Decentralised authorisation: OAuth2

COSC312 / COSC412

Learning objectives

- Describe the notion of security '**capabilities**'
- Describe the purpose of web technology for **distributed authorisation**
- Contrast between **OAuth2 and Kerberos** authorisation and authentication systems

OAuth2

- HTTP-based set of protocols to allow **resource owners** to **delegate access** to **their resources**
  - Has different interaction modes: *e.g.*, for browser / smartphone

- OAuth2 is a token-based authorisation system
  - Tokens are similar to Kerberos tickets
  - Both abstract a notion of a capability
  - To me, ‘token’ implies something opaque
    - We know that Kerberos tickets have many attributes

Defining security ‘capabilities’

- Abstract notion of an **access control matrix**:
  - ACLs list role permissions alongside each asset
  - Capabilities list permissions on assets for each user

	Asset 1	Asset 2	File	Device
Role 1	read, write, execute, own	execute	read	write
Role 2	read	read, write, execute, own		

- Permission to perform some action can be **decoupled** from identity
  - Also, have different timescales: **capabilities are short-lived** compared to the user’s privilege

Cryptography in capabilities?

- For token-based capabilities, knowledge of an 'opaque' token may be sufficient:
  - *e.g.*, token is indirectly passed to (OAuth) client through an intermediary authorisation service
  - Transport-level security required—token is **password equivalent**
- Alternatively, can encode data that only the target service can decrypt
  - thus the capability can be 'checked', as in Kerberos tickets

Delegation of capabilities

- Authorisation using capabilities **allows for delegation**
  - Transfer the capability to some other principal

- For example, using ‘add-on’ software:
  - You want it to access your resources, so it can help you
  - However, you don’t want this helper software to **be** you
  - Ideally: know which helper software did what, when
    - (…but our uses often don’t have this level of audit trail yet!)

Have 4 participants, compared to Kerberos

- Aim: you **delegate** privilege to an **independent service** to **access your data**
  - … so adds another principal compared to Kerberos:
  - that new principal is accessing target service, rather than you
- But compared to Kerberos also still have (approximately):
  - user agent, target service, and a security service
- … however in some cases above parties may combine
  - *e.g.*, service seeking access might be on the same device as the user agent

OAuth history

- OAuth 1.0 released in 2007
  - Twitter developers realised that OpenID was not going to support delegated API access well
  - OAuth then adopted into IETF: RFC 5849
  - 2009: OAuth 1.0a fixes a session fixation flaw (see next slide)
- OAuth 2.0 is current evolution [RFC6749,6750,8252]
  - Supported by Facebook, Twitter, Google, Microsoft, *etc.*
  - … however this committee effort has made it **complex**
  - Released in 2012 (… intended for 2010 release)

More on session fixation attacks

- An **attacker sets the session** of their victim
  - Attacker can then join that session
- Common web application workflow:
  - No active session? Authenticate user within new session
  - Authentication check and session check may be separate
  - Trick user into authenticating into **session ID set by attacker**
    - Attack vector such as server accepting URL containing session ID
- Not a cryptographic attack: authentication is skipped

CSRF: also a session-based problem

- Cross-Site Request Forgery (CSRF)
  - Another case of skipping cryptography

- Attacker embeds data on `a.org` that causes an HTTP request that targets `b.org` :
  - *e.g.*, an HTML image tag on a page, an HTML iframe, *etc.*

- If victim still has a valid session on `b.org` the target site may honour the attacker's request

History repeating ... literally

- A recurring COSC[34]12 theme of failure in cryptographic implementations:
  - Early OAuth 2.0 code often failed to use nonces

- OAuth 2.0 makes **compromises of convenience**
  - Requiring the 'state' parameter would limit some of the potential OAuth 2.0 use cases
  - (the 'state' parameter facilitates nonce checks)
  - Ideally systems would indicate their intended security level

Past OAuth controversy

- OAuth operates at the level of HTTP requests
  - *e.g.*, GET requests with parameters—URLs with sensitive data
  - … but browsers weren't designed to handle this
    - What sorts of vectors spring to mind?
  - Also, parameters aren't appropriately checked
    - (many layers of technology to worry about: URI encoding, *etc.*)

- ... however OAuth **is** in use, and has formal verification
  - (Something like it will be in demand always, in any case)

Roles in OAuth 2.0

- **Resource Owner**: the 'end-user' (or similar)
  - RO is granting access to part of their account
- **Client**: software trying to access RO's data
- **Resource Server**: where RO's data is stored
- **Authorization server**: (may also be the resource server)
  - Authenticates RO, obtains authorisation
  - Issues access tokens to client
- (RS / AS interaction not specified in OAuth 2)

Setting up OAuth 2.0

- OAuth 2.0 requires registration of the client application with the authorisation server
  - The means of registration are not specified
  - Registration is a one-time operation: no RO mentioned
- Registration of the application involves:
  - Specifying the client type
  - Providing redirection URIs (mandatory)
  - Other metadata required by authorisation server
    - *e.g.*, application name, logo, description, terms & conditions, *etc*.

Redirection URIs in OAuth 2.0

- Redirection URIs need to use TLS, *e.g.*, HTTPS
  - … because the **parameter values are sensitive**
  - (For development purposes, HTTP may be supported)

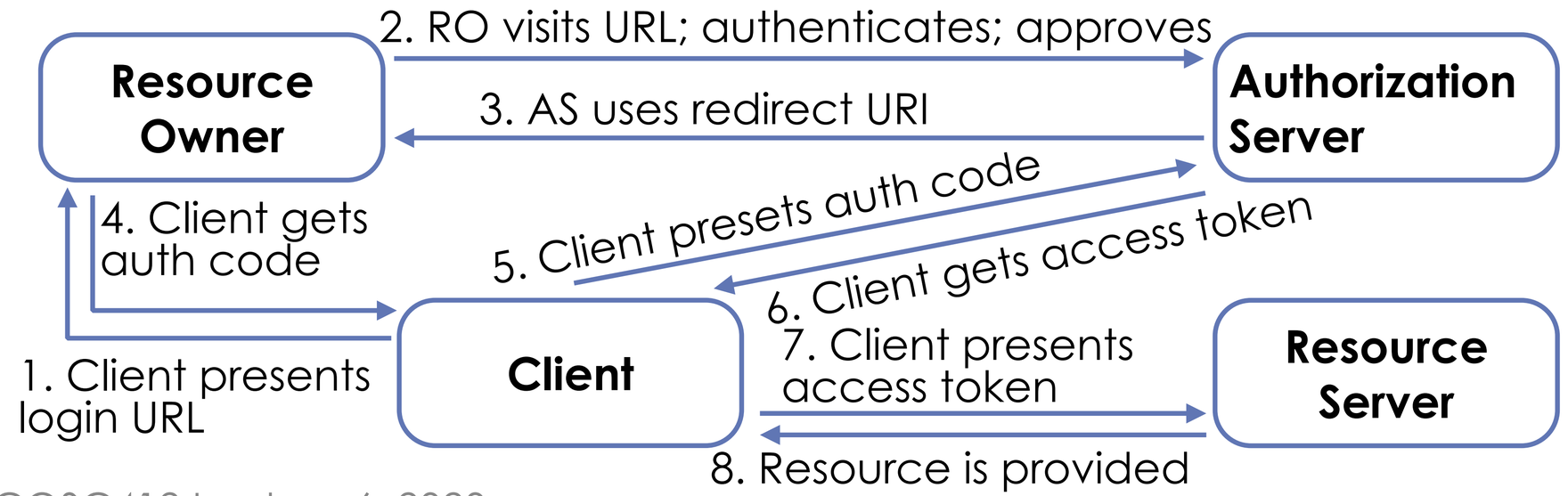
- The redirection URI is how focus returns to the client from the authorisation server: *e.g.*,
  - could be to a target web server
  - or to a ‘user-agent-based’ application (*e.g.*, local JavaScript)
  - or to some other ‘native’ application

Client's record of registration

- Authorisation Service provides client with two records of registration:
  - **Client ID** (length undefined in the specification)
  - **Client secret**
- Client ID is how the application is identified
- Two types of client: confidential and public
  - **Confidential** clients can keep secrets
  - **Public** clients can't keep secrets, *e.g.*, JavaScript in browser

OAuth 'authorisation code flow' steps

- Authorisation workflow is per access session
  - Client aims to get access to RO's data (*i.e.*, data stored on RS)
- Figure below is indicative of order of flow
  - (Some further steps may be needed in practice)



OAuth 2.0 grant types (1)

- We traced the authorisation code workflow
  - FYI: similar in pattern to decentralised authentication using protocols such as OpenID, Shibboleth, *etc*.
- OAuth 2 provides several 'grant types':
  - **Authorization code** for apps on a web server
    - **PKCE** is like 'authorization code', but without client secret
  - **Implicit** for browser-based/mobile apps… but should use PKCE
  - **RO Password Credentials** for gaining RO's login
  - **Client credentials** for application access

OAuth 2.0 grant types (2)

- For **authorisation code**, the AS is an intermediary between client and RO
  - RO's credentials never shared with client
  - Client's credentials never shared with RO
    - (*e.g.*, RO's web browser might leak access tokens)
- **Implicit** flow skips the authorisation code step
  - Token delivered straight to client
  - Client does not present a client secret
  - Suits JavaScript in-browser use cases

OAuth 2.0 grant types (3)

- **RO Password Credentials** grant type does what it says: the client gets the RO's username+password (!)
  - This requires a lot of trust in the client!
  - Does **not** represent controlled delegation
- May make sense for clients developed by the resource server's org., *e.g.*, the X app. accessing X
- Still creates tokens from the RO's password
  - So can be used as a transition plan, awaiting better security

OAuth 2.0 grant types (4)

- **Client credentials** grant type is when the client is not acting on behalf of an RO
  - *e.g.*, a helper application might retrieve a general set of data from the resource server
  - It would be unnecessary and inappropriate for general client requests to be linked to a particular RO (*i.e.*, user)

- Grant types are an evolution from OAuth 1.0
  - Handle a wider range of user agents

OAuth 2.0 token response

- Let's assume a request for an access token is valid
- Response adds JSON to HTTP 200 body:
  - `access_token`
  - `token_type` (`bearer` or `mac` currently)
- Optionally may add:
  - `expires_in` (lifetime of token in seconds)
  - `refresh_token` (think Kerberos 'renewable' tickets)
  - `scope` (client requests some scope; RS can restrict it)

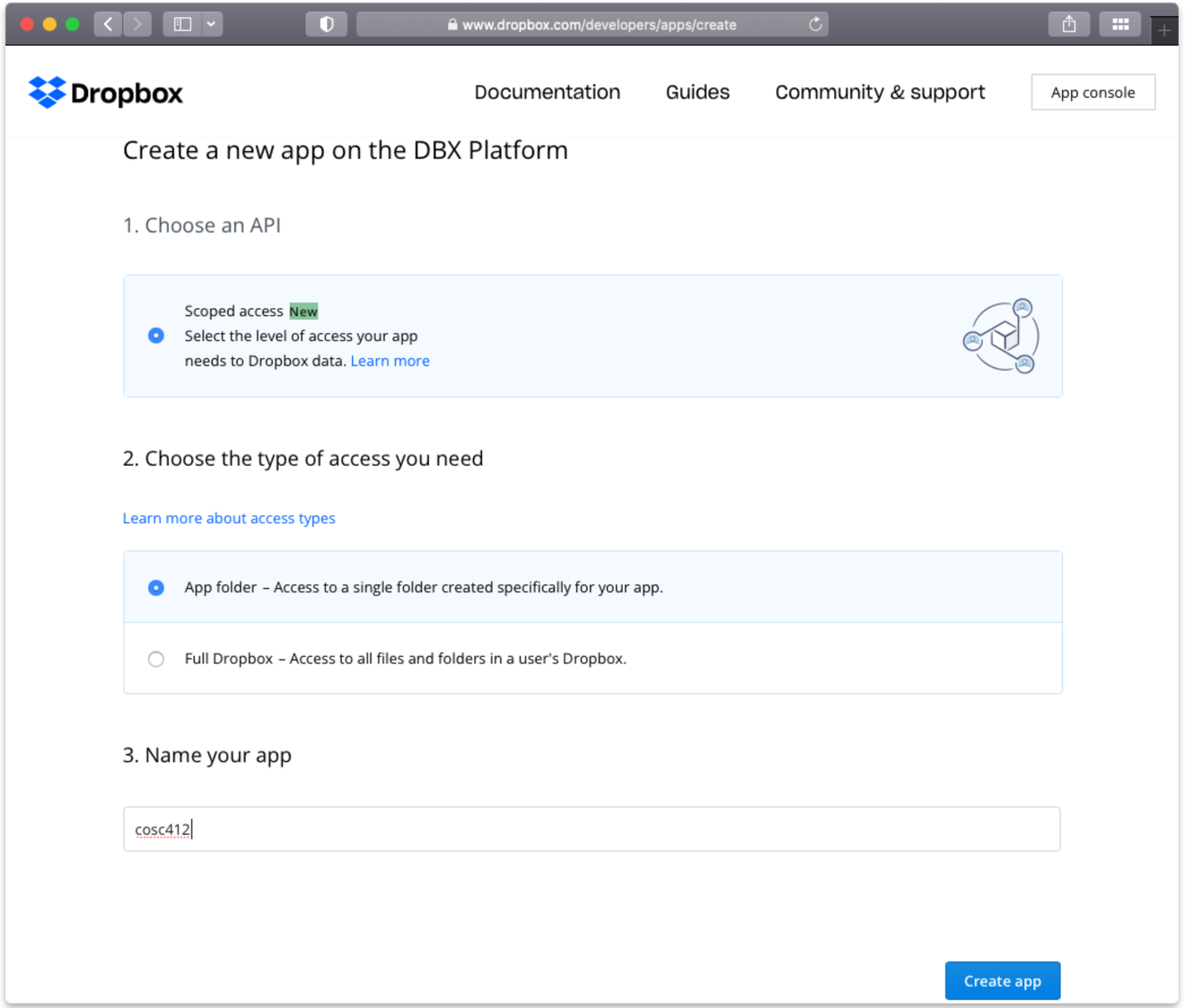
OAuth token types

- **Bearer** token type:
  - if you are bearing the token, you are authorised
- **MAC** token type:
  - Client demonstrates it has symmetric session key
  - Key is shared with resource server
- Client builds ‘authenticator’ of request fields
  - Uses session key to encrypt this data
  - Resource Server can check it

Let's see some OAuth 2.0 in practice

- Deploy a Dropbox 'App':
  - The Dropbox user is the resource owner
  - Dropbox is the RS and AS
  - Client is a PHP application running on our VM's web server
- Dropbox provides documentation and examples
  - Many languages are supported by Dropbox;
    - … and even more supported from the community
  - Demo app we use lists files within app folder on your Dropbox
    - (demo app is independent from Dropbox software clients, though)

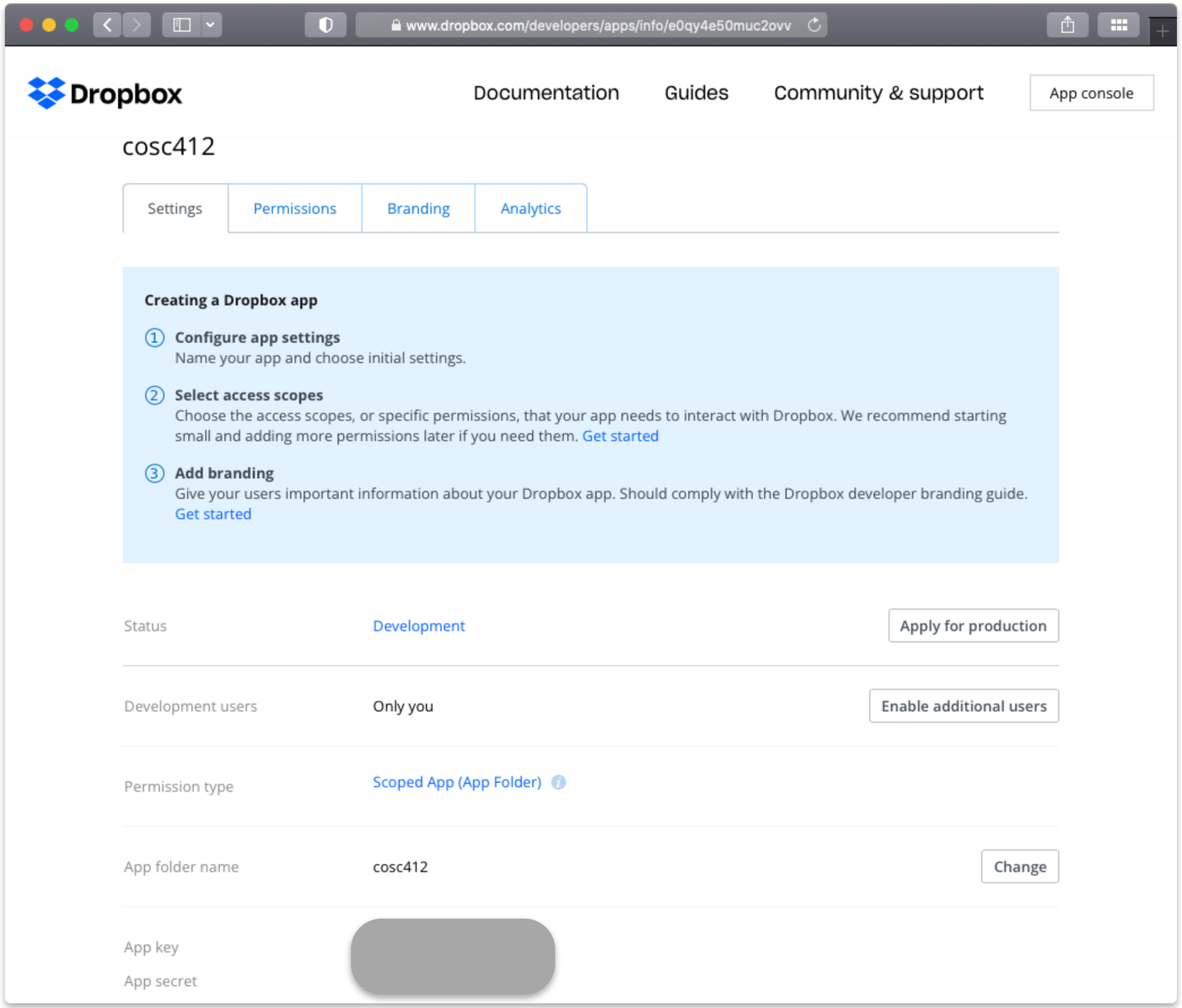
Register the application



Configure app.

- Set permissions:
  - files.content.read

- As expected:
  - App key
  - App secret
  - App name, etc.
  - Redirect URI:



http://localhost:8080/DropPHP/samples/simple.php?auth_redirect=1

Set up local application state

- Web pages served through Apache web server in this demo are using the 'authorization code' flow

- Set up the OAuth2 demo:

```
: ~$; /vagrant/setup-apache.sh
: ~$; /vagrant/setup-oauth2.sh
```

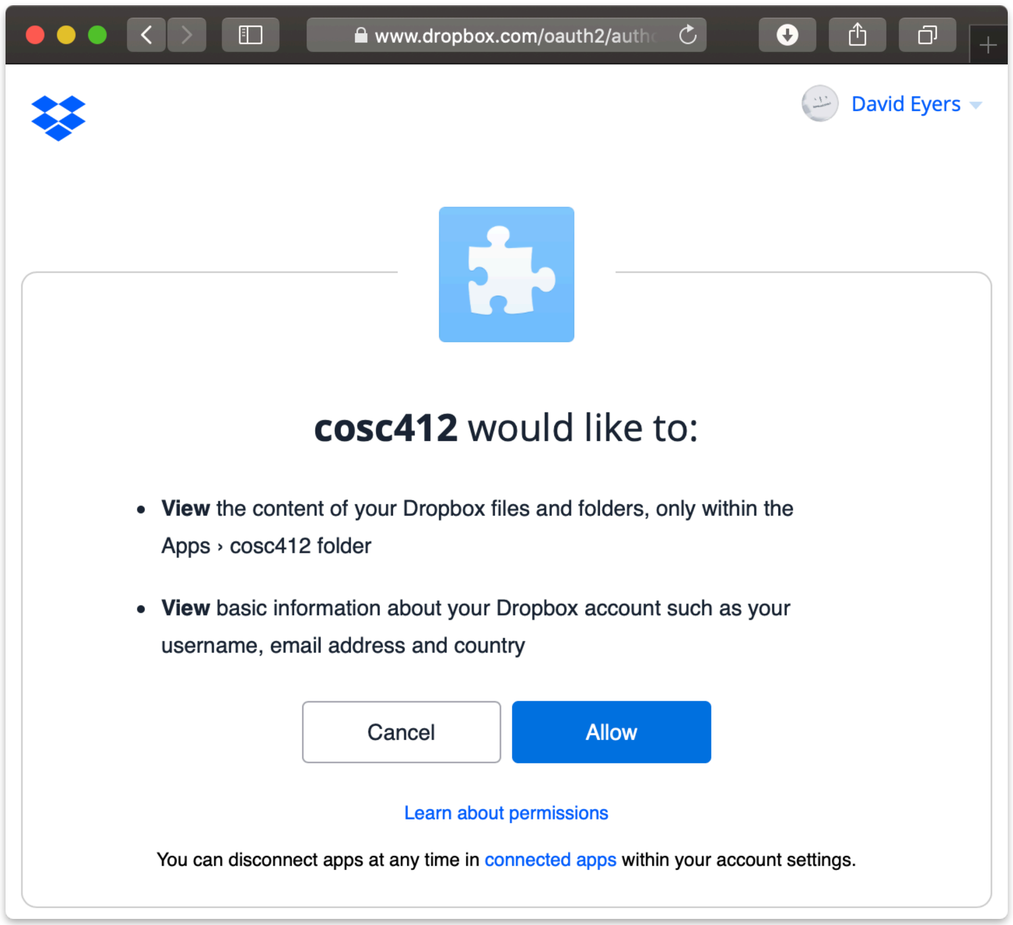
  - Configure App key in the PHP file within the VM
    - This is line 26 and 27 of the file mentioned below, for me
    - You replace the app_key + app_secret string with your app's value

```
: ~$; nano /vagrant/www/DropPHP/samples/simple.php
```

    - You can run the network monitoring commands shown in previous lectures within the VM if you want to see what exchanges occur

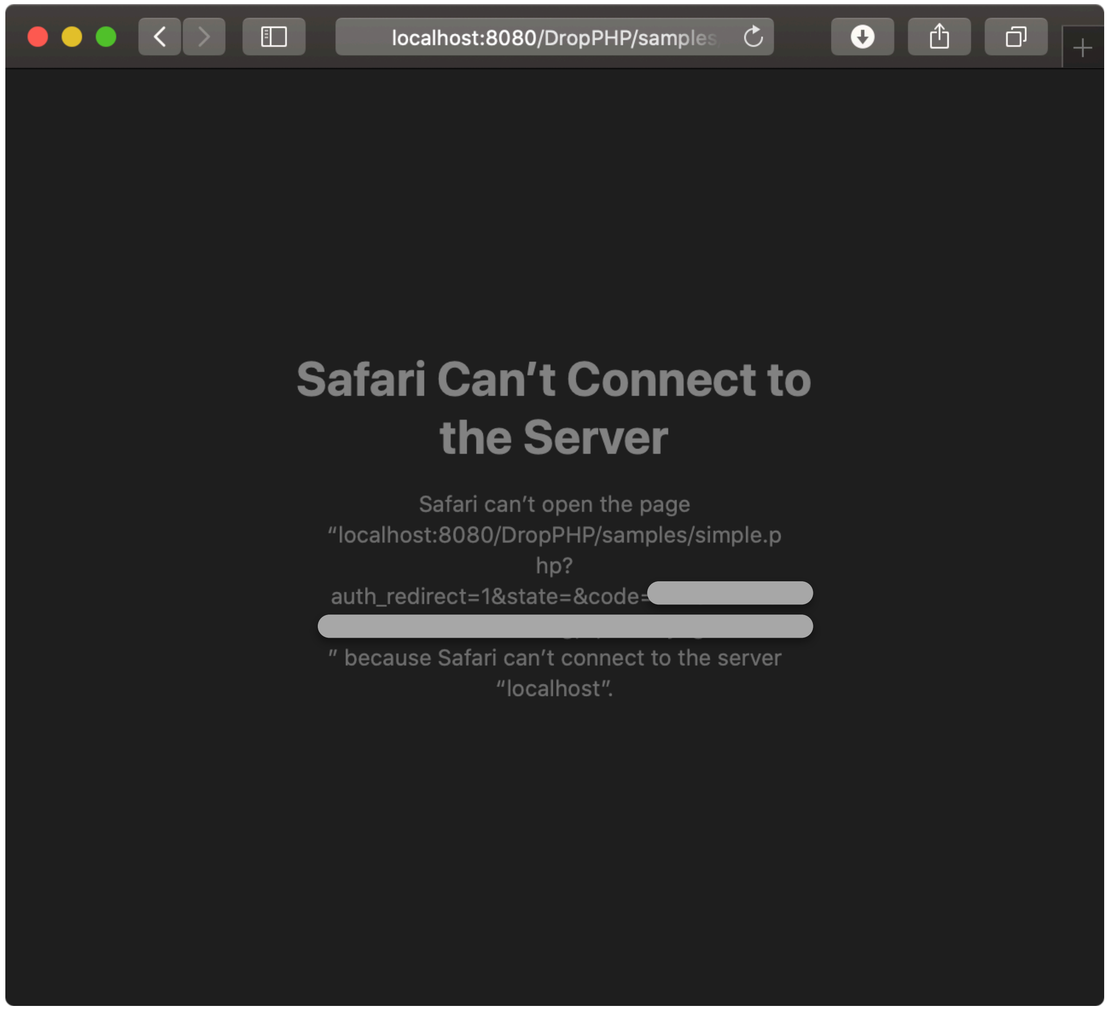
Now access our local client app

- Local client lists files within a Dropbox app folder
  - http://localhost:8180/DropPHP/samples/simple.php
- "Authentication required." is stated by PHP script with continue link to click…
- On the first visit, Dropbox checks with me (I'm the RO) whether or not to authorise this client (our PHP script)



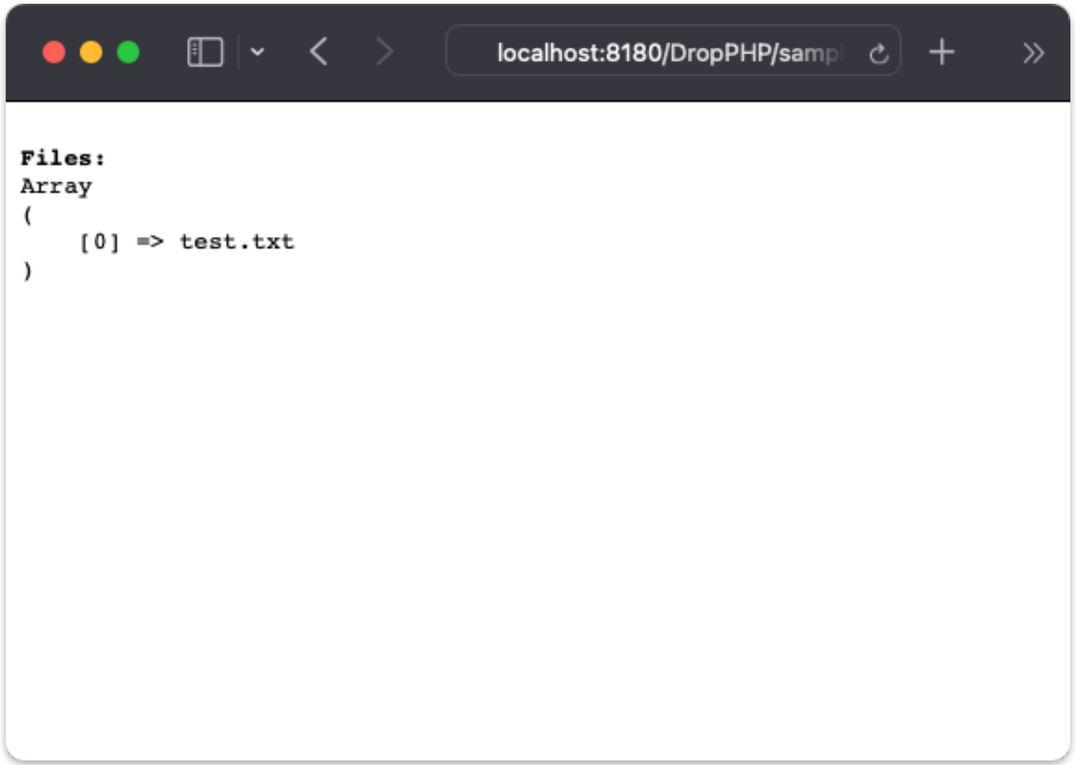
The redirect URL is intentionally wrong…

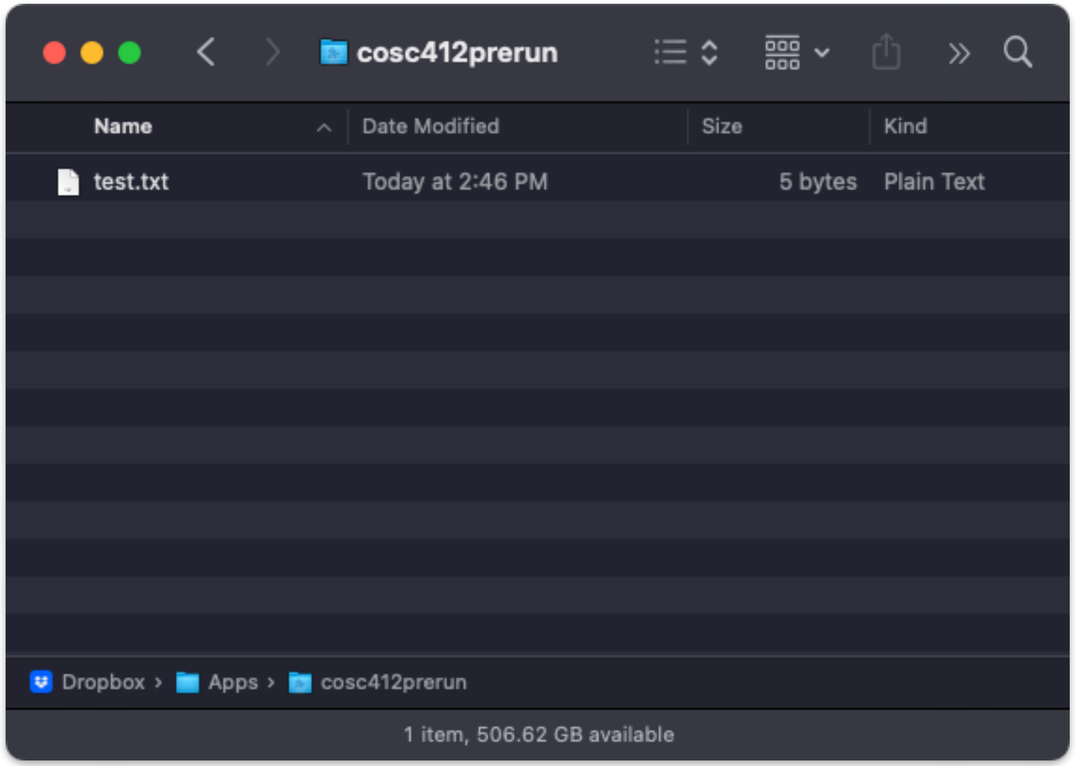
- Normally this step would proceed without any explicit status reporting
- Intentionally give the wrong port number so browser shows URL to you…
- Change 8080 to 8180 to pass the token back to the app
  - (You can fix Dropbox app too)



Delegated authorisation complete

- Application is accessing files on my Dropbox
  - Reloading will show the PHP script stored a bearer token





In summary

- Distributed authorisation allows **controlled data sharing**
  - Useful for orchestrating interacting services
- **OAuth 2.0** is a leading standard for HTTP(S)-based distributed authorisation
  - However it raises some security concerns
- Its **focus on authorisation** makes OAuth 2.0 a good point of contrast to Kerberos, and web authentication