Distributed Systems

24. Authentication

Paul Krzyzanowski

Rutgers University

Fall 2017

Security Goals

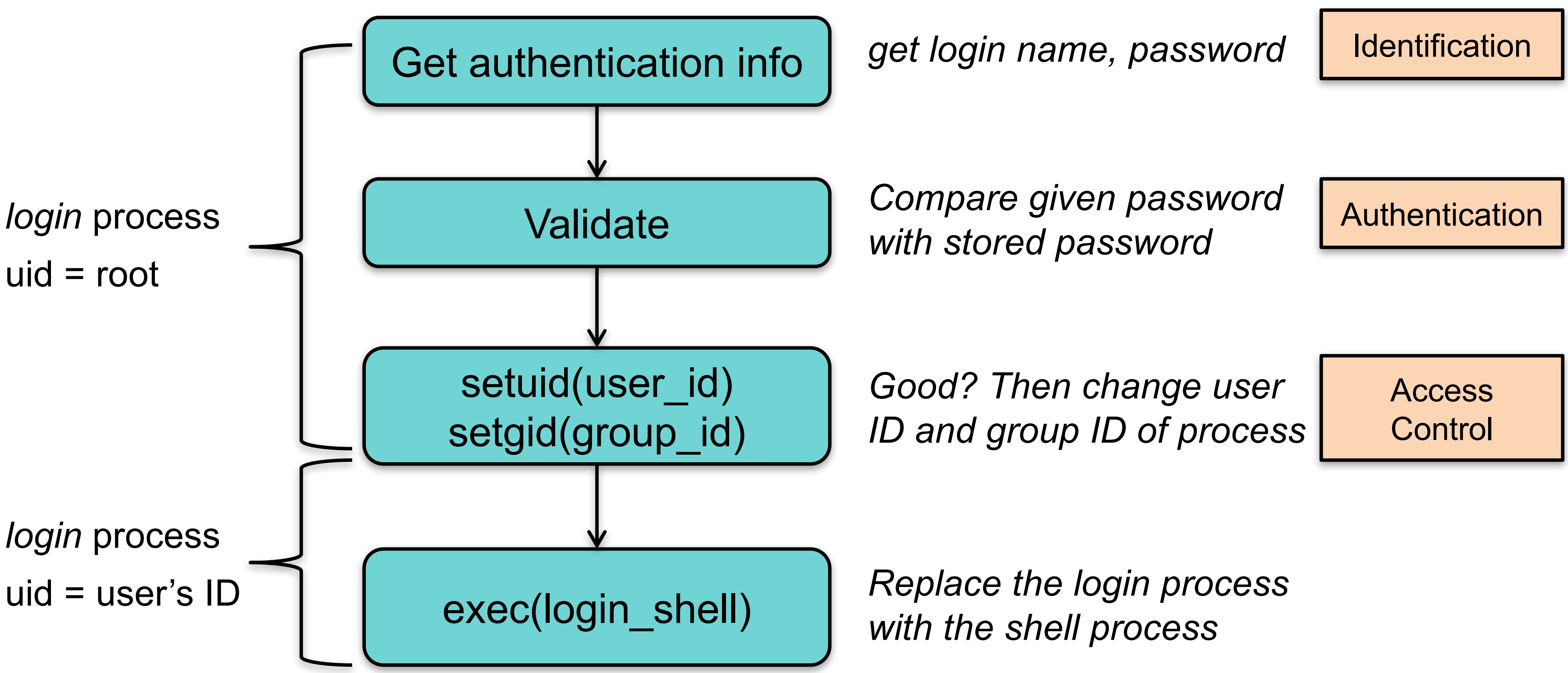
- Authentication
  - Ensure that users, machines, programs, and resources are properly identified
- Integrity
  - Verify that data has not been compromised: deleted, modified, added
- Confidentiality
  - Prevent unauthorized access to data
- Availability
  - Ensure that the system is accessible

Authentication

- For a user (or process):
  - Establish & verify identity
  - Then decide whether to allow access to resources (= authorization)

- For a file or data stream:
  - Validate that the integrity of the data; that it has not been modified by anyone other than the author
  - E.g., digital signature

Local authentication example: login



Identification vs. Authentication

- Identification:
  - Who are you?
  - User name, account number, …
- Authentication:
  - Prove it!
  - Password, PIN, encrypt nonce, …

Versus Authorization

Authorization defines access control

Once we know a user's identity:

- Allow/disallow request
- **Operating systems**
  - Enforce access to resources and data based on user's credentials
- **Network services** usually run on another machine
  - Network server may not know of the user
  - Application takes responsibility
  - May contact an *authorization server*
    - Trusted third party that will grant credentials
    - Kerberos ticket granting service
    - RADIUS (centralized authentication/authorization service)
    - OAuth service

Security

The Three A's

Authentication

Authorization

Accounting
(+ Auditing)

Authentication

Three factors:

- something you have *key, card*
  - Can be stolen
- something you know *passwords*
  - Can be guessed, shared, stolen
- something you are *biometrics*
  - Usually needs hardware, can be copied (sometimes)
  - Once copied, you're stuck

Multi-Factor Authentication

Factors may be combined

– ATM machine: 2-factor authentication

- ATM card something you have
- PIN something you know

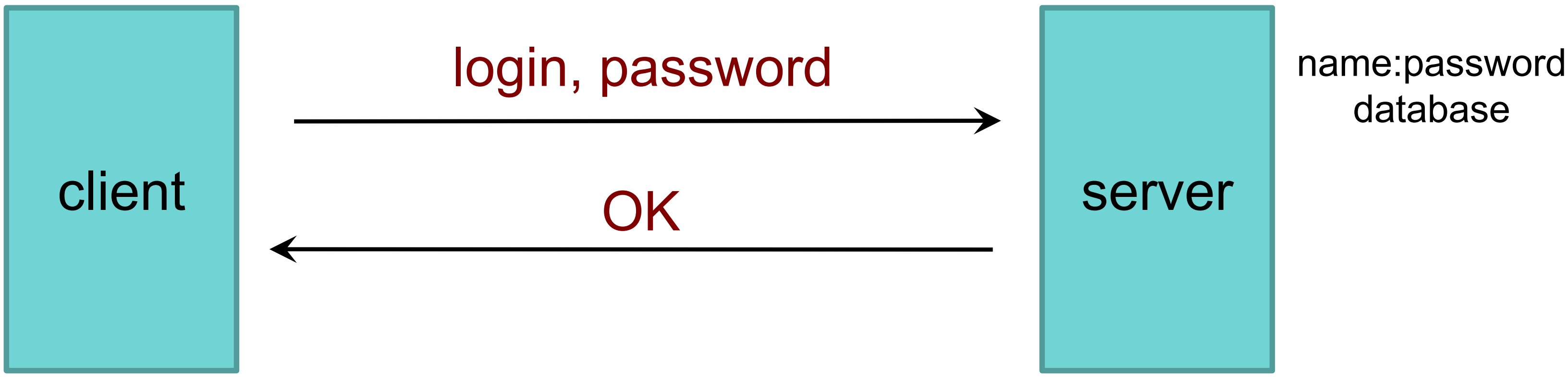
– Password + code delivered via SMS: 2-factor authentication

- Password something you know
- Code validates that you possess your phone

Two passwords ≠ Two-factor authentication

Authentication: PAP

Password Authentication Protocol



- Unencrypted, reusable passwords
- Insecure on an open network
- Also, password file must be protected from open access
  - But administrators can still see everyone's passwords

PAP: Reusable passwords

Problem #1: Open access to the password file

What if the password file isn't sufficiently protected and an intruder gets hold of it? All passwords are now compromised!

Even if a trusted admin sees your password, this might also be your password on other systems.

Solution:

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
- Example, passwords hashed with SHA-512 hashes (SHA-2)

Common Passwords

Adobe security breach (November 2013)

- 152 million Adobe customer records … with encrypted passwords
- Adobe encrypted passwords with a symmetric key algorithm
- … and used the same key to encrypt every password!

Top 26 Adobe Passwords

	Frequency	Password
1	1,911,938	123456
2	446,162	123456789
3	345,834	password
4	211,659	adobe123
5	201,580	12345678
6	130,832	qwerty
7	124,253	1234567
8	113,884	111111
9	83,411	photoshop
10	82,694	123123
11	76,910	1234567890
12	76,186	000000
13	70,791	abc123

	Frequency	Password
14	61,453	1234
15	56,744	adobe1
16	54,651	macromedia
17	48,850	azerty
18	47,142	iloveyou
19	44,281	aaaaaa
20	43,670	654321
21	43,497	12345
22	37,407	666666
23	35,325	sunshine
24	34,963	123321
25	33,452	letmein
26	32,549	monkey

What is a dictionary attack?

- **Suppose you got access to a list of hashed passwords**
- **Brute-force, exhaustive search: try every combination**
  - Letters (A-Z, a-z), numbers (0-9), symbols (!@#$%...)
  - Assume 30 symbols + 52 letters + 10 digits = 92 characters
  - Test all passwords up to length 8
  - Combinations = $92^8 + 92^7 + 92^6 + 92^5 + 92^4 + 92^3 + 92^2 + 92^1 = 5.189 \times 10^{15}$
  - If we test 1 billion passwords per second: ≈ 60 days
- **But some passwords are more likely than others**
  - 1,991,938 Adobe customers used a password = “123456”
  - 345,834 users used a password = “password”
- **Dictionary attack**
  - Test lists of common passwords, dictionary words, names
  - Add common substitutions, prefixes, and suffixes

What is salt?

- How to speed up a dictionary attack
  - Create a table of precomputed hashes
  - Now we just search a table

    Example: SHA-512 hash of "password" =
    sQnzu7wkTrgkQZF+0G1hi5AI3Qmzvv0bXgc5THBqi7mAsdd4XlI27ASbRt
    9fEyavWi6m0QP9B8lThf+rDKy8hg==

- Salt = random string (typically up to 16 characters)
  - Concatenated with the password
  - Stored with the password file (it's not secret)
  - Even if you know the salt, you cannot use precomputed hashes to search for a password (because the salt is prefixed)

    Example: SHA-512 hash of "am$7b22QLpassword", salt = "am$7b22QL":
    ntIxjDMnueMWig4dtWoMbaguucW6xV6cHJ+7yNrGvdoyFFRVb/LLqS01/pXS
    8xZ+ur7zPO2yn88xcliUPQj7xg==

    - You will not have precomputed *hash("am$7b22QLpassword")*

PAP: Reusable passwords

Problem #2: Network sniffing

Passwords can be stolen by observing a user's session in person or over a network:

- snoop on telnet, ftp, rlogin, rsh sessions
- Trojan horse
- social engineering
- brute-force or dictionary attacks

Solutions:

(1) Use one-time passwords

(2) Use an encrypted communication channel

One-time passwords

Use a different password each time

– If an intruder captures the transaction, it won't work next time

Three forms

1. Sequence-based: password = *f*(previous password)
2. Time-based: password = *f*(time, secret)
3. Challenge-based: *f*(challenge, secret)

S/key authentication

- One-time password scheme
- Produces a limited number of authentication sessions
- Relies on one-way functions

S/key authentication

<u>Authenticate Alice for 100 logins</u>

- pick random number, R
- using a one-way function, f(x):

$$
\begin{aligned}
x_1 &= f(R) \\
x_2 &= f(x_1) = f(f(R)) \\
x_3 &= f(x_2) = f(f(f(R))) \\
&\dots \quad \dots \\
x_{100} &= f(x_{99}) = f(\dots f(f(f(R)))\dots)
\end{aligned}
$$

Give this list to Alice

- then compute:

$$x_{101} = f(x_{100}) = f(\dots f(f(f(R)))\dots)$$

S/key authentication

<u>Authenticate Alice for 100 logins</u>

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

alice: x_{101}

S/key 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

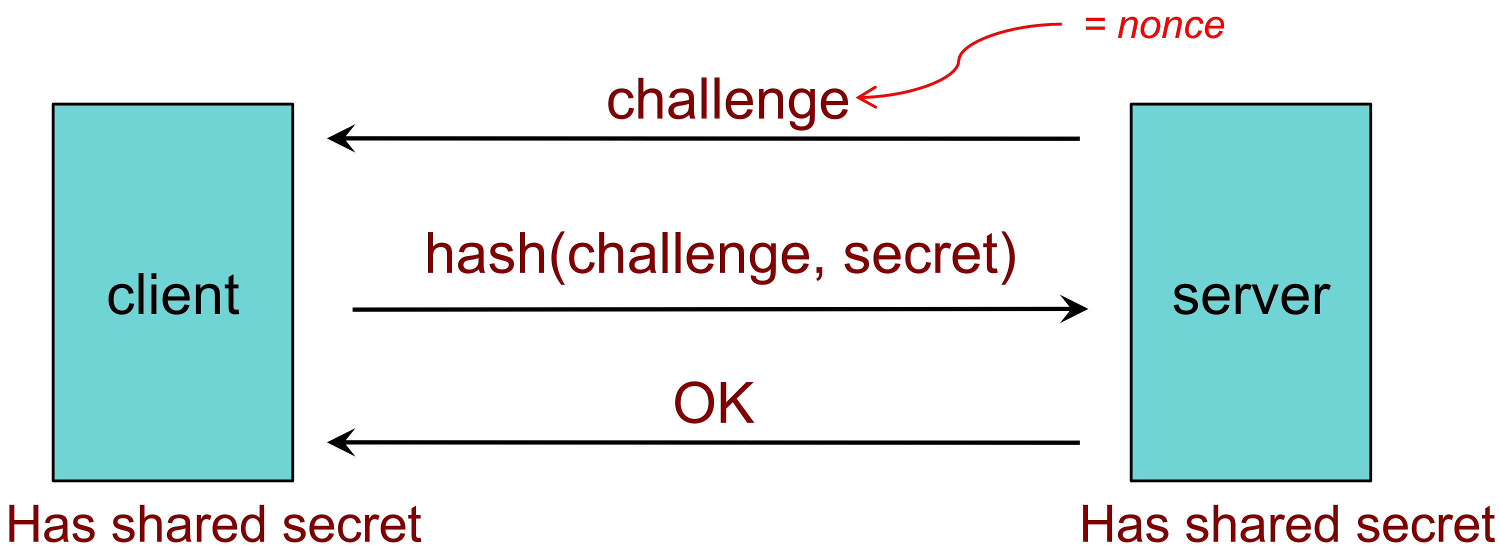
if (x_{100} provided by alice) = passwd("alice")
    replace x_{101} in db with x_{100} provided by alice
    return success
else
    fail

next time: Alice presents x_{99}

if someone sees x_{100} there is no way to generate x_{99}.

Authentication: CHAP

Challenge-Handshake Authentication Protocol



The challenge is a *nonce* (random bits).

We create a hash of the nonce and the secret.

An intruder does not have the secret and cannot do this!

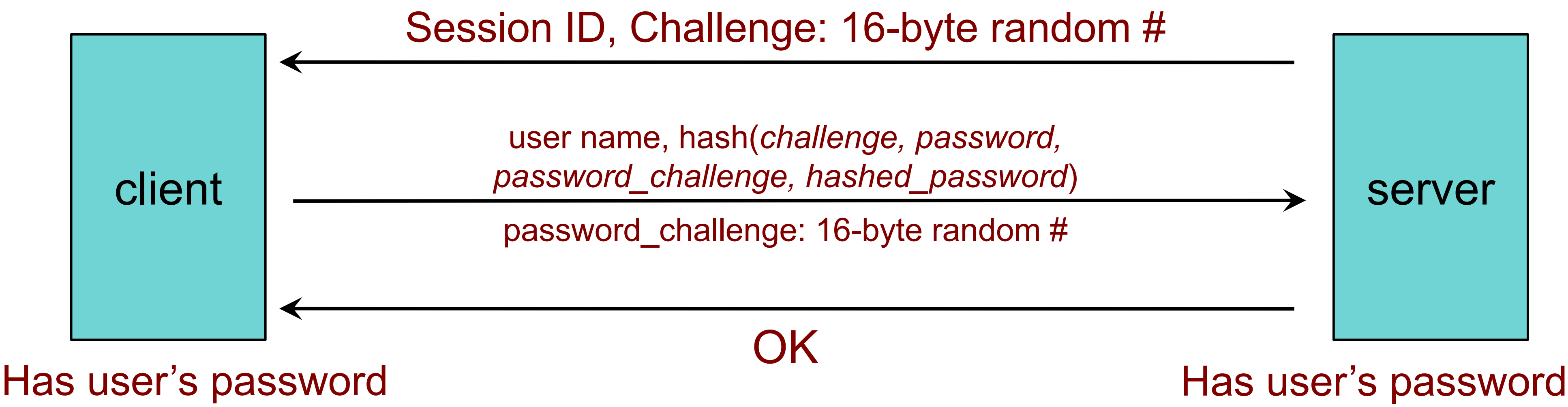
CHAP authentication

Alice	**network**	**host**
"alice"	**"alice"** →	look up alice's key, *K*
		generate random challenge number *C*
$R' = f(K,C)$	← $\boldsymbol{C}$	
	$\boldsymbol{R'}$ →	$R = f(K, C)$
	← **"welcome"**	$\boldsymbol{R = R'}$ **?**

an eavesdropper does not see K

Authentication: MS-CHAP

Microsoft's Challenge-Handshake Authentication Protocol



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

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 password & enter

Password:

354982

SecurID card

- Proprietary device from RSA
  - SASL mechanism: RFC 2808

- Two-factor authentication based on:
  - **Shared secret key** (seed) ⇦ Something you have
    - stored on authentication card
  - **Shared personal ID** – PIN ⇦ Something you know
    - known by user

SecurID (SASL) authentication: server side

- Look up user's PIN and seed associated with the token
- Get the time of day
  - Server stores relative accuracy of clock in that SecurID card
  - historic pattern of drift
  - adds or subtracts offset to determine what the clock chip on the SecurID card believes is its current time
- Passcode is a cryptographic hash of seed, PIN, and time
  - server computes ***f* (seed, PIN, time)**
- Server compares results with data sent by client

SecurID

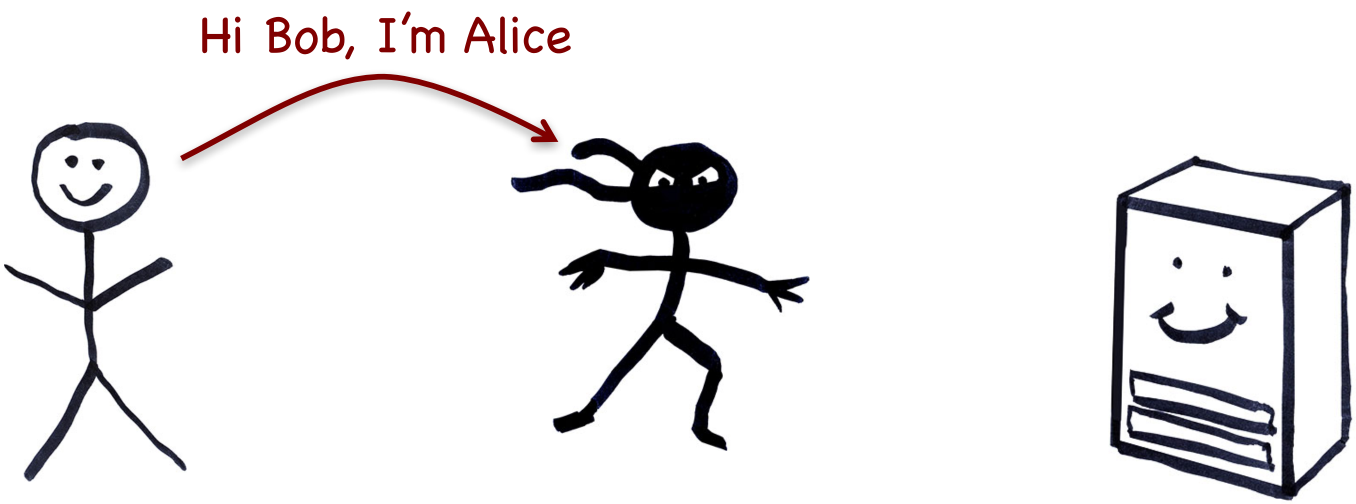
- An intruder (sniffing the network) does not have the information to generate the password for future logins
  - Needs the seed number (in the card), the algorithm (in the card), and the PIN (from the user)

- An intruder who steals your card cannot log in
  - Needs a PIN (the benefit of 2-factor authentication)
- An intruder who sees your PIN cannot log in
  - Needs the card (the benefit of 2-factor authentication)

Man-in-the-Middle Attacks

Password systems are vulnerable to man-in-the-middle attacks

– Attacker acts as the server

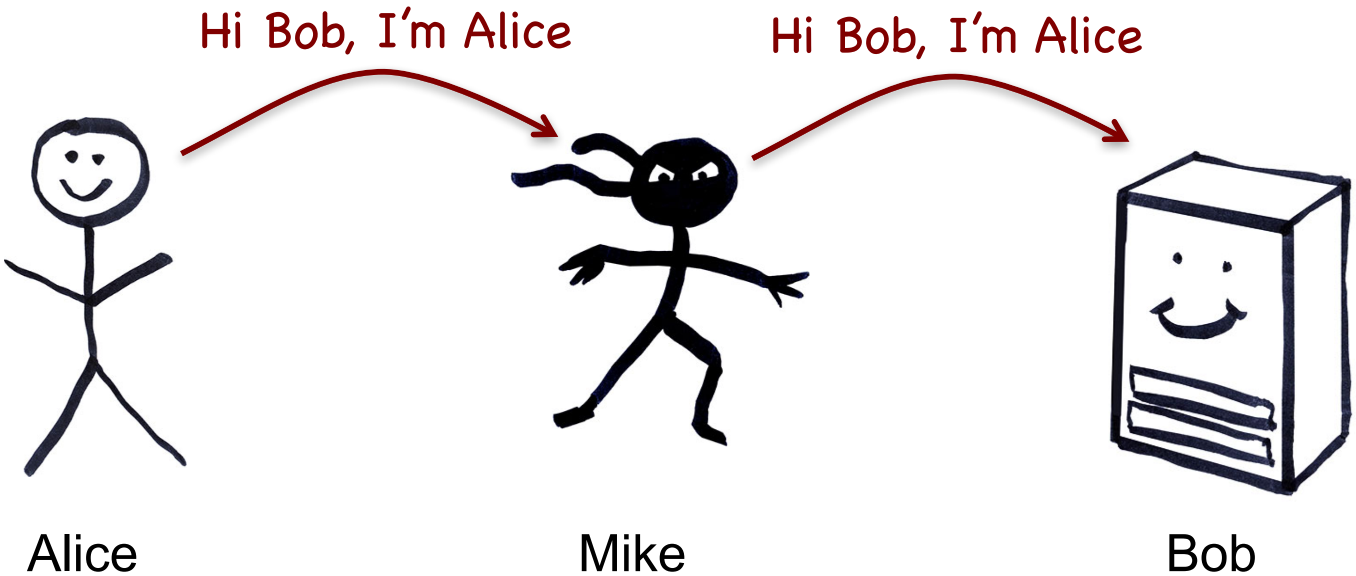


Alice Mike Bob

Man-in-the-Middle Attacks

Password systems are vulnerable to man-in-the-middle attacks

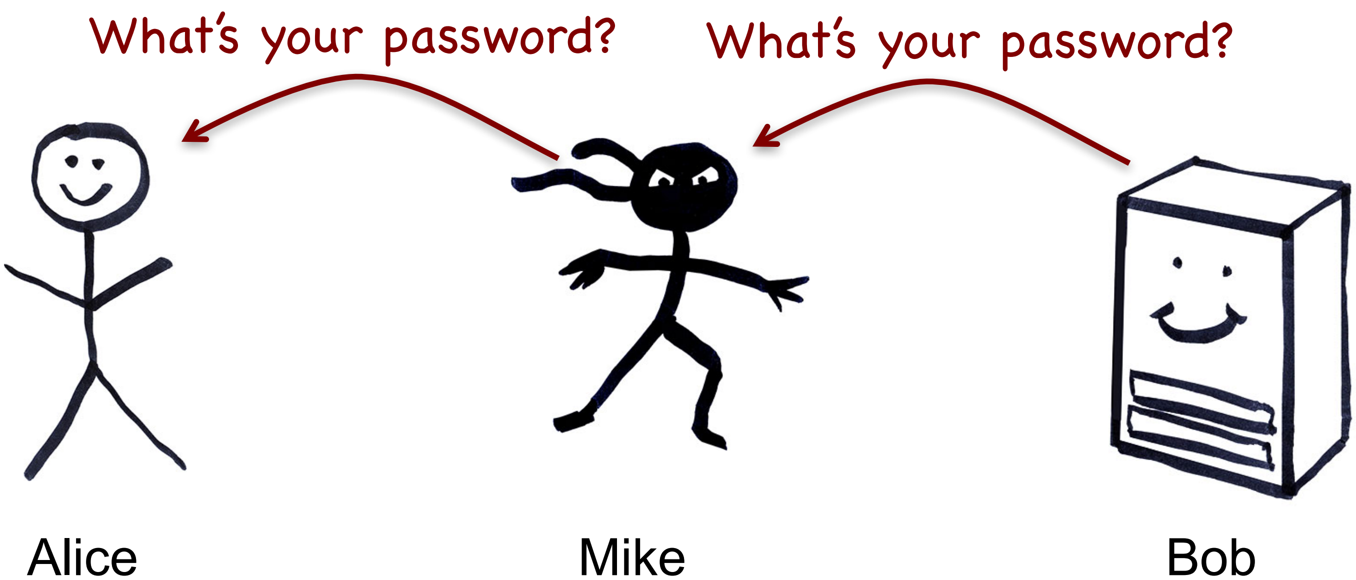
– Attacker acts as the server



Man-in-the-Middle Attacks

Password systems are vulnerable to
man-in-the-middle attacks

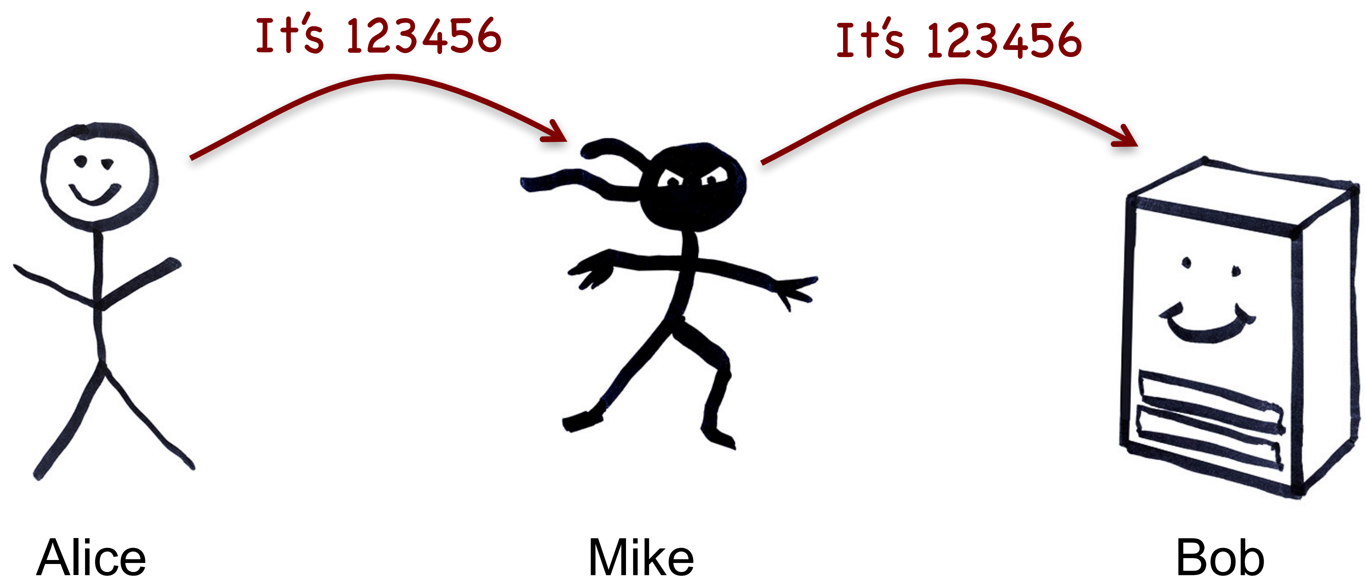
– Attacker acts as the server



Man-in-the-Middle Attacks

Password systems are vulnerable to
man-in-the-middle attacks

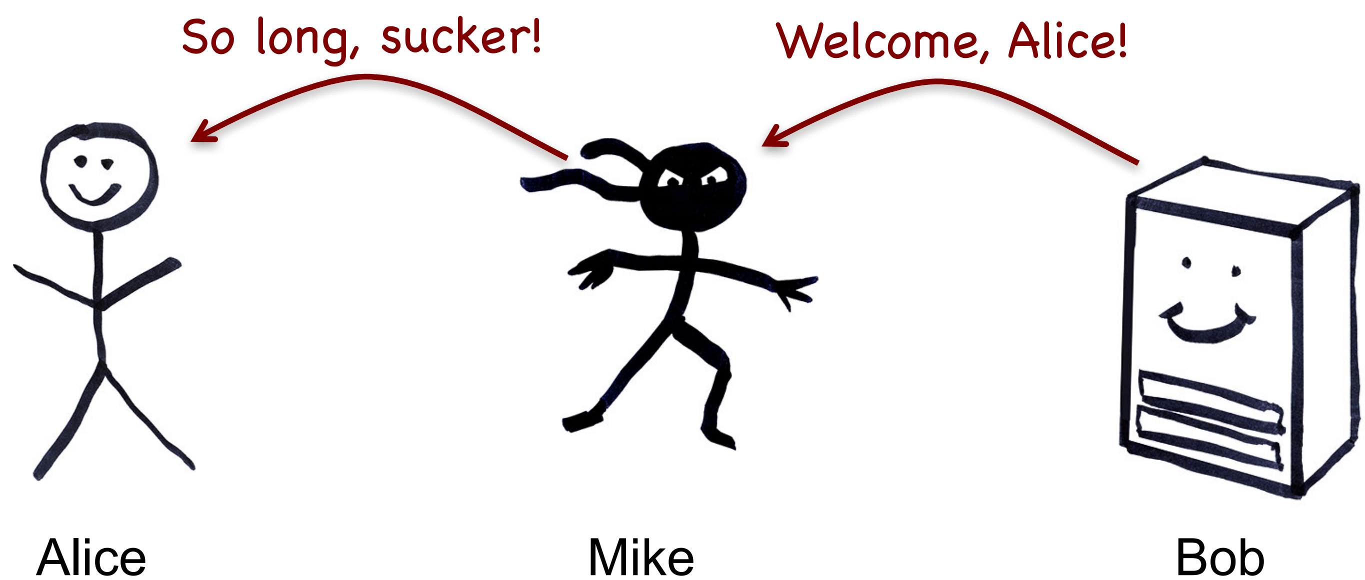
– Attacker acts as the server



Man-in-the-Middle Attacks

Password systems are vulnerable to
man-in-the-middle attacks

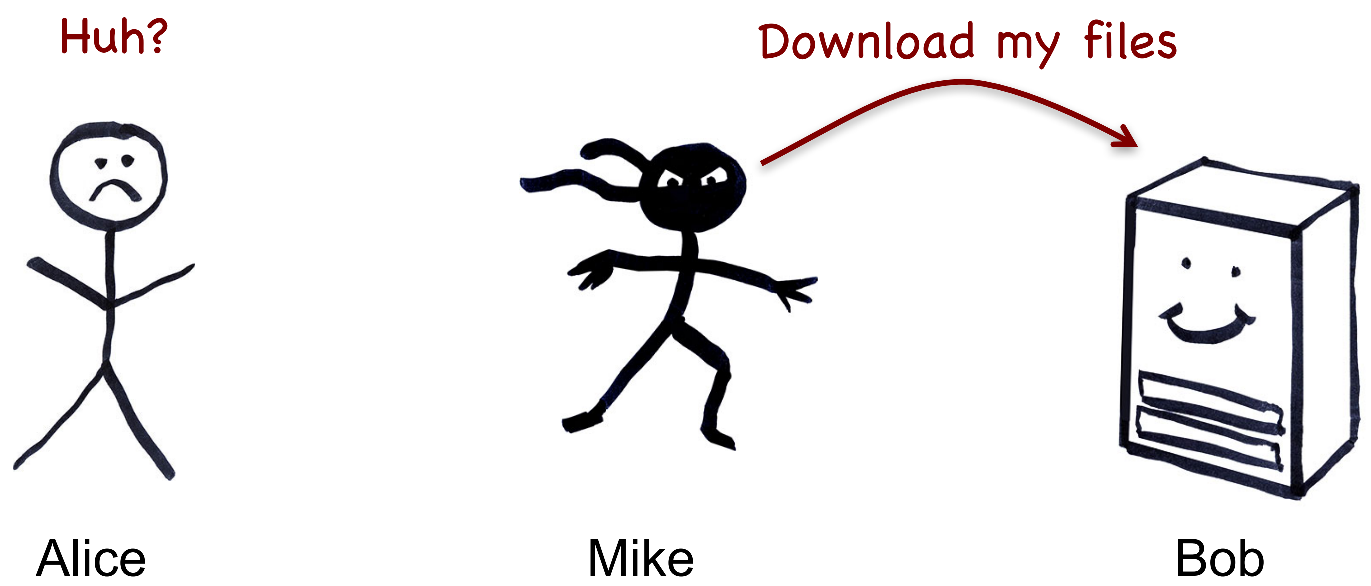
– Attacker acts as the server



Man-in-the-Middle Attacks

Password systems are vulnerable to
man-in-the-middle attacks

– Attacker acts as the server



Guarding against man-in-the-middle

- Use a covert communication channel
  - The intruder won't have the key
  - Can't see the contents of any messages
  - But you can't send the key over that channel!

- Use signed messages
  - Signed message = { *message* and *encrypted hash of message* }
  - Both parties can reject unauthenticated messages
  - The intruder cannot modify the messages
    - Signatures will fail (they will need to know how to encrypt the hash)

Combined authentication and key exchange

Wide-mouth frog

Alice $\longrightarrow$ Trent

$$\text{“alice”}, E_A(T_A, \text{“bob”}, K)$$

session key

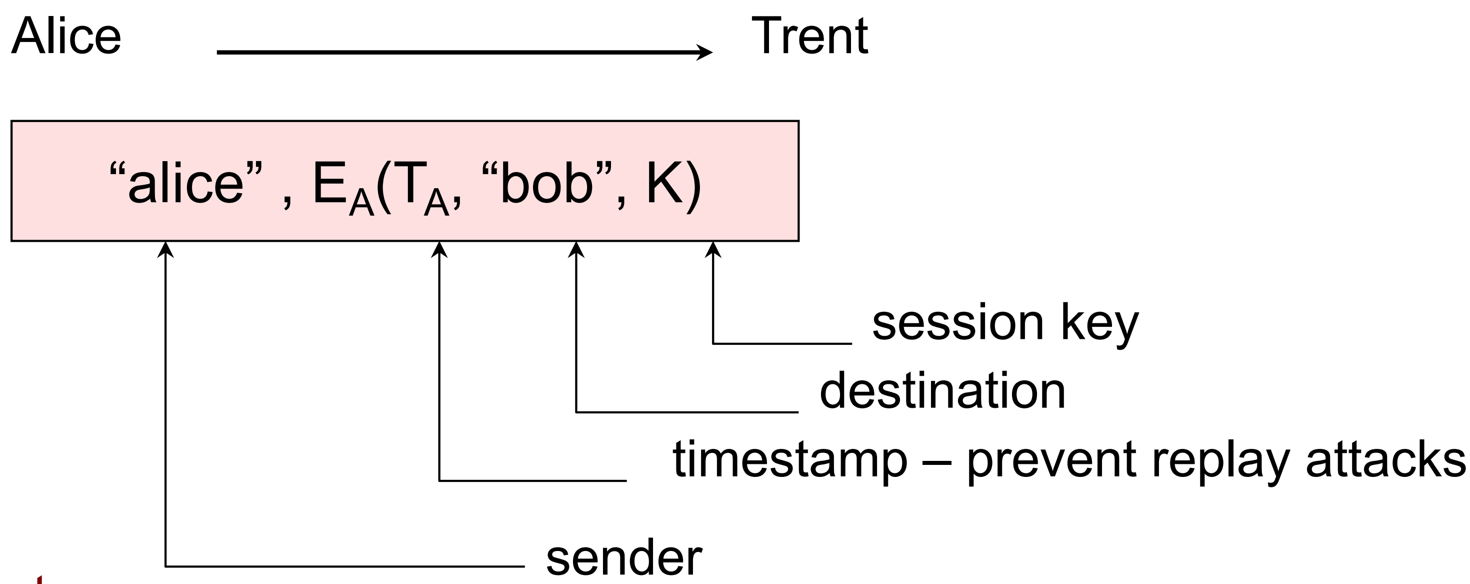
destination

timestamp – prevent replay attacks

sender

- Arbitrated protocol – Trent (3rd party) has all the keys
- Symmetric encryption used for transmitting a session key

Wide-mouth frog



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 $\longrightarrow$ Trent $\longrightarrow$ Bob

“alice” , $E_A(T_A$, “bob”, $K)$

$E_B(T_T$, “alice”, $K)$

session key

source

timestamp – 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

timestamp – 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

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 is a trusted third party

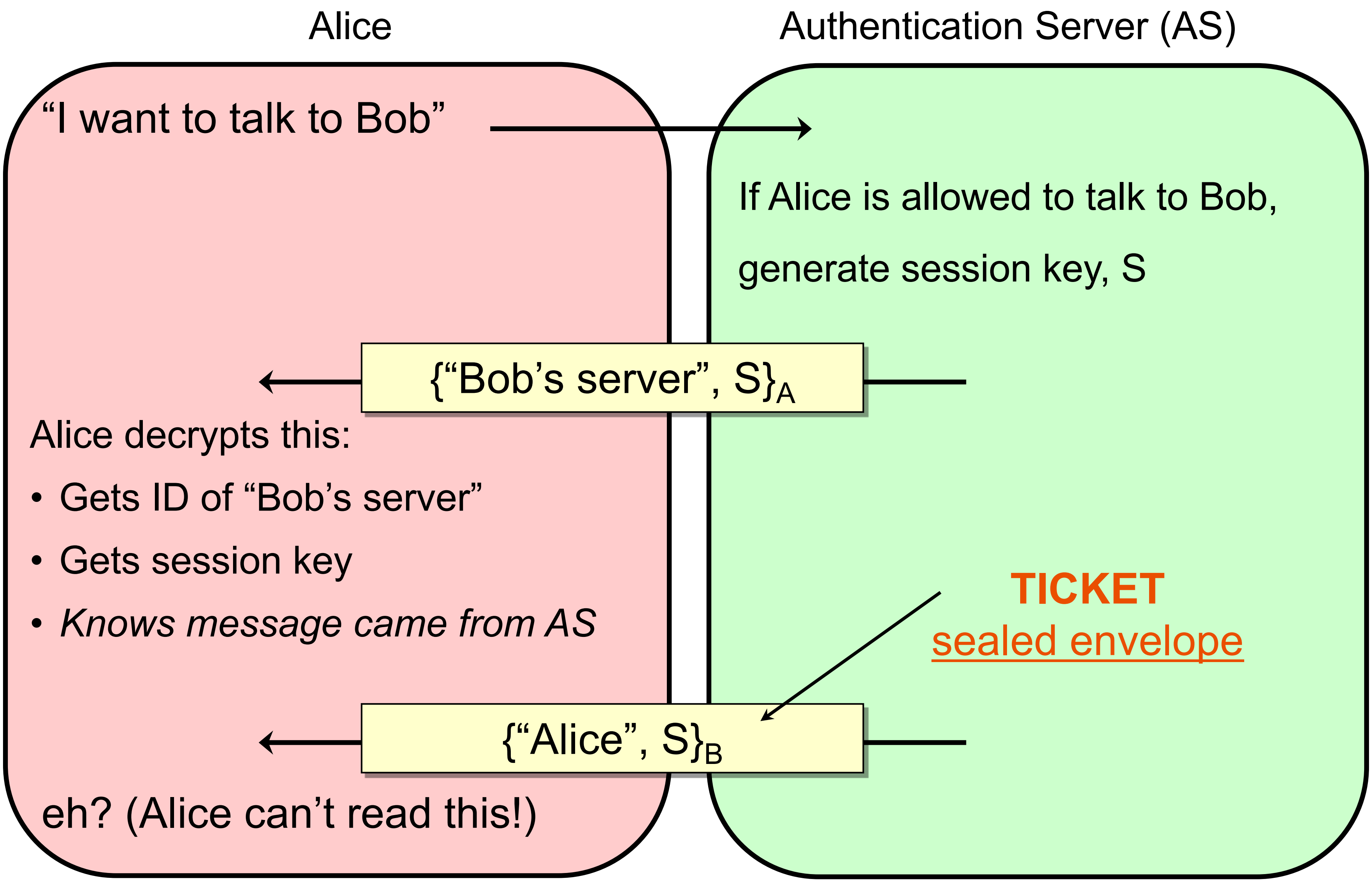
– Knows all (users and services) passwords
– Responsible for
  - Authentication: validating an identity
  - Authorization: deciding whether someone can access a service
  - Key exchange: giving both parties an encryption key (securely)

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



Send key

Alice

Bob

Alice encrypts a timestamp with session key

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

sealed envelope

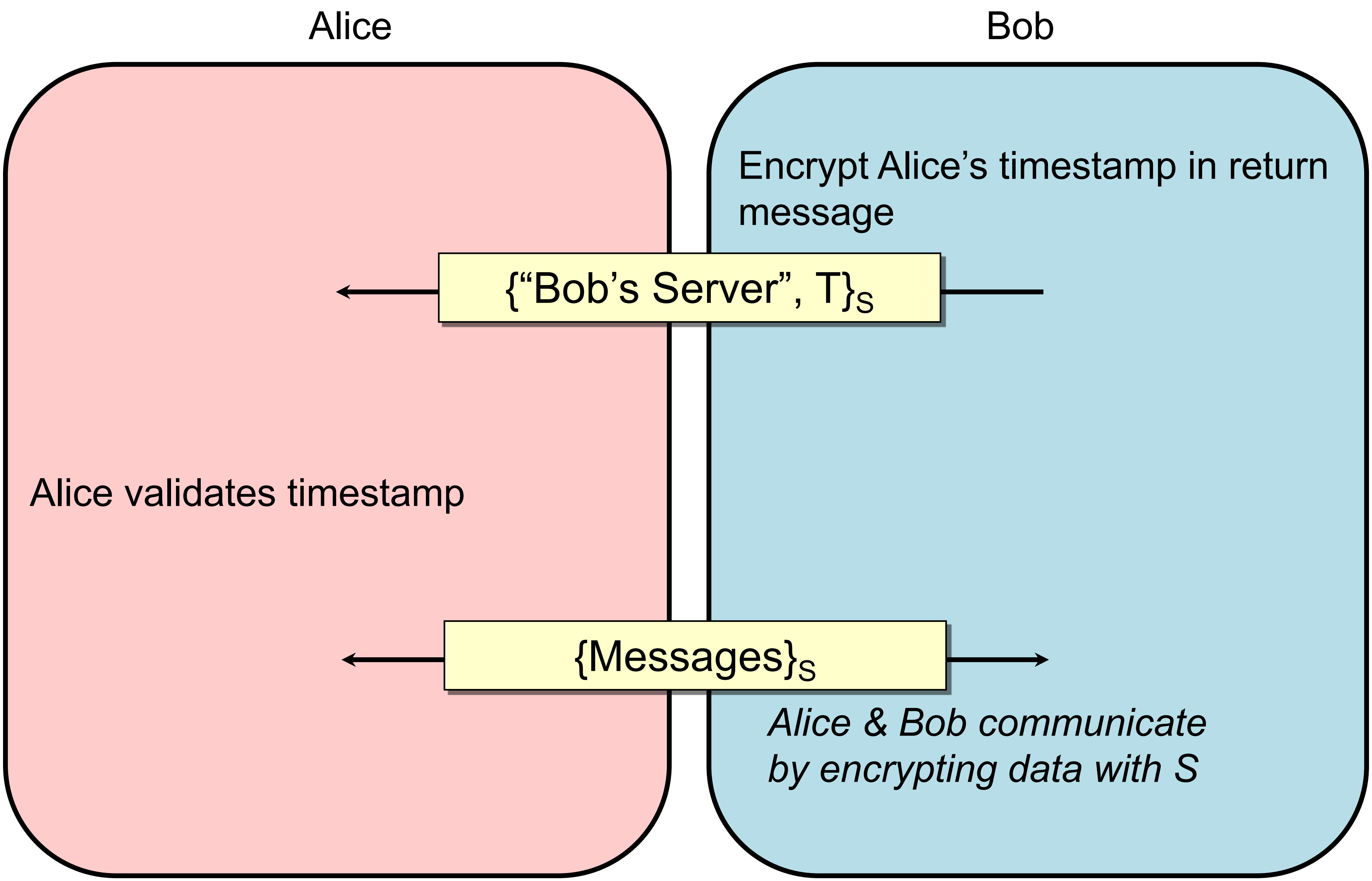
Bob decrypts envelope:

- Envelope was created by Kerberos on request from Alice
- Gets session key

Decrypts time stamp

- Validates time window
- Prevent replay attacks

Authenticate recipient of message



Kerberos key usage

- Every time a user wants to access a service
  - User's password (key) must be used to decode the message from Kerberos
- We can avoid this by caching the password in a file
  - Not a good idea
- Another way: create a temporary password
  - We can cache this temporary password
  - Similar to a session key for Kerberos – to get access to other services
  - Split Kerberos server into

    Authentication Server + Ticket Granting Server

Ticket Granting Service (TGS)

TGS + AS = KDC (Kerberos Key Distribution Center)

- Authentication Server
  - Authenticates user, gives a session key to access the TGS
  - Before accessing any service, user requests a ticket to contact TGS
- Ticket Granting Server
  - Anytime a user wants a service, request a ticket from TGS
  - Reply is encrypted with the TGS session key
- TGS works like a temporary ID

Using Kerberos

`$ kinit`

`Password:` *enter password*

ask AS for permission (session key) to access TGS

Alice gets:

{"TGS", S}$_A$ ← *Session key*
{"Alice", S}$_{TGS}$ ← *TGS Ticket*

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

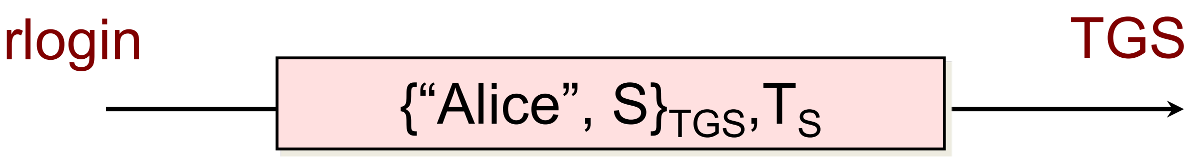
You now have a ticket to access the Ticket Granting Service

Using Kerberos

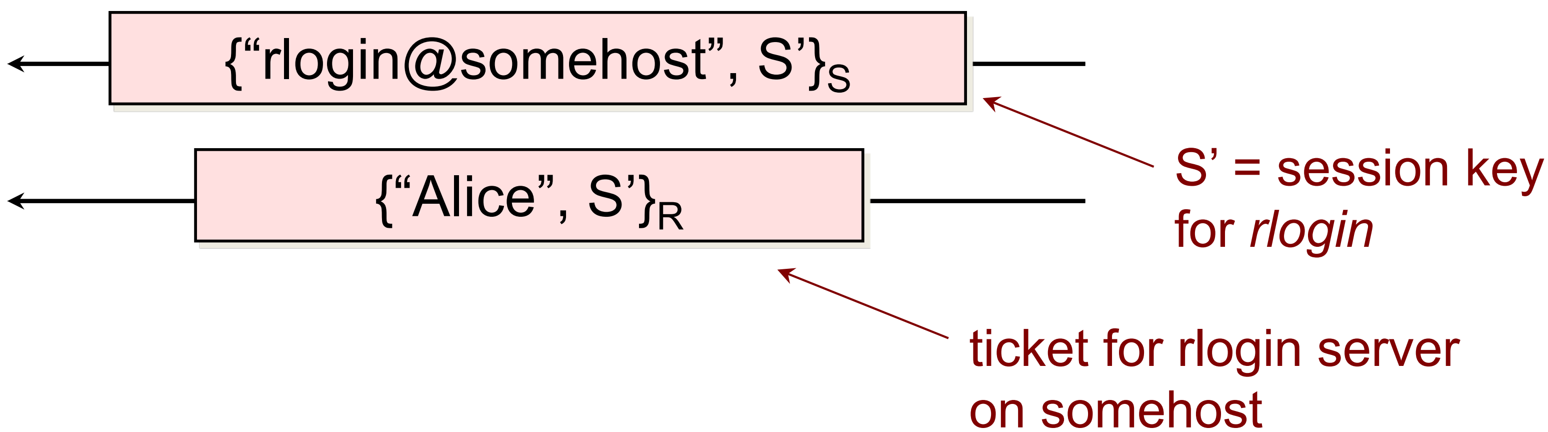
$ rlogin *somehost*

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

Alice sends session key, S, to TGS



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



Public Key Authentication

Public key authentication

Demonstrate we can encrypt or decrypt a *nonce*

This shows we have the right key

A random bunch of bits

- Alice wants to authenticate herself to Bob:
- Bob: generates nonce, *S*
  – Sends it to Alice
- Alice: encrypts *S* with her private key (signs it)
  – Sends result to Bob

Public key authentication

Bob:

1. Look up “alice” in a database of public keys
2. Decrypt the message from Alice using Alice’s public key
3. If the result is *S*, then Bob is convinced he’s talking with Alice

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

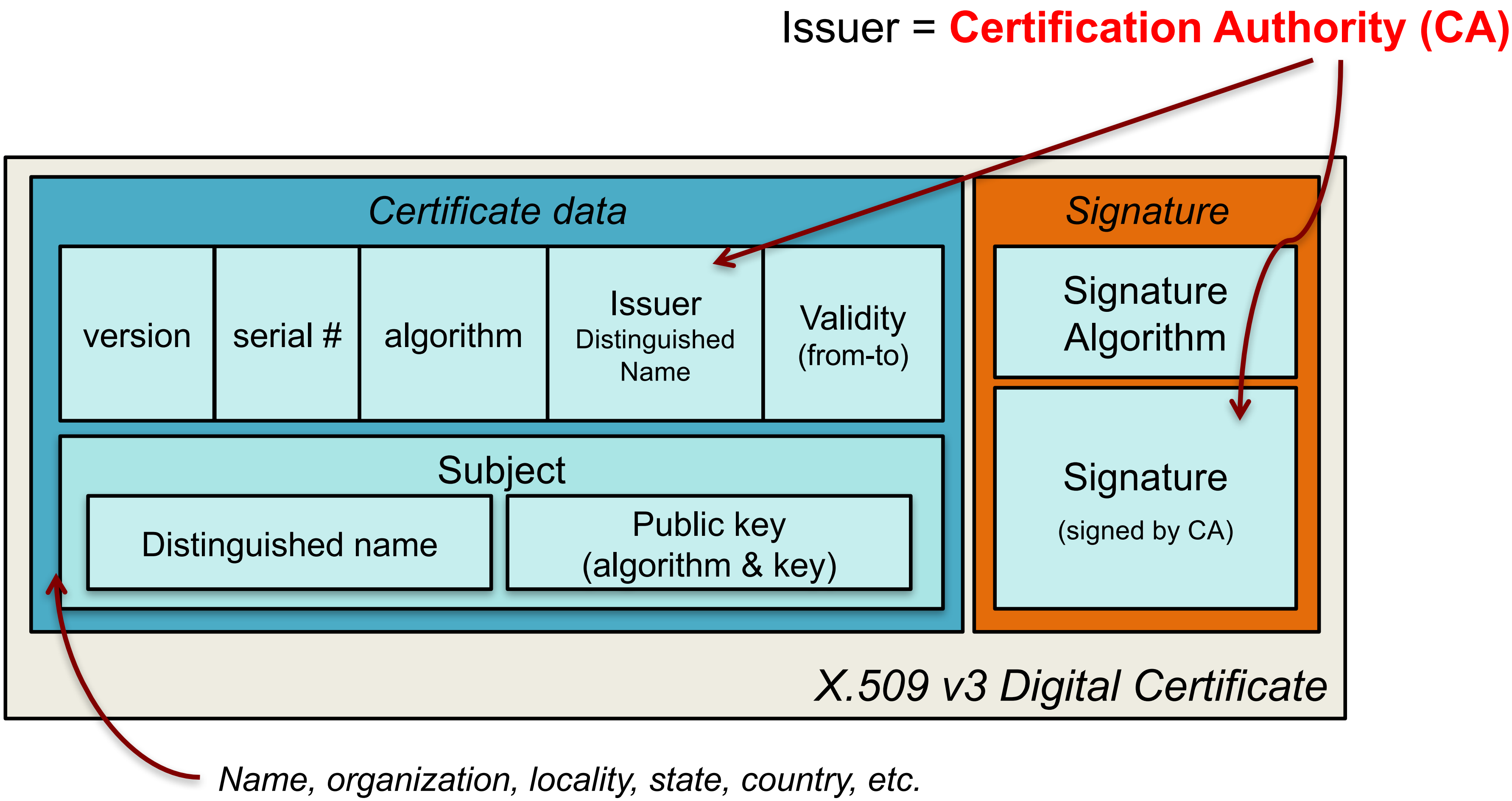
Public key authentication

- Public key authentication relies on binding identity to a public key
  - *How do you know it really is Alice's 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*
  - Contents cannot be modified
  - **digital certificate**

X.509 Certificates

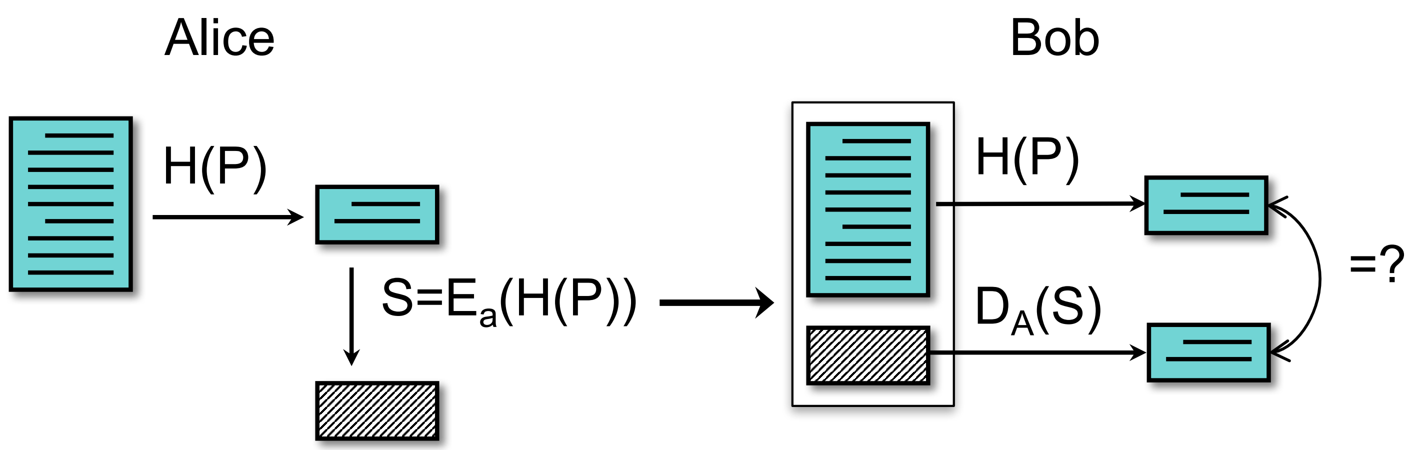
ISO introduced a set of authentication protocols

X.509: Structure for public key certificates:



Reminder: What's a digital signature?

Hash of a message encrypted with the signer's private key



X.509 certificates

When you get a certificate

- Verify its 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

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

…if you trust the CA

SSL/TLS

Transport Layer Security

- Provide a transport layer security protocol
- After setup, applications feel like they are using TCP sockets

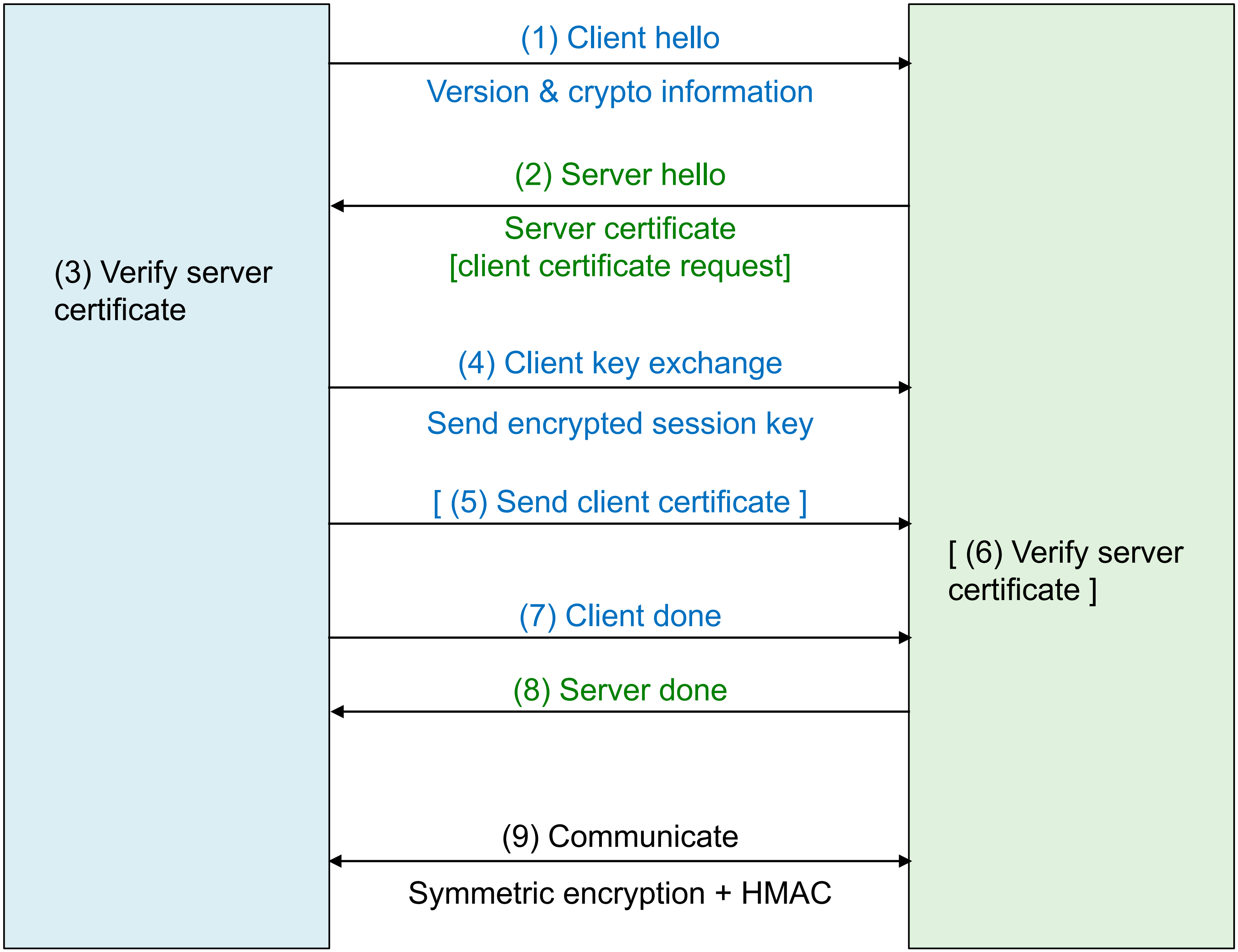
SSL: Secure Socket Layer

- Created with HTTP in mind
  - Web sessions should be secure
  - Mutual authentication is usually not needed
    - Client needs to identify the server but the server won't know all clients
    - Rely on passwords after the secure channel is set up
- SSL evolved to TLS (Transport Layer Security)
  - SSL 3.0 was the last version of SSL … and is considered insecure
  - We use TLS now … but often still call it SSL

Transport Layer Security (TLS)

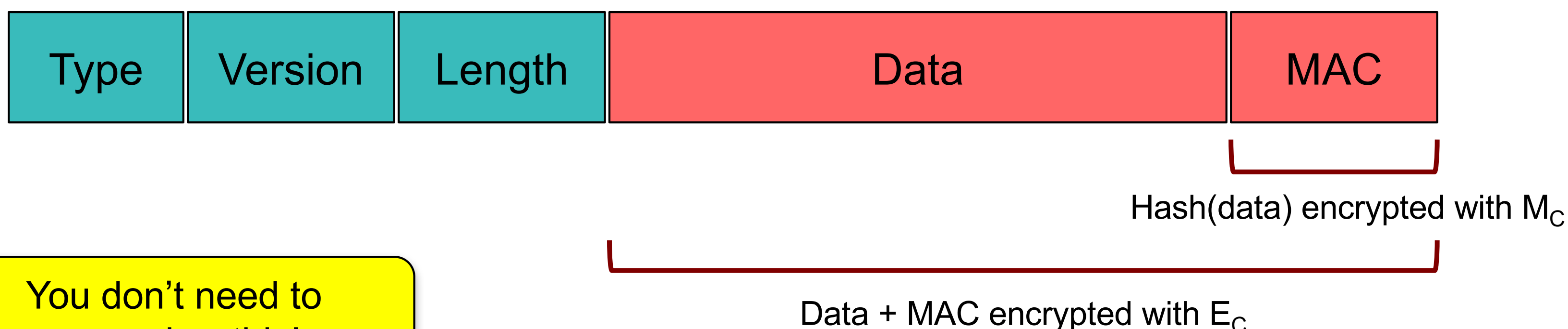
- *aka* Secure Socket Layer (SSL), which is an older protocol
- Sits on top of TCP/IP
- Goal: provide an encrypted and possibly authenticated communication channel
  - Provides authentication via RSA and X.509 certificates
  - Encryption of communication session via a symmetric cipher
- Hybrid cryptosystem: (usually, but also supports Diffie-Hellman)
  - Public key for authentication
  - Symmetric for data communication
- Enables TCP services to engage in secure, authenticated transfers
  - http, telnet, ntp, ftp, smtp, …

TLS Protocol



SSL Keys … more details

- SSL really uses four session keys
  - E_C – encryption key for messages from Client to Server
  - M_C – MAC encryption key for messages from Client to Server
  - E_S – encryption key for messages from Server to Client
  - M_S – MAC encryption key for messages from Server to Client
- They are all derived from the random key selected by the client



You don't need to remember this!

OAuth 2.0

Service Authorization

- You want an app to access your data at some service
  - E.g., access your Google calendar data
- But you want to:
  - Not reveal your password to the app
  - Restrict the data and operations available to the app
  - Be able to revoke the app's access to the data

OAuth 2.0: Open Authorization

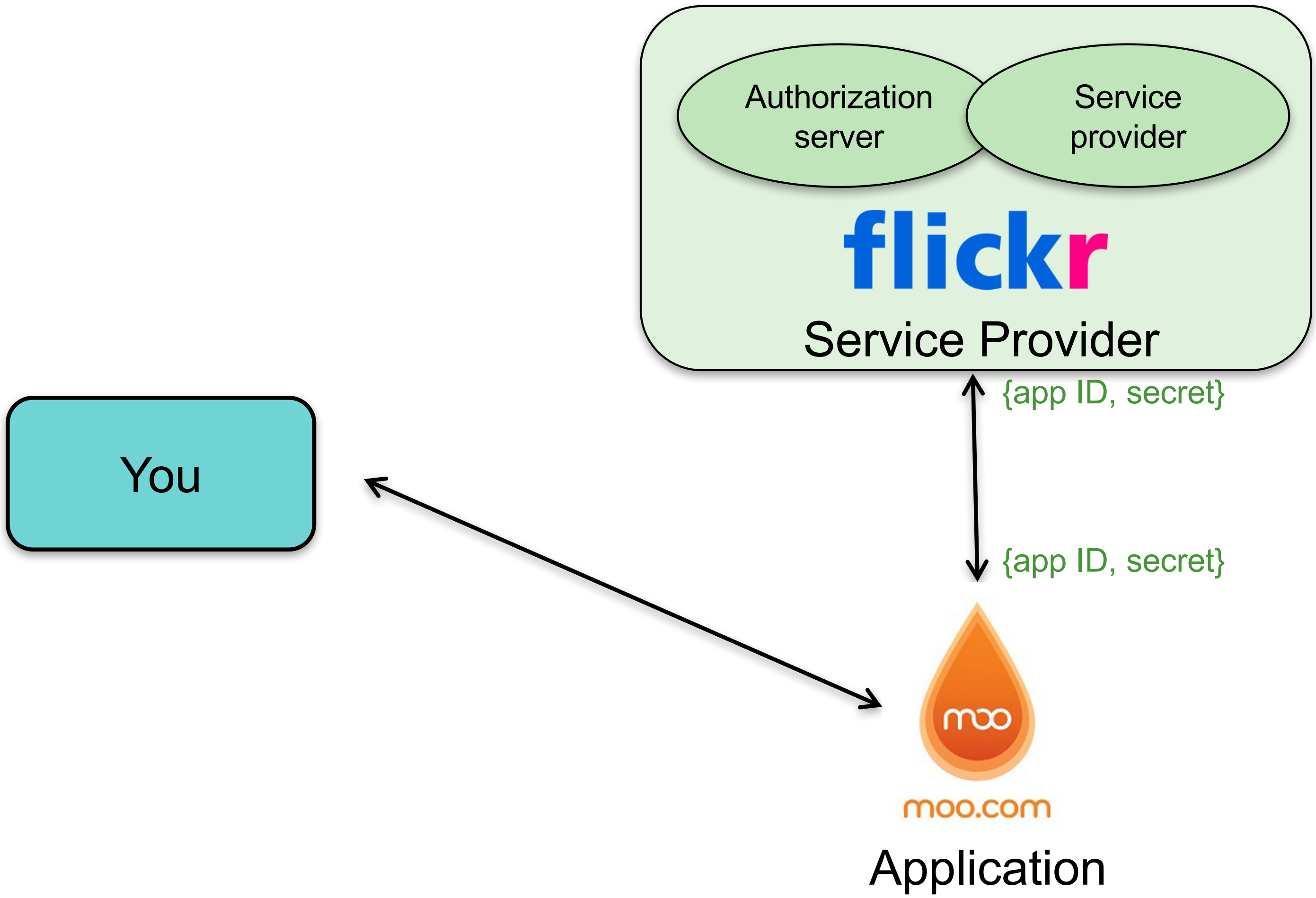
- OAuth: framework for service authorization
  - Allows you to authorize one website (consumer) to access data from another website (provider) – *in a restricted manner*
  - Designed initially for web services
  - Examples:
    - *Allow the Moo photo printing service to get photos from your Flickr account*
    - *Allow the NY Times to tweet a message from your Twitter account*
- OpenID Connect
  - Remote identification: use one login for multiple sites
  - Encapsulated within OAuth 2.0 protocol

OAuth setup

- OAuth is based on
  - Getting a token from the service provider & presenting it each time an application accesses an API at the service
  - URL redirection
  - JSON data encapsulation

- Register a service
  - Service provider (e.g., Flickr):
    - Gets data about your application (name, creator, URL)
    - Assigns the application (consumer) an ID & a secret
    - Presents list of authorization URLs and scopes (access types)

OAuth Entities

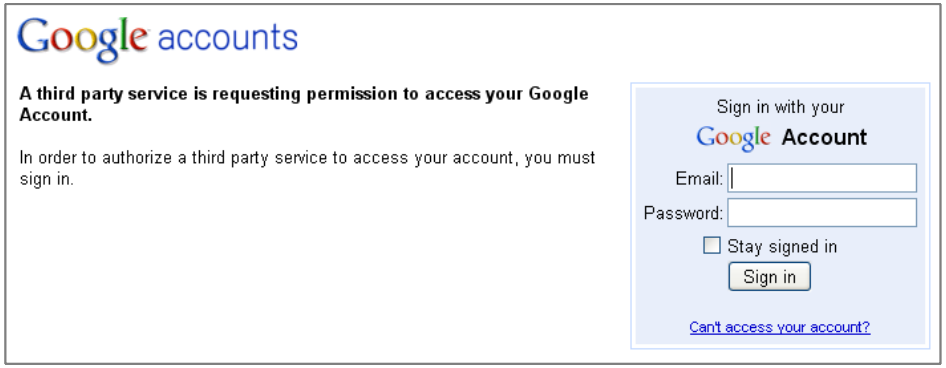


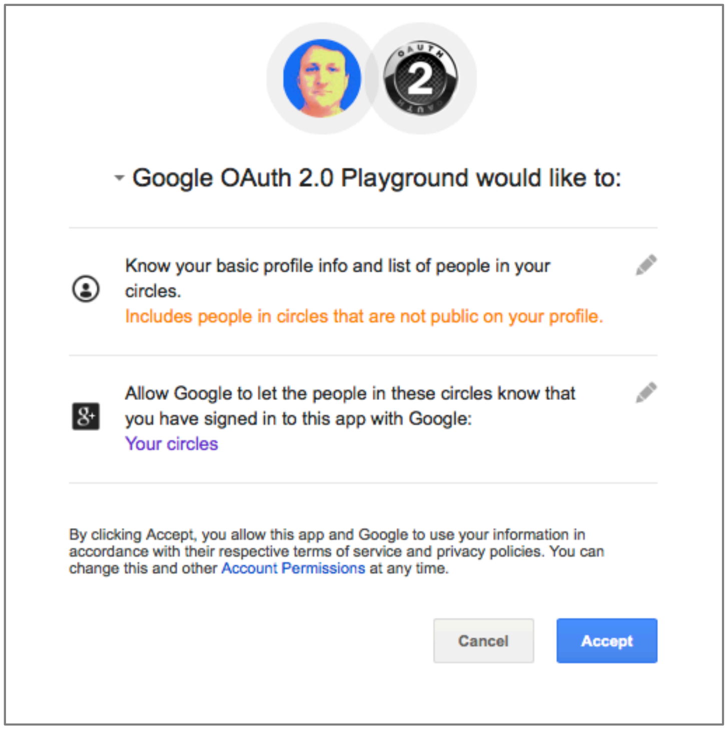
You want moo.com to access your photos on flickr

How does authorization take place?

- Application needs a *Request Token* from the Service (e.g., moo.com needs an *access token* from flickr.com)
  - Application redirects user to Service Provider
    - Request contains: *client ID, client secret, scope* (list of requested APIs)
    - User may need to authenticate at that provider
    - User authorizes the requested access
    - Service Provider redirects back to consumer with a one-time-use authorization code
  - Application now has the *Authorization Code*
    - The previous redirect passed the Authorization Code as part of the HTTP request – therefore not encrypted
  - Application exchanges *Authorization Code* for *Access Token*
    - The legitimate app uses HTTPS (encrypted channel) & sends its secret
    - The application now talks securely & directly to the Service Provider
    - Service Provider returns Access Token
  - Application makes API requests to Service Provider using the **Access Token**

Key Points





- You still may need to log into the Provider's OAuth service when redirected

- You approve the specific access that you are granting

- The Service Provider validates the requested access when it gets a token from the Consumer

Play with it at the ***OAuth 2.0 Playground***:
https://developers.google.com/oauthplayground/

Identity Federation: OpenID Connect

OpenID Connect

- Designed to solve the problem of
  - Having to get an ID per service (website)
  - Managing passwords per site
  - Layer on top of OAuth 2.0
- Decentralized mechanism for single sign-on
  - Access different services (sites) using the same identity
    - Simplify account creation at new sites
  - User chooses which OpenID provider to use
    - OpenID does not specify authentication protocol – up to provider
  - Website never sees your password
- *OpenID Connect* is a standard but not the only solution
  - Used by Google, Microsoft, Amazon Web Services, PayPal, Salesforce, …
  - Facebook Connect – popular alternative solution
    (similar in operation but websites can share info with Facebook, offer friend access, or make suggestions to users based on Facebook data)

OpenID Connect Authentication

- OAuth requests that you specify a "scope"
  - List of access methods that the app needs permission to use
- To enable user identification
  - Specify "openid" as a requested scope
- Send request to server (identity provider)
  - Server requests user ID and handles authentication
- Get back an access token
  - If authentication is successful, the token contains:
    - user ID
    - approved scopes
    - expiration
    - etc.

same as with OAuth requests for authorization

Cryptographic toolbox

- Symmetric encryption
- Public key encryption
- One-way hash functions
- Random number generators
  - Used for nonces and session keys

Examples

- Key exchange
  - Public key cryptography
- Key exchange + secure communication
  - Random # + Public key + symmetric cryptography
- Authentication
  - Nonce (random #) + encryption
- Message authentication codes
  - Hashes
- Digital signature
  - Hash + encryption with private key

The End