Network Security: Replay and freshness

ENCS5322, NETWORK SECURITY PROTOCOLS

First Semester 2024-2025

Outline

1. Alice and Bob
2. Replay and freshness
3. Timestamp
4. Sequence number
5. Nonce

The first broken protocol

- Please meet Alice and Bob!
- Alice sends a signed message to Bob:

  A → B: M, $S_A(M)$ // Example: S_A("Attack now!")

  Assumption: Alice and Bob know each others' public keys

- What things are wrong with this protocol?

If you want to learn, stop here and think a few minutes before looking at the solution

Being explicit

- Should include recipient id:
  A → B: B, M, S_A(B,M) // Example: S_A("Bob, attack now!")
- Include important information, such as endpoint identities, explicitly in the authenticated message
- What about Alice's identity?
- What else is wrong with this protocol?

Replay and freshness

$A \rightarrow B$: B, M, $S_A(B, M)$ // S_A("Bob, attack now!")

- Replay attack: attacker sniffs the original message and sends it again on the next day
- Authentication is usually not enough in network security! Need to also check freshness of the message
  - Fresh = sent recently, not received before (exact definition depends on the application)
  - Freshness mechanisms: timestamp, nonce, sequence number

Timestamps

- Checking freshness with A's timestamp:

  $A \rightarrow B$: $B, T_A, M, S_A(B, T_A, M)$

  Example: S_A("2024-10-4 14:15 GMT", "Bob, attack now!")

- Timestamp implementations:
  - Sender's clock value, UTC
  - Message expiration time
  - Validity start and end times

Timestamp limitations

- Timestamp requires clocks at the sender and receiver
- Timestamp requires secure clock synchronization
  - Secure fine-grained synchronization is difficult to implement
  - Loose synchronization (minutes or over 24 h) is easier
- Clock must never turn back
- Problematic in IoT devices, smartcards, locks etc.

When can timestamps be used without clock synchronization?

- Fast replays while the timestamp is fresh:

$S_A(B, T_A, \text{“Transfer \$10.”}), \ S_A(B, T_A, \text{“Transfer \$10.”})$

  - Solutions: (helpless) operations, duplicate detection with sequence numbers

Sequence numbers

- Sequence numbers for detecting message deletion, reordering and replay

  A → B: B, seq, M, S_A(B, seq, M)

  Example:
  S_A("Transaction 43542. Transfer 30€ to account 1006443.")

Sequence number limitations

- Sequence number must grow monotonically
  - Require synchronization of distributed endpoints, e.g. server farm, multi-threaded server
- Counter must not be reset, except when rekeying
- Sender and receiver counters must stay in sync
  - Plan resynchronization after message loss and endpoint failure

- Attacker can delay the message:

S_A(seq, “Bob, attack now!”) // intercept and replay tomorrow

Nonces

- Checking freshness with B's nonce:
  1. A → B: "Hello, I'd like to send you a message."
  2. B → A: N_B
  3. A → B: B, N_B, M, S_A(B, N_B, M)
- Bob's nonce is usually a long random number selected by Bob
  - Long means at least 128 to 256 bits
- Reasoning: any authenticated message that contains N_B must have been sent after Bob generated N_B

Nonce implementation

- Nonce must be never reused
- In many applications, nonce must be unpredictable to attackers
- Best nonce: 128-bit or 256-bit random number
  - Very unlikely to repeat and impossible to guess
- Another nonce: timestamp and random number (or their hash)
  - Protects against Random number generation (RNG) problems, e.g., if entropy pool is empty after device reset

Nonce limitations

1. A → B: "Hello"
2. B → A: N_B
3. A → B: B, N_B, M, S_A(B, N_B, M)

- Nonce requires a random number generator, entropy source
- Nonce requires an extra message or roundtrip
- Ok for connections but not well suited for asynchronous communication: store-and-forward such as email, events, or message bus
- Not suitable for broadcast communication
  - Too many Bobs, too many nonces
  - Radio and satellite broadcast, multicast

Freshness mechanism summary

1. Use a random nonce from the receiver where possible
2. Timestamp to limit message lifetime + sequence number for duplicate detection
3. Use pure sequence number only when nothing else is available (leads to complex designs)

Network Security:
Classic protocol flaws

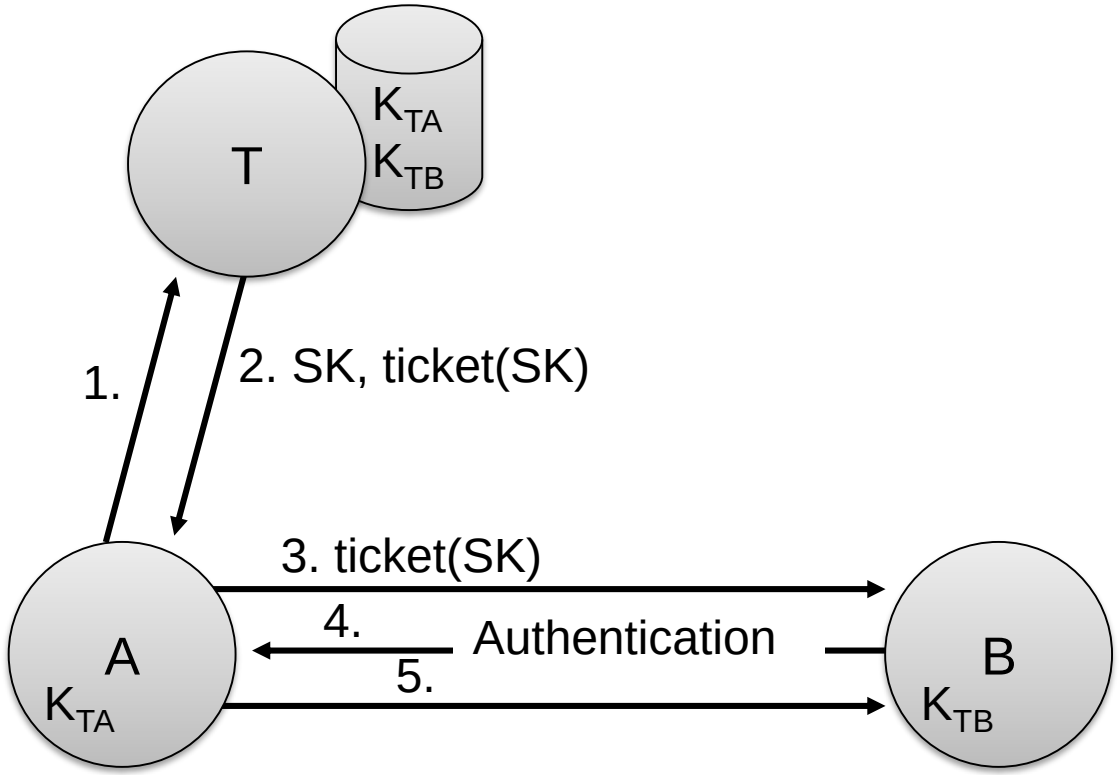
Outline

- Needham-Schroeder secret-key protocol
- Denning-Sacco protocol
- Needham-Schroeder public-key protocol
- Wide-mouth-frog protocol
- Encrypt and sign

These protocol are old designs or early research ideas that must not be used in practice. They are covered in security courses because they illustrate specific security flaws.

Needham-Schroeder secret-key protocol

- The first secret-key key-exchange protocol 1978; basis for Kerberos
- Trusted third party T shares a secret master key with each user
- Alice asks T to create a session key SK for communication with Bob



Needham-Schroeder secret-key protocol

T creates a random session key SK and distributes it encrypted with A's and B's the master keys K_{TA}, K_{TB}

1. A → T: A, B, N_{A1} // ticket request
2. T → A: $E_{TA}(N_{A1}, B, SK, ticket_{AB})$ // ticket grant
3. A → B: $ticket_{AB}$, $E_{SK}(N_{A2})$
4. B → A: $E_{SK}(N_{A2}-1, N_B)$ // authentication and
5. A → B: $E_{SK}(N_B-1)$ // key confirmation

$ticket_{AB} = E_{TB}(SK, A)$ // encrypt and MAC

Needham-Schroeder analysis

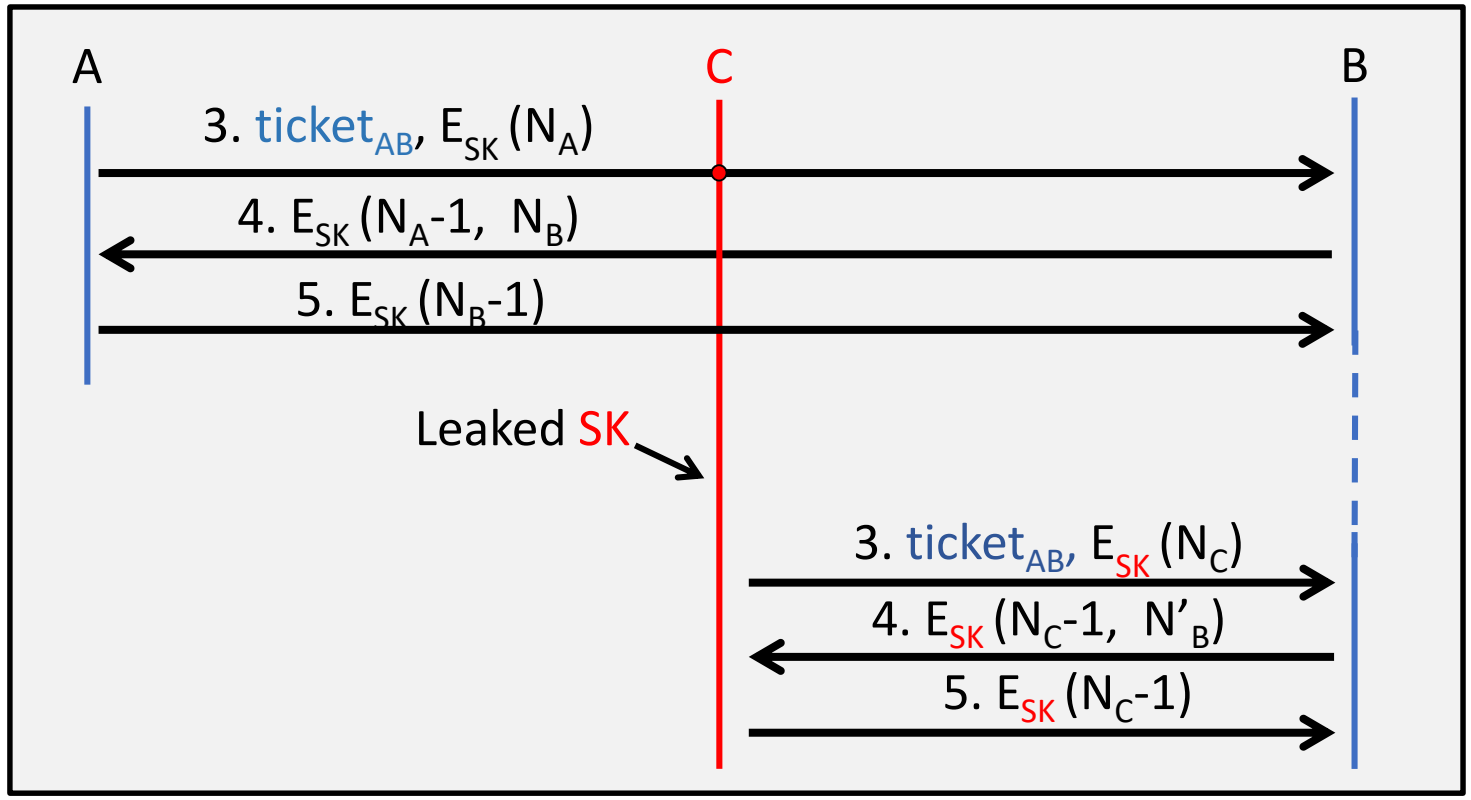
1. A → T: A, B, N_{A1}
2. T → A: $E_{TA}(N_{A1}, B, SK, ticket_{AB})$ // $ticket_{AB} = E_{TB}(SK,A)$
3. A → B: $ticket_{AB}$, $E_{SK}(N_{A2})$
4. B → A: $E_{SK}(N_{A2}-1, N_B)$
5. A → B: $E_{SK}(N_B-1)$

- T encrypts a session key under A's and B's master keys
- Master keys K_{TA} and K_{TB} must be strong secrets; weak passwords could can be cracked by trying to decrypt message 2 and the ticket
- Messages 4–5 provide key confirmation

- N_{A1} guarantees freshness of ticket and session key to A
- N_{A2} and N_B guarantee freshness of authenticators to A and B, respectively
- No freshness of the ticket to B…

Needham-Schroeder vulnerability

- Vulnerability discovered by Denning and Sacco 1981
  - B cannot check freshness of the ticket
- Assume attacker C has an old (sniffed) ticket, and that the old session key SK leaks. C pretends to be A:



Lesson: protocol designers should assume compromise of old short-term secrets

How to fix?
How fixed in in Kerberos?

Denning-Sacco protocol

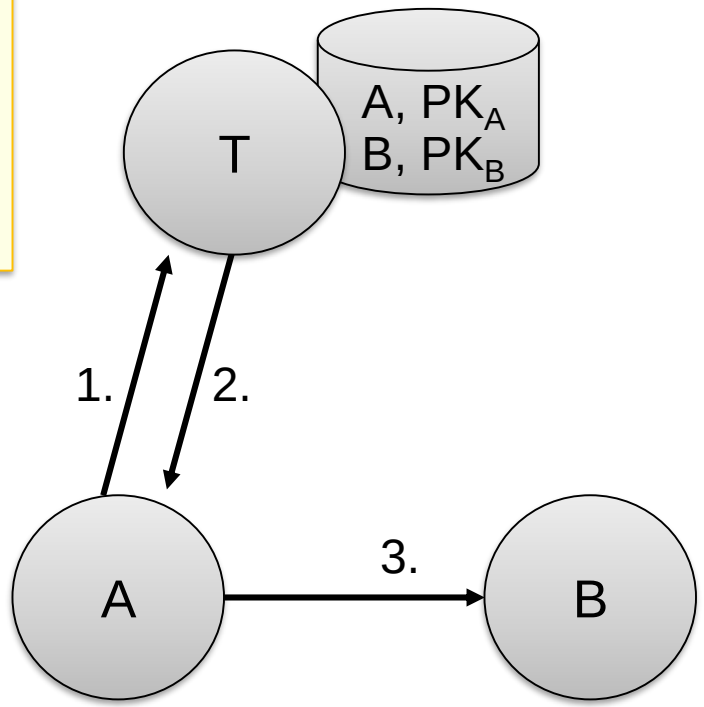
- Public-key key exchange 1981; flaw found in 1994
- A obtains certificates from trusted server T

1. A → T: A, B
2. T → A: $Cert_A$, $Cert_B$
3. A → B: $E_B(T_A, SK, S_A(T_A, SK))$, $Cert_A$, $Cert_B$

SK = session key selected by A

E_B = encryption with B's public key

$Cert_A = A, PK_A, S_T(A, PK_A)$



Denning-Sacco analysis

1. A → T: A, B
2. T → A: $Cert_A$, $Cert_B$
3. A → B: $E_B(T_A, SK, S_A(T_A, SK))$, $Cert_A$, $Cert_B$

SK = session key selected by A

E_B = encryption with B's public key

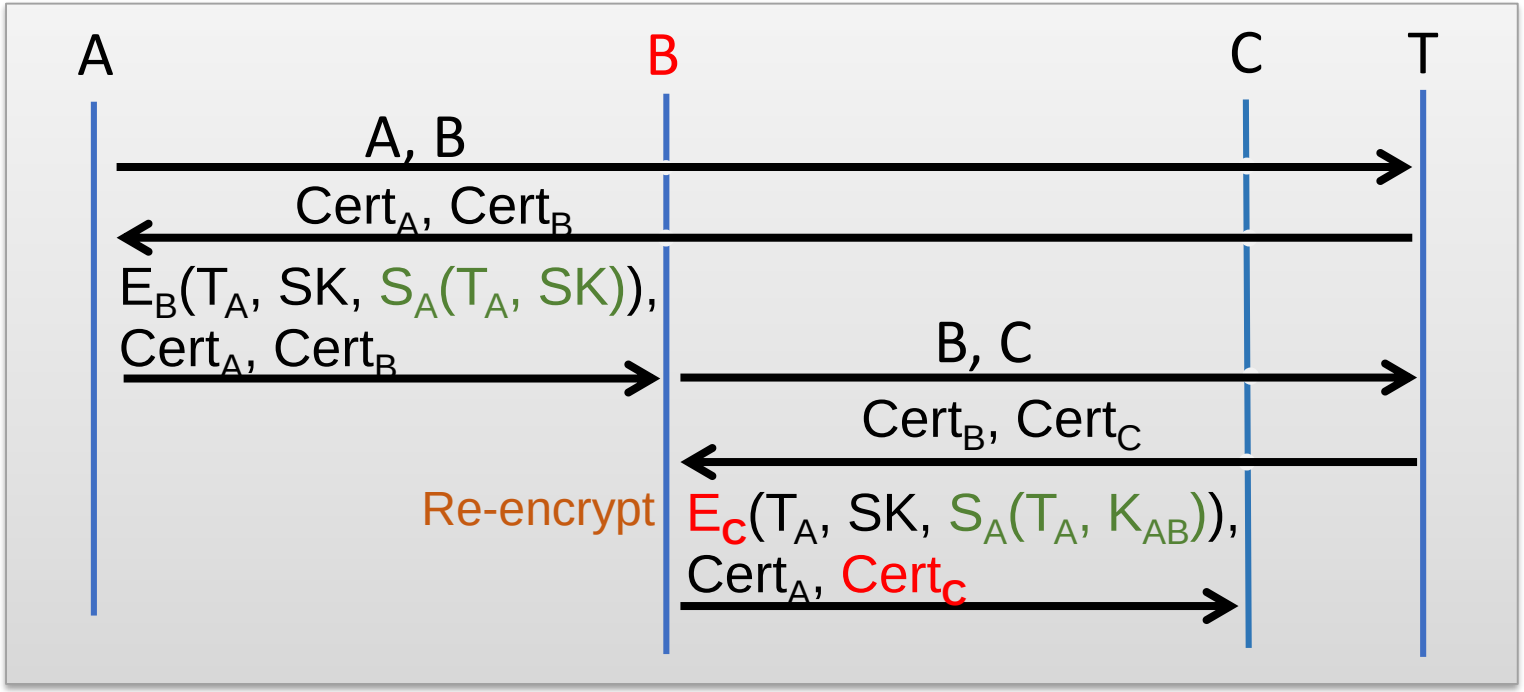
$Cert_A = A, PK_A, S_T(A, PK_A)$

- Should use standard X.509 certificates with expiration time
- Public-key encryption for secrecy of SK → ok
- Time stamp for freshness of the session key → ok
- Public-key signature for authentication →

  what information exactly is authenticated?

Denning-Sacco vulnerability

- The signed part is missing some information: not bound to B's identity
  - Does it matter? Yes, because B could be bad!



Lesson: consider what is *not* authenticated

Lesson: protocols should withstand insider attacks where a legitimate user impersonates another

How to fix?

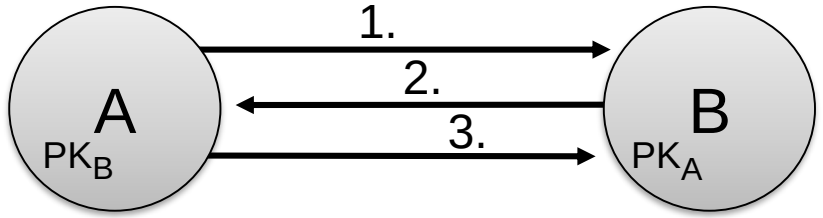
Compare with audience attribute in OpenID Connect identity token.

- Forwarding attack: B can re-encrypt and forward message 3 to others:
  C will think it shares SK with A, but also Bob knows it

Needham-Schroeder public-key protocol

- The first public-key protocol 1978; flaw found in 1995 [Lowe95]
- A and B know each other's public encryption keys (or certificates). Then, A and B exchange encrypted nonces:

1. $A \rightarrow B: E_B(N_A, A)$
2. $B \rightarrow A: E_A(N_A, N_B)$
3. $A \rightarrow B: E_B(N_B)$



N_A, N_B = secret nonces, used both for freshness and as key material

E_A, E_B = encryption with A's or B's public key

$SK = h(N_A, N_B)$

Needham-Schroeder analysis

1. A → B: $E_B(N_A, A)$
2. B → A: $E_A(N_A, N_B)$
3. A → B: $E_B(N_B)$

N_A, N_B = secret nonces, also serving as key material

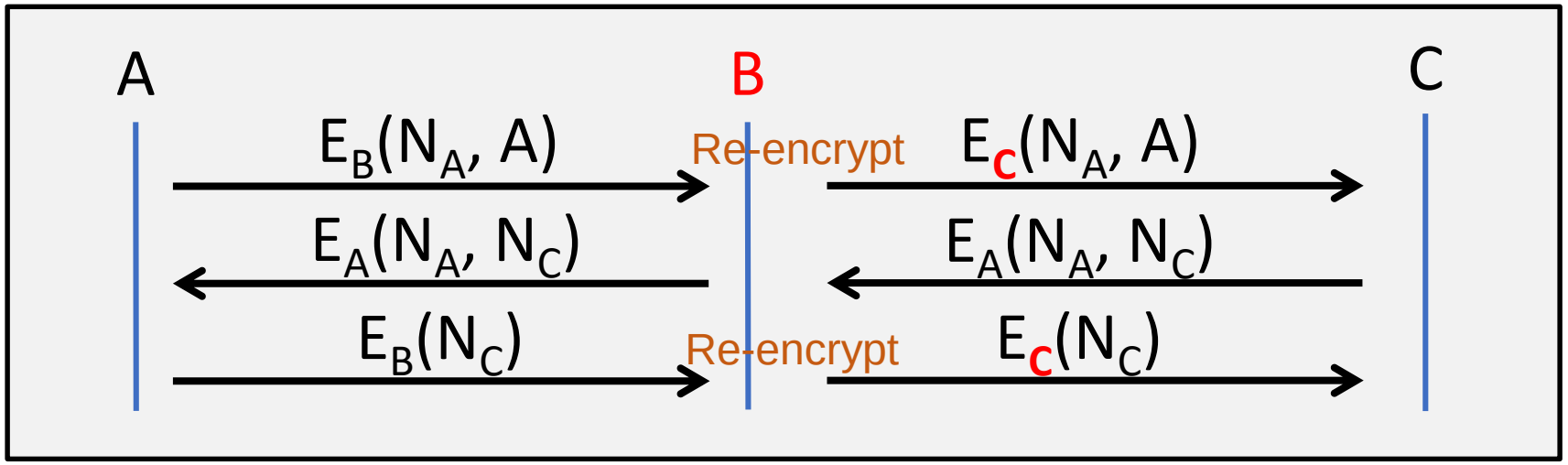
E_A, E_B = encryption with A's or B's public key

$SK = h(N_A, N_B)$

- Session key secret and fresh → ok
- Entity authentication → ok with authenticated encryption
- Key material bound to A but not to B

Needham-Schroeder public-key vulnerability

- A authenticates to B. B can forward the authentication to C:



How to fix?

- C thinks it shares SK with A, but also B knows SK
- Insider attack: legitimate user B impersonates another user A

Another lesson: Consider two or more parallel protocol executions and attacker forwarding messages between them (interleaving attack)

Wide-mouth-frog protocol

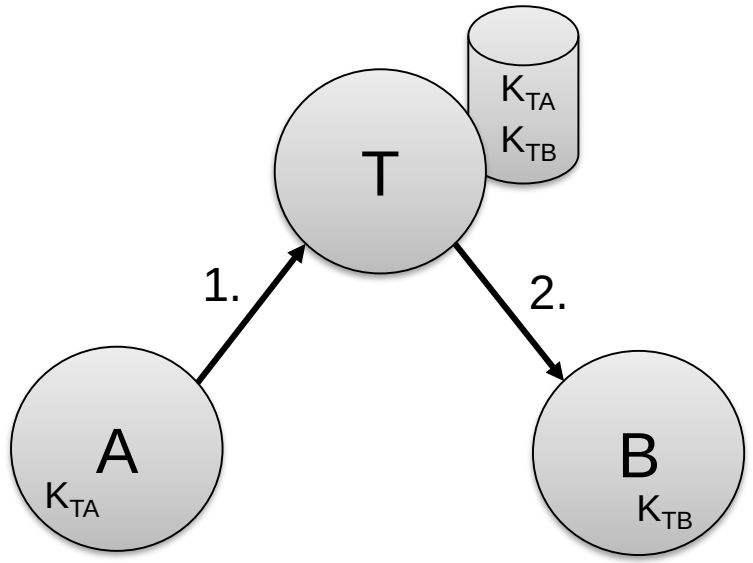
- Toy protocol with interesting flaws
- A and B share secret master keys with trusted server T.
  T distributes session keys:

1. $A \rightarrow T: A, E_{TA}(T_A, B, SK)$
2. $T \rightarrow B: E_{TB}(T_T, A, SK)$

E_{TA}, E_{TB} = encryption with A's and B's master keys

T_A, T_T = time stamps

SK = session key selected by A



Wide-mouth-frog analysis

1. A → T: A, $E_{TA}(T_A, B, SK)$
2. T → B: $E_{TB}(T_T, A, SK)$

E_{TA}, E_{TB} = symmetric encryption with A's and B's master keys

T_A, T_T = time stamps

SK = session key selected by A

- Encryption must protect integrity

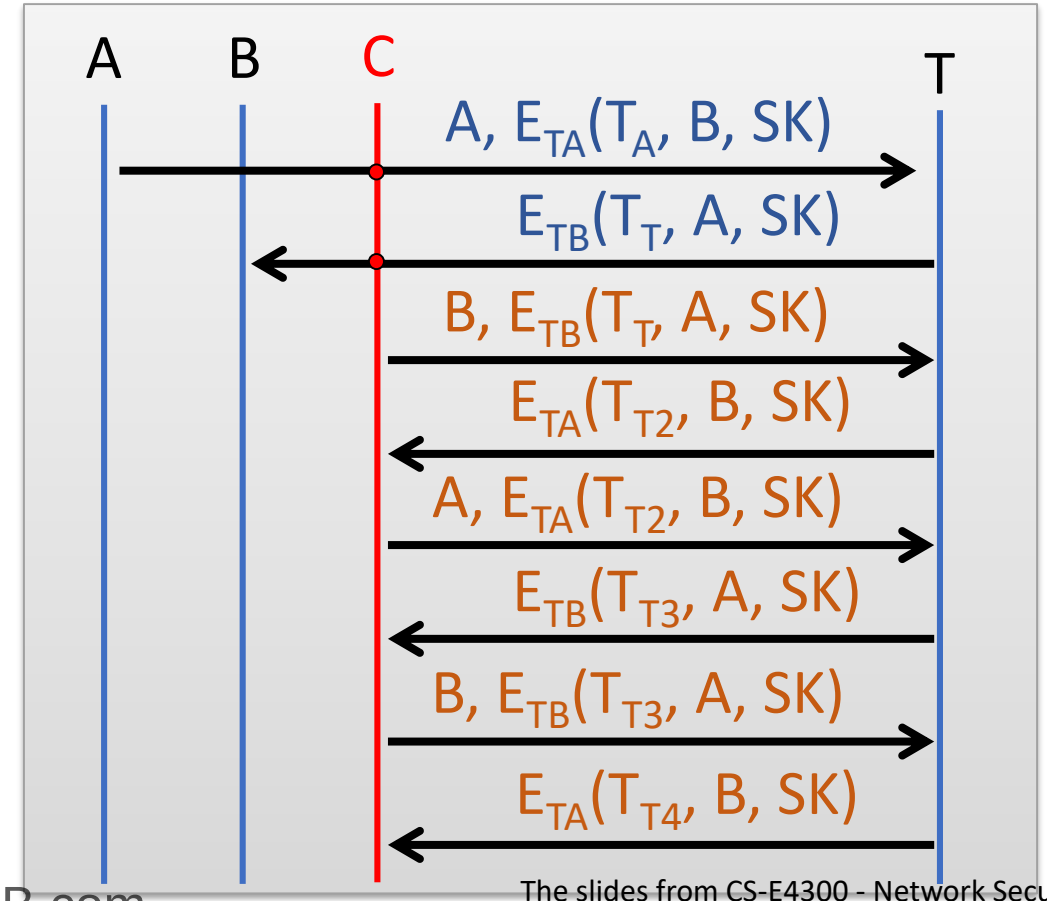
→ implement with a MAC or authenticated encryption

- Subtle issue with the time stamps and message formats…

- It requires a global clock.
- Server has access to all the keys.
- the session key SK is determined by user A.
- It can only replay the messages within the valid timestamp period.
- User A is not certain that user B exists.
- It is a stateful protocol

Wide-mouth-frog vulnerability

- Messages 1 and 2 can be confused with each other → replay attack
- Attacker can refresh timestamps and keep sessions alive for ever



Lesson: Use type tags in all authenticated messages to avoid accidental similarities

Lesson: Don't allow unlimited refreshing of credentials or messages that should expire

How to fix?

Encrypt and sign

- A sends encrypted session key to B:

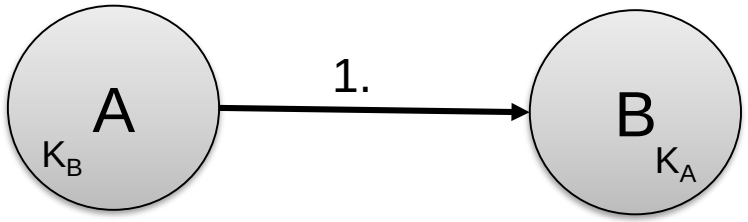
1. $A \rightarrow B$: $A, B, T_A, E_B(SK), S_A(A, B, T_A, E_B(SK))$

SK = session key selected by A

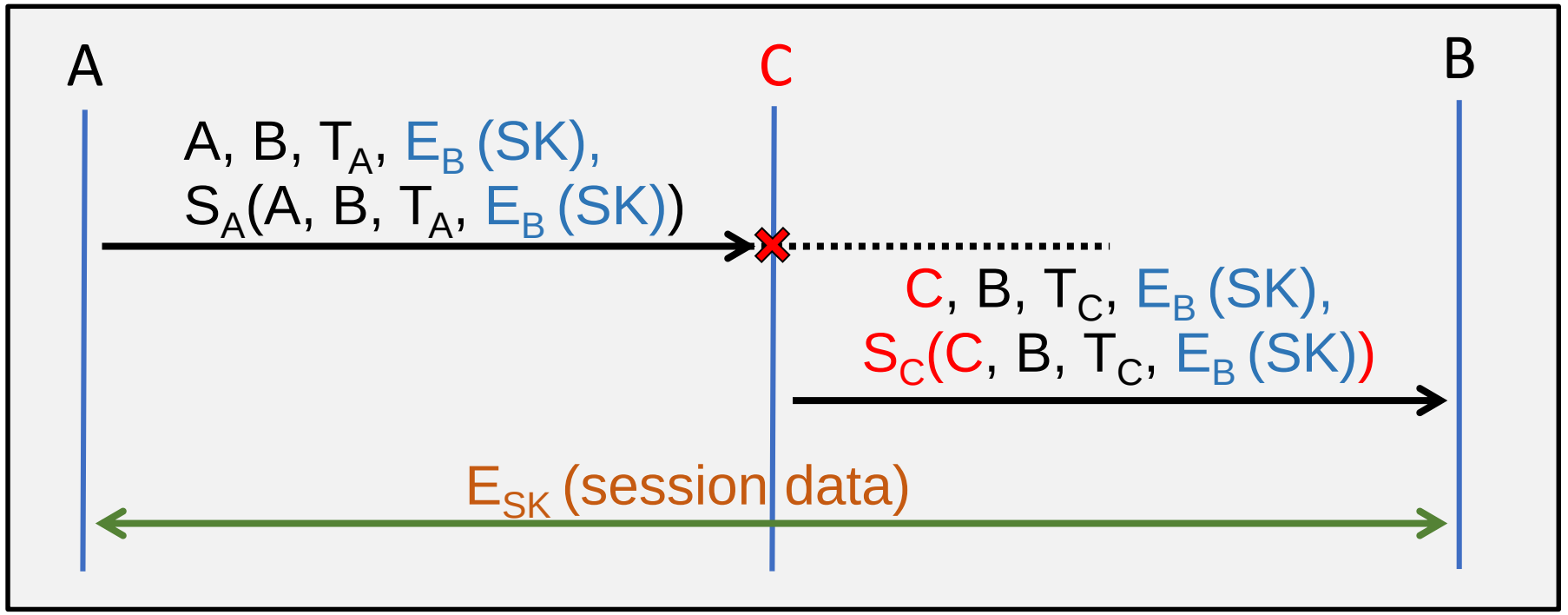
E_B = encryption with B's public key

S_A = A's public-key signature

T_A = time stamp



Encrypt and sign vulnerability



How to fix?

- C sniffs message 1, replaces the signature, and forwards the key as her own

→ Session between A and B, but B thinks it is between C and B

Lesson: In misbinding attacks, attacker causes confusion about who is communicating without learning any keys or secrets herself