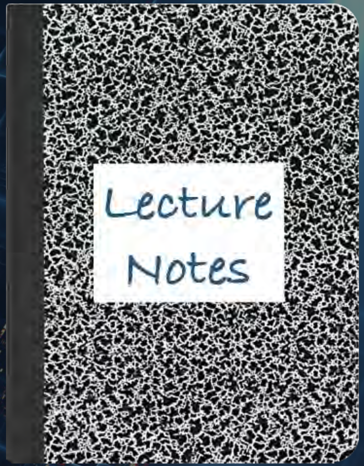
CS 417 – DISTRIBUTED SYSTEMS



Week 12: Security in Distributed Systems

Part 3: Authentication

Paul Krzyzanowski

Authentication

For a user (or process):

- Get the user's identity = **identification**
- Verify the identity = **authentication**
- Then decide whether to allow access to resources = **authorization**

Three Factors of Authentication

1. Ownership Something you have	*Key, card*	*Can be stolen*
2. Knowledge Something you know	*Passwords, PINs*	*Can be guessed, shared, stolen*
3. Inherence Something you are	*Biometrics (face, fingerprints)*	*Requires hardware* *May be copied* *Not replaceable if lost or stolen*

Multi-Factor Authentication

Factors may be combined

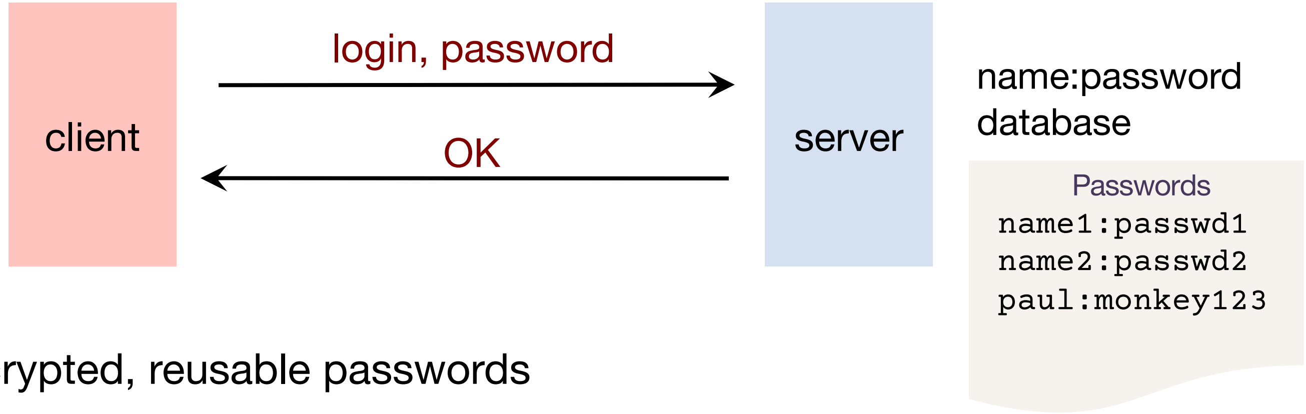
- ATM machine: 2-factor authentication (2FA)
  - ATM card — something you have
  - PIN — something you know
- Password + code delivered via SMS: 2-factor authentication
  - Password — something you know
  - Code — something you have: your phone

Two passwords ≠ Two-factor authentication

The factors must be different

Authentication: PAP

Password Authentication Protocol



- Unencrypted, reusable passwords
- Insecure on an open network
- Also, the password file must be protected from open access
  - But administrators can still see everyone’s passwords
    What if you use the same password on Facebook as on Amazon?

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

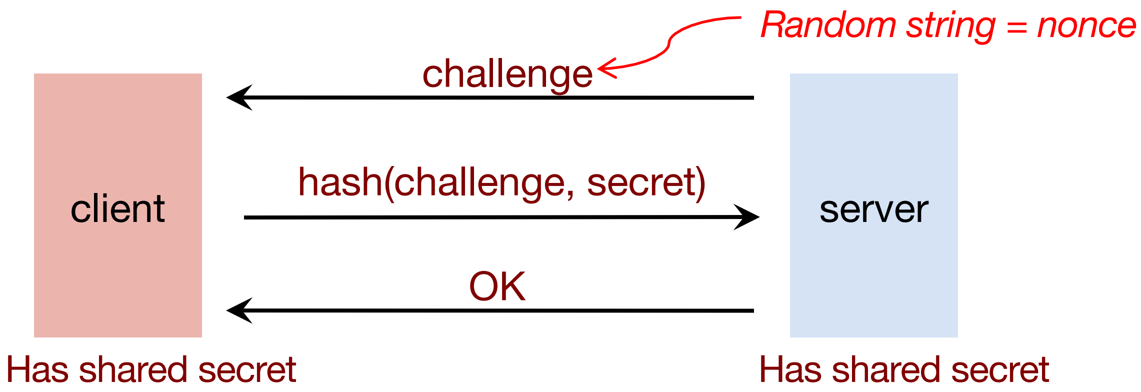
- Given a file, you don't get the passwords
- The attacker must resort to a dictionary or brute-force attack
- For example, Linux passwords are hashed with SHA-512 hashes (512-bit SHA-2 hash)

PROBLEM 2: Sniffing

Someone who can see network traffic (or over your shoulder) can see the password!

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*

$R' = f(K,C)$ ← ***C*** ←

→ ***R'*** → $R = f(K, C)$

← **"welcome"** ← $\boldsymbol{R = R'}$ **?**

an eavesdropper does not see K

TOTP: Time-Based Authentication

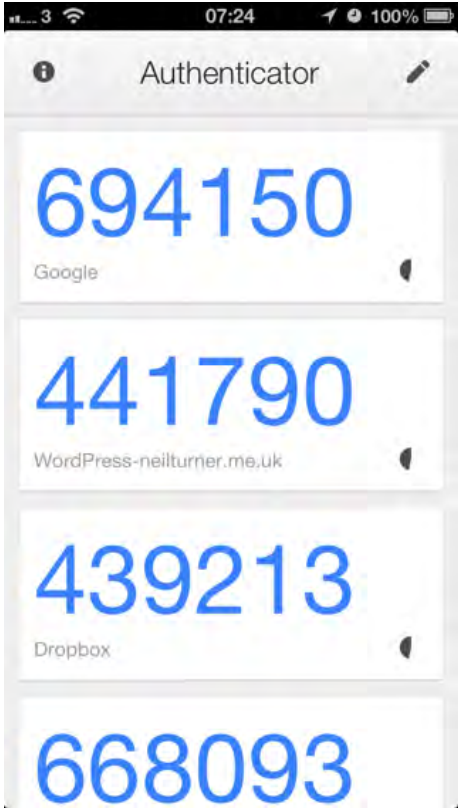
Time-based One-time Password (TOTP) algorithm

- Both sides share a secret key
  - Sometimes sent via a QR code so user can scan it into the TOTP app
- User runs TOTP function to generate a one-time password

  one_time_password = hash(*secret_key*, *time*)
- User logs in with:

  Name, *password*, and *one_time_password*
- Service generates the same password

  one_time_password = hash(*secret_key*, *time*)



Public Key Authentication

Public key authentication

Demonstrate we can encrypt or decrypt a *nonce*

This shows we know the 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 must present Bob with a nonce that Bob will encrypt with his private key and return

Public Keys as Identities

- A **public key** can be treated as an identity
  - Only the owner of the corresponding private key will be able to create the signature
- New identities can be created by generating new random {private, public} key pairs
- Anonymous identity – no identity management
  - A user is known by a random-looking public key
  - Anybody can create a new identity at any time
  - Anybody can create as many identities as they want
  - A user can throw away an identity when it is no longer needed
  - Example: Bitcoin identity = hash(public key)

Passkeys - WebAuthn

Passkeys = Passwordless login – endorsed by Apple, Google, Microsoft

- Avoid problems of having users choose strong, unique passwords
- Avoids phishing attacks
- Based on public key cryptography
  - Credentials can be backed up and replicated across user devices

Device generates public/private key pair for a specific service

- Private key is stored locally – the service never sees it
  - Its use can be authorized with Touch ID, Face ID, local device/user password
- Public key is sent to the server – it associates it with the user account

Passkeys – Setup

User Alice

Service

username: alice

Password and/or authentication code

Conventional login

Create public/private key pair for the service

Elliptic Curve or RSA algorithms

Public key

Note: the public key is not secret

Store private key securely

Accessible via local password or biometrics.
Associate this key with the service.

Store public key with user info

Enable lookup when presented with a user login name

Each passkey is unique for each service

See https://passage.id/post/what-is-webauth

Passkeys – Login

User Alice

Service

username: alice →

← *Here's a challenge:* `XdQLAxB1L1...`

Generate signature for challenge:
Encrypt hash(challenge) with your private key for this service

Authorize access to private key via Touch ID, Face ID, password, …

signature(challenge) →

Validate signature:
Decrypt signature with the user's public key and compare it with hash(challenge)

← *Welcome, Alice!*

See https://passage.id/post/what-is-webauth

Public key authentication – Identity Binding

Public key authentication relies on associating an identity with a public key

- *How do you know it really is Alice's public key?*

Sign the public key

- Once signed, it is tamper-proof
- But we need to know it's Bob's public key and who signed it
  - Create & sign a data structure that
  - Identifies Bob
  - Contains his public key
  - Identifies who is doing the signing

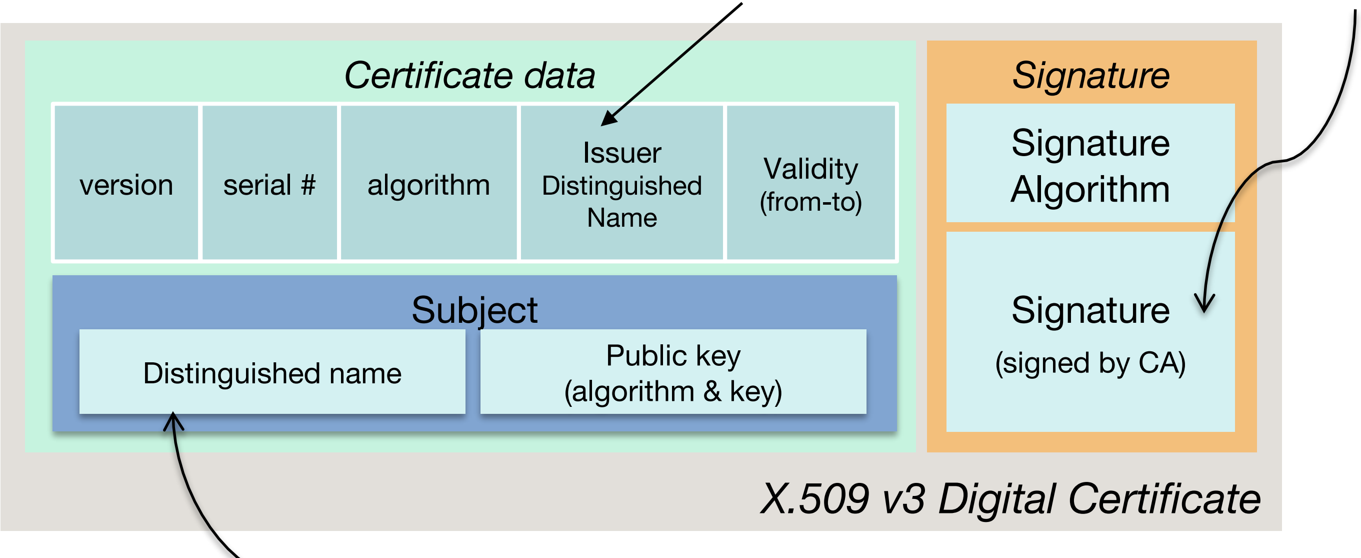
⇒ digital certificate

X.509 certificates

ISO introduced a set of authentication protocols

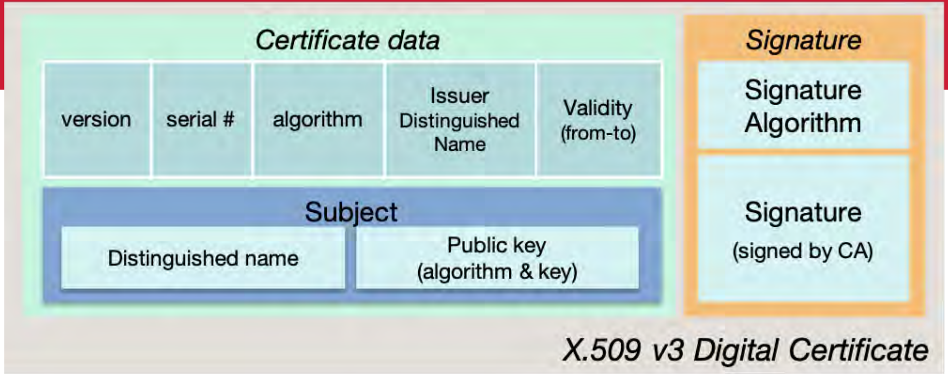
X.509: Structure for public key certificates:

Issuer = **Certification Authority (CA)**



User's name, organization, locality, state, country, etc.

X.509 certificates



To validate a certificate

Verify its signature:

1. Get the issuer (CA) from the certificate
2. Validate the certificate's signature against the issuer's public key
   - 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

Transport Layer Security (TLS)

Transport Layer Security (TLS)

Goal: 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

Enables TCP services to engage in secure, authenticated transfers

- http, telnet, nntp, ftp, smtp, xmpp, …

SSL evolved to **TLS** (**Transport Layer Security**)

TLS Protocol

Goal

Provide authentication (usually one-way), privacy, & data integrity between two applications

Principles

- **Data encryption**
  - Use symmetric cryptography to encrypt data
  - **Key exchange**: keys generated uniquely at the start of each session
- **Data integrity**
  - Include a MAC with transmitted data to ensure message integrity
- **Authentication**
  - Use public key cryptography & X.509 certificates for authentication
  - Optional – can authenticate 0, 1, or both parties
- **Interoperability & evolution**
  - Support many different key exchange, encryption, integrity, & authentication protocols – negotiate what to use at the start of a session

TLS Protocol & Ciphers

Two sub-protocols

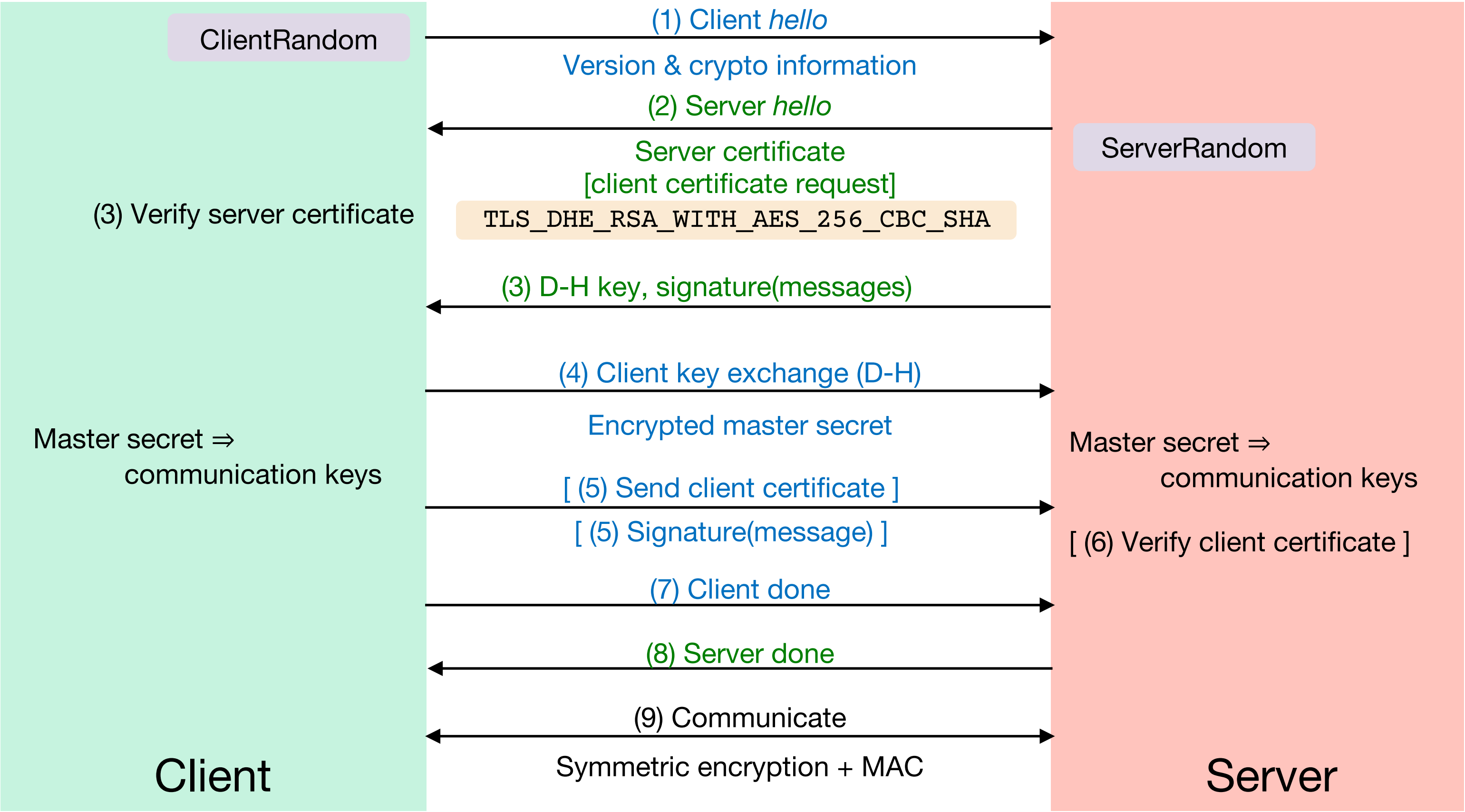
1. Authenticate & establish keys

- Authentication
  - Public keys (X.509 certificates and RSA or Elliptic Curve cryptography)
- Key exchange options
  - Ephemeral Diffie-Hellman (D-H) keys (generated for each session)

2. Communicate

- Data encryption options – *symmetric cryptography*
  - AES GCM, AES CBC, ChaCha20-Poly1305, …
- Data integrity options – *message authentication codes*
  - HMAC-SHA256/384, …

TLS Protocol



Benefits & Downsides of TLS

Benefits

- Validates the authenticity of the server (if you trust the CA)
- Protects integrity of communications
- Protects the privacy of communications

Downsides

- Longer latency for session setup
- Older protocols had weaknesses
- Attackers can use TLS too!

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 post a message from your X account*

OpenID Connect

- Remote identification: use one login for multiple sites
- Encapsulated within OAuth 2.0 protocol

OAuth setup

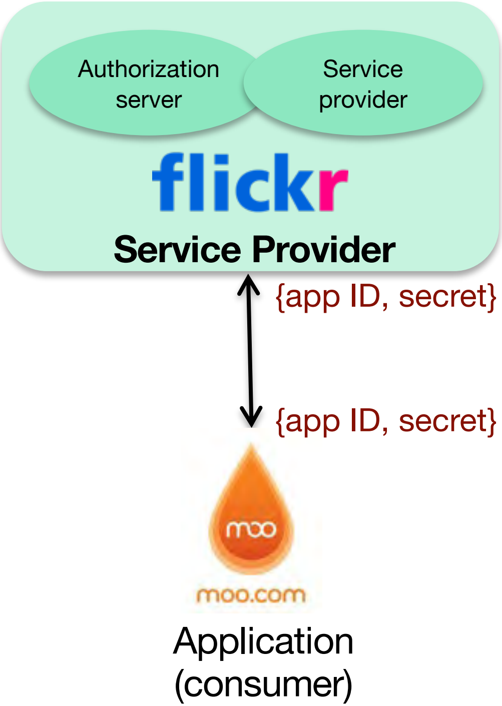
OAuth is based on

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

Before users can use OAuth, the app (consumer) must register with the service provider

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

OAuth Entities



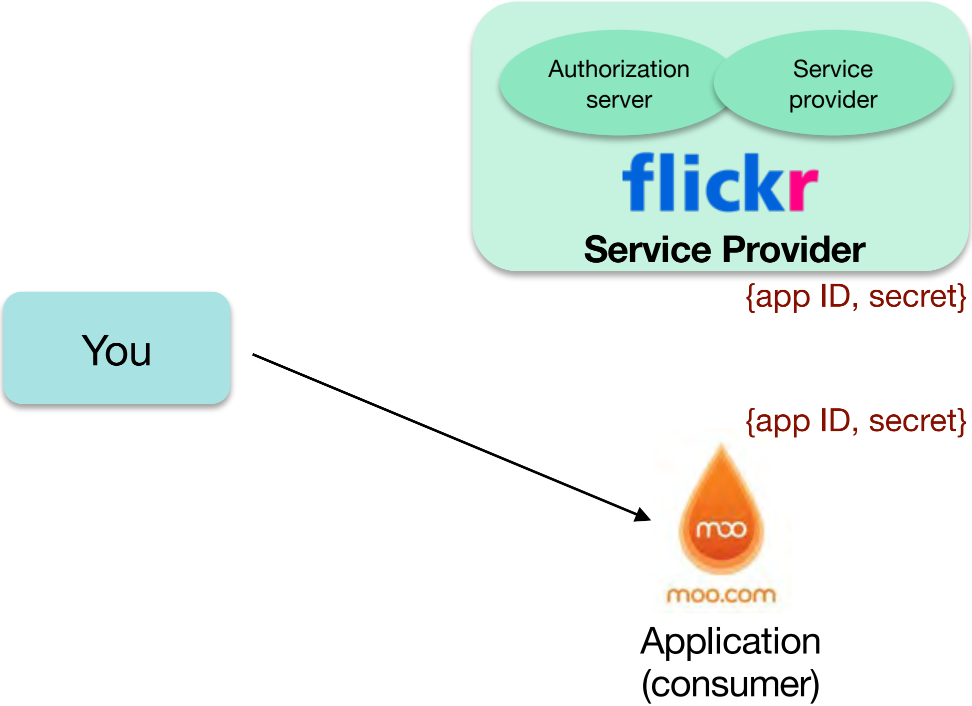
Initial setup

How does authorization take place?

Application needs an *Access Token* from the Service
(e.g., moo.com needs an *access token* from flickr.com)

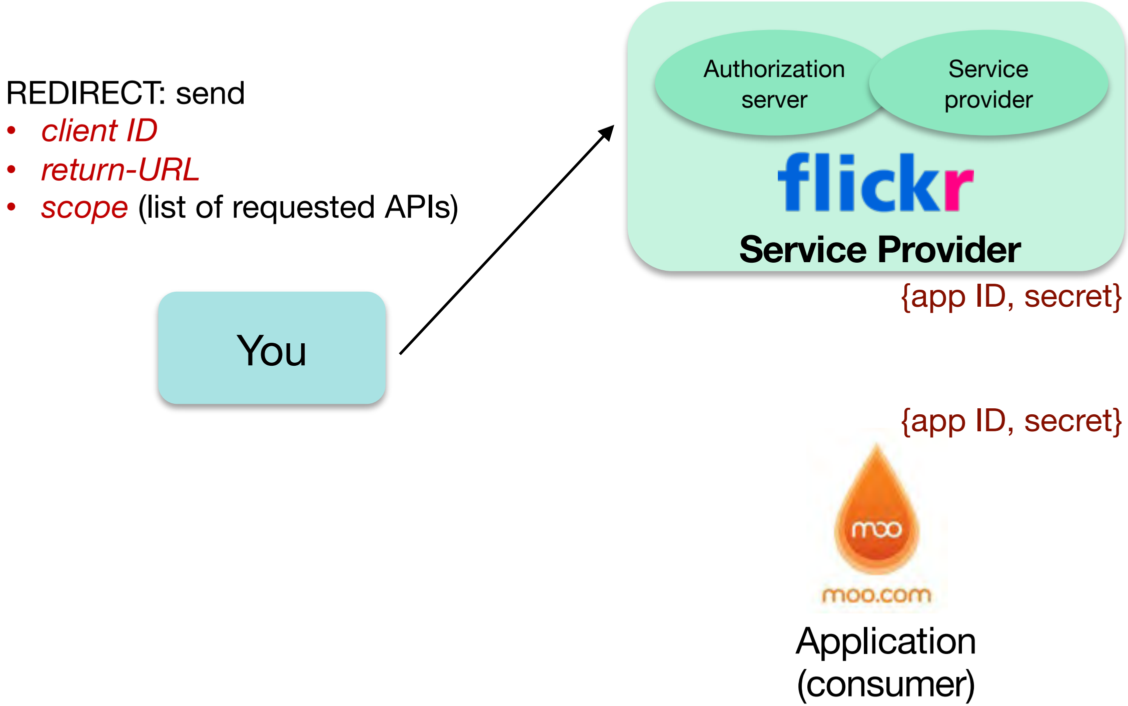
- Application redirects user to **Service Provider**
  - Request contains: *client ID, client secret, scope* (list of requested APIs)
  - User may need to authenticate at that provider
  - User authorizes the requested access
  - Service Provider redirects back to consumer with a one-time-use authorization code
- Application now has the *Authorization Code*
  - The previous redirect passed the Authorization Code as part of the HTTP request
- 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**

OAuth Entities



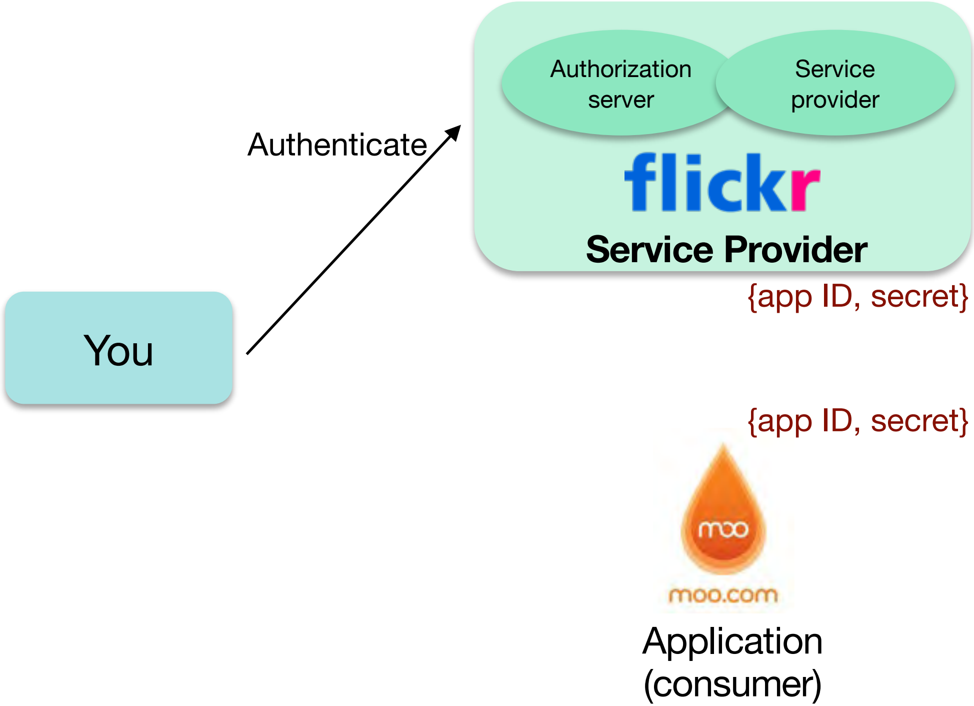
You want moo.com to access your photos on flickr

OAuth Entities



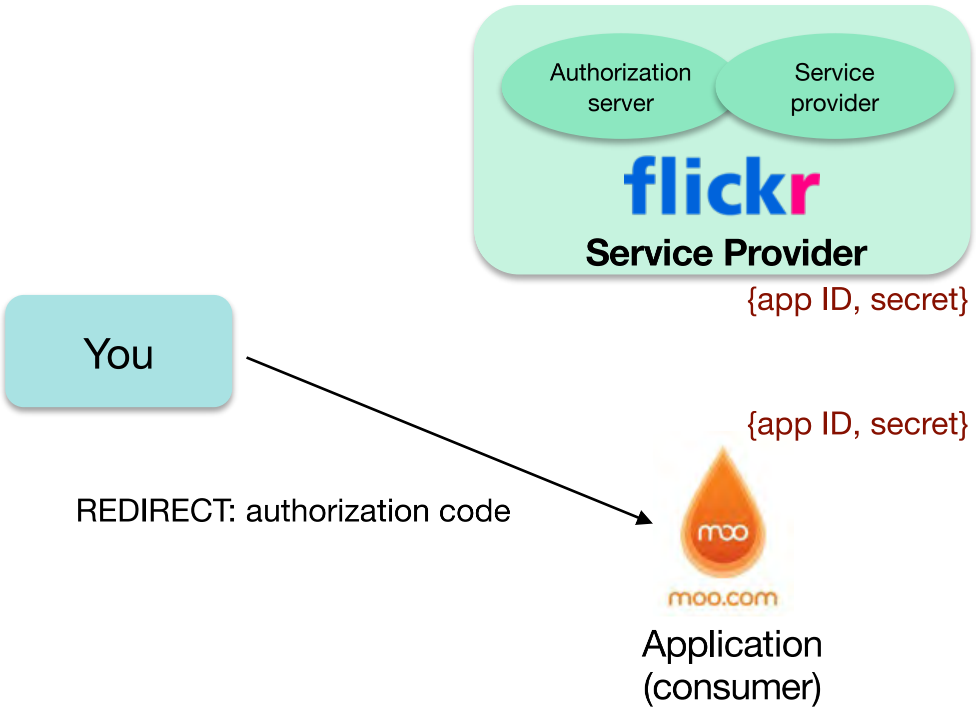
Moo.com app redirects you to the service provider

OAuth Entities



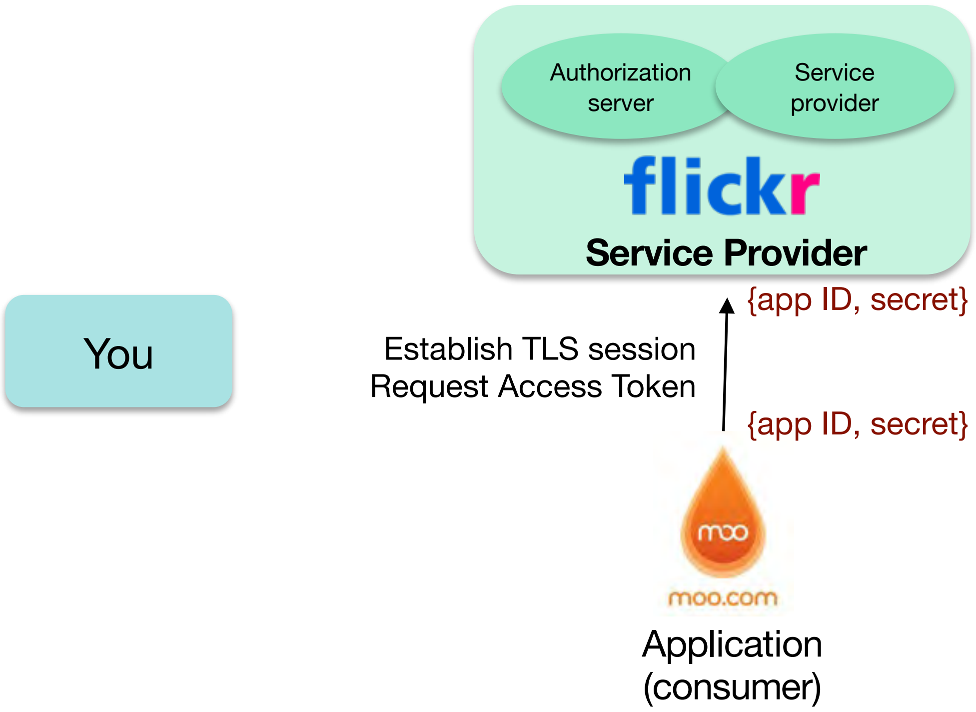
You authenticate (optional) & authorize the request at flickr

OAuth Entities



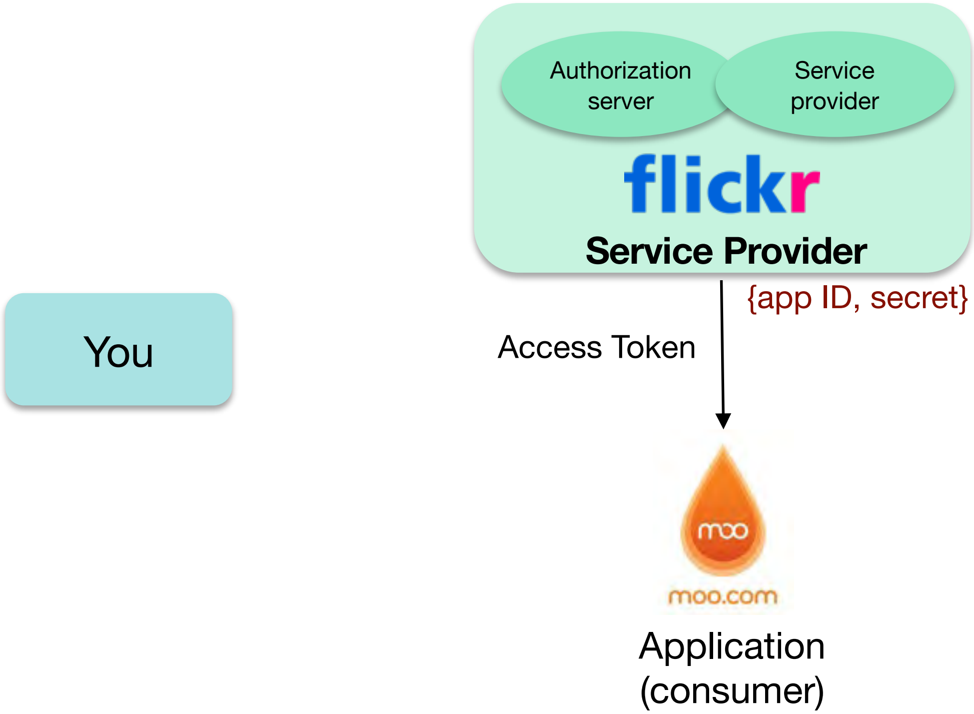
Flicker sends a redirect back with an authorization code

OAuth Entities



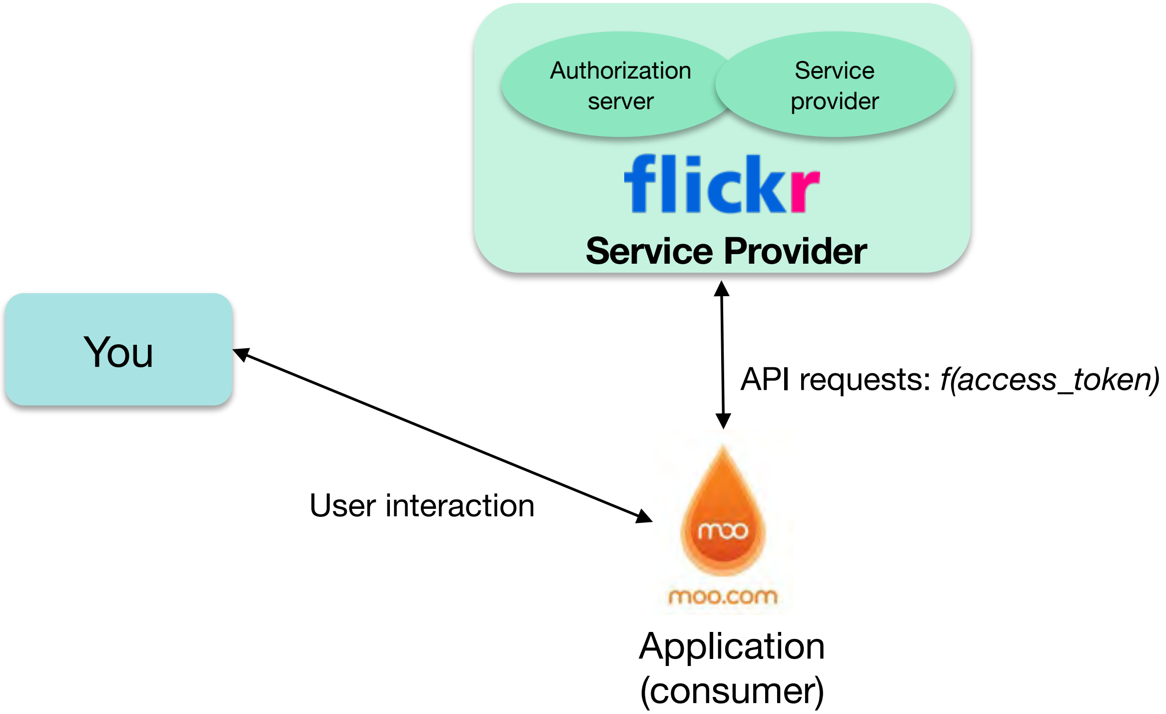
Moo requests an access token (securely)

OAuth Entities



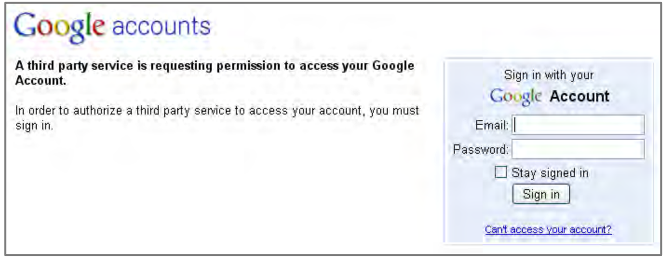
Moo gets the. access token (securely)

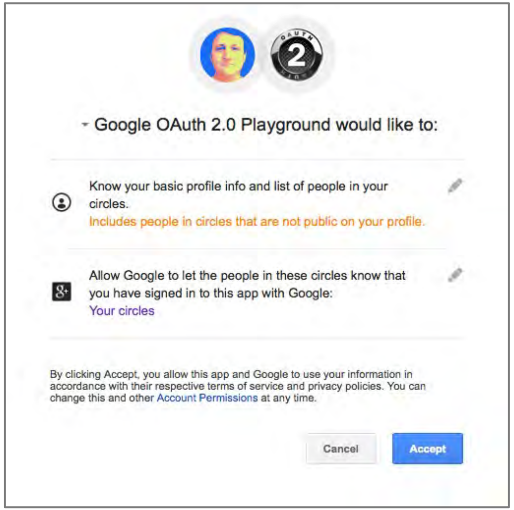
OAuth Entities



Moo can send requests to flickr (securely)

Key Points





- You may still 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

Single Sign-On: OpenID Connect

- Designed to solve the problems of
  - Having to get an ID per service (website)
  - Managing passwords per site

- **Decentralized mechanism for single sign-on** — *layer on top of Oauth 2.0*
  - 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, …
  - Sign in with Apple – based on OAuth 2.0 and OpenID Connect
  - 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 the identity provider
  - Handles user authentication
  - Redirects the user back to the client
- Provider returns an **access token** and an **ID token**
  - The **access token** contains:
    - approved scopes
    - expiration
    - etc.

    } same as with OAuth requests for authorization
  - The **ID token** can be read by the consumer (client) and contains
    - *Name, screen name, email, birthdate,* … whatever the Identity Provider chose to send

Cryptographic toolbox

- Symmetric encryption
- Public key encryption
- Hash functions
- Random number generators

Examples

- **Key exchange**
  - Public key cryptography
- **Key exchange + secure communication**
  - Random # + public key cryptography + symmetric cryptography
- **Authentication**
  - Nonce (random #) + encryption
- **Message authentication code**
  - Hash + symmetric keys (random #s)
- **Digital signature**
  - Hash + public key cryptography

The End