Distributed Systems

Authentication Protocols

Paul Krzyzanowski
pxk@cs.rutgers.edu

Authentication

Establish and verify identity
- allow access to resources

Authentication

Three factors:
- something you have   *key, card*
  - can be stolen

- something you know   *passwords*
  - can be guessed, shared, stolen

- something you are   *biometrics*
  - costly, can be copied (sometimes)

Authentication

factors may be combined

    – ATM machine: <u>2-factor authentication</u>

        • ATM card   *something you have*
        • PIN   *something you know*

Password Authentication Protocol (PAP)

- Reusable passwords

- Server keeps a database of username:password mappings

- Prompt client/user for a login name & password

- To authenticate, use the login name as a key to look up the corresponding password in a database (file) to authenticate

  if (supplied_password == retrieved_password)
      user is authenticated

Authentication: PAP

Password Authentication Protocol
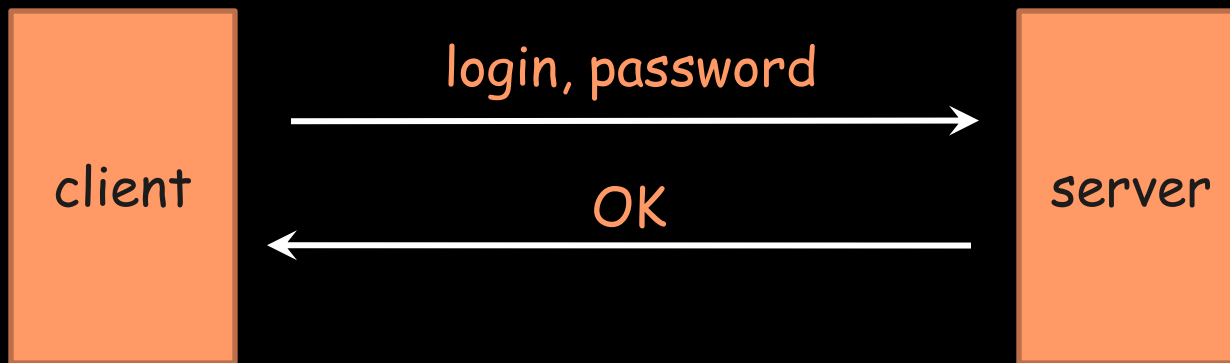
- Unencrypted passwords
- Insecure on an open network

PAP: Reusable passwords

One problem: what if the password file isn't sufficiently protected and an intruder gets hold of it, he gets all the passwords!

Enhancement:

Store a hash of the password in a file
- given a file, you don't get the passwords
- have to resort to a dictionary or brute-force attack

PAP: Reusable passwords

Passwords can be stolen by observing a user's session over the network:

- snoop on telnet, ftp, rlogin, rsh sessions

- Trojan horse

- social engineering

- brute-force or dictionary attacks

One-time password

Different password used each time

- generate a list of passwords

or:

- use an authentication card

Skey authentication

- One-time password scheme

- Produces a limited number of authentication sessions

- relies on one-way functions

Skey authentication

Authenticate Alice for 100 logins

- pick random number, R

- using a one-way function, $f(x)$:

> $x_1 = f(R)$
> $x_2 = f(x_1) = f(f(R))$
> $x_3 = f(x_2) = f(f(f(R)))$
>
>   ...   ...
> $x_{100} = f(x_{99}) = f(...f(f(f(R)))...)$

give this list to Alice

- then compute:

> $x_{101} = f(x_{100}) = f(...f(f(f(R)))...)$

Skey authentication

Authenticate Alice for 100 logins

store x_{101} in a password file or database record associated with Alice

alice: x_{101}

Skey authentication

Alice presents the *last* number on her list:

Alice to host: { "alice", x_{100} }

Host computes $f(x_{100})$ and compares it with the value in the database

if (x_{100} provided by alice) = passwd("alice")

replace x_{101} in db with x_{100} provided by alice
return success

else

fail

next time: Alice presents x_{99}
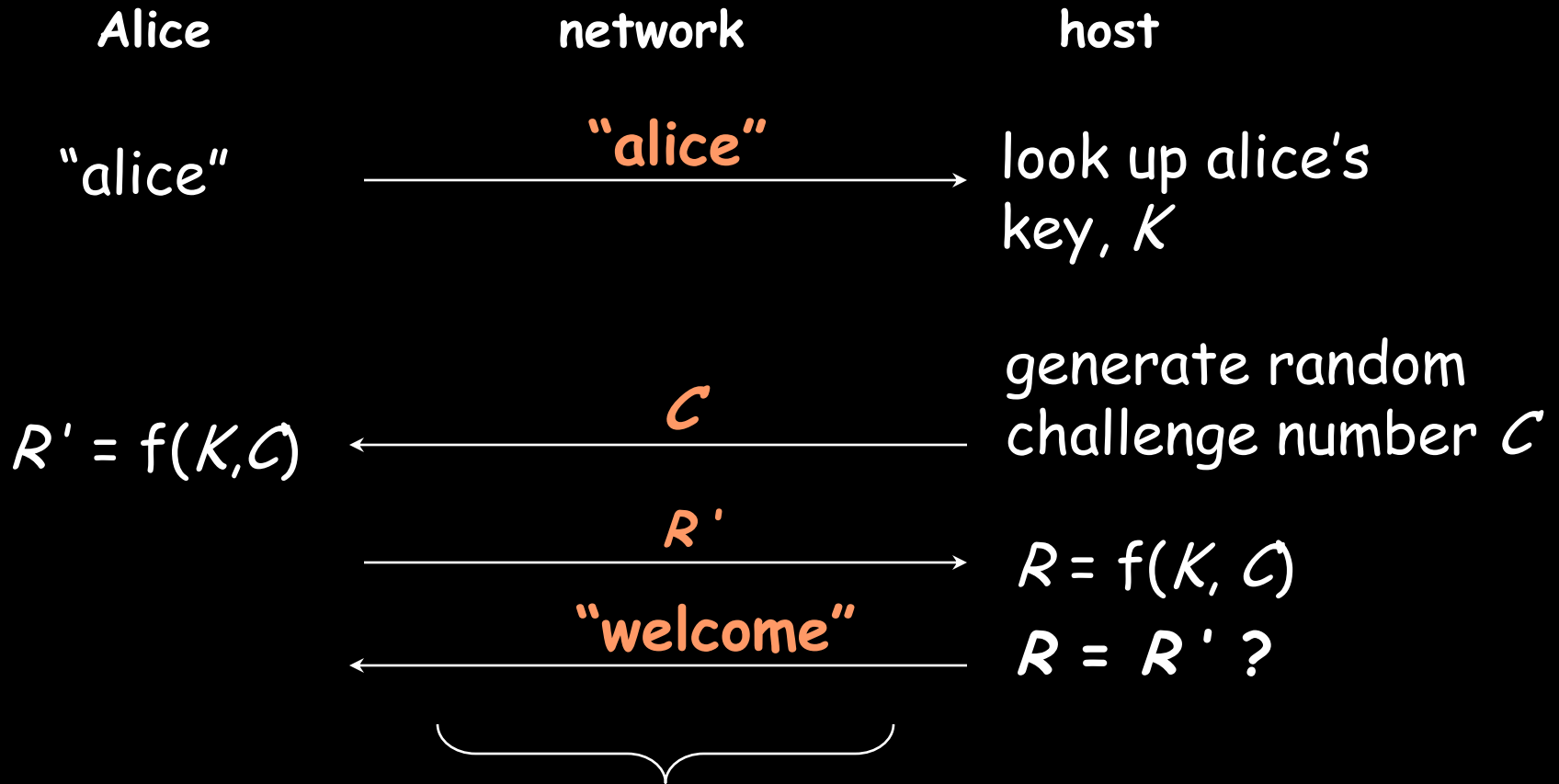
if someone sees x_{100} there is no way to generate x_{99}.

Two-factor authentication with an authenticator card

Challenge/response authentication

- user provided with a challenge number from host
- enter challenge number to challenge/response unit
- enter PIN
- get response: f(PIN, challenge)
- transcribe response back to host

- host maintains PIN
  - computes the same function
  - compares data

- rely on one-way function

Challenge-Response authentication

Alice   network   host

"alice" ─────────── "alice" ──────────► look up alice's
                                          key, K

                                          generate random
$R' = f(K,C)$ ◄─────── C ────────   challenge number C

         ─────────── R' ──────────► $R = f(K, C)$
         ◄────────── "welcome" ───── $R = R'$?

an eavesdropper does not see K

SecurID card

Username:

paul

Password:

1234032848

PIN + passcode from card

Something you know

Something you have

Passcode changes every 60 seconds

1. Enter PIN
2. Press ◇
3. Card computes password
4. Read off password

Password:

354982

SecurID card

- from RSA, SASL mechanism: RFC 2808
- Compute: AES-hash on:
  - 128-bit token-specific seed
  - 64-bit ISO representation of time of day (Y:M:D:H:M:S)
  - 32-bit serial number of token
  - 32-bits of padding
- Server computes three hashes with different clock values to account for drift.

SecurID

Vulnerable to man-in-the-middle attacks
- attacker acts as application server
- user does not have a chance to authenticate server

SKID2/SKID3 authentication

- uses symmetric cryptography
  - shared secret key
- generate a random token
  - *nonce*
- give it to the other party, which encrypts it
  - returns encrypted result
- verify that the other party knows the secret key

SKID2/SKID3 authentication

Alice chooses a random
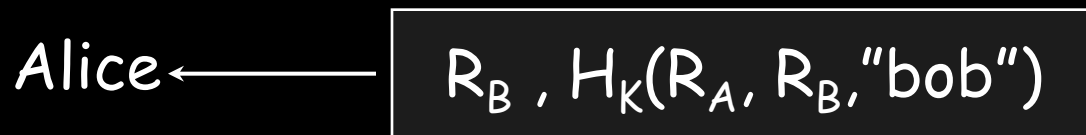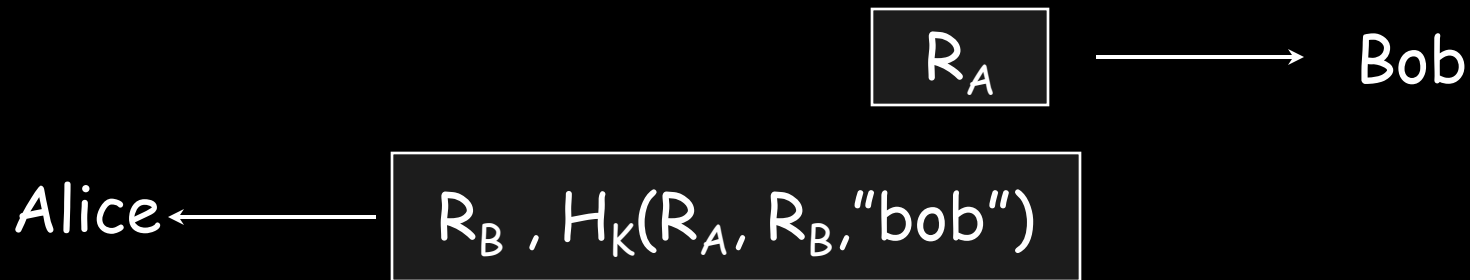number (nonce) R_A
and sends it to Bob

$$\boxed{R_A} \longrightarrow \text{Bob}$$

SKID2/SKID3 authentication

R_A $\longrightarrow$ Bob
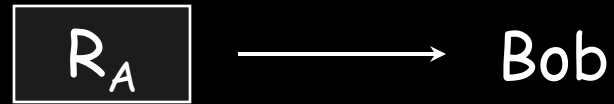
Bob chooses a random number (nonce): R_B.

He computes $H_K(R_A, R_B, \text{"bob"})$ and sends it to Alice with R_B

Alice $\longleftarrow$ R_B , $H_K(R_A, R_B, \text{"bob"})$

Bob shows that he can encrypt Alice's nonce

SKID2/SKID3 authentication

$$R_A \longrightarrow \text{Bob}$$

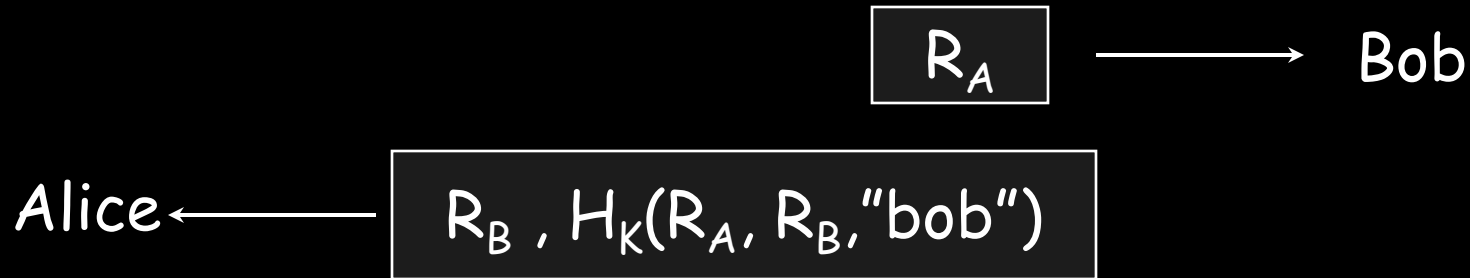$$\text{Alice} \longleftarrow R_B , H_K(R_A, R_B, \text{"bob"})$$

Alice receives R_B and has R_A.
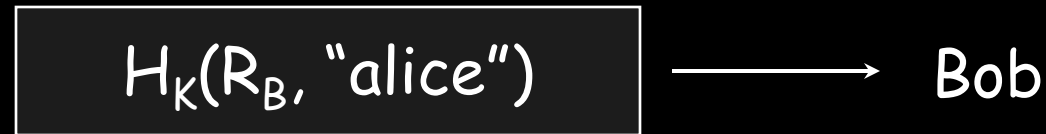Computes: $H_K(R_A, R_B, \text{"bob"})$

compares result to verify that Bob was able to encrypt data with key K.

Authentication is complete as far as Alice is concerned (Bob knows the key).

SKID2/SKID3 authentication

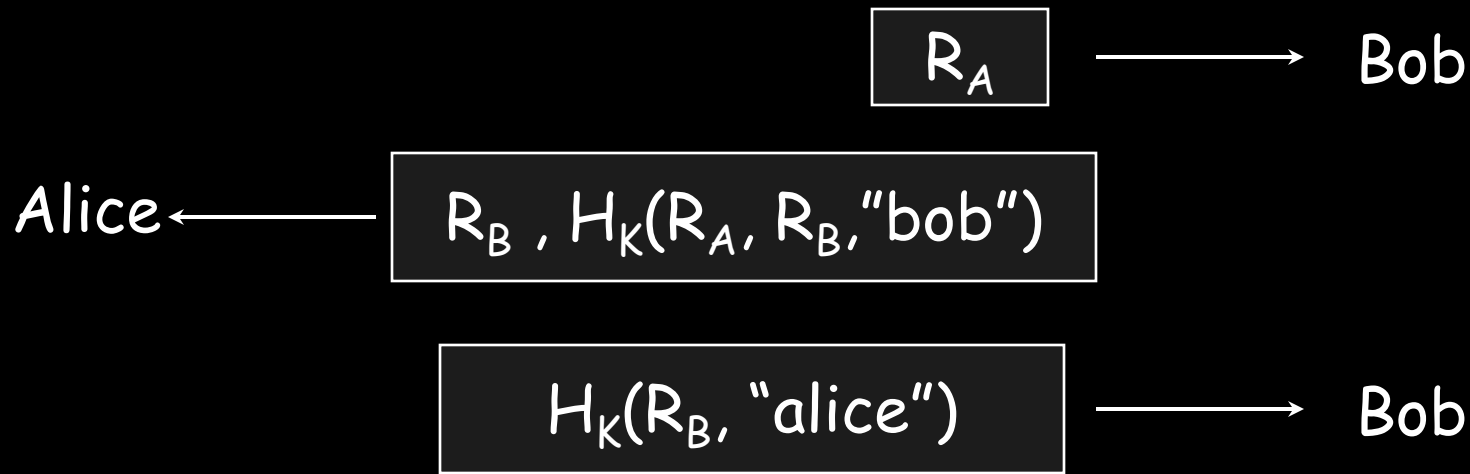R_A $\longrightarrow$ Bob

Alice $\longleftarrow$ R_B , $H_K(R_A, R_B, \text{"bob"})$

Now Alice has to convince Bob (<u>mutual authentication</u>)

$H_K(R_B, \text{"alice"})$ $\longrightarrow$ Bob

Alice demonstrates that she can encrypt Bob's nonce

SKID2/SKID3 authentication

$$R_A \longrightarrow \text{Bob}$$

$$\text{Alice} \longleftarrow R_B, H_K(R_A, R_B, \text{"bob"})$$

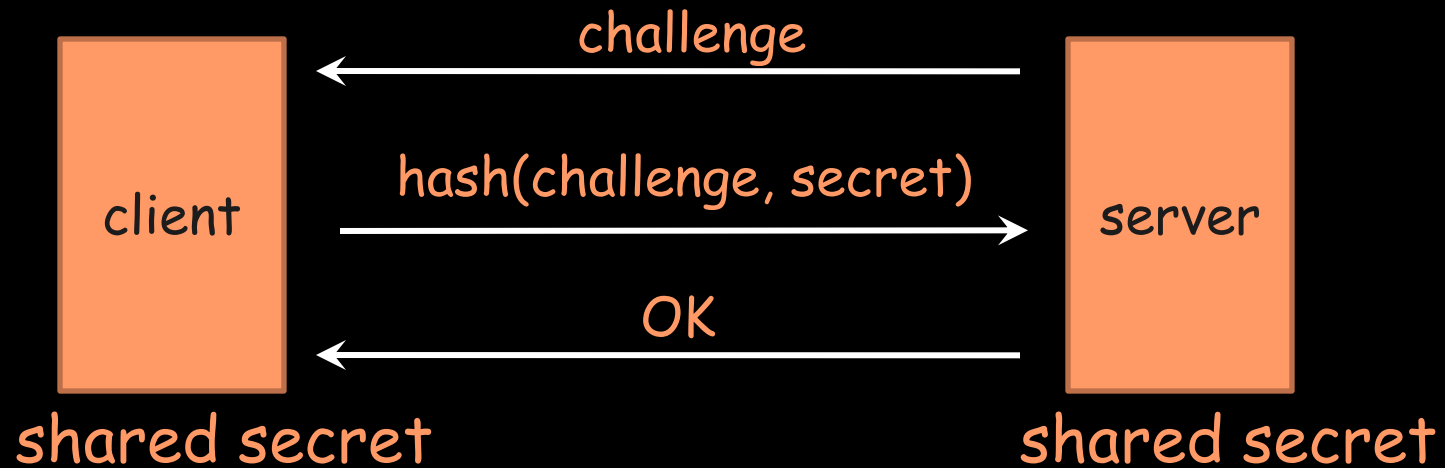$$H_K(R_B, \text{"alice"}) \longrightarrow \text{Bob}$$

Bob computes $H_K(R_B, \text{"alice"})$ and compares Alice's message. If they match, he trusts Alice's identity

Key point: Each party permutes data generated by the other. Challenge the other party with data that will be different each time.

Authentication: CHAP

Challenge-Handshake Authentication Protocol

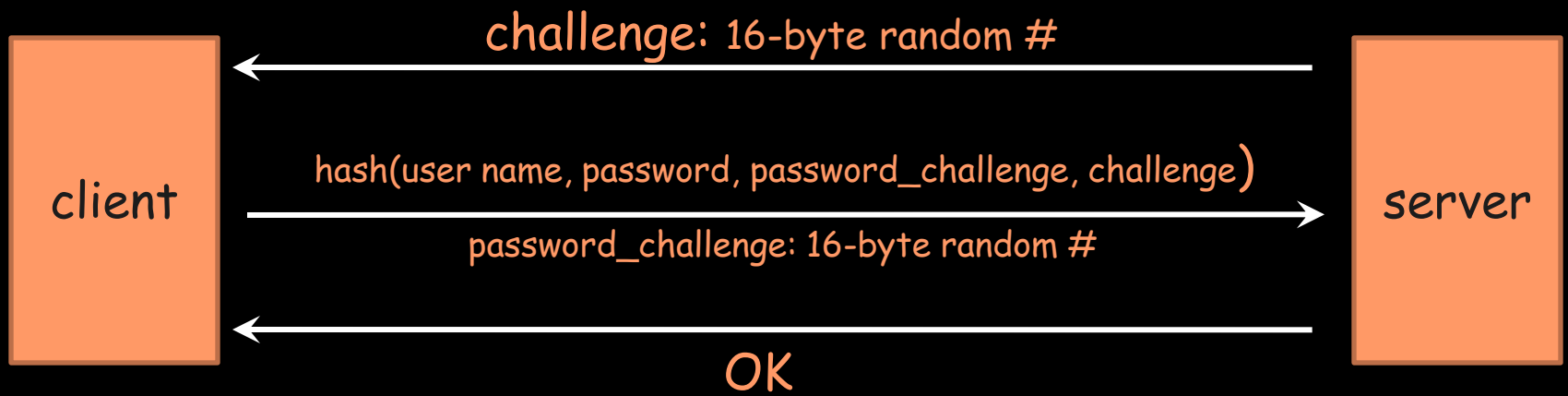The challenge-response scheme in a slightly different form.

This is functionally the same as SKID2 (single party authentication)

The challenge is a nonce. Instead of encrypting the nonce with a shared secret key, we create a hash of the nonce and the secret.

Authentication: MS-CHAP

Microsoft's Challenge-Handshake Authentication Protocol

The same as CHAP – we're just hashing more things in the response

Combined authentication and key exchange

Wide-mouth frog

Alice $\longrightarrow$ Trent

"alice" , $E_A(T_A,$"bob", $K)$

session key
destination
time stamp – prevent replay attacks
sender

- arbitrated protocol – Trent (3rd party) has all the keys
- symmetric encryption for transmitting a **session key**

Wide-mouth frog

Alice ———————————→ Trent

"alice" , $E_A(T_A,$ "bob", K)

session key

destination

time stamp – prevent replay attacks

sender

Trent:
- looks up key corresponding to sender ("alice")
- decrypts remainder of message using Alice's key
- validates timestamp (this is a new message)
- extracts destination ("bob")
- looks up Bob's key

Wide-mouth frog

Alice ⟶ Trent ⟶ Bob

| "alice" , $E_A(T_A,$"bob", K)$ | $E_B(T_T,$"alice", K)$ |

session key

source

time stamp – prevent replay attacks

Trent:
- creates a new message
- new timestamp
- identify source of the session key
- encrypt the message for Bob
- send to Bob

Wide-mouth frog

Alice $\longrightarrow$ Trent $\longrightarrow$ Bob

"alice" , $E_A(T_A,"bob", K)$	$E_B(T_T,"alice", K)$

session key

source

time stamp – prevent replay attacks

Bob:
- decrypts message
- validates timestamp
- extracts sender ("alice")
- extracts session key, K

Wide-mouth frog

Alice $\longleftrightarrow$ Bob

$E_k(M)$
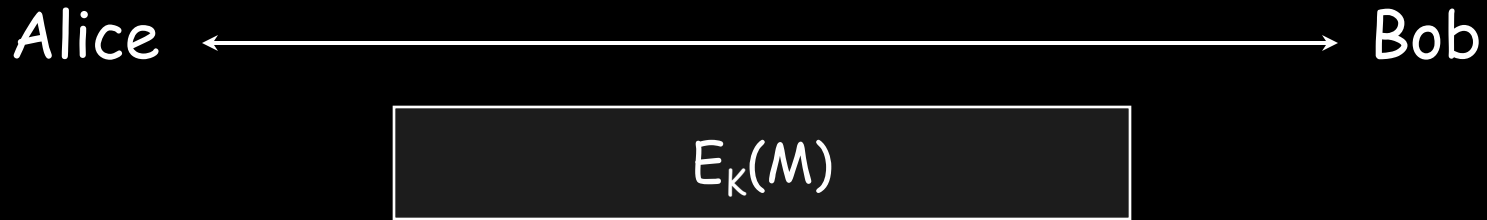
Since Bob and Alice have the session key,
they can communicate securely using the key

Kerberos

- authentication service developed by MIT
  - project Athena 1983-1988
- trusted third party
- symmetric cryptography
- passwords not sent in clear text
  - assumes only the network can be compromised

Kerberos

Users and services authenticate themselves to each other

To access a service:

- – user presents a ticket issued by the Kerberos authentication server
- – service examines the ticket to verify the identity of the user

Kerberos

- user *Alice* wants to communicate with a service *Bob*

- both Alice and Bob have keys

- Step 1:
  - Alice authenticates with Kerberos server
    - Gets session key and *sealed envelope*
- Step 2:
  - Alice gives Bob a session key (securely)
  - Convinces Bob that she also got the session key from Kerberos

Authenticate, get permission

Alice   Authentication Server (AS)

"I want to talk to Bob" ———————→

if Alice is allowed to talk to Bob,
generate session key, S

←——— {"Bob's server", S}$_A$ ———

Alice decrypts this:
• gets ID of "Bob's server"
• gets session key
• *knows message came from AS*

TICKET
<u>sealed envelope</u>

←——— {"Alice", S}$_B$ ———

eh? (Alice can't read this!)

Send key

Alice

Alice encrypts a timestamp with session key

$\{\text{"Alice"}, S\}_B, T_S$

sealed envelope

Bob

Bob decrypts envelope:
- envelope was created by Kerberos on request from Alice
- gets session key

Decrypts time stamp
- validates time window
- Prevent replay attacks

Authenticate recipient

Alice

Bob

Encrypt Alice's timestamp in return message

$\longleftarrow$ {"Bob's Server", T}$_S$ ———

Alice validates timestamp
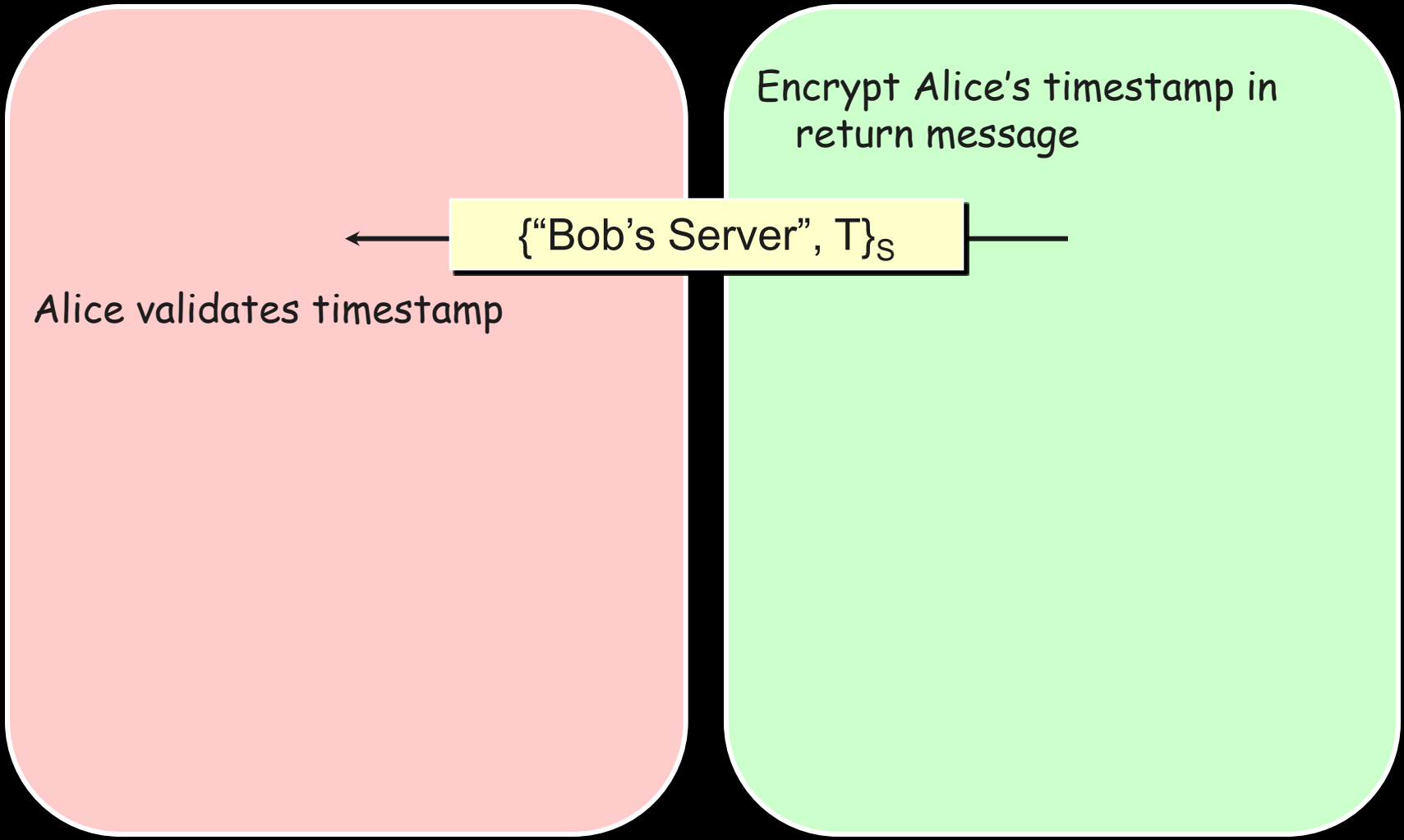
Kerberos key usage

- Every time a user wants to access a service
  - User's password (key) must be used each time (in decoding message from Kerberos)

- Possible solution:
  - Cache the password (key)
  - Not a good idea

- Another solution:
  - Split Kerberos server into Authentication Server + Ticket Granting Server

Ticket Granting Service (TGS)

TGS + AS = KDC (Kerberos Key Distribution Center)

- Before accessing any service, user requests a ticket to contact the TGS

- Anytime a user wants a service
  - Request a ticket from TGS
  - Reply is encrypted with session key from AS for use with TGS

- TGS works like a temporary ID

Using Kerberos

`$ kinit`

`Password:` *enter password*

ask AS for permission (session key) to access TGS
Alice gets:

$$\{\text{``TGS''}, S\}_A$$

$$\{\text{``Alice''}, S\}_{TGS}$$

Compute key (A) from password to decrypt session key
S and get TGS ID.

*You now have a ticket to access the Ticket Granting
Service*

Using Kerberos

$ rlogin *somehost*

rlogin uses TGS Ticket to request a ticket for the *rlogin* service on *somehost*

Alice sends session key, S, to TGS

rlogin   TGS

$\longrightarrow$ {"Alice", S}$_{TGS}$, T$_S$ $\longrightarrow$

Alice receives session key for rlogin service
& ticket to pass to rlogin service

$\longleftarrow$ {"rlogin@somehost", S'}$_S$ ——

$\longleftarrow$ {"Alice", S'}$_R$ ——

session key
for *rlogin*

ticket for rlogin server
on somehost
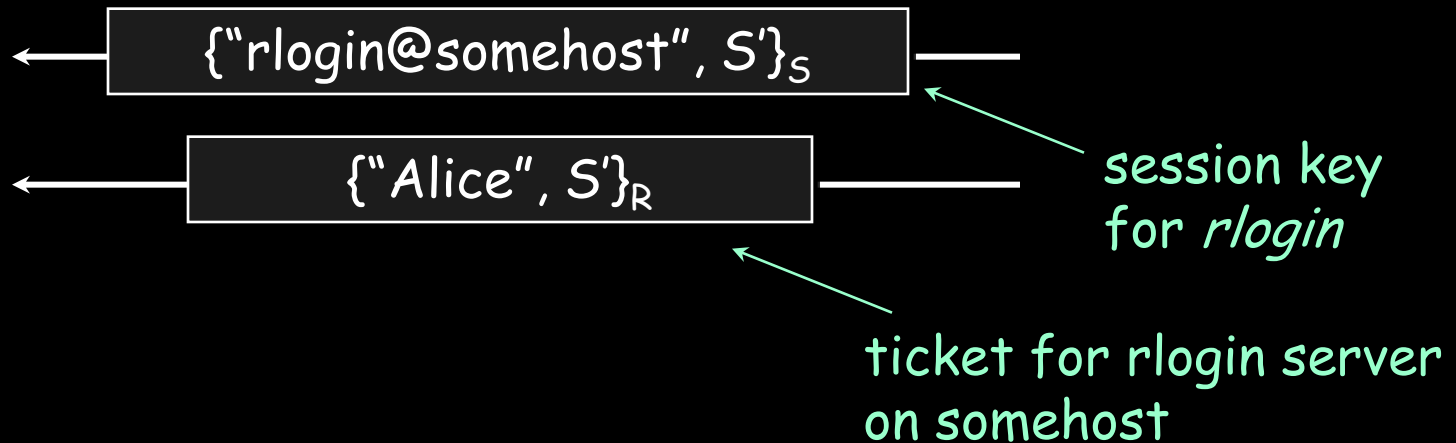
Public key authentication

Like SKID, demonstrate we can encrypt or decrypt a nonce:

- Alice wants to authenticate herself to Bob:
- <u>Bob</u>: generates nonce, S
  - presents it to Alice
- <u>Alice</u>: encrypts S with her private key (sign it) and send to Bob

Public key authentication

Bob:
   look up "alice" in a database of public keys
   - decrypt the message from Alice using Alice's public key
   - If the result is S, then it was Alice!
- Bob is convinced.

For mutual authentication, Alice has to present Bob with a nonce that Bob will encrypt with his private key and return
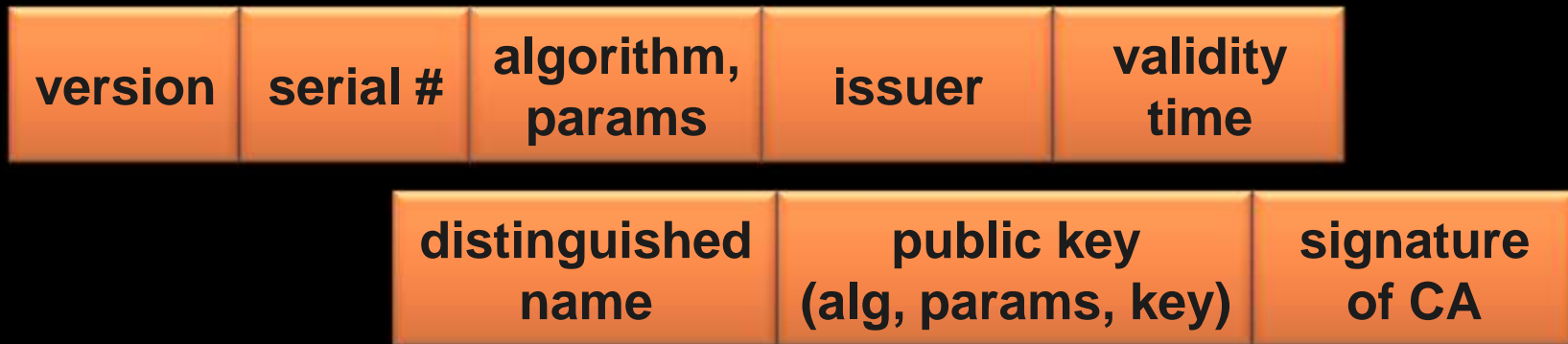
Public key authentication

- Public key authentication relies on binding identity to a public key
- One option:
    get keys from a trusted source
- Problem: requires always going to the source
    - cannot pass keys around

- Another option: *sign the public key*
    - **digital certificate**

X.509 Certificates

ISO introduced a set of authentication protocols: X.509

Structure for public key <u>certificates</u>:

version	serial #	algorithm, params	issuer	validity time

distinguished name	public key (alg, params, key)	signature of CA

Trusted <u>Certification Authority</u> issues a signed certificate

X.509 certificates

When you get a certificate
- Verify signature
  - hash contents of certificate data
  - Decrypt CA's signature with <u>CA's public key</u>

Obtain CA's public key (certificate) from trusted source
- Certification authorities are organized in a hierarchy
- A CA certificate may be signed by a CA above it
  - <u>**certificate chaining**</u>

Certificates prevent someone from using a phony public key to masquerade as another person

Example: Root Certificates in IE

Agencia Catalana de Certificacio

ANCERT

AOL

Arge Daten

AS Sertifitseerimiskeskuse

Asociacion Nacional del Notariado Mexicano

A-Trust

Austria Telekom-Control Commission

Autoridad Certificadora Raiz de la Secretaria de Economia

Autoridad de Certificacion Firmaprofesional

Autoridade Certificadora Raiz Brasileira

Belgacom E-Trust

CAMERFIRMA

Example: Root Certificates in IE

CC Signet

Certicámara S.A.

Certipost s.a./n.v.

Certisign

CertPlus

Colegio de Registradores

Comodo Group

ComSign

Correo

Cybertrust

Deutsche Telekom

DigiCert

DigiNotar B.V.

Dirección General de la Policía – Ministerio del Interior – España.

DST

Example: Root Certificates in IE

Echoworx

Entrust

eSign

EUnet International

First Data Digital Certificates

FNMT

Gatekeeper Root CA

GeoTrust

GlobalSign

GoDaddy

Government of France

Government of Japan Ministry of Internal Affairs and Communications

Government of Tunisia National Digital Certification Agency

Hongkong Post

IPS SERVIDORES

Example: Root Certificates in IE

IZENPE

KMD

Korea Information Security Agency

Microsec Ltd.

NetLock

Network Solutions

Post.Trust

PTT Post

Quovadis

RSA

Saunalahden Serveri

SECOM Trust.net

SecureNet

SecureSign

SecureTrust Corporation

Example: Root Certificates in IE

Serasa

SIA

Sonera

Spanish Property & Commerce Registry

Swisscom Solutions AG

SwissSign AG

S-TRUST

TC TrustCenter

TDC

Thawte

Trustis Limited

TurkTrust

TW Government Root Certification Authority

U.S. Government Federal PKI

Example: Root Certificates in IE

Unizeto Certum

UserTRUST

ValiCert

VeriSign

Visa

Wells Fargo

WISeKey

XRamp
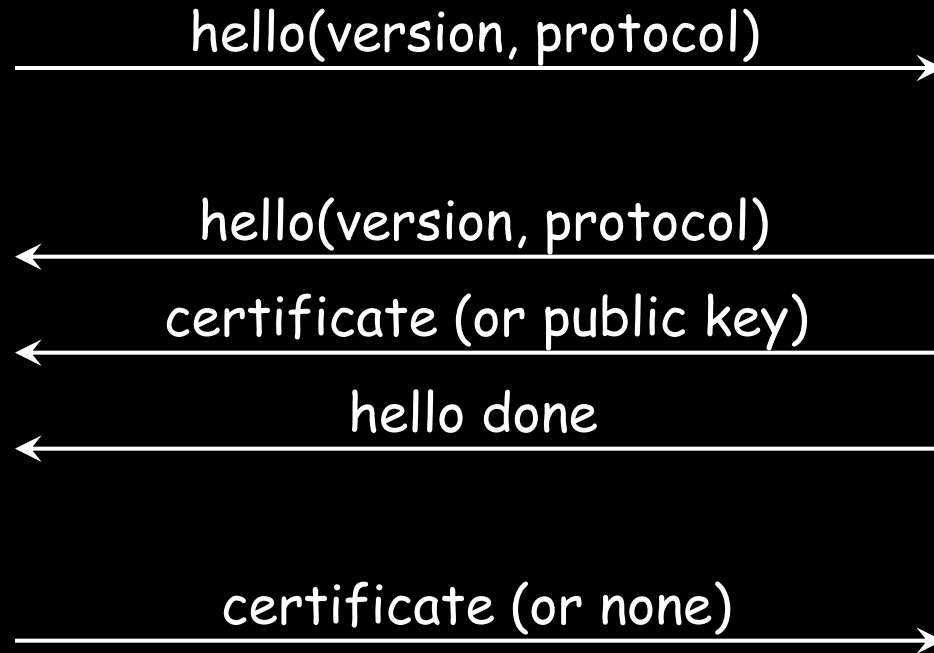
Transport Layer Security (TLS)
aka Secure Socket Layer (SSL)

- Sits on top of TCP/IP
- Goal: provide an encrypted and possibly authenticated communication channel
  - Provides authentication via RSA and X.509 certificates
  - Encryption of communication session via a symmetric cipher
- Enables TCP services to engage in secure, authenticated transfers
  - http, telnet, ntp, ftp, smtp, …

Secure Sockets Layer (SSL)

client   server

client → server: hello(version, protocol)

server → client: hello(version, protocol)

server → client: certificate (or public key)

server → client: hello done

client → server: certificate (or none)

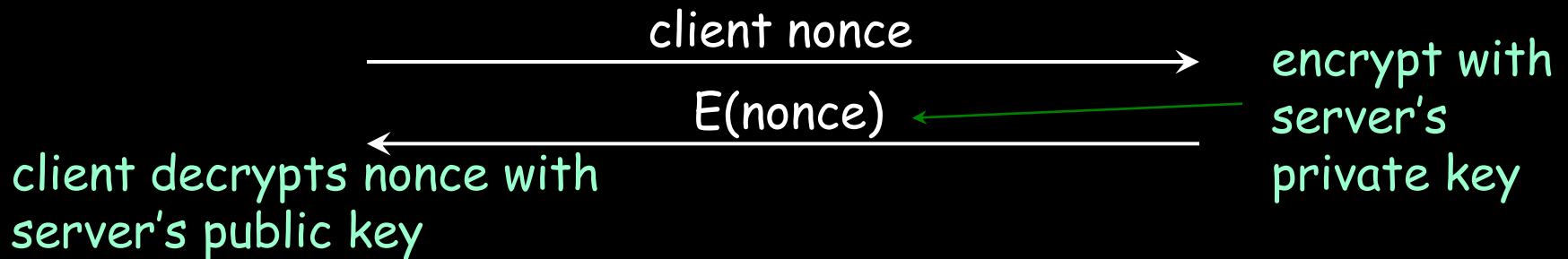1. Establish protocol, version, cipher suite, compression mechanism, exchange certificates (or send public key)

Secure Sockets Layer (SSL)

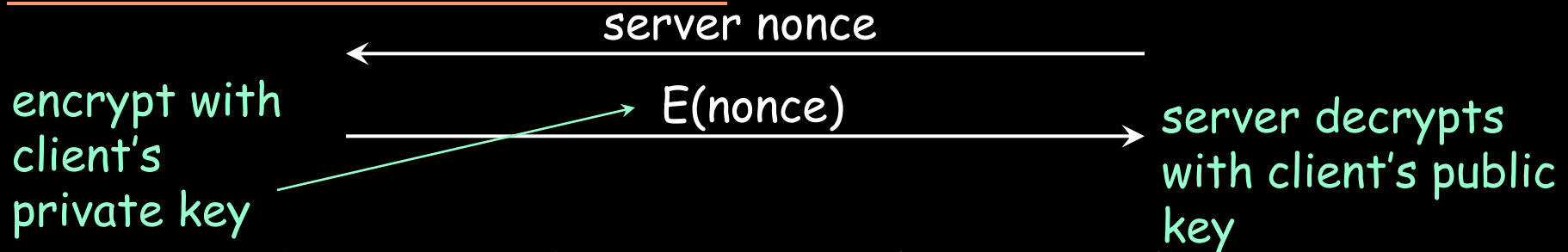<u>client</u>   <u>server</u>

client authenticates server

client nonce →

E(nonce) ←   encrypt with
                              server's
client decrypts nonce with   private key
server's public key

server authenticates client

← server nonce

encrypt with   E(nonce) →   server decrypts
client's   with client's public
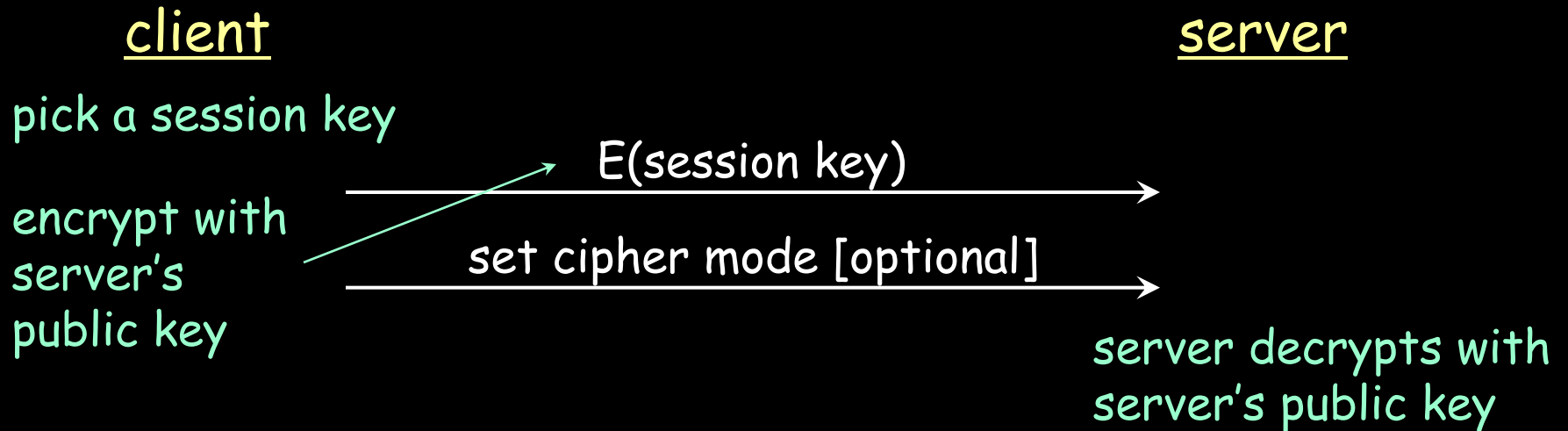private key   key

2. Authenticate (unidirectional or mutual)
   [optional]

Secure Sockets Layer (SSL)

<u>client</u>   <u>server</u>

pick a session key

                            E(session key)
encrypt with
server's   set cipher mode [optional]
public key

                                                server decrypts with
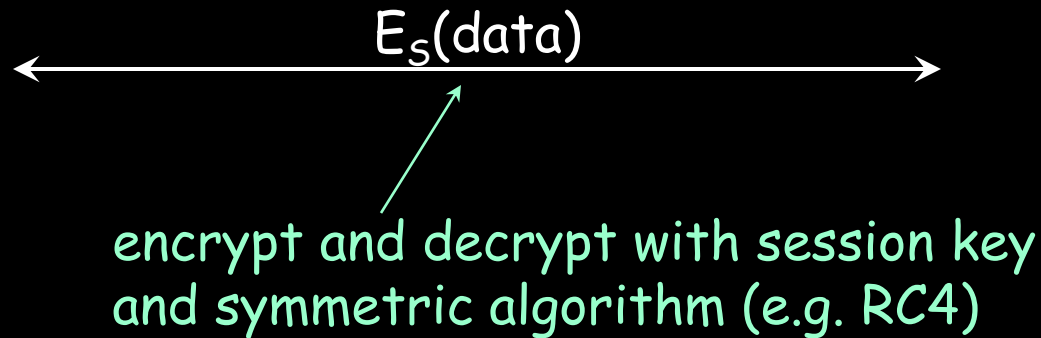   server's public key

3. Establish session key
   (for symmetric cryptography)

Secure Sockets Layer (SSL)

client   server

$E_S(\text{data})$

encrypt and decrypt with session key
and symmetric algorithm (e.g. RC4)

4. Exchange data (symmetric encryption)

The end.