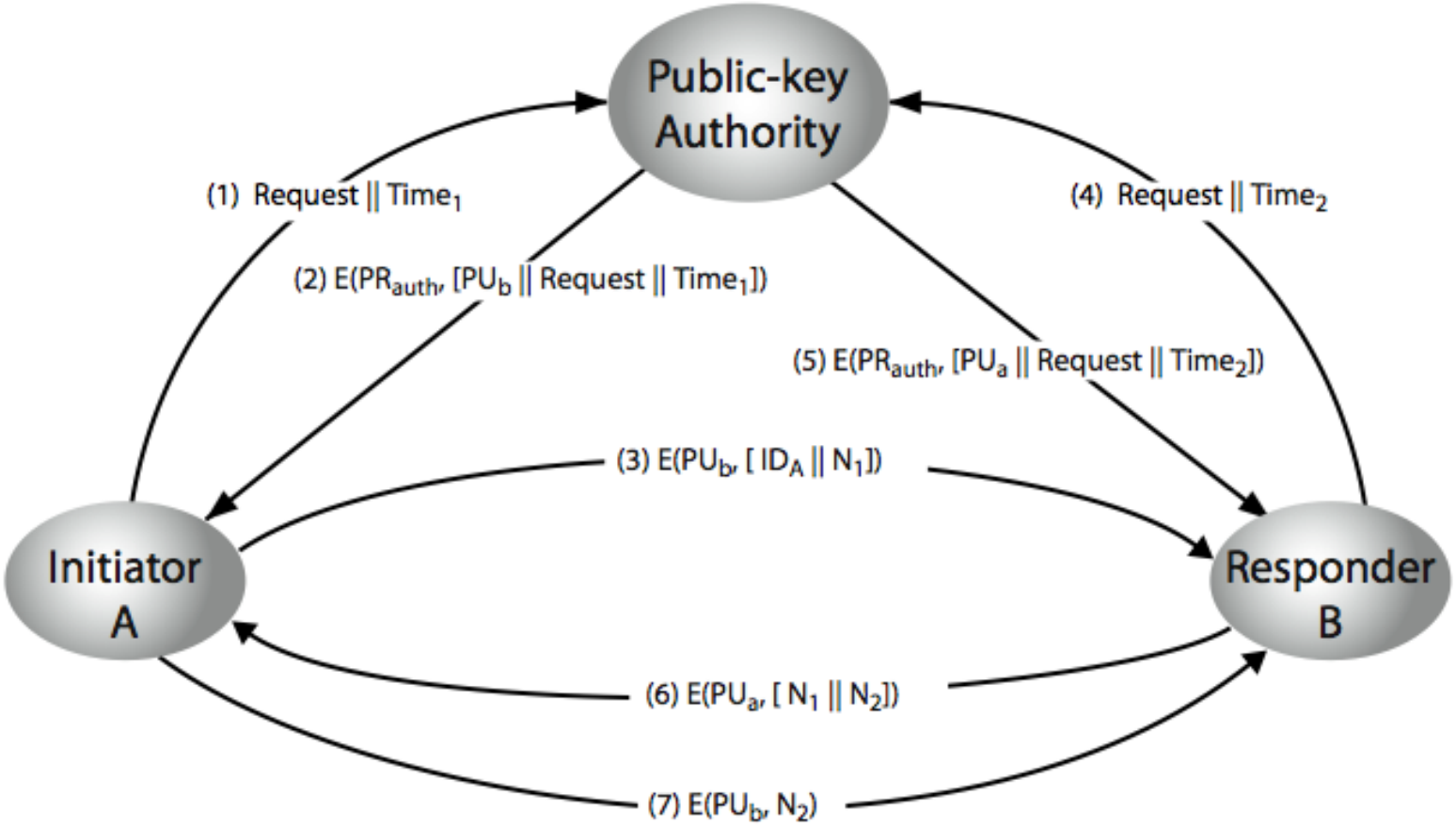
4 & 5. B retrieves A' public key from the authority in the same manner as A retrieved B's public key



Public-Key Authority

6. B sends a mesg to A encrypted using A's public key that contains A's nonce and a nonce generated by B.

7. A returns N2 encrypted using B's public key, to ensure B that its correspondent is A.



Drawback: Public-Key Authority

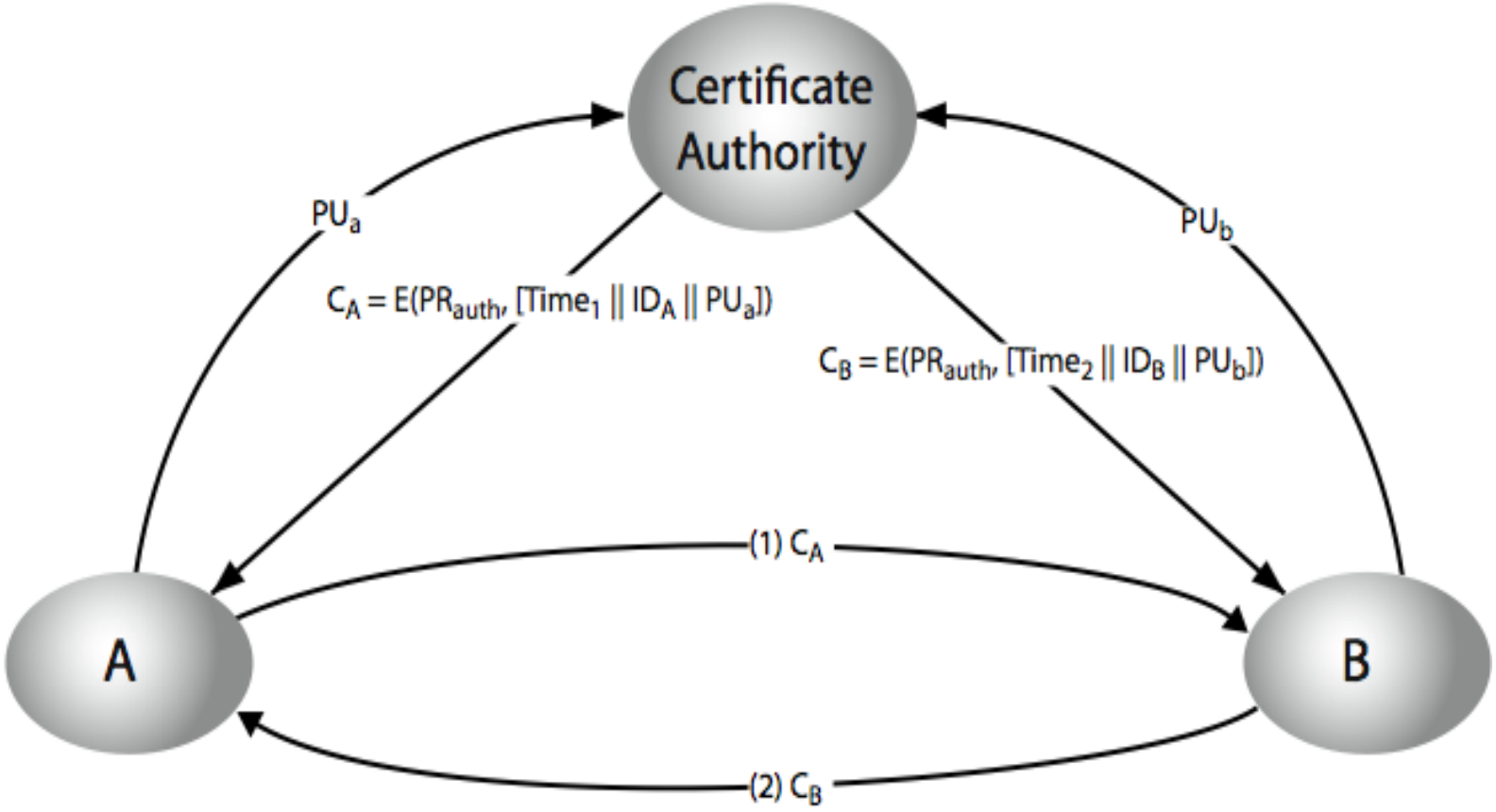
- Drawback of public-key authority: the public-key authority is a bottleneck because a user must appeal to the authority for a public key for every other user that it wishes to contact.

Public-Key Certificates

- Certificates allow key exchange without contacting a public-key authority
- A certificate consists of a public key plus an identifier of the key owner, with the whole block signed by a trusted third party (certificate authority).
- A user can present his/her public key to the authority in a secure manner and obtain a certificate.
- The user then publishes the certificate.
- Other participant can verify that the certificate was created by the authority.

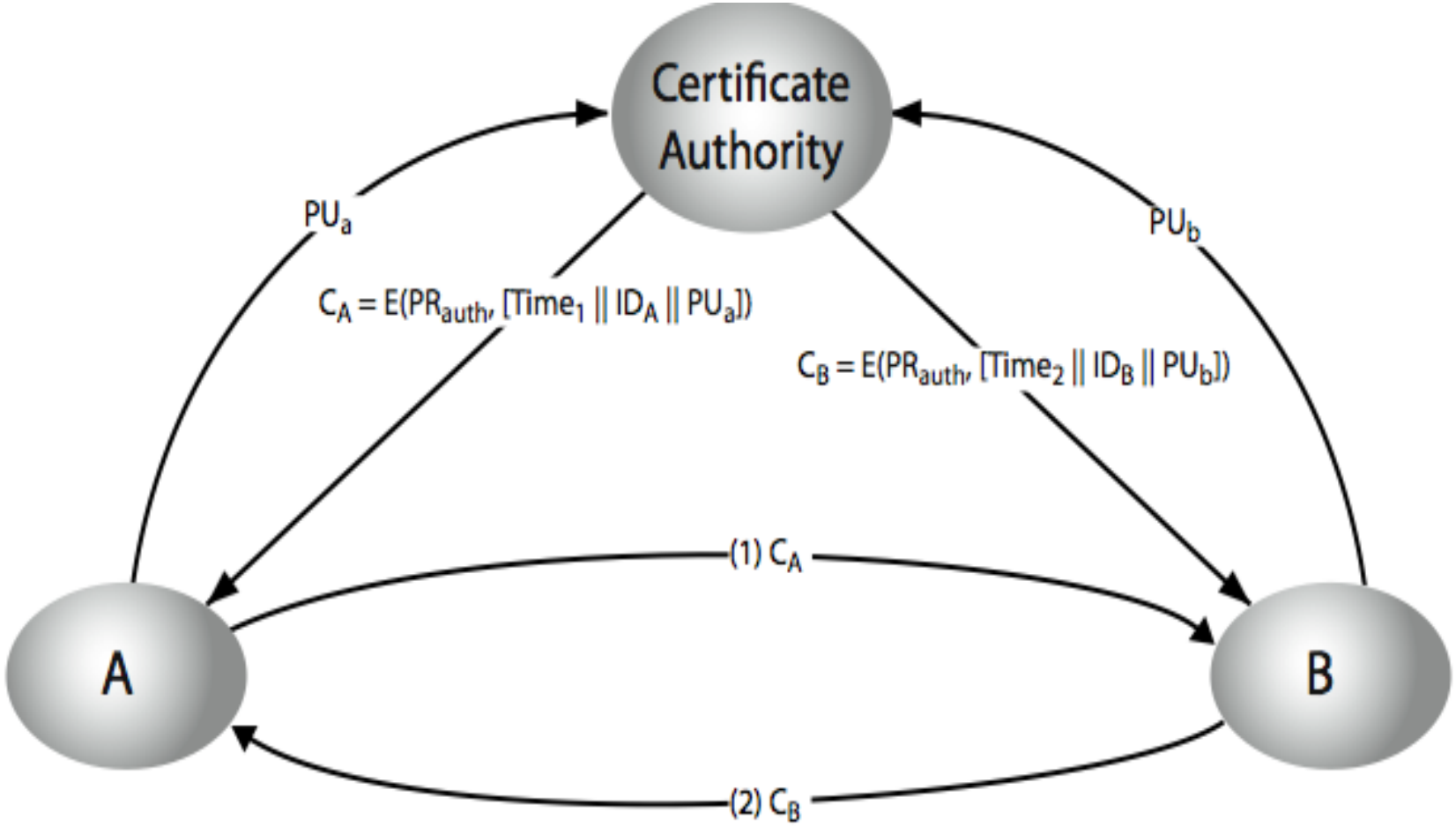
Public-Key Certificates

1. Participant A applies to the certificate authority, supplying a public key and requesting a certificate.
   - Application must be in person or by some form of secure authenticated communication.



Public-Key Certificates

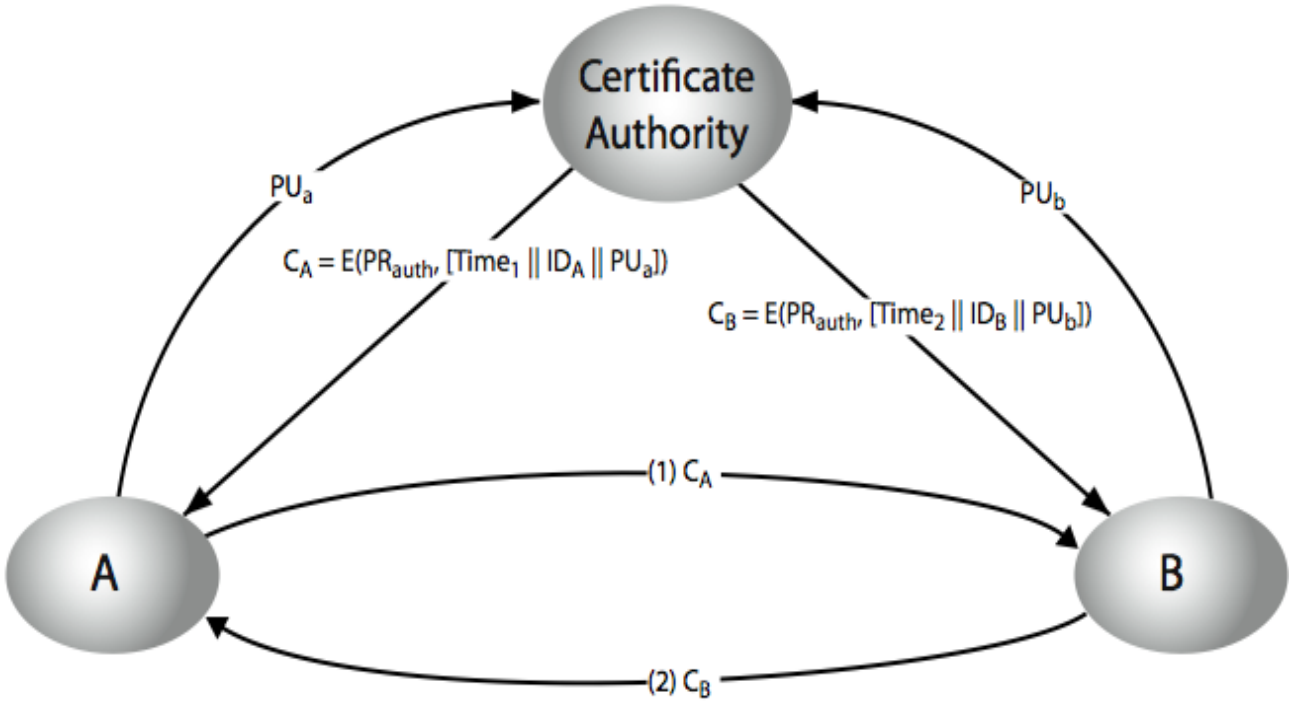
2. The authority provides the certificate of the form CA = E(PRauth, [Time1||IDa||PUa]), where PRauth is authority's private key and Time1 is a timestamp.



Public-Key Certificates

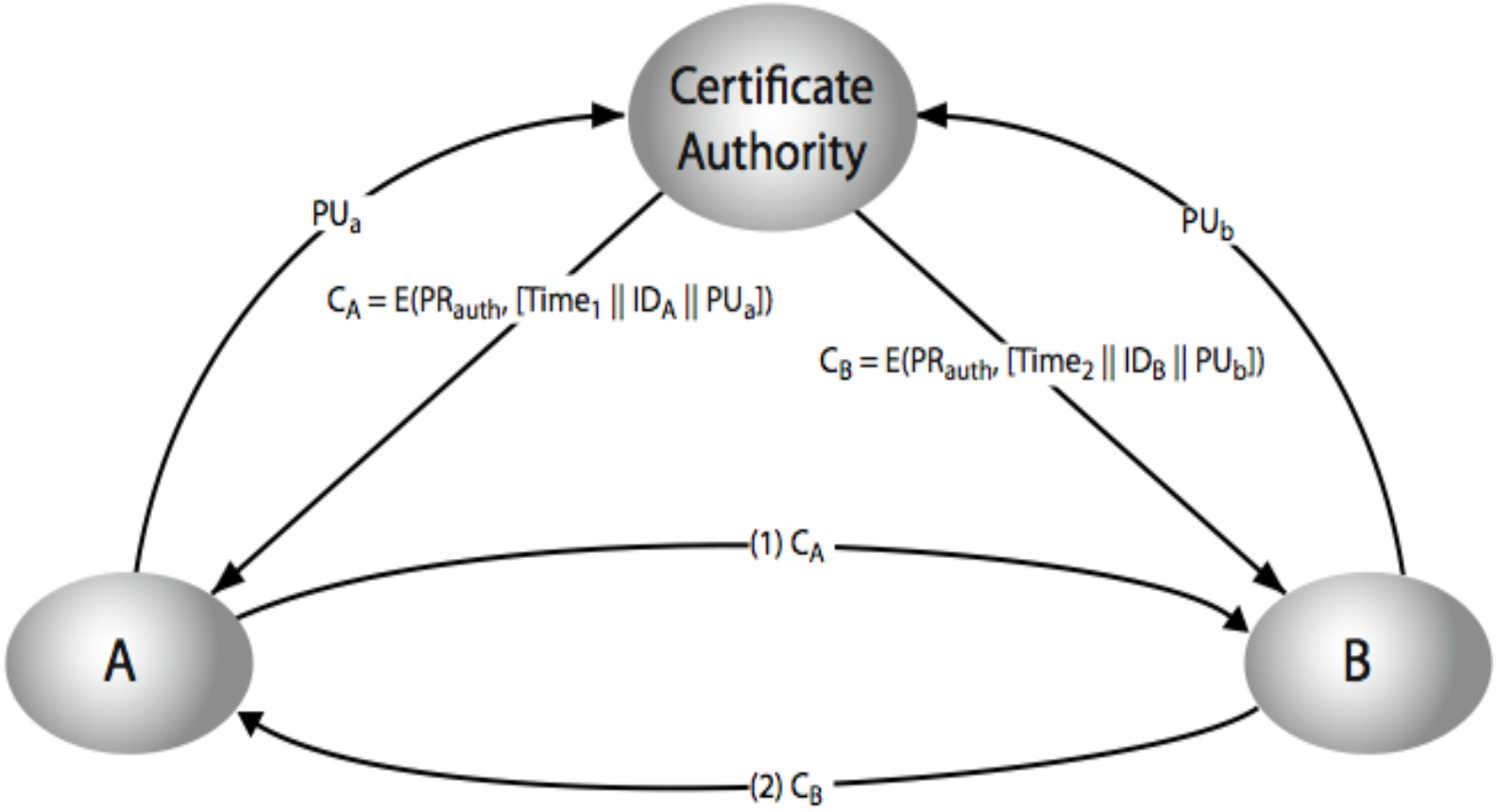
3. A may then pass this certificate on to any other participant, who reads and verifies that the certificate comes from the certificate authority (by decrypting the certificate using authority's public key):

D(PUauth, CA) = D(PUauth, E(PRauth, [T||IDA||PUa]))= (T||IDA||PUa)



Public-Key Certificates

- Any participant can verify that the certificate originated from the certificate authority and is not counterfeit.
- Only the certificate authority can create and update certificate.

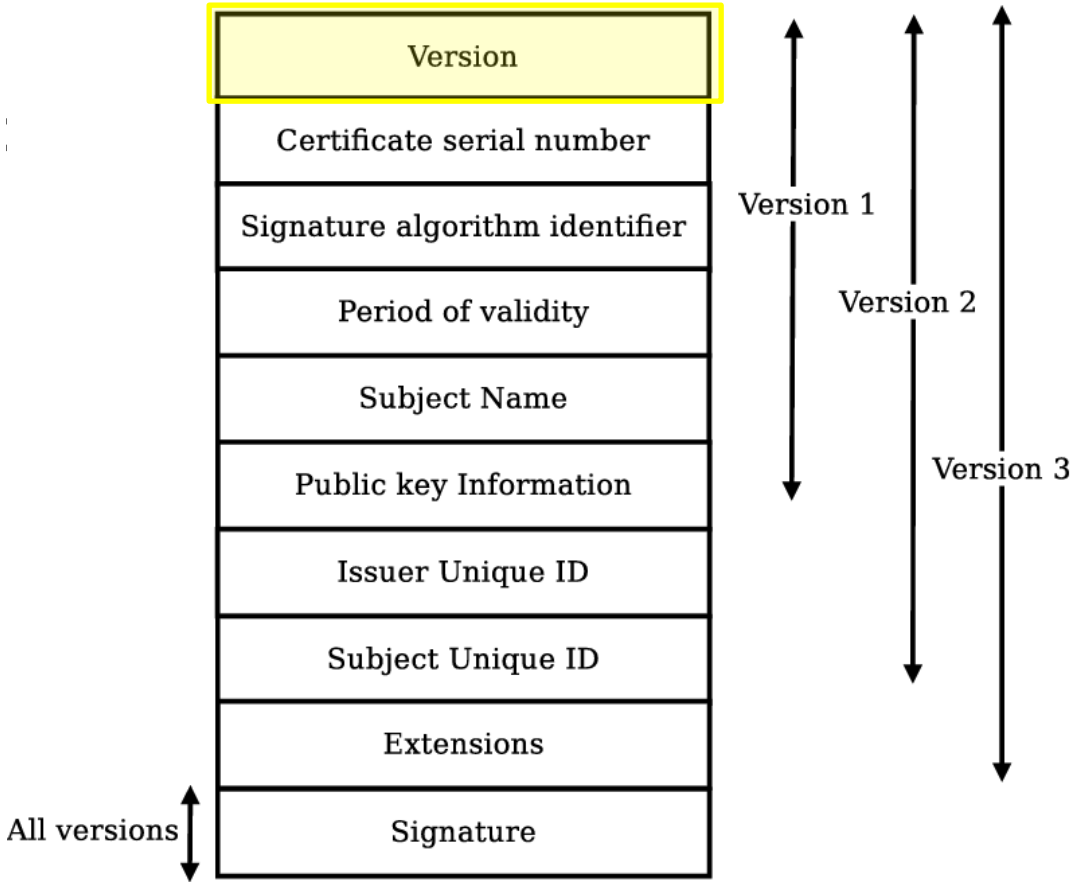


X.509 Certificates

- X.509 is a standard format for public key certificate:
  - Securely associates cryptographic key pairs with identities such as websites, individuals, or organizations.
  - Has three versions (1,2, and 3)
  - Each version extends the previous version with the new fields
  - Used in:
    - SSL/TLS protocols used to encrypt the web traffic in HTTPS
    - S/MIME email security applications
    - Code signing
    - Government-Issued Electronic ID
    - Signing Documents

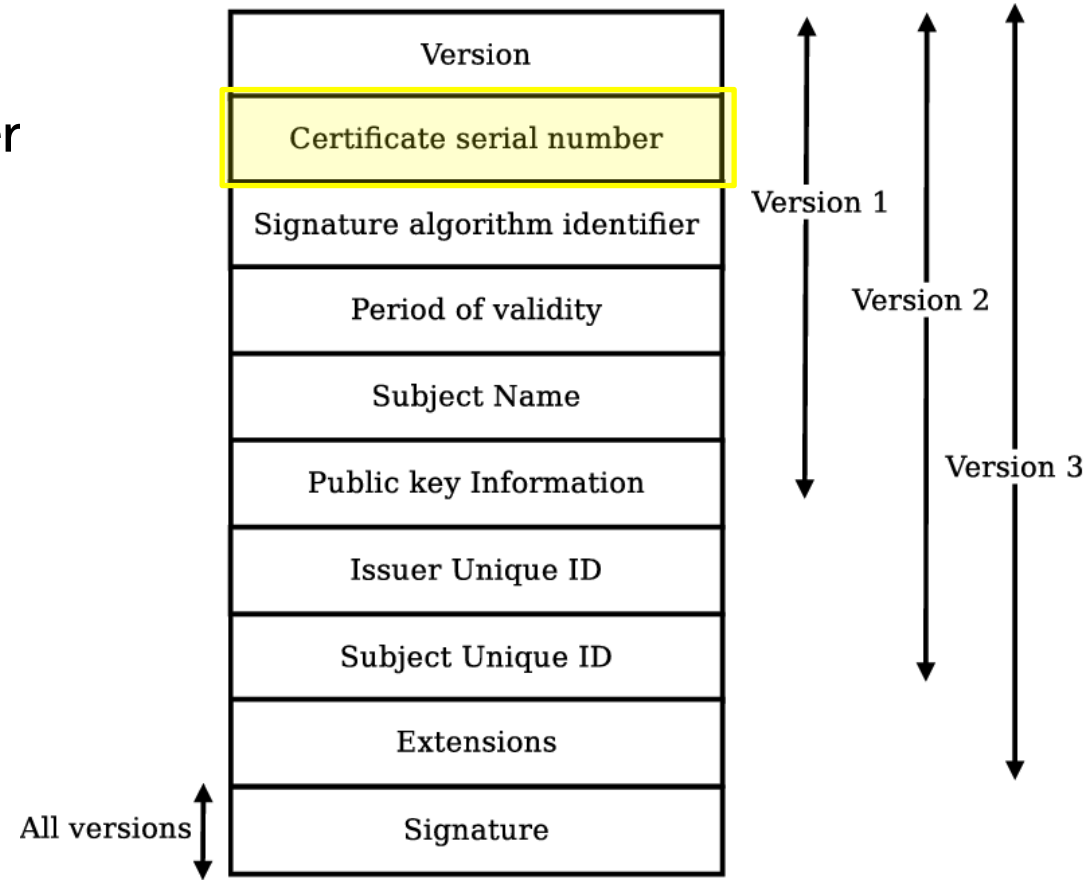
X.509 Certificates

- Version 3 has the following fields:
  - **Certificate version**: A value of 1, 2, or



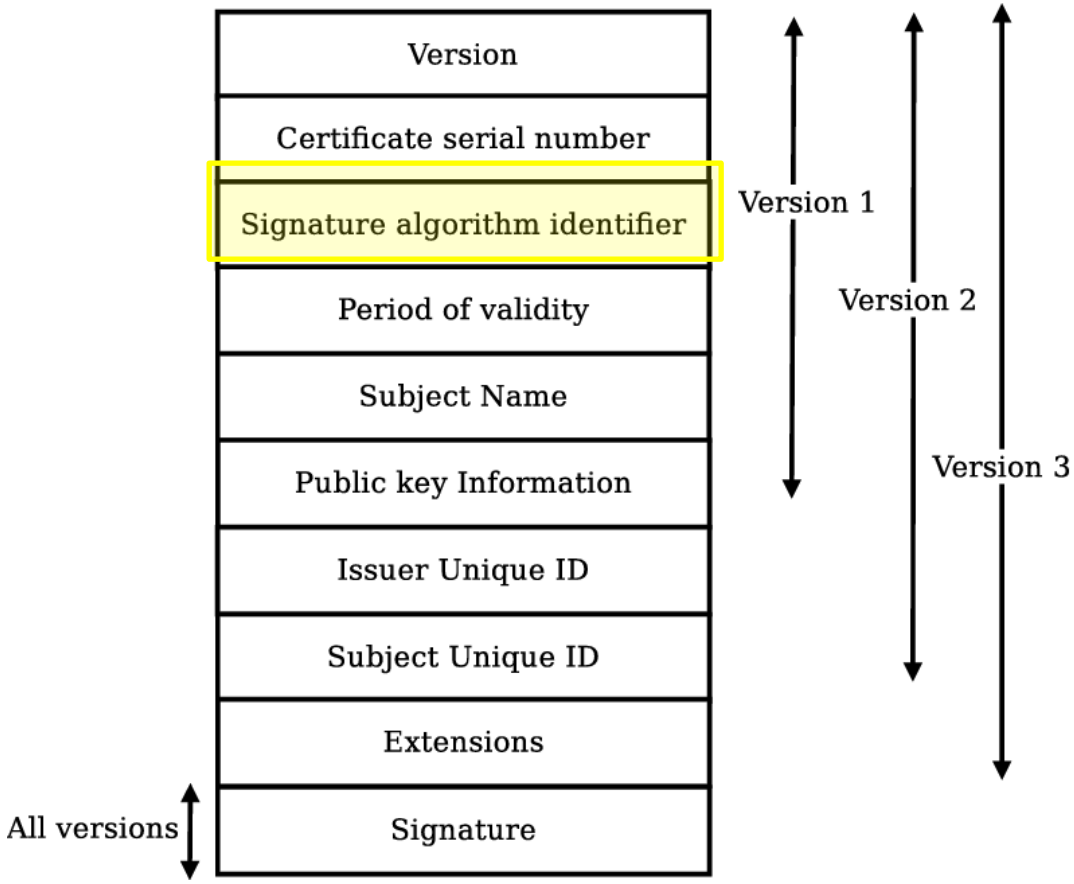
X.509 Certificates

- Version 3 has the following fields:
  - **Certificate serial number**: A unique ider of the certificate



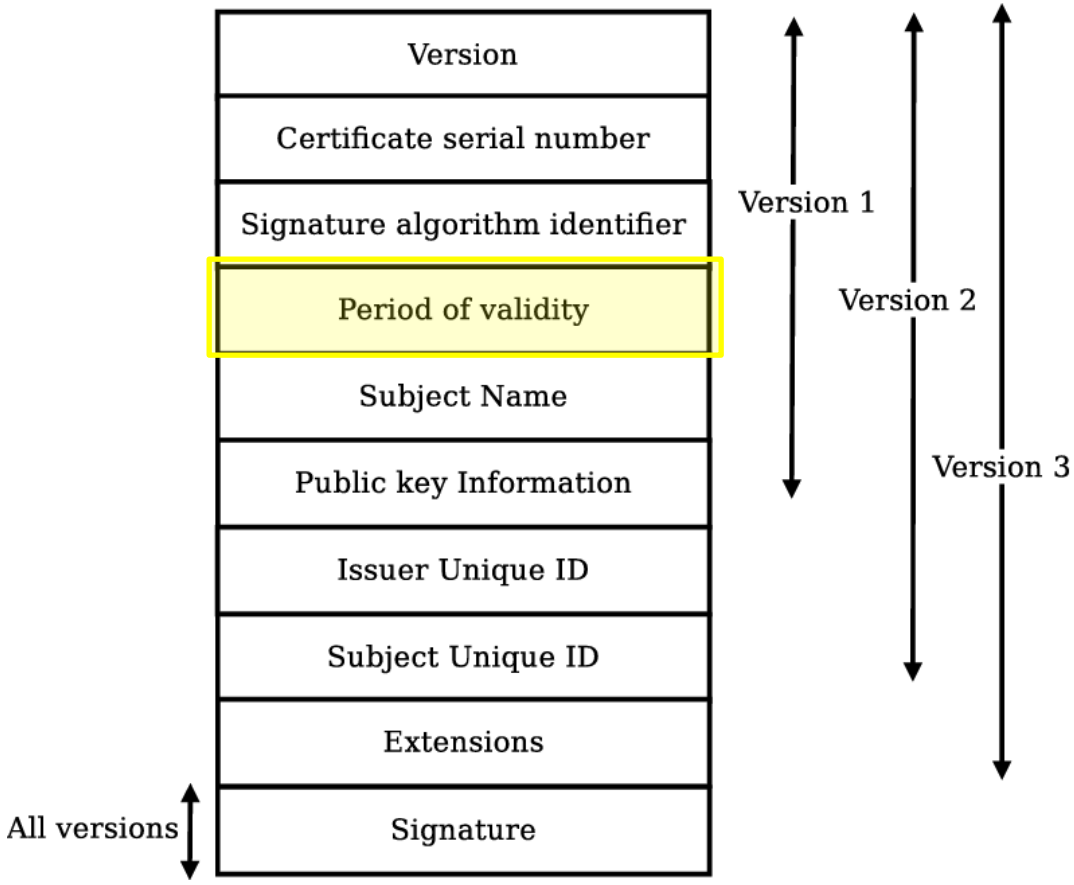
X.509 Certificates

- Version 3 has the following fields:
  - **CA Signature Algorithm**: The digital signature algorithm the CA used to sign the certificate



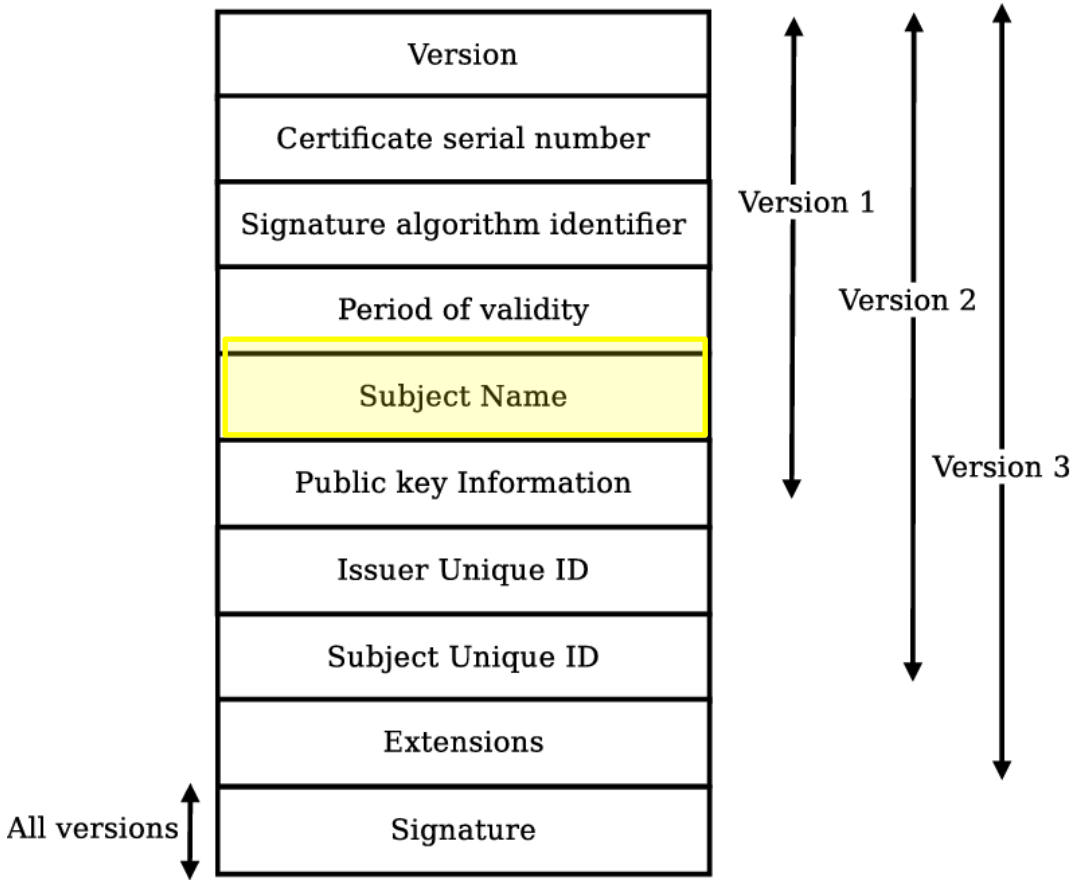
X.509 Certificates

- Version 3 has the following fields:
  - **Validity Period**: The time period for which the certificate is considered valid



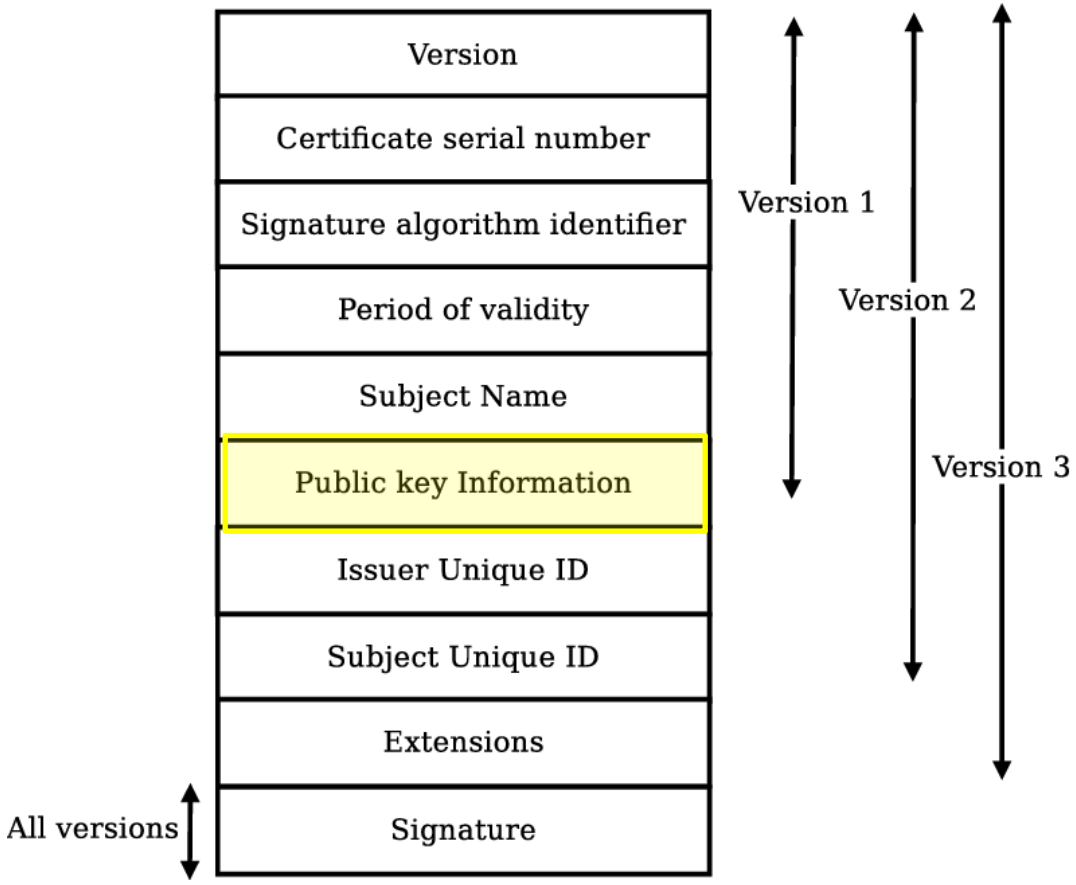
X.509 Certificates

- Version 3 has the following fields:
  - **Subject Name**: Name of the entity represented by the certificate



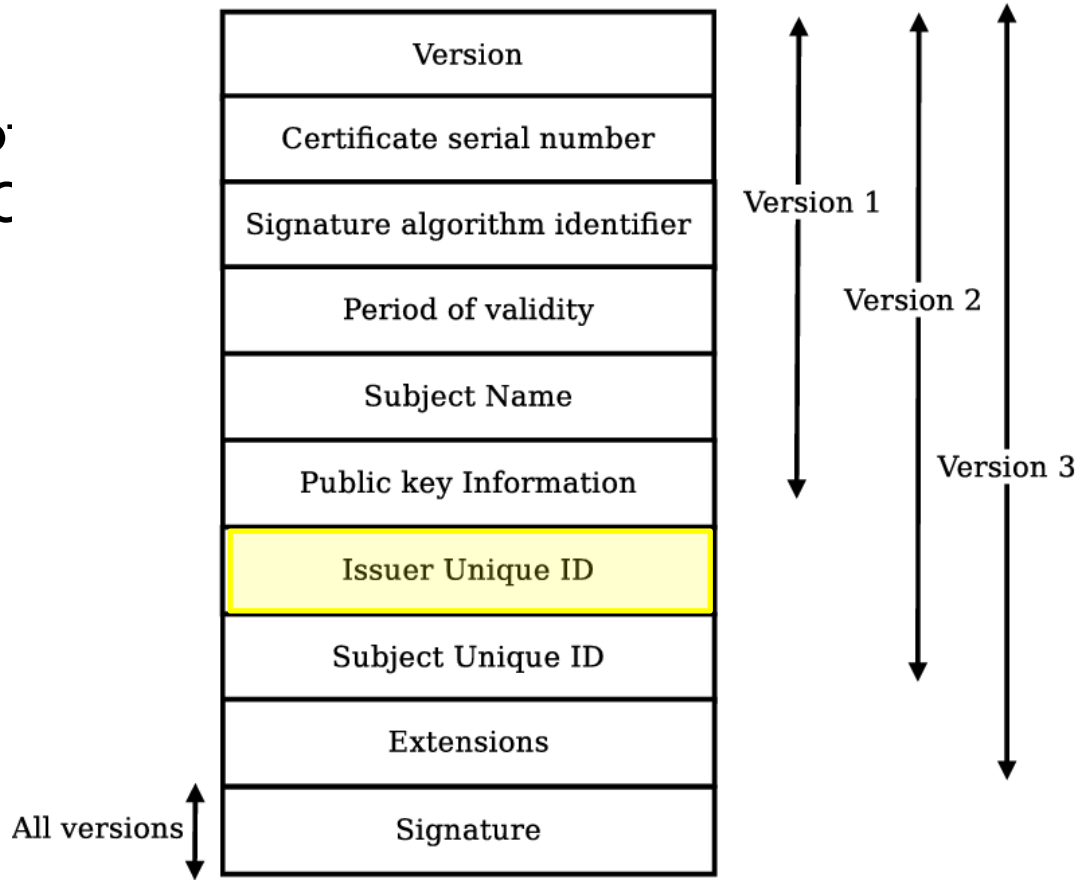
X.509 Certificates

- Version 3 has the following fields:
  - **Public Key Info**: Public key owned by the subject



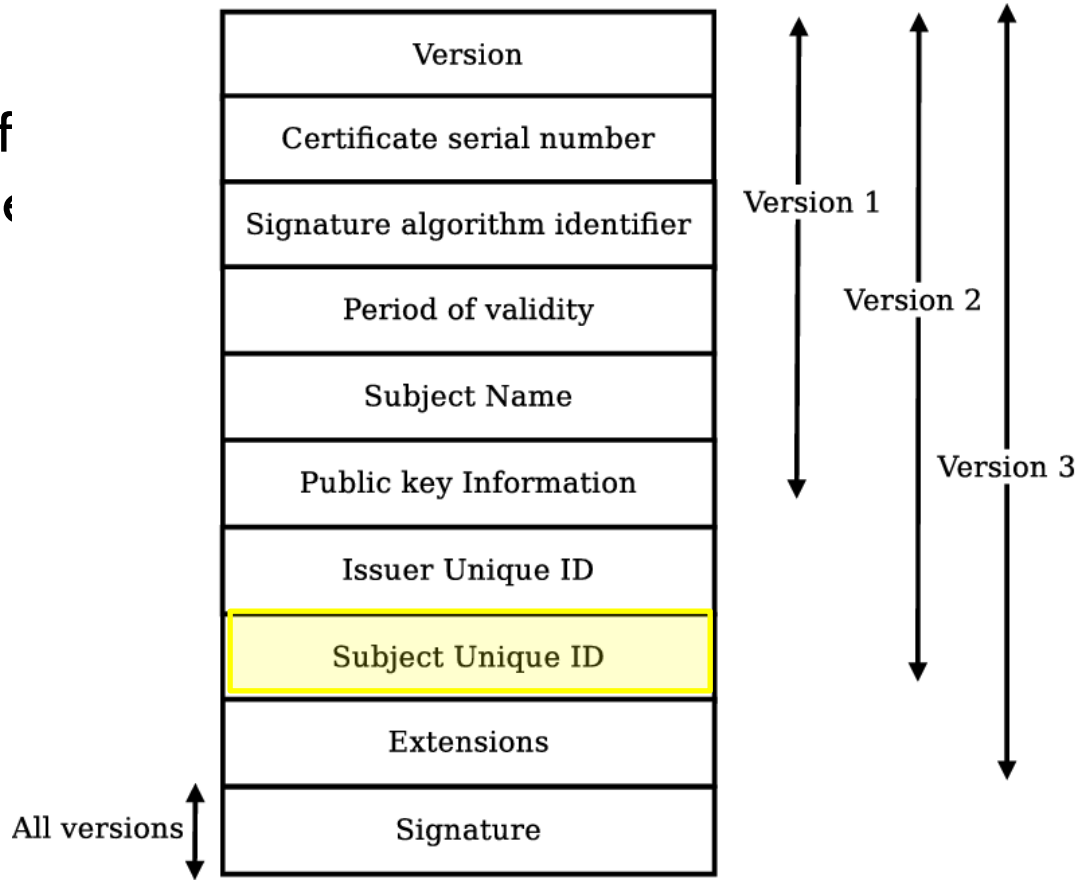
X.509 Certificates

- Version 3 has the following fields:
  - **Issuer Unique ID**: A unique identifier o
    issuing CA as defined by the issuing C



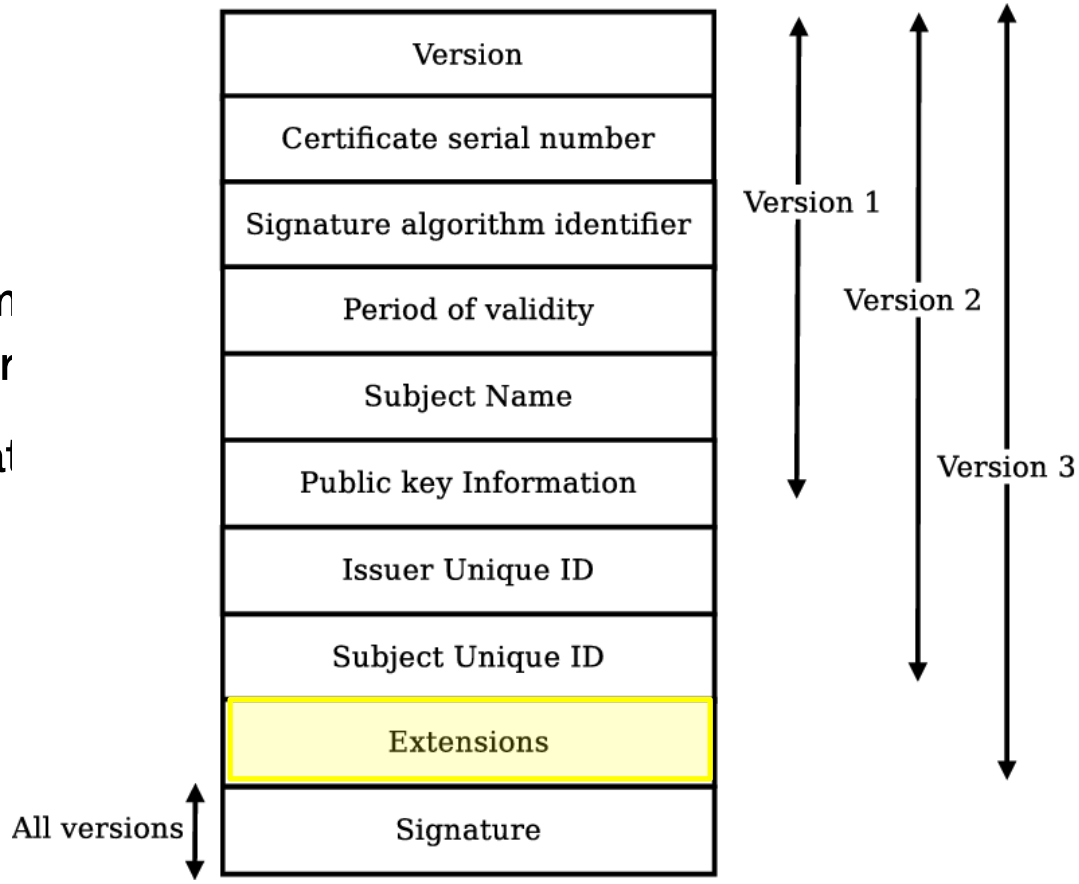
X.509 Certificates

- Version 3 has the following fields:
  - **Subject Unique ID**: A unique identifier f the certificate subject as defined by the issuing CA



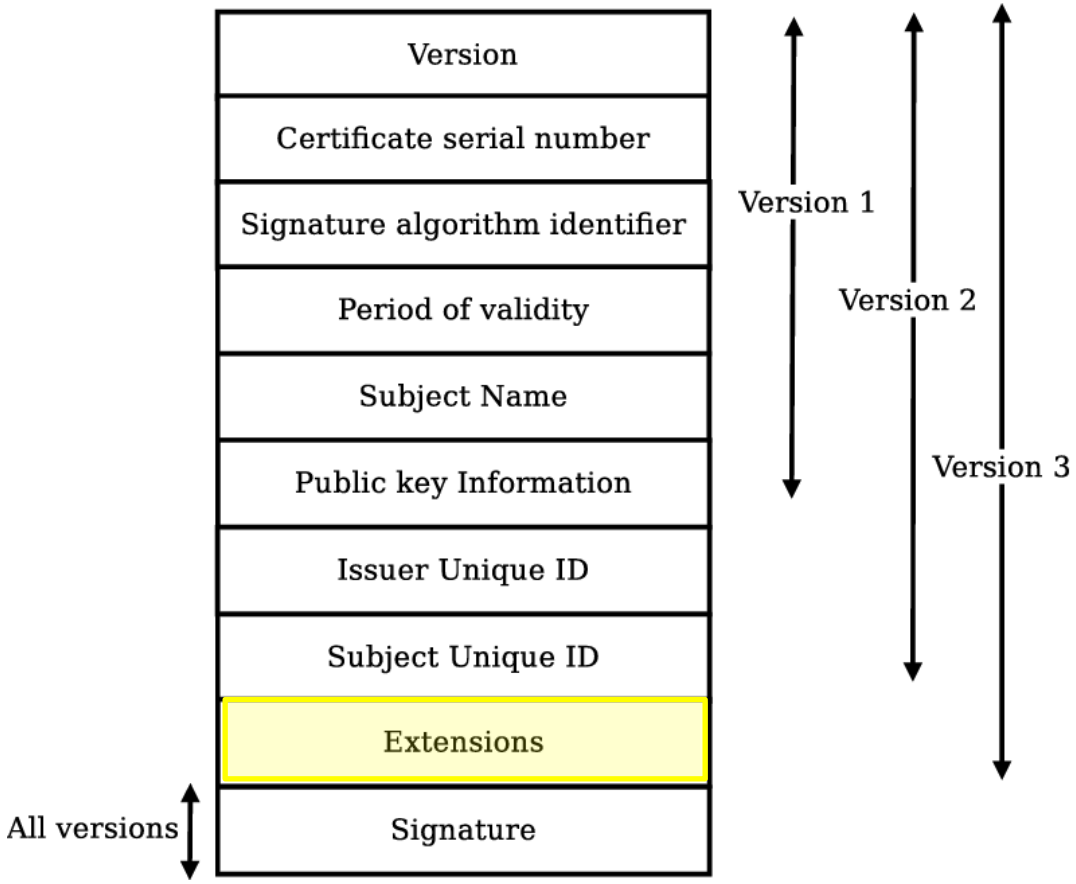
X.509 Certificates

- Version 3 has the following fields:
  - **Extensions: Authority Key Identifier**: is one of the following:
    - The subject of the CA and a serial num of the CA that issued the certificate; or
    - A hash of the public key of the CA that issued the certificate



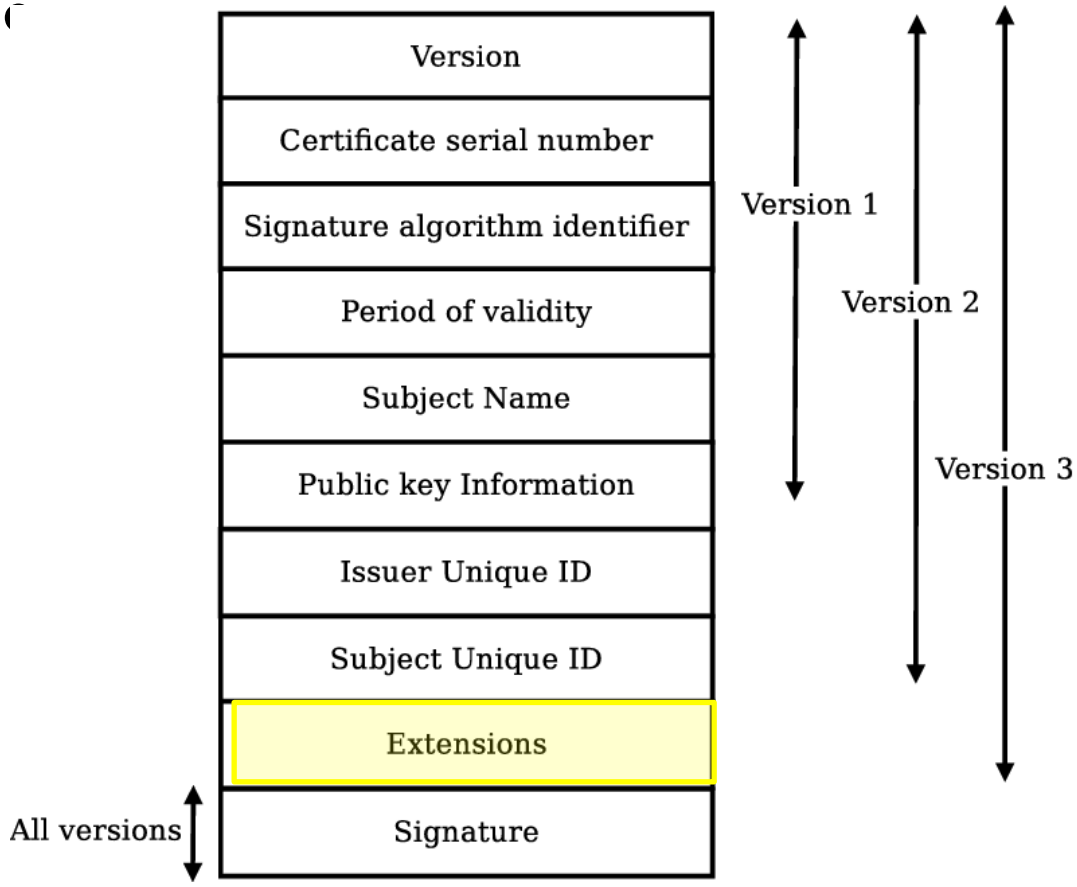
X.509 Certificates

- Version 3 has the following fields:
  - **Extensions: Subject Key Identifier**: A hash of the public key of the CA that issued the certificate



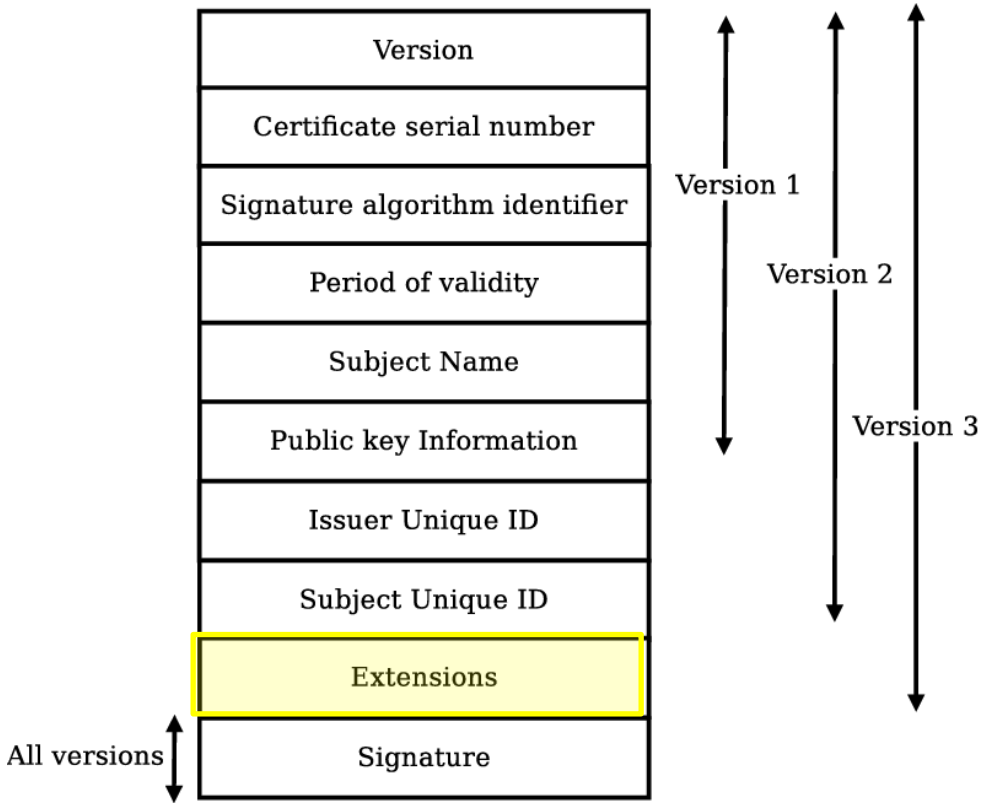
X.509 Certificates

- Version 3 has the following fields:
  - **Extensions: Key Usage**: defines the service for which the certificate was issued (i.e. the purpose of its usage):
    - Digital Signature
    - Non-Repudiation
    - Key Encipherment
    - Data Encipherment
    - Key Agreement
    - Key Cert Sign
    - CRL Sign
    - Encipher Only
    - Decipher Only



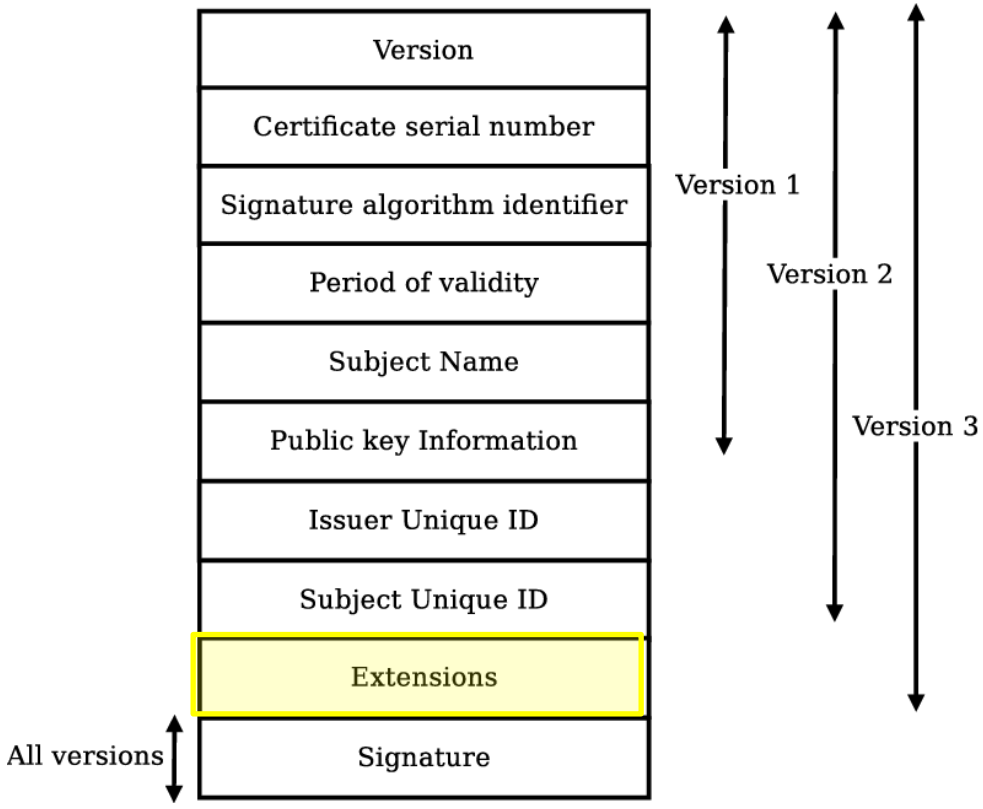
X.509 Certificates

- Version 3 has the following fields:
  - **Extensions: Other fields (see https://docs.microsoft.com/en-us/azure/iot-hub/tutorial-x509-certificates):**
    - Private Key Usage Period:
    - Certificate Policies:
    - Policy Mappings:
    - Subject Alternative Name:
    - Issuer Alternative Name:
    - Subject Dir Attribute:
    - Basic Constraints:
    - Name Constraints:
    - Policy Constraints:
    - Extended Key Usage:
    - CRL Distribution Points:
    - Inhibit anyPolicy:



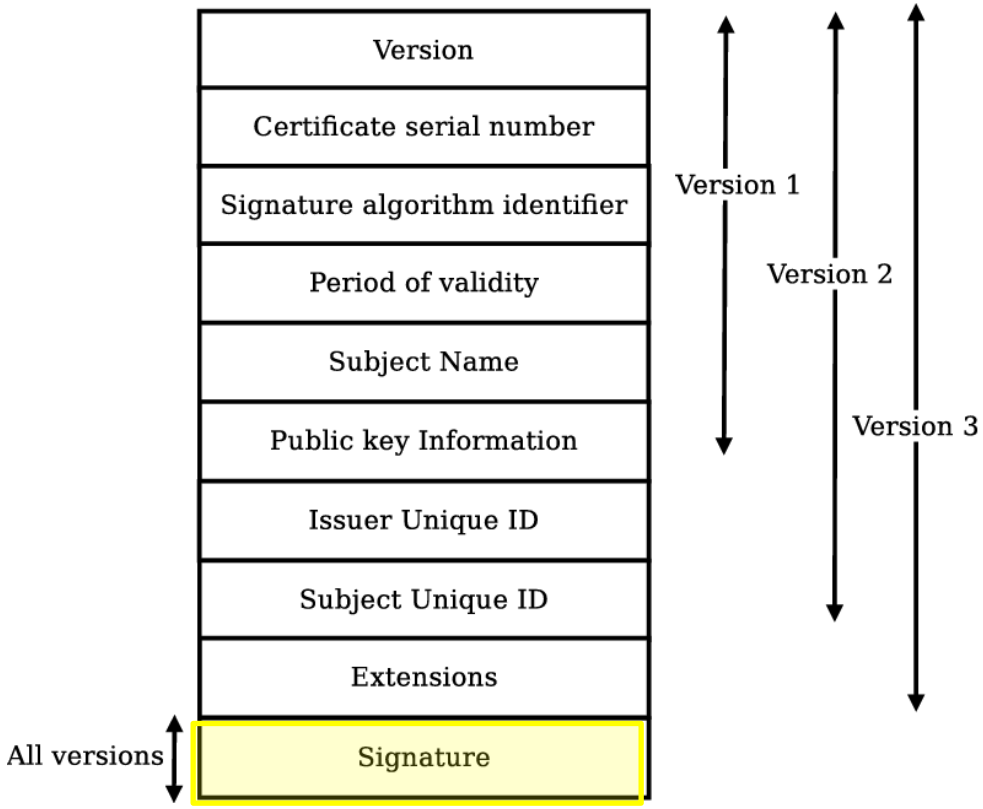
X.509 Certificates

- Version 3 has the following fields:
  - **Extensions: Other fields (see https://docs.microsoft.com/en-us/azure/iot-hub/tutorial-x509-certificates):**
    - Fresh CRL
    - Subject Information Access



X.509 Certificates

- Version 3 has the following fields:
  - **Digital Signature:** The digital signature over the entire contents of the certificate



X.509 Certificates (1)

- Example:

X.509 Certificates (2)

- Example:

Issuer Name	
Country or Region	US
State/Province	Texas
Locality	Houston
Organization	SSL Corp
Common Name	SSL.com EV SSL Intermediate CA RSA R3

Serial Number	72 14 11 D3 D7 E0 FD 02 AA B0 4E 90 09 D4 DB 31
Version	3
Signature Algorithm	SHA-256 with RSA Encryption (1.2.840.113549.1.1.11)
Parameters	None
Not Valid Before	Thursday, April 18, 2019 at 5:15:06 PM Central Daylight Time
Not Valid After	Saturday, April 17, 2021 at 5:15:06 PM Central Daylight Time

X.509 Certificates (3)

- Example:

Public Key Info	
Algorithm	RSA Encryption (1.2.840.113549.1.1.1)
Parameters	None
Public Key	256 bytes : AD 0F EF C1 97 5A 9B D8 1E B0 44 8D C6 C9 A0 28 C3 0E 68 1B 94 91 2E 77 EC AC AE BE 6C 78 04 5B A4 78 04 CE FB 07 4B 5D 34 F3 57 E5 0F FB 6B A4 2A A5 53 D3 D5 7F 3A 3C 54 4C EB 73 7B 5E A1 0A D9 7E 5F A9 5A C0 71 71 43 9D 6F BD 4C CC CC 43 8C CF 77 4B 9D 1A 75 CB 1F BD F7 3B D3 66 C6 CE 7C B0 5A FC D4 14 24 3A 2A C5 A8 61 6D 04 4D A6 36 2D B0 FC C4 B0 BF FC 41 27 71 E4 C3 90 AD 37 07 67 BE 5A 1A 81 9D AB 8A 71 92 A3 85 1D 99 E7 20 19 CF C4 FD AD 9F 6E 98 9F 5B CE 17 A1 FE 7B 4A 4F C9 F2 AD 21 C8 F7 1B 5D 10 79 59 85 DF 7E B8 A8 FE 3A D7 2F E2 02 DF D8 67 67 F4 63 9F FA B3 E7 47 63 48 3A C1 98 73 3D 9A 8D 8D DA AC C8 DF 50 32 BC A1 21 A6 10 56 AE E6 C6 10 2A 4E 54 41 5D 38 C1 37 77 78 1E 43 F8 70 2A 4B 4D EA B7 F9 51 CC 1C 17 4F 2A 1B 67 1C 2E E0 E0 2D 7C 59
Exponent	65537
Key Size	2,048 bits
Key Usage	Encrypt, Verify, Wrap, Derive
Signature	512 bytes : 36 07 E7 3B B7 45 97 CA 4D 6C B0 2A 3F 3F 38 43 12 3D 1C 4C 8E F6 87 18 5C 66 54 C5 E2 5B 4B ED ED DC 4C 23 EC 93 21 A1 19 28 DD 78 6D A6 0D E7 F4 F5 64 2E 1B 49 22 B4 EE FE E7 D3 0B 34 85 6A 12 14 09 33 4F 4E 52 FD 6B B0 04 9A EF 62 3C E3 78 6C 08 7A 87 25 63 61 28 B2 2C 22 10 5E 51 0F 03 7B 53 41 48 74 47 7D 3C 06 C3 E6 56 4D 96 9C 09 62 B2 76 00 9F 1A 3C C8 08 67 05 A1 C1 55 48 C2 37 EA 32 69 6A 12 E2 53 26 DB AC AB 79 94 88 8B 5B 5A 72 76 04 76 0D 53 CC 3D A9 38 95 E6 C1 BE E0 A4 C8 7E F6 AC 7E FF 34 ED 3B 5D 38 46 67 1C C5 79 D4 A8 81 8E 9C D0 CA F7 75 64 4F DC F8 4A 38 7C 88 18 DC D1 9B 50 F1 DB E8 61 D4 7D AE D8 9E 6E 86 E9 73 4A D4 2A F1 C7 CA 69 19 89 56 B5 FC BE 8D 90 F4 5A 21 89 A4 9A B7 3B F5 BA 24 34 A0 FD 5E 59 80 7A 45 93 3B 56 89 62 E3 4E E3 7E EB 13 2B 28 24 B9 86 EC DA 93 49 A1 0F 14 EF 54 93 BE 1E F4 55 CF 17 20 C5 01 C5 84 62 D5 64 38 1D 1C 59 08 D1 31 F8 AE 05 A4 1B BA 0A 67 51 9E A8 15 F2 E8 CF 8E 9E D8 88 52 21 89 CC 4F 98 13 0A 41 40 71 69 79 B0 A5 6A BE 77 AB 5E A1 D4 89 66 6C 02 C2 D1 43 0D A2 CA D7 7A 71 01 8B F7 98 21 74 89 E8 8B 27 38 28 CD 3E EA A7 78 AD 2A 3A 63 DB 3A D0 05 6B 4F C9 20 4E 01 38 DF 05 75 49 F7 9F 2E DC 19 31 A9 96 D7 2F 2D 4E 84 7C FA 7E F6 67 5A A1 E7 5C A1 72 3B 22 DC A5 FA F2 E7 DC D6 A8 6D A0 4D FD 78 C5 5C DC 34 D9 86 76 5B 1C 0D BB B1 E5 DB 64 2A 55 7F 20 4D 5D 4D 44 01 1D 79 A3 2D EC F5 6B CD BE 7B 52 67 1D FF 05 42 FB 42 7A A1 BC 4C 23 DF AF 16 B9 76 C9 69 86 02 34 F2 A9 CB B8 15 39 BA A5 F1 E6 72 7C 1D 5E 0C 48 D7 99 1F 50 98 2B 75 2D 67 58 79 A1 1A 05 5A

X.509 Certificates (4)

- Example:

Fingerprints	
SHA-256	79 E0 E2 8E ED C9 A9 52 D3 6B 41 3B A9 F9 09 DD 60 70 E5 A7 C9 05 B1 67 A8 6C C6 5E 57 C0 F7 A7
SHA-1	CB A9 CA 35 60 64 6A D3 47 23 E3 AD DA C6 2B 1D D1 A4 0A 52

Generating an X.509 Certificate in Linux

- Example:
  - 1. Generate a 4096 RSA public key (many options are available for the type of key and the algorithm):

```
┌──[mike@ResearchBackend]─[~]
└─$openssl genrsa -out mypublickey.key 4096
```

Generating an X.509 Certificate in Linux

- Example:
  - 2. Create the (not yet signed certificate) for the key (the certificate will be stored in the file mycert.csr). You will be prompted to enter a bunch of information regarding your company etc:

```
└── $openssl req -new -key mypublickey.key -out mycert.csr
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:CA
Locality Name (eg, city) []:Fullerton
Organization Name (eg, company) [Internet Widgits Pty Ltd]:TuffyEnterprises
Organizational Unit Name (eg, section) []:Security
Common Name (e.g. server FQDN or YOUR name) []:Security
Email Address []:mgofman@fullerton.edu

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

Generating an X.509 Certificate in Linux

- Example:
  - 2. Submit mycert.csr to the CA. They will sign it with their private key and you can then install your certificate in the web server, VPN application, etc.

Generating an X.509 Self-Signed Certificate

- We can also generate self-signed certificates (ones that are signed by ourselves)
- Such certificates do not provide a way to authenticate the certified public key because it is not signed by a known CA
- However, such certificates are still useful, for example, setting up encrypted channels between trusted points (e.g., your home web server and browsers) as the public key in the certificate is used for exchanging the session keys.

Generating an X.509 Certificate in Linux

- Example: The following will generate the private key (file privatekey.pem) and will use it for signing the certificate (file certificate.pem) it will also generate.

```
└──╼ $openssl req -newkey rsa:4096  -x509  -sha512  -days 365 -nodes -out certif
icate.pem -keyout privatekey.pem
```

- The signature will be generated using RSA 4096 public key encryption and SHA-512 hash

Certificate Chains

- SSL/TLS security protocols used for HTTPs, VPNs, and other applications use the X.509 public key certificates
- **Problem:** there are many CA's out there. Some organizations have their own internal CA's for signing their products etc.
  - Applications (e.g., browsers) often ship with the public keys of the well-known CA's (e.g., VeriSign), but what about other less well-known CAs?

Certificate Chains

- SSL/TLS security protocols used for HTTPs, VPNs, and other applications use the X.509 public key certificates
- **Problem:** there are many CA's out there. Some organizations have their own internal CA's for signing their products etc.
  - Applications (e.g., browsers) often ship with the public keys of the well-known CA's (e.g., VeriSign), but what about other less well-known CAs?
- **Solution:** use **chains of certificates:**
  - **Main Idea:** verify the certificate of the CA that has certified the unknown CA. Keep regressing backward from the latter CA until a certificate of a known CA is found.

Certificate Chains

- SSL/TLS security protocols used for HTTPs, VPNs, and other applications use the X.509 public key certificates
- **Example:** the owner of domain www.example.com wants an a X.509 for their domain
  - 1. The owner pays the Certificate Authority (CA) to generate an X.509 certificate signed with the CA's private key

Certificate Chains

- SSL/TLS security protocols used for HTTPs, VPNs, and other applications use the X.509 public key certificates
- **Example:** the owner of domain www.example.com wants an a X.509 for their domain
  - 1. The owner pays the Certificate Authority (CA) to generate an X.509 certificate signed with the CA's private key
  - 2. When the user visits the domain with e.g., their browser using HTTPs, the protocol requests the server's certificate

Certificate Chains

- SSL/TLS security protocols used for HTTPs, VPNs, and other applications use the X.509 public key certificates
- **Example:** the owner of domain www.example.com wants an a X.509 for their domain
  - 3. The browser looks up the CA name in it's local database of known certificate authority names and their associated public keys

Certificate Chains

- SSL/TLS security protocols used for HTTPs, VPNs, and other applications use the X.509 public key certificates
- **Example:** the owner of domain www.example.com wants an a X.509 for their domain
  - 4. If the CA name is found, the browser verifies the certificate using the CA's associated public key

Certificate Chains

- SSL/TLS security protocols used for HTTPs, VPNs, and other applications use the X.509 public key certificates
- **Example:** the owner of domain www.example.com wants an a X.509 for their domain
  - 4. If the CA name is found, the browser verifies the certificate using the CA's associated public key
  - **What happens if the browser does not find the certificate authority in it's local data base?**

Certificate Chains

- SSL/TLS security protocols used for HTTPs, VPNs, and other applications use the X.509 public key certificates
- **Example:** the owner of domain www.example.com wants an a X.509 for their domain:
  - **What happens if the browser does not find the certificate authority in it's local data base?**
    - The browser will fetch the certificate that certificate of the CA
    - The browser checks if the issuer CA (let's call it CAb) of the unknown CA's certificate is in the local database, then verifies the certificate using the associated public key
    - If CAb is not in in the local database, then fetch CAb's certificate
    - Repeat until a certificate whose issuer appears in the browser's local database is found

Certificate Chains

- SSL/TLS security protocols used for HTTPs, VPNs, and other applications use the X.509 public key certificates

- Root Certificate: the certificate of the last CA in the chain of certificates. It is self-signed.

Certificate Chains

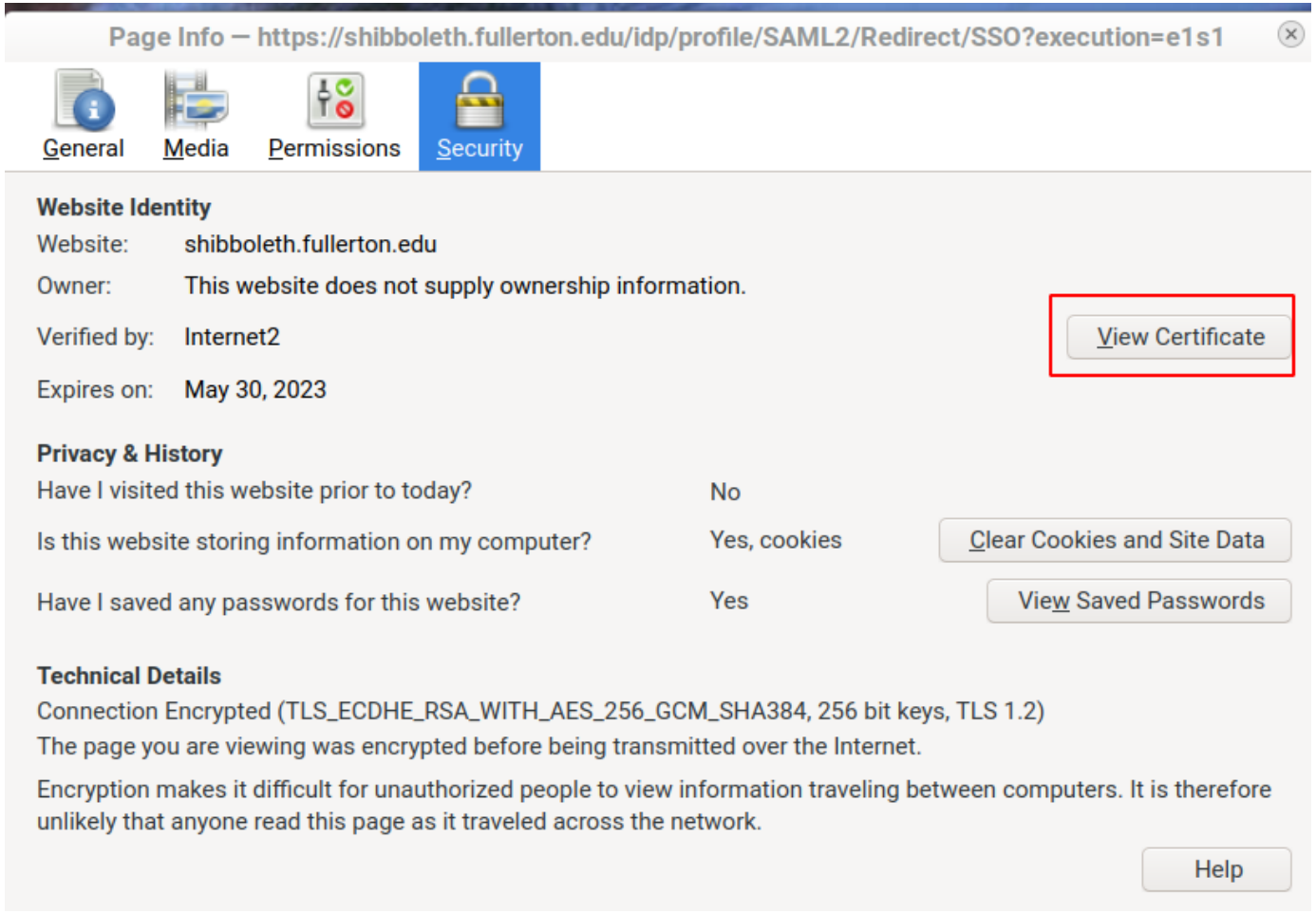
- SSL/TLS security protocols used for HTTPs, VPNs, and other applications use the X.509 public key certificates

- **Case Study:** The certificate of shibboleth.fullerton.edu:
  - Go to www.Fullerton.edu and navigate to the login page (this assumes the Firefox browser is being used).
  - Lock icon in the URL bar -> Connection Secure -> More Information
  - In the window click "View Certificate" (see the next slide)

Certificate Chains

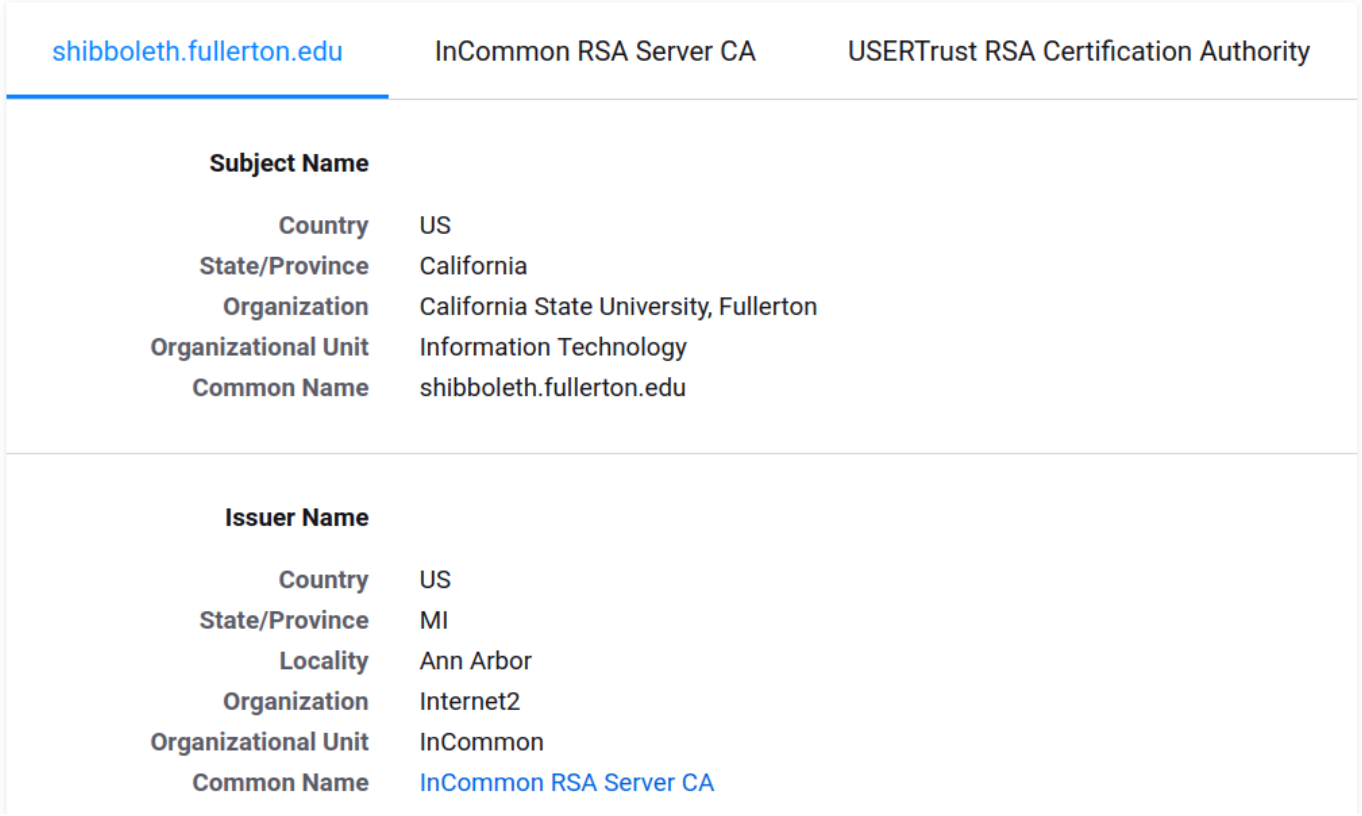
- SSL/TLS security protocols used for HTTPs, VPNs, and other applications use the X.509 public key certificates

- **Case Study:** The certificate of shibboleth.fullerton.edu:



Certificate Chains

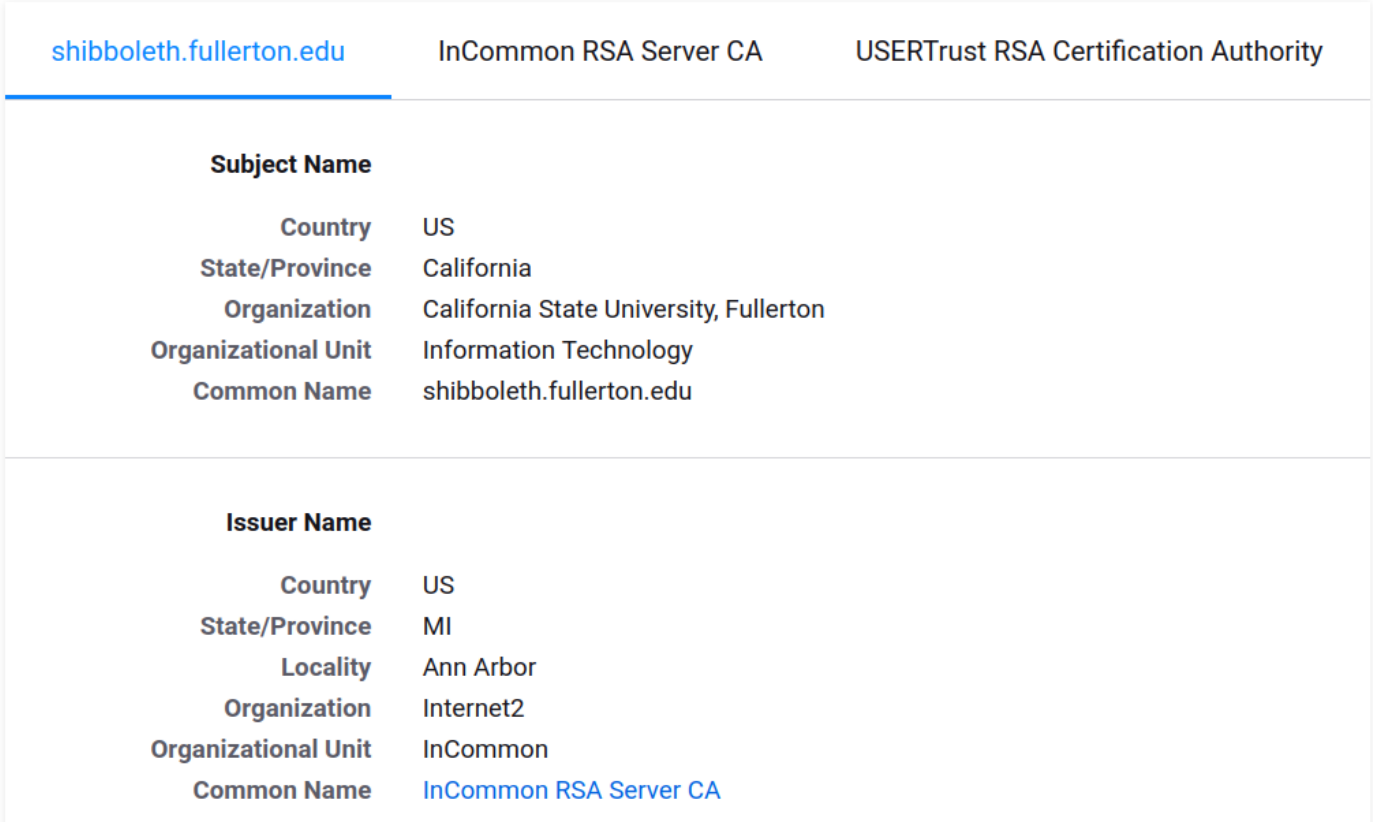
- SSL/TLS security protocols used for HTTPs, VPNs, and other applications use the X.509 public key certificates
- **Case Study:** The certificate of shibboleth.fullerton.edu: We can see the tabbed interface that shows the chain of certificates:



Certificate Chains

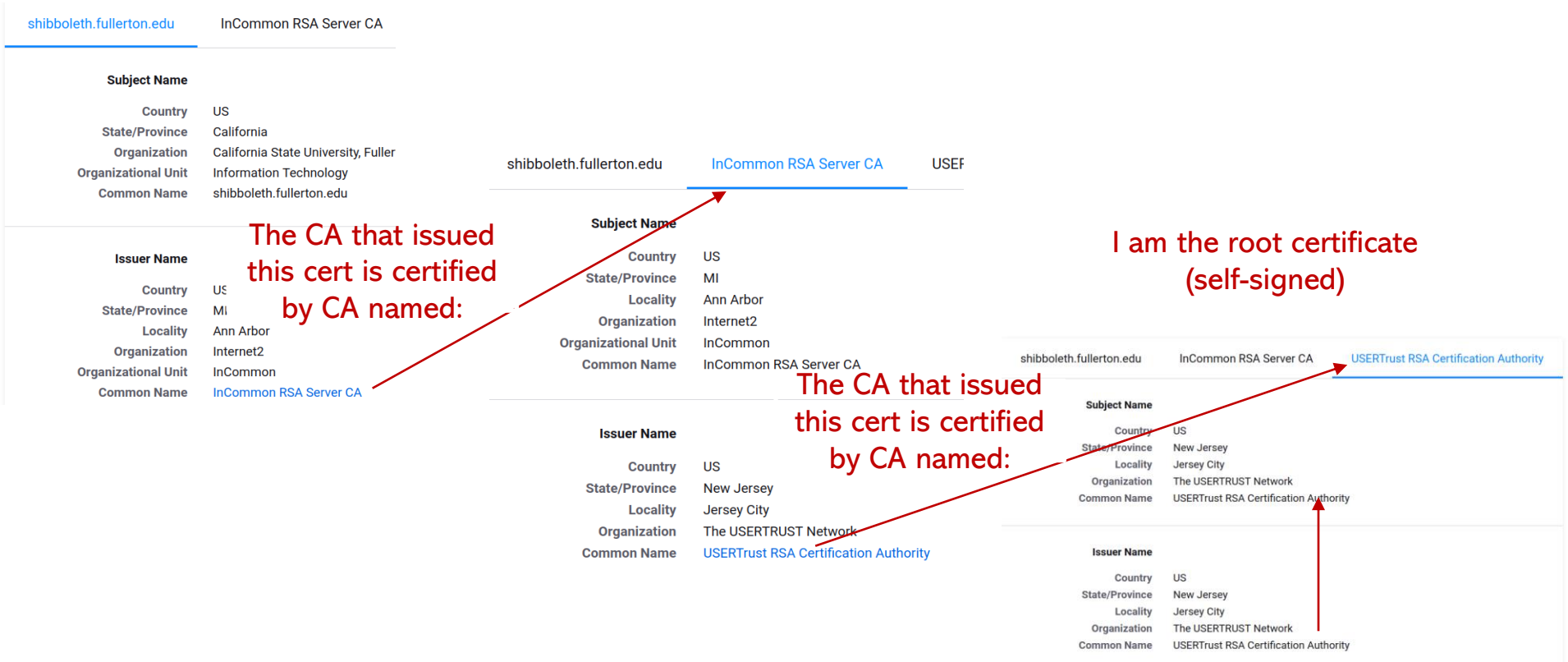
- SSL/TLS security protocols used for HTTPs, VPNs, and other applications use the X.509 public key certificates

- **Case Study:** shibboleth cert. is signed by InCommon whose cert is in turn signed by USERTrust (click on tabs to view certs)



Certificate Chains

- **Case Study:** shibboleth cert. is signed by InCommon whose cert is in turn signed by USERTrust (click on tabs to view certs)



Public Key Infrastructure

- A set of policies, processes, server platforms, software and workstations used for the purpose of administering certificates and public-private key pairs.
- Includes the ability to:
  - Issue, maintain, and revoke public key certificates
  - Validate digital signatures

Credits

- Some slides borrowed from Dr. Ping Yang from State University of New York at Binghamton