Authentication and Authorization

Basic Introduction for Web Apps

Authenticate & Authorize

- **Authentication** - validate the identity of a "user", agent, or process
- **Authorization** - specifying rights to access a resource

Authentication is responsible for identifying who the user is.

Authorization is responsible for deciding what the user has permission to do.

Other Aspects of Security

- Access Control - how app controls access to resources
- Data Integrity - ability to prevent data from being modified, and *prove* that data hasn't been modified
- Confidentiality & Privacy - (privacy is about people, confidentiality is about data)
- Non-repudiation - ability to prove that user has made a request
  - "repudiate" means to deny doing something
- Auditing - make a pamper-resistant record of security related events

Authentication Methods

Authentication methods for humans:

1. Username & password

2. Username & one-time password (TOTP, codes, SMS)

2. Biometrics

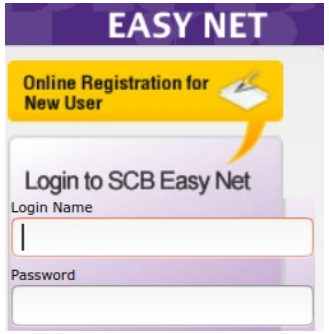
3. Trusted 3rd party - OAuth and OpenId
   "Login with Google" or "Login with Facebook"

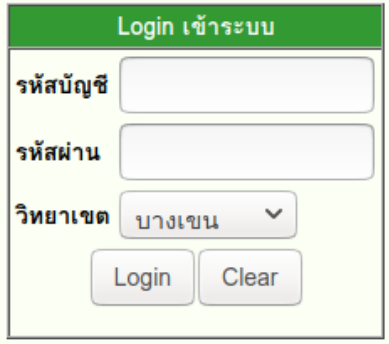
4. Public-private keys

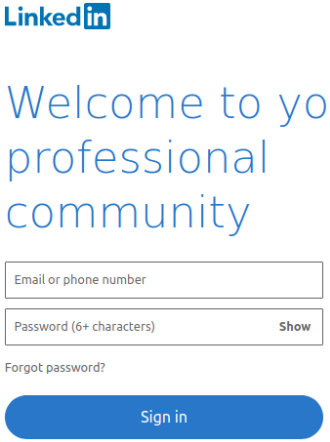
5. SQRL - a new method by Steve Gibson

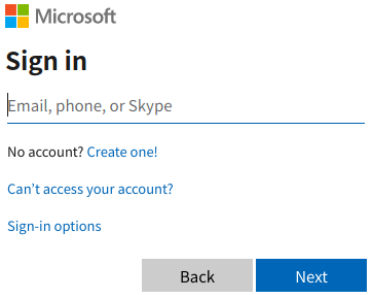
Username & Password

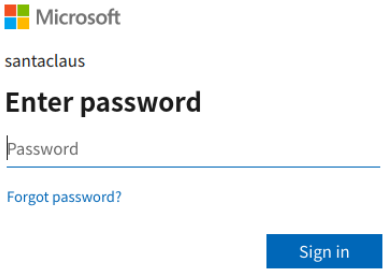
The oldest and one of the worst authentication methods.











Username & Password

The oldest computer-based auth method.

Not very secure.

- passwords can be stolen
- passwords can be guessed
- vulnerable to man-in-the-middle attack
- does not use the computational ability of user's device
  (it just sends a constant string)

Exercise: Have You Been Pwned?

Has your email address (and data) been stolen?

`https://haveibeenpwned.com/`

Has your password been seen in a breach?

`https://haveibeenpwned.com/Passwords`

OAuth & OpenID

Use a 3rd party to validate the user's identity

OAuth 2.0

OAuth is *really* about granting access to resources.

But, as a side effect, you confirm your identity.

Example:

Google:

"*Grant shopee.co.th access to your name and email?*"

- tells shopee who you are, and proves that you can authenticate yourself to Google.

OAuth Use Cases or "Flows"

Server-side web app: The web app gets a "secret" that it uses when requesting access to a user's resources.

Browser-based app: Javascript code running in web browser. Cannot keep a secret, so the flow is different.

Mobile app: uses the mobile browser as intermediary to grant OAuth access. Cannot keep a secret.

When you apply for <u>your app</u> to use OAuth on Google, etc, it is important to choose the correct "flow" or "grant type".

OAuth Exercise

1. Login to Google. What apps or web sites have you granted access to your Google resources?

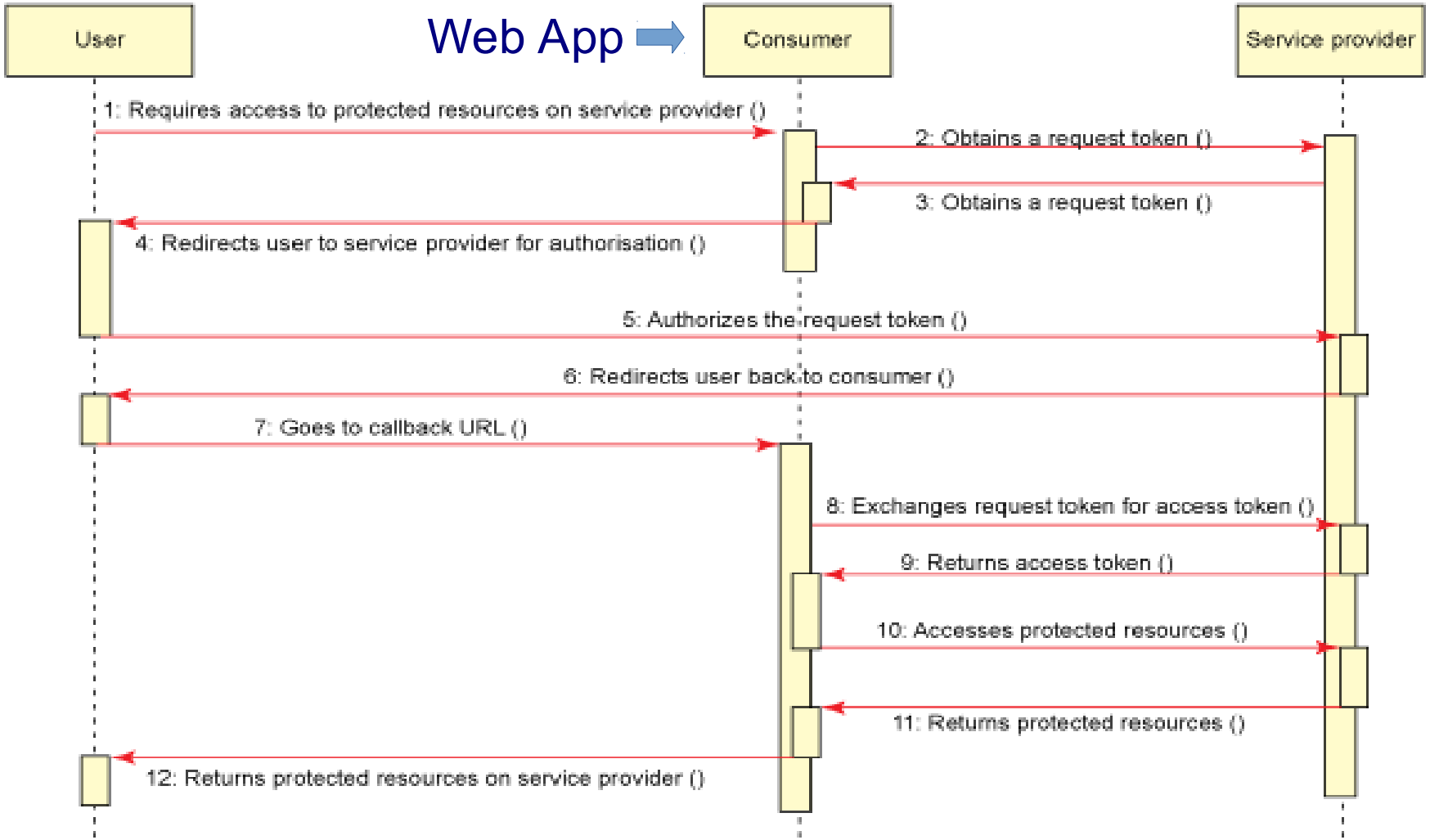
   Any that you don't use anymore? Delete them.

2. What apps have been granted access to your Facebook account?

Software developers need to have good cybersecurity habits.

This exercise is part of that.

OAuth Flow for Server-Side Web Apps



This is the OAuth 1.0 flow, some names and parameters are different in OAuth 2.0

Step 0: Register your app

Go to the OAuth provider and request OAuth access.

"Register an application" using "Authorization Code" flow

give the server:

app. name and URL,

a callback URL,

requested scopes

server gives you:

- client_id
- client_secret
- authorization URL - where you send the user's browser

Exercise: OAuth Playground

Go to `https://www.oauth.com/playground/`

Choose "Authorization Code" flow.

Work through the exercise.

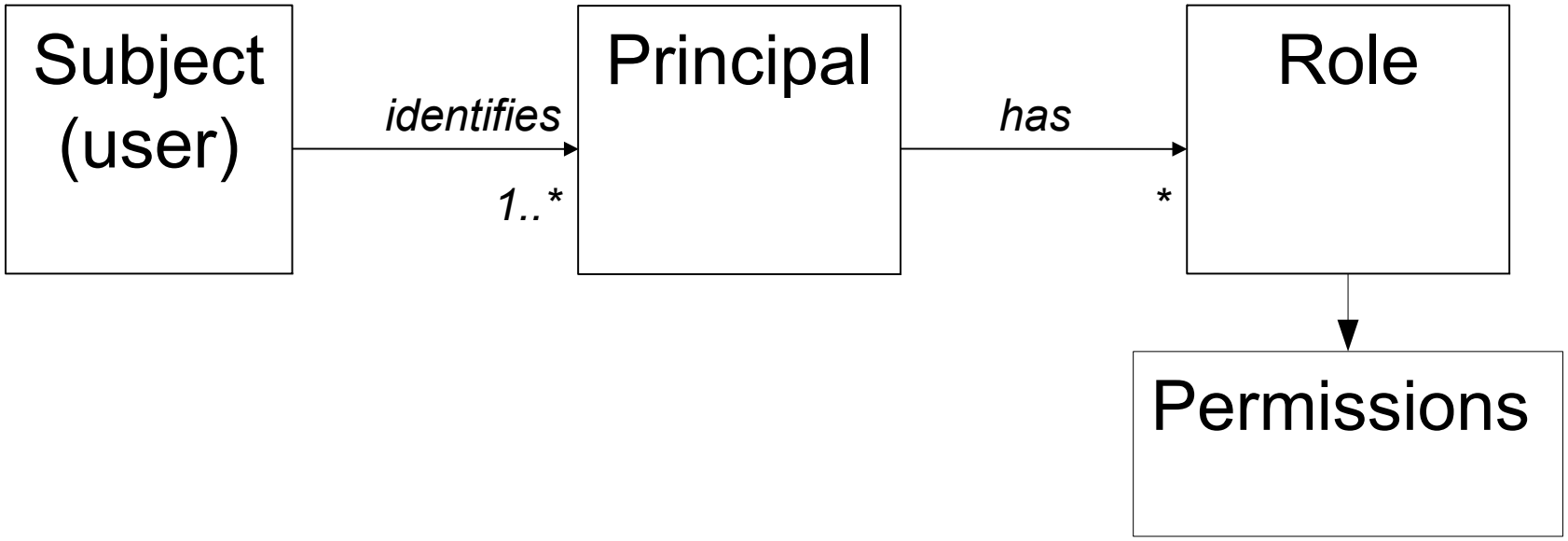
Note: you need to remember (or write down) the username and password the site gives you!

Role Based Authorization

Permissions are based on the *roles* a user possesses.

A user may have many roles.

Example: “joe” has roles “voter” and “administrator”

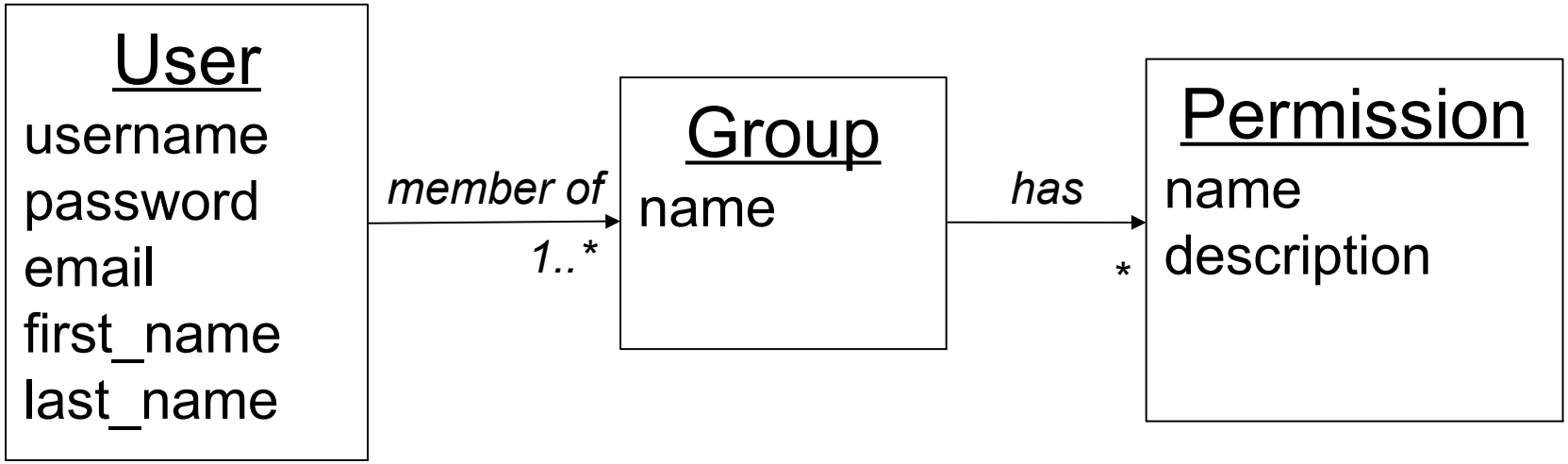


How Django Does It

User - identifies a user, authenticate by one of many *backends.*

Group - User is assigned one or more groups. Each group possesses some Permissions.

Permission - key-value pair (anything you like) used in code to enforce authorization



Checking Authorization in Code

```
from django.contrib.auth
            import authenticate, login
# django.contrib.auth has views to do this:
user = authenticate(request, "hacker", "Hack!")
login(request, user)

if user.is_authenticated:
    # allow any logged in user to do something

if user.has_perm('blog.can_post_comment'):
    # allow user to comment on blog
```

Checking Auth in Views

The request object has reference to current user.

```
def comment(request, blog_entry):
    if not request.user.is_authenticated:
        return redirect('login')

    if request.user.has_perm(
            'blog.can_post_comment')
        # add comment to blog entry
```

Use Decorators on Views

Decorators reduce risk of errors

```
from django.contrib.auth.decorators
   import login_required, permission_required

@login_required
def comment(request, blog_entry):
    """comment on a blog entry"""

@permission_required('blog.can_post')
def post_blog(request, blog_entry):
    """post a new blog entry"""
```

Define Your Own Decorators

If none of Django's decorators do what you want...

https://docs.djangoproject.com/en/2.2/topics/auth/default/

```
def kasetsart_email(user):
    return user.email.endswith('@ku.ac.th')

@user_passes_test( kasetsart_email )
def vote(request, question_id):
    # only users at KU can vote
```

Mixins for Class-based Views

"Mixin" means to combine or "mix in" behavior from several different classes.

```
from django.contrib.auth.mixins
          import LoginRequiredMixin

class PollsIndex(LoginRequiredMixin, ListView):
    template_name = 'polls/index.html'
    ...
```

Authorization in Templates

Templates can use the **`user`** and **`perms`** objects.

```
{% if user.is_authenticated %}
    Hello, {% user.username %}
{% else %}
    Please <a href="{% url 'login'%}">Login</a>
{% endif %}

{# same as user.has_perm('blog.post_entry') #}
{% if perms.blog.post_entry %}
   You can post a blog entry
{% endif %}
```

Where to Apply Authorization?

1. In templates. This gives desired appearance and page flow, but can be by-passed. Don't rely on it.

2. In views. Requests are always passed to a view, so this is fairly secure. Prefer decorators or Mixins instead of checks in code.

3. In models? In some frameworks, you can configure required permissions directly into model classes. Apparently not in Django.