Key Management

ECS 153 Spring Quarter 2021

Module 15

Notation

- $X \rightarrow Y : \{ Z \,||\, W \} k_{X,Y}$
  - X sends Y the message produced by concatenating Z and W enciphered by key $k_{X,Y}$, which is shared by users X and Y
- $A \rightarrow T : \{ Z \} k_A \,||\, \{ W \} k_{A,T}$
  - A sends T a message consisting of the concatenation of Z enciphered using k_A, A's key, and W enciphered using $k_{A,T}$, the key shared by A and T
- r_1, r_2 nonces (nonrepeating random numbers)

Session, Interchange Keys

- Alice wants to send a message *m* to Bob
  - Assume public key encryption
  - Alice generates a random cryptographic key k_s and uses it to encipher *m*
    - To be used for this message *only*
    - Called a *session key*
  - She enciphers k_s with Bob;s public key k_B
    - k_B enciphers all session keys Alice uses to communicate with Bob
    - Called an interchange *key*
  - Alice sends $\{ m \} k_s \{ k_s \} k_B$

Benefits

- Limits amount of traffic enciphered with single key
  - Standard practice, to decrease the amount of traffic an attacker can obtain
- Prevents some attacks
  - Example: Alice will send Bob message that is either “BUY” or “SELL”. Eve computes possible ciphertexts { “BUY” } k_B and { “SELL” } k_B. Eve intercepts enciphered message, compares, and gets plaintext at once

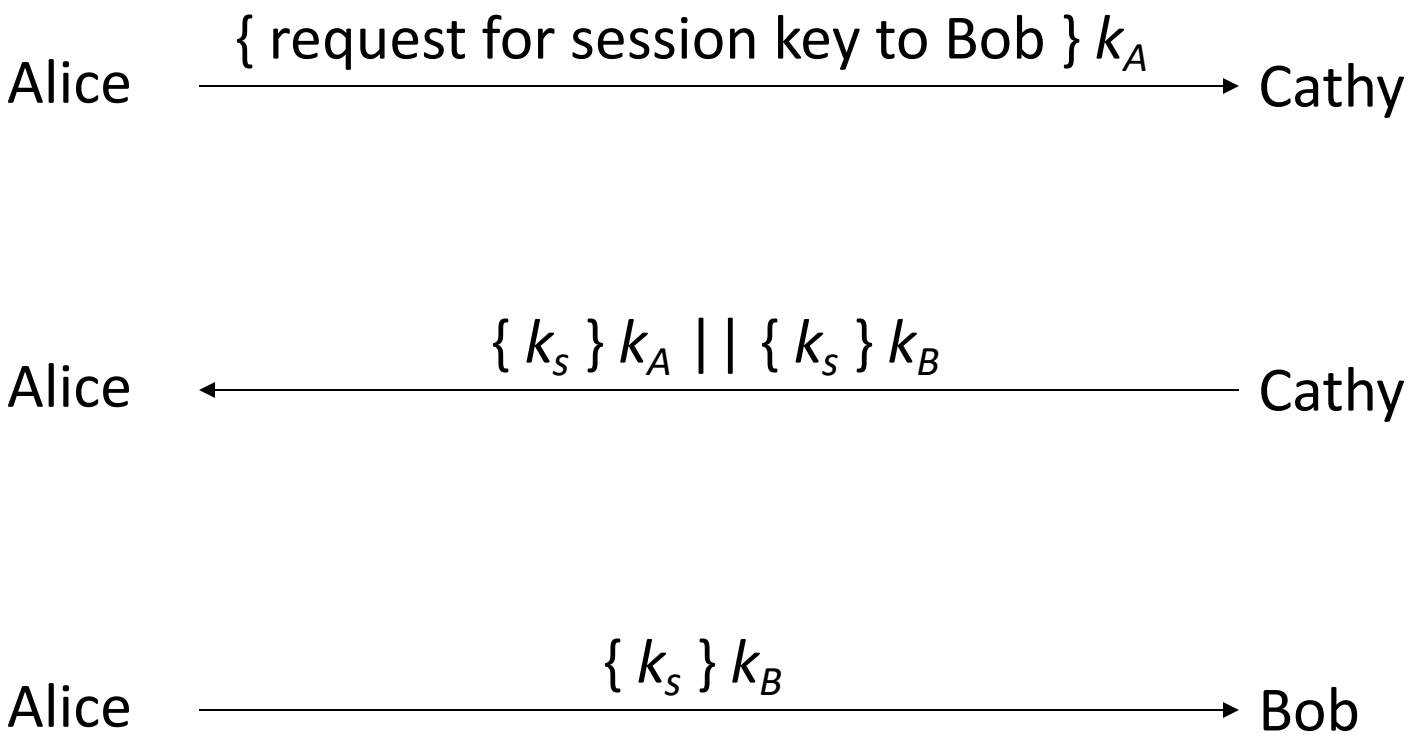
Key Exchange Algorithms

- Goal: Alice, Bob get shared key
  - Key cannot be sent in clear
    - Attacker can listen in
    - Key can be sent enciphered, or derived from exchanged data plus data not known to an eavesdropper
  - Alice, Bob may trust third party
  - All cryptosystems, protocols publicly known
    - Only secret data is the keys, ancillary information known only to Alice and Bob needed to derive keys
    - Anything transmitted is assumed known to attacker

Symmetric Key Exchange

- Bootstrap problem: how do Alice, Bob begin?
  - Alice can't send it to Bob in the clear!
- Assume trusted third party, Cathy
  - Alice and Cathy share secret key k_A
  - Bob and Cathy share secret key k_B
- Use this to exchange shared key k_s

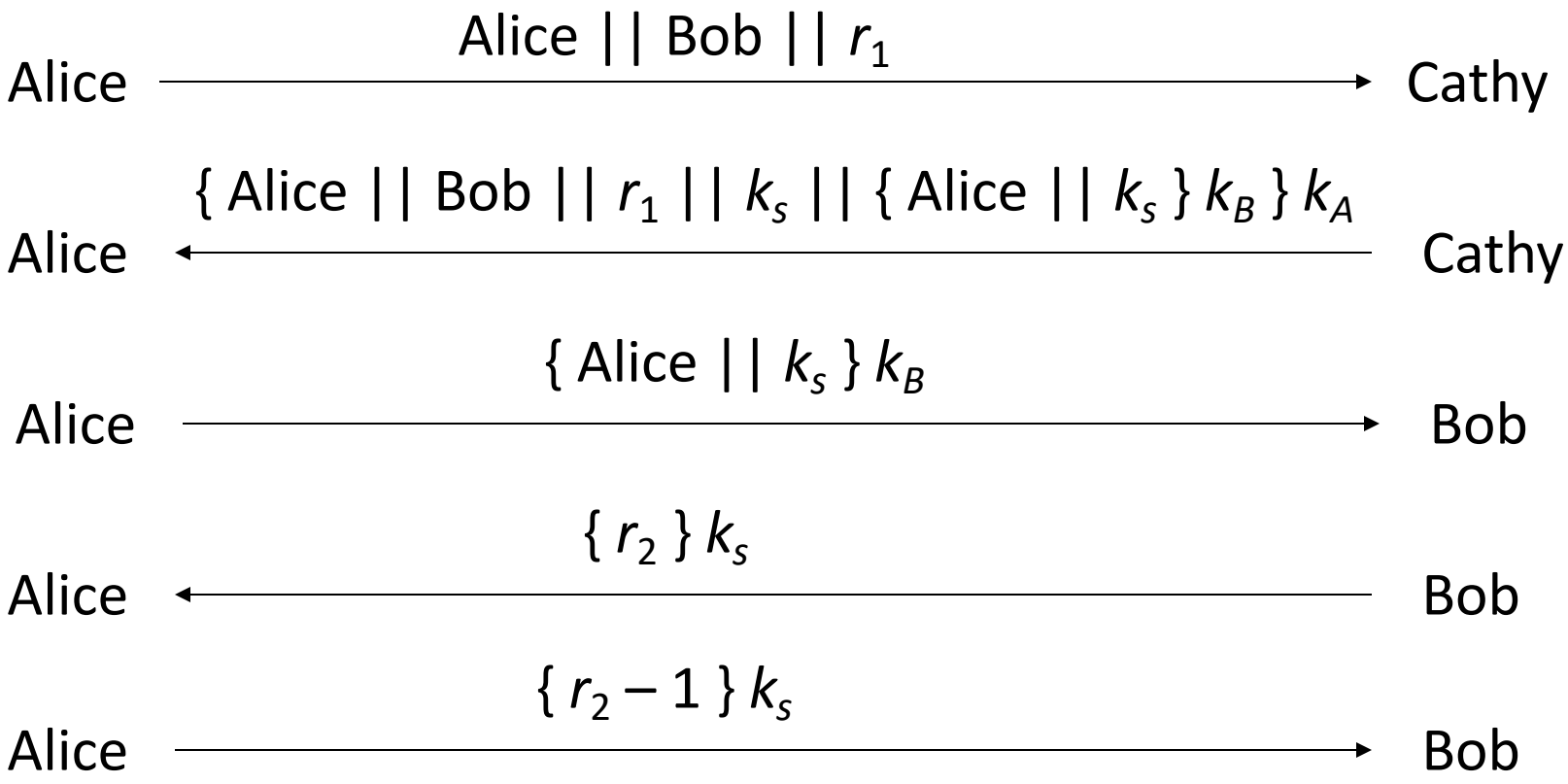
Simple Protocol



Problems

- How does Bob know he is talking to Alice?
  - Replay attack: Eve records message from Alice to Bob, later replays it; Bob may think he's talking to Alice, but he isn't
  - Session key reuse: Eve replays message from Alice to Bob, so Bob re-uses session key
- Protocols must provide authentication and defense against replay

Needham-Schroeder



Argument: Alice talking to Bob

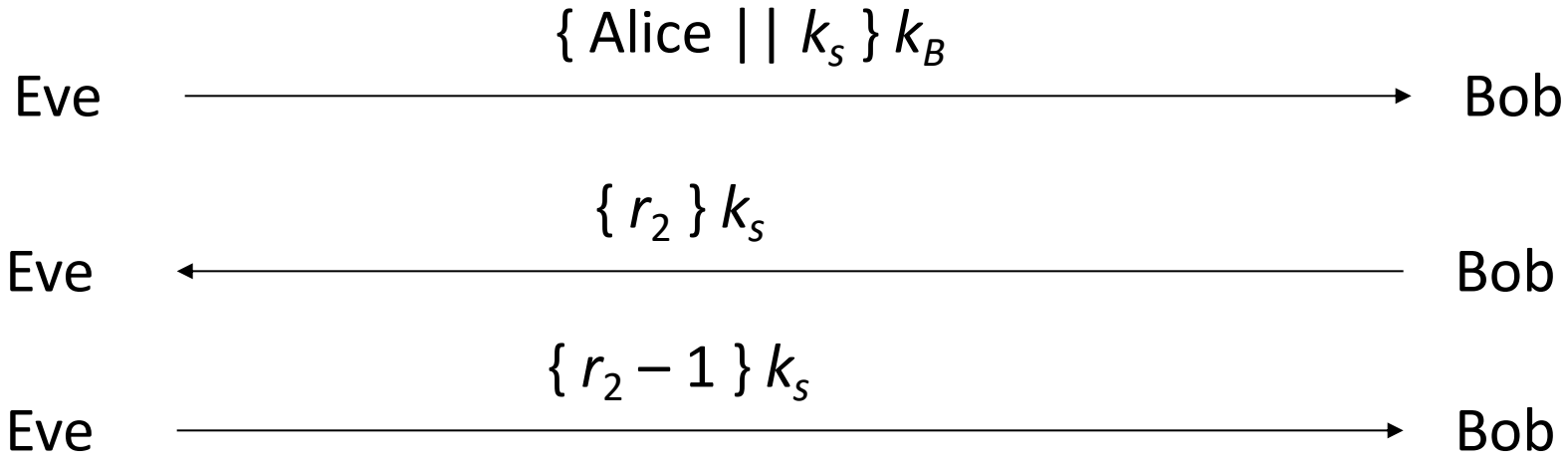
- Second message
  - Enciphered using key only she, Cathy knows
    - So Cathy enciphered it
  - Response to first message
    - As r_1 in it matches r_1 in first message
- Third message
  - Alice knows only Bob can read it
    - As only Bob can derive session key from message
  - Any messages enciphered with that key are from Bob

Argument: Bob talking to Alice

- Third message
  - Enciphered using key only he, Cathy know
    - So Cathy enciphered it
  - Names Alice, session key
    - Cathy provided session key, says Alice is other party
- Fourth message
  - Uses session key to determine if it is replay from Eve
    - If not, Alice will respond correctly in fifth message
    - If so, Eve can't decipher r_2 and so can't respond, or responds incorrectly

Denning-Sacco Modification

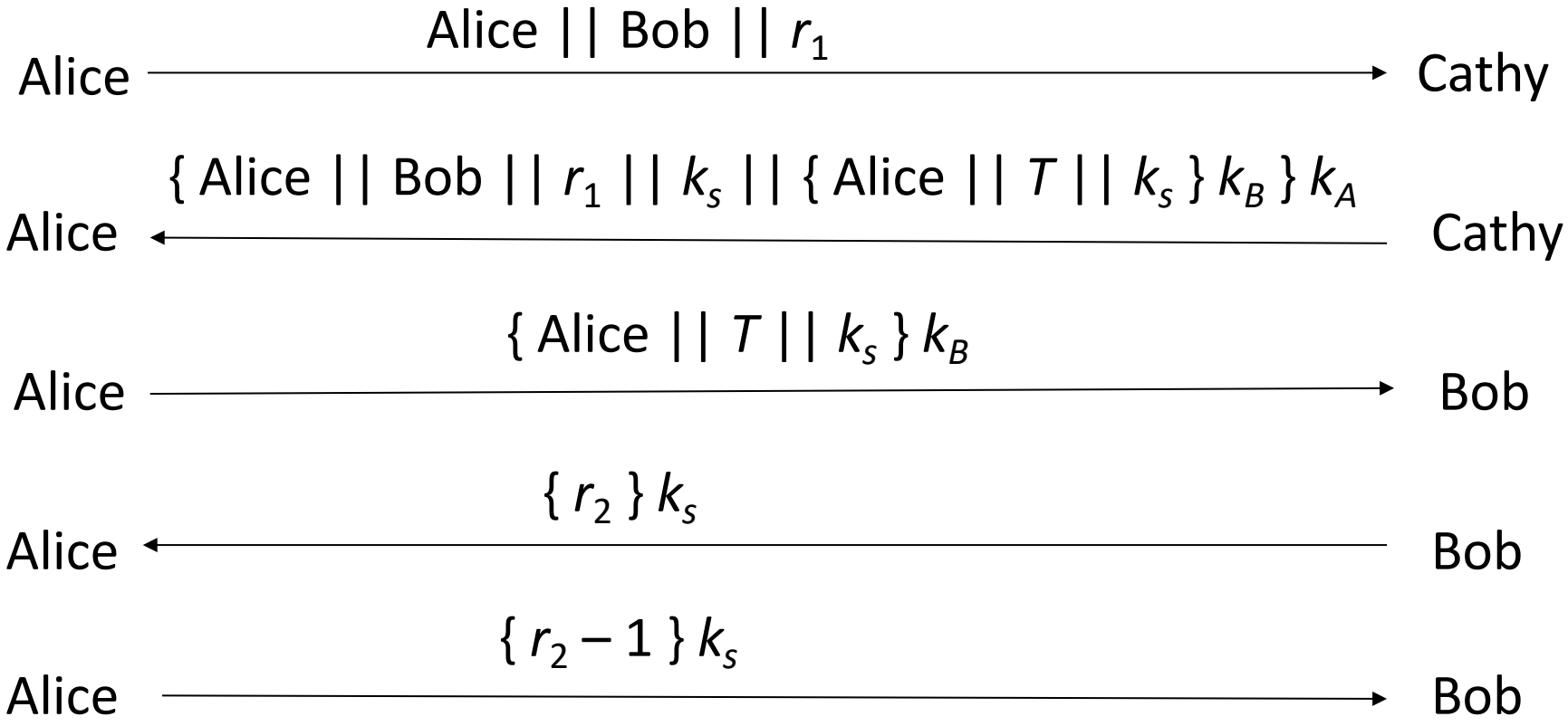
- Assumption: all keys are secret
- Question: suppose Eve can obtain session key. How does that affect protocol?
  - In what follows, Eve knows k_s



Problem and Solution

- In protocol above, Eve impersonates Alice
- Problem: replay in third step
  - First in previous slide
- Solution: use time stamp *T* to detect replay
- Weakness: if clocks not synchronized, may either reject valid messages or accept replays
  - Parties with either slow or fast clocks vulnerable to replay
  - Resetting clock does *not* eliminate vulnerability

Needham-Schroeder with Denning-Sacco Modification



Kerberos

- Authentication system
  - Based on Needham-Schroeder with Denning-Sacco modification
  - Central server plays role of trusted third party (“Cathy”)
- Ticket
  - Issuer vouches for identity of requester of service
- Authenticator
  - Identifies sender

Idea

- User *u* authenticates to Kerberos server
  - Obtains ticket $T_{u,TGS}$ for ticket granting service (TGS)
- User *u* wants to use service *s*:
  - User sends authenticator A_u, ticket $T_{u,TGS}$ to TGS asking for ticket for service
  - TGS sends ticket $T_{u,s}$ to user
  - User sends A_u, $T_{u,s}$ to server as request to use *s*
- Details follow

Ticket

- Credential saying issuer has identified ticket requester
- Example ticket issued to user *u* for service *s*

$$T_{u,s} = s \,||\, \{\, u \,||\, u\text{'s address} \,||\, \text{valid time} \,||\, k_{u,s} \,\}\, k_s$$

where:

- $k_{u,s}$ is session key for user and service
- Valid time is interval for which ticket valid
- *u*'s address may be IP address or something else
  - Note: more fields, but not relevant here

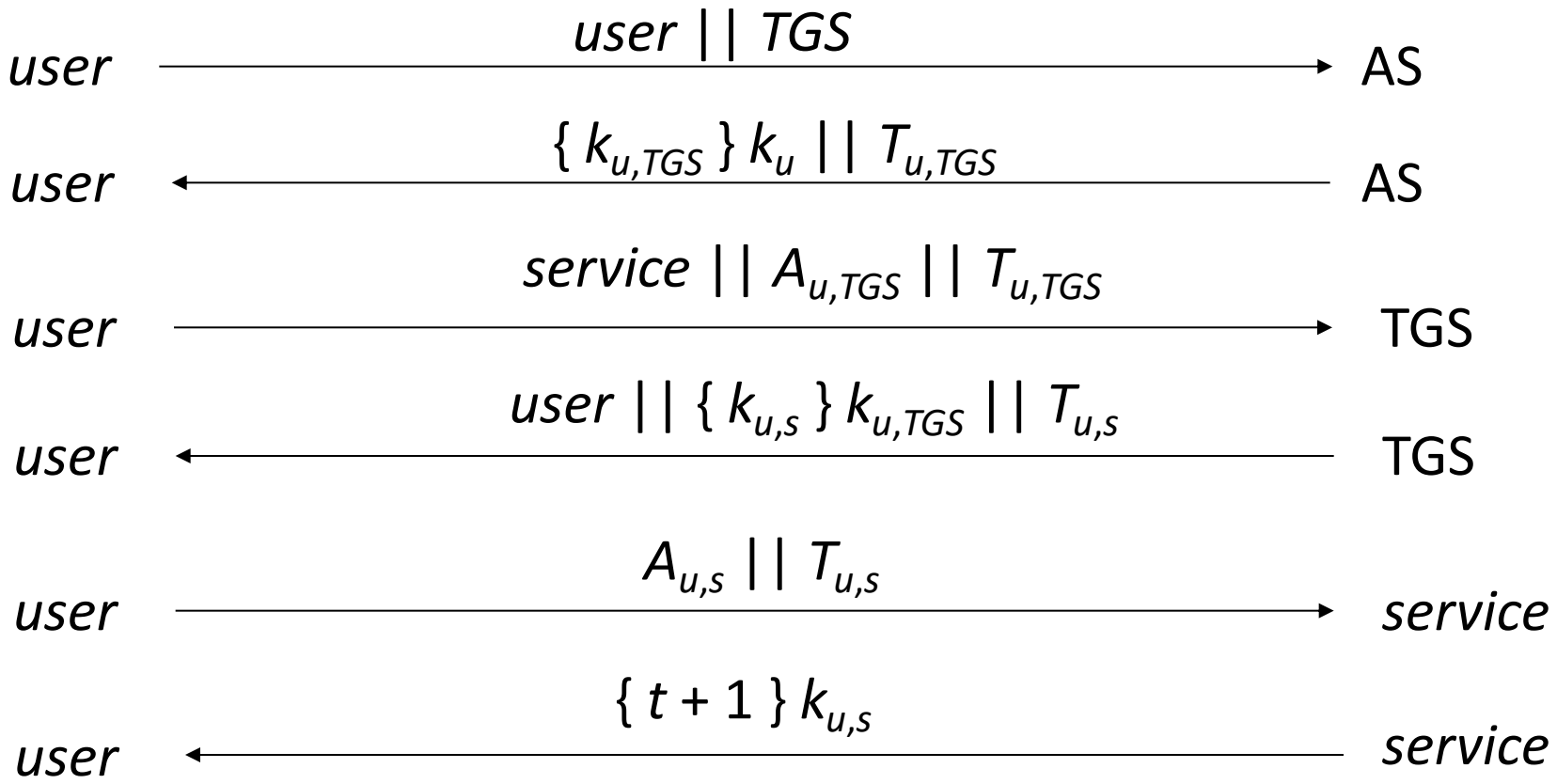
Authenticator

- Credential containing identity of sender of ticket
  - Used to confirm sender is entity to which ticket was issued
- Example: authenticator user *u* generates for service *s*

$$A_{u,s} = \{ u \,||\, \text{generation time} \,||\, k_t \}\, k_{u,s}$$

  where:
  - k_t is alternate session key
  - Generation time is when authenticator generated
    - Note: more fields, not relevant here

Protocol


user
user || TGS
AS
user
{ $k_{u,TGS}$ } k_u || $T_{u,TGS}$
AS
user
service || $A_{u,TGS}$ || $T_{u,TGS}$
TGS
user
user || { $k_{u,s}$ } $k_{u,TGS}$ || $T_{u,s}$
TGS
user
$A_{u,s}$ || $T_{u,s}$
service
user
{ t + 1 } $k_{u,s}$
service

Analysis

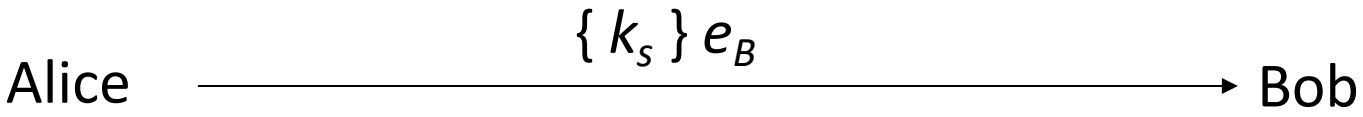
- First two steps get user ticket to use TGS
  - User *u* can obtain session key only if *u* knows key shared with AS
- Next four steps show how *u* gets and uses ticket for service *s*
  - Service *s* validates request by checking sender (using $A_{u,s}$) is same as entity ticket issued to
  - Step 6 optional; used when *u* requests confirmation

Problems

- Relies on synchronized clocks
  - If not synchronized and old tickets, authenticators not cached, replay is possible
- Tickets have some fixed fields
  - Dictionary attacks possible
  - Kerberos 4 session keys weak (had much less than 56 bits of randomness); researchers at Purdue found them from tickets in minutes

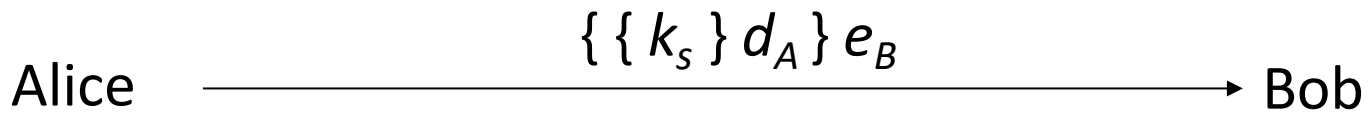
Public Key Key Exchange

- Here interchange keys known
  - e_A, e_B Alice and Bob's public keys known to all
  - d_A, d_B Alice and Bob's private keys known only to owner
- Simple protocol
  - k_s is desired session key



Problem and Solution

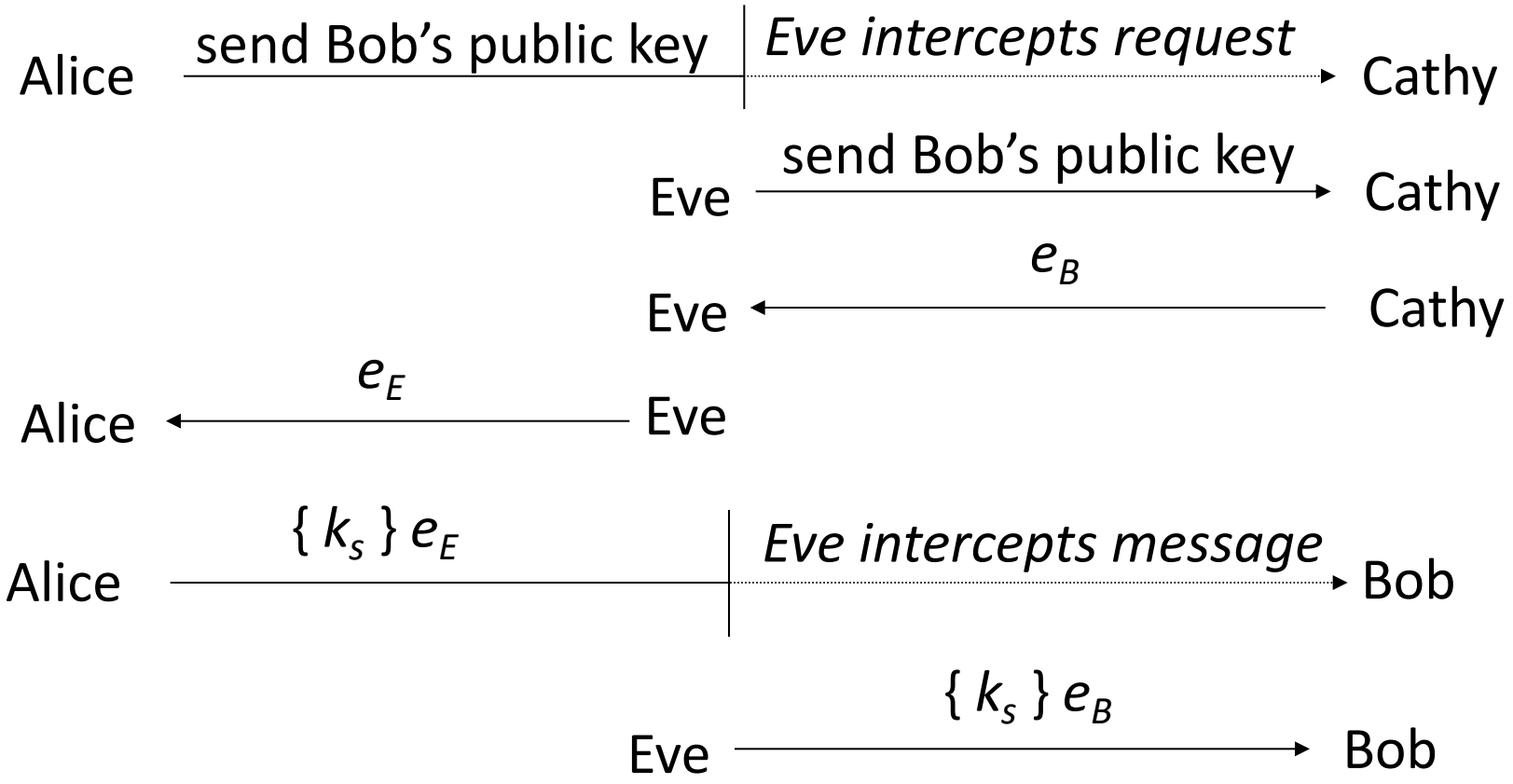
- Vulnerable to forgery or replay
  - Because e_B known to anyone, Bob has no assurance that Alice sent message
- Simple fix uses Alice's private key
  - k_s is desired session key



Notes

- Can include message enciphered with k_s
- Assumes Bob has Alice's public key, and *vice versa*
  - If not, each must get it from public server
  - If keys not bound to identity of owner, attacker Eve can launch a *man-in-the-middle* attack (next slide; Cathy is public server providing public keys)
    - Solution to this (binding identity to keys) discussed later as public key infrastructure (PKI)

Man-in-the-Middle Attack



Diffie-Hellman

- Compute a common, shared key
  - Called a *symmetric key exchange protocol*
- Based on discrete logarithm problem
  - Given integers n, g and prime number p, compute k such that $n = g^k \bmod p$
  - Solutions known for small p
  - Solutions computationally infeasible as p grows large

Algorithm

- Constants: prime p, integer $g \neq 0, 1, p-1$
  - Known to all participants
- Alice chooses private key k_{Alice}, computes public key $K_{Alice} = g^{k_{Alice}} \bmod p$
- Bob chooses private key k_{Bob}, computes public key $K_{Bob} = g^{k_{Bob}} \bmod p$
- To communicate with Bob, Anne computes $K_{Alice,Bob} = K_{Bob}^{\ k_{Alice}} \bmod p$
- To communicate with Anne, Bob computes $K_{Bob,Alice} = K_{Alice}^{\ k_{Bob}} \bmod p$
- It can be shown $K_{Alice,Bob} = K_{Bob,Alice}$

Example

- Assume $p = 121001$ and $g = 6981$
- Alice chooses $k_{Alice} = 526784$
  - Then $K_{Alice} = 6981^{26874} \bmod 121001 = 22258$
- Bob chooses $k_{Bob} = 5596$
  - Then $K_{Bob} = 6981^{5596} \bmod 121001 = 112706$
- Shared key:
  - $K_{Bob}^{\ k_{Alice}} \bmod p = 112706^{26874} \bmod 121001 = 78618$
  - $K_{Alice}^{\ k_{Bob}} \bmod p = 22258^{5596} \bmod 121001 = 78618$

Example (Elliptic Curve Version)

- Alice, Bob agree to use the curve $y^2 = x^3 + 4x + 14 \bmod 2503$ and the point $P = (1002, 493)$; curve has $n = 2428$ integer points
- Alice chooses $k_{Alice} = 1379$
  - Then $K_{Alice} = k_{Alice} P \bmod p = 1379(1002,493) \bmod 2503 = (1041,1659)$
- Bob chooses $k_{Bob} = 2011$
  - Then $K_{Bob} = k_{Bob} P \bmod p = 2011(1002,493) \bmod 2503 = (629,548)$
- Shared key:
  - $K_{Bob} k_{Alice} \bmod p = 2011(1041,1659) \bmod 2503 = (2075,2458)$
  - $K_{Alice} k_{Bob} \bmod p = 1379(629,548) \bmod 2503 = (2075,2458)$

Key Generation

- Goal: generate keys that are difficult to guess
- Problem statement: given a set of *k* potential keys, choose one randomly
  - Equivalent to selecting a random number between 0 and *k*–1 inclusive
- Why is this hard: generating random numbers
  - Actually, numbers are usually *pseudorandom*, that is, generated by an algorithm

What is "Random"?

- *Sequence of cryptographically random numbers*: a sequence of numbers n_1, n_2, … such that for any integer $k > 0$, an observer cannot predict n_k even if all of n_1, …, n_{k-1} are known
- Best: physical source of randomness
  - Random pulses
  - Electromagnetic phenomena
  - Characteristics of computing environment such as disk latency
  - Ambient background noise

What is “Pseudorandom”?

- *Sequence of cryptographically pseudorandom numbers*: sequence of numbers intended to simulate a sequence of cryptographically random numbers but generated by an algorithm
- Very difficult to do this well
  - Linear congruential generators [$x_k = (ax_{k-1} + b) \bmod n$] broken
  - Polynomial congruential generators [$x_k = (a_j x_{k-1}^{\,j} + \ldots + a_1 x_{k-1}\, a_0) \bmod n$] broken too
  - Here, “broken” means next number in sequence can be determined

Best Pseudorandom Numbers

- *Strong mixing function*: function of 2 or more inputs with each bit of output depending on some nonlinear function of all input bits
  - Examples: AES, SHA-512, SHA-3
  - Use on UNIX-based systems:

```
(date; ps gaux) | sha512
```

    where "`ps gaux`" lists all information about all processes on system

Biometrics

- Physical variations cause slight differences in successive biometric readings and so is good source of randomness
  - This causes randomness in the least significant bits of the data
- Biometrics for generating keys tied to individuals
  - Requires: adversary unlikely to determine them, but must be regenerated consistently
- Represent data as bit string (*feature descriptor*)
  - Transform it in some way
  - Generate cryptographic key from this
  - Add some randomness so if key compromised, a new and different one can be created

Cryptographic Key Infrastructure

- Goal: bind identity to key
- Symmetric: not possible as all keys are shared
  - Use protocols to agree on a shared key (see earlier)
- Public key: bind identity to public key
  - Crucial as people will use key to communicate with principal whose identity is bound to key
  - Erroneous binding means no secrecy between principals
  - Assume principal identified by an acceptable name

Certificates

- Create token (message) containing
  - Identity of principal (here, Alice)
  - Corresponding public key e_{Alice}
  - Timestamp (when issued)
  - Other information (perhaps identity of signer)

signed by trusted authority (here, Cathy)

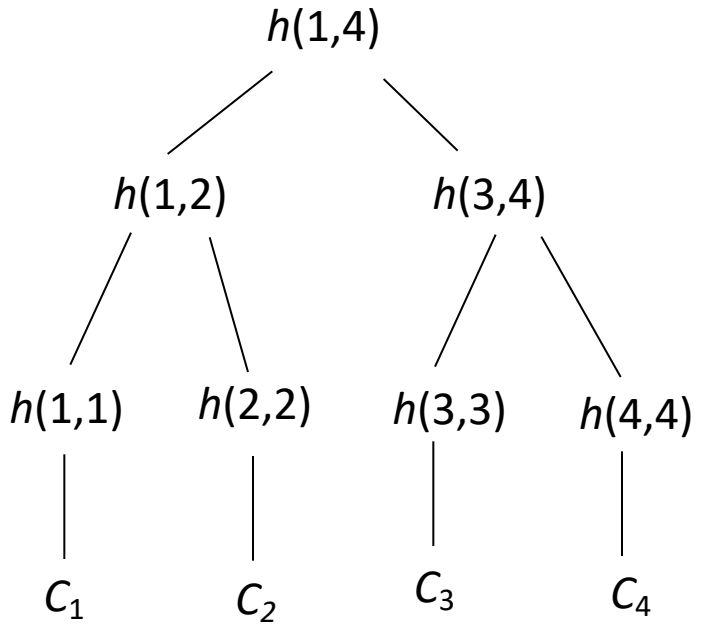
$$C_{\text{Alice}} = \{ e_{\text{Alice}} \,||\, \text{Alice} \,||\, T \}\, d_{\text{Cathy}}$$

Use

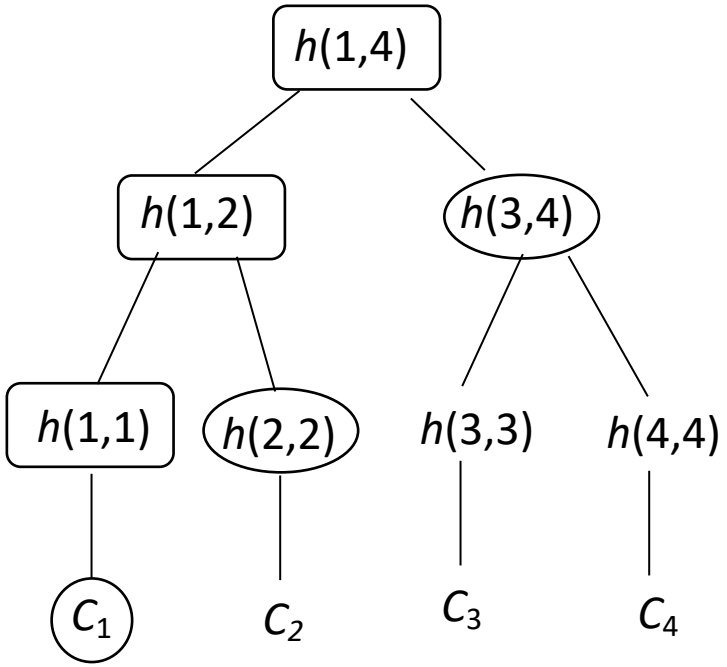
- Bob gets Alice's certificate
  - If he knows Cathy's public key, he can decipher the certificate
    - When was certificate issued?
    - Is the principal Alice?
  - Now Bob has Alice's public key
- Problem: Bob needs Cathy's public key to validate certificate
  - Problem pushed "up" a level
  - Two approaches: Merkle's tree, signature chains

Merkle's Tree Scheme

- Keep certificates in a file
  - Changing any certificate changes the file
  - Use crypto hash functions to detect this
- Define hashes recursively
  - h is hash function
  - C_i is certificate i
- Hash of file ($h(1,4)$ in example) known to all



Validation



- To validate C_1:
  - Compute $h(1, 1)$
  - Obtain $h(2, 2)$
  - Compute $h(1, 2)$
  - Obtain $h(3, 4)$
  - Compute $h(1,4)$
  - Compare to known $h(1, 4)$
- Need to know hashes of children of nodes on path that are not computed
- In drawing at left:
  - Circle contains what is to be validated
  - Ovals are what are to be obtained
  - Curved rectangles are what are to be computed

Details

- f: $D \times D \rightarrow D$ maps bit strings to bit strings
- h: $N \times N \rightarrow D$ maps integers to bit strings
  - if $i \geq j$, $h(i, j) = f(C_i, C_j)$
  - if $i < j$, $h(i, j) = f(h(i, \lfloor (i+j)/2 \rfloor), h(\lfloor (i+j)/2 \rfloor + 1, j))$

Problem

- File must be available for validation
  - Otherwise, can't recompute hash at root of tree
  - Intermediate hashes would do
- Not practical in most circumstances
  - If any public key changed, validation fails unless tree is updated
  - This includes compromised certificates as well as legitimate public key changes
  - If copies of tree are widely distributed, a change to one must be reflected by all

Certificate Signature Chains

- Create certificate
  - Generate hash of certificate
  - Encipher hash with issuer's private key
- Validate
  - Obtain issuer's public key
  - Decipher enciphered hash
  - Recompute hash from certificate and compare
- Problem: getting issuer's public key

X.509 Chains

- Some certificate components in X.509v3:
  - Version
  - Serial number
  - Signature algorithm identifier: hash algorithm
  - Issuer's name; uniquely identifies issuer
  - Interval of validity
  - Subject's name; uniquely identifies subject
  - Subject's public key
  - Signature: enciphered hash

X.509 Certificate Validation

- Obtain issuer's public key
  - The one for the particular signature algorithm
- Decipher signature
  - Gives hash of certificate
- Recompute hash from certificate and compare
  - If they differ, there's a problem
- Check interval of validity
  - This confirms that certificate is current

Issuers

- *Certification Authority* (*CA*): entity that issues certificates
  - Multiple issuers pose validation problem
  - Alice's CA is Cathy; Bob's CA is Don; how can Alice validate Bob's certificate?
  - Have Cathy and Don cross-certify by issuing certificates for each other

Validation and Cross-Certifying

- Notation: *X*<<*Y*>> means *X* issues certificate for *Y*
- Certificates:
  - Cathy<<Alice>>
  - Dan<<Bob>
  - Cathy<<Dan>>
  - Dan<<Cathy>>
- Alice validates Bob's certificate
  - Alice obtains Cathy<<Dan>>
  - Alice uses (known) public key of Cathy to validate Cathy<<Dan>>
  - Alice uses Cathy<<Dan>> to validate Dan<<Bob>>

PGP Chains

- OpenPGP certificates structured into packets
  - One public key packet
  - Zero or more signature packets
- Public key packet:
  - Version (3 or 4; 3 compatible with all versions of PGP, 4 not compatible with older versions of PGP)
  - Creation time
  - Validity period (present in version 3 only)
  - Public key algorithm, associated parameters
  - Public key

OpenPGP Signature Packet

- Version 3 signature packet
  - Version (3)
  - Signature type (level of trust)
  - Creation time (when next fields hashed)
  - Signer's key identifier (identifies key to encipher hash)
  - Public key algorithm (used to encipher hash)
  - Hash algorithm
  - Part of signed hash (used for quick check)
  - Signature (enciphered hash)
- Version 4 packet more complex

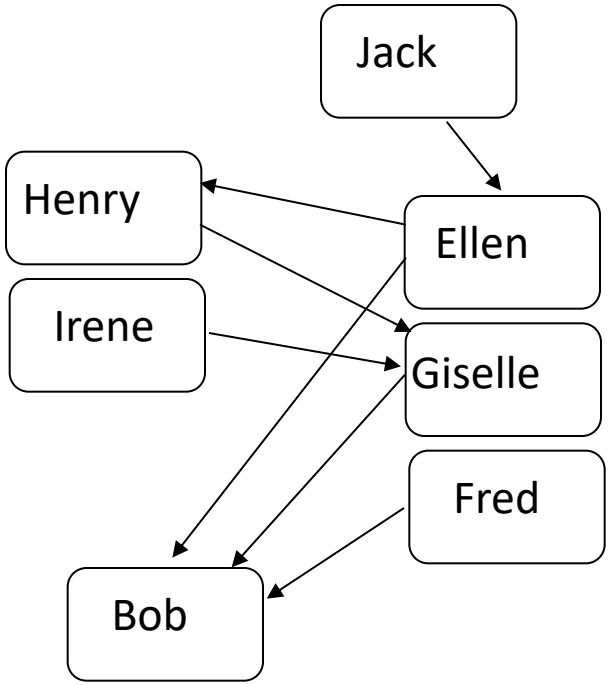
Signing

- Single certificate may have multiple signatures
- Notion of “trust” embedded in each signature
  - Range from “untrusted” to “ultimate trust”
  - Signer defines meaning of trust level (no standards!)
- All version 4 keys signed by the subject of the certificate
  - Called “self-signing”
  - Version 3 certificates can be too

Validating Certificates

- Alice needs to validate Bob's OpenPGP cert
  - Does not know Fred, Giselle, or Ellen
- Alice gets Giselle's cert
  - Knows Henry slightly, but his signature is at "casual" level of trust
- Alice gets Ellen's cert
  - Knows Jack, so uses his cert to validate Ellen's, then hers to validate Bob's

Arrows show signatures
Self signatures not shown



Public Key Infrastructures (PKIs)

- An infrastructure that manages public keys and certificate authorities
  - This includes registration authorities and other entities involved in creating and issuing certificates

Internet X.509 PKI

- *End entity certificate*: a certificate issued to entities not authorized to issue certificates
- *Certificate authority certificate*: a certificate issued to a CA
  - *Self-issued*: issuer, subject are the same entity
  - *Self-signed*: self-issued certificate in which public key in certificate can be used to validate that certificate's digital signature
- *Trust anchor*: CA that begins a certificate signature chain
- *Cross-certificate*: certificate for one CA issued by another CA
- *Registration authority*: entity delegated the registration task by a CA
  - CA trusts RA to properly identify, authenticate, validate entity

Certificate Extensions

- Critical: mandatory accept or reject, depending on content
  - If application can't recognize or process it, certificate rejected
- Non-critical: can be ignored if unrecognized
- All conforming CAs must support the following:
  - Authority key identifier: identifies public key used to validate certificate's digital signature
    - Must not be marked critical
  - Subject key identifier: same value of authority key field, but if subject is CA, this must be present
    - Must not be marked critical

CA Certificate Extension Support

- All conforming CAs must support the following:
  - Key usage: describes purposes for which public key can be used
    - If certificate used to validate digital signatures on certificates, must be present
    - Should be marked critical
  - Basic constraints: identifies whether subject is CA if the certificate can be used to validate another certificate's digital signature, number of intermediate certificates that may follow this one in a chain and that are not self-signed
    - Must be critical if certificate used to validate digital signatures of certificates
    - May be critical or non-critical otherwise
  - Certificate policies: describes policy under which certificate is issued and what it can be used for
    - Should be marked critical

CA Certificate Extensions

- Authority key identifier eliminates need to try different keys of issuing CA to determine whether certificate valid
  - In earlier versions of Internet PKI, this also indicated applicable policy
  - Key usage, certificate policy extensions now do this explicitly
- Key usage makes clear what public key is to be used for
  - Before, assumed valid for any purpose, or embedded in issuer's policy
- Basic constraints limits length of certificate chain beginning here
  - Doesn't include self-signed certificates

Conforming Applications Certificate Extensions

- Conforming applications that process certificates must recognize:
  - *Key usage*, *certificate policies*, *basic constraints* extensions
  - *Subject alternative name*: another name for subject; must be verified by CA or RA
    - Must be critical
  - *Name constraints*: constrains names in subject, subject alternative name of non-self-signed certificates following it in certificate chain
  - *Policy constraints*: controls when policy for chain containing this certificate must be explicit or when policy of issuer need not be same as that of subject
    - Must be critical

Conforming Applications Certificate Extensions

- Conforming applications that process certificates must recognize:
  - Extended key usage: issuer uses this to specify uses of public key beyond those in the key usage extension
  - Inhibit anyPolicy: wildcard (anyPolicy) matches policies only if it occurs in intermediate self-signed certificate in certificate chain
    - Must be critical
- Subject alternative name allows multiple subject names in certificate
  - Previous versions allowed only one subject name per certificate
- Extended key usage allows public key to be used in ways not identified in key usage

PKI Problems

Basis for any PKI is trust

- Trust that the binding of identity to public key is correct
  - Degree of confidence depends on CA or RA
- Trust that appropriate CA issued the certificate
  - Also that issuance policies are understood
  - Also that implementation of signing, and PKI mechanisms,
- Certificate does not embody authorization
  - Identity may, but that is external to PKI
- Trust that no 2 certificates will have same public (and hence private) key

Key Revocation

- Certificates invalidated *before* expiration
  - Usually due to compromised key
  - May be due to change in circumstance (*e.g.,* someone leaving company)
- Problems
  - Entity revoking certificate authorized to do so
  - Revocation information circulates to everyone fast enough
    - Network delays, infrastructure problems may delay information

CRLs

- *Certificate revocation list* lists certificates that are revoked
- X.509: only certificate issuer can revoke certificate
  - Added to CRL
- PGP: signers can revoke signatures; owners can revoke certificates, or allow others to do so
  - Revocation message placed in PGP packet and signed
  - Flag marks it as revocation message