



Authentication and Authorisation

Learning Objectives

- ▶ On completion of this section, you will be able to:
- ▶ Explain the difference between authentication and authorisation - providing use cases
- ▶ Explain the construct and use-case of a Json Web Token (JWT)
- ▶ Explain the difference between base64 and base64url
- ▶ Provide some key attributes of a JWT when used in authentication and authorisation
- ▶ Explain at a detailed level, step by step:
 - ▶ Authorisation code flow in detail to illustrate , authorisation and authentication using OIDC

First Party Authentication Approach

- ▶ Create an account
 - ▶ User enters a username and password into an html form to register an account
 - ▶ This is posted to the server using https POST
- ▶ Add the hashed password to a database (maybe salt and pepper)
- ▶ The users roles and access authorisation are added to the database if required
- ▶ User logs in by sending credentials, server creates a hash of the password, searches for user by username, hashes the password and checks it against the stored hash
- ▶ If ok, the server creates a session object with a random session ID, user ID (primary key), roles/permissions and an expiry date and time. This is stored in memory or a memory-based database like Redis
- ▶ Server creates a session cookie to avoid re-authentication during the session
 - ▶ Typical opaque session cookie: `connect.sid=s:<sessionId>.<signature>;`
 - ▶ `sessionId`=random number. HMAC of the ID is created and added to the string. HMAC checked on return for tampering
 - ▶ Opaque meaning the content is of no value to anyone as no user data is stored in it
 - ▶ Cookie is returned to client. HTTP headers are set: `HttpOnly; Secure; SameSite=Lax`
 - ▶ `HttpOnly` means protect from JavaScript. JavaScript cannot access the cookie on the browser
 - ▶ `Secure` means only send cookie from client using https
 - ▶ `SameSite=Lax` means allow get and redirect cross-site but not POST

- ▶ When something is clicked on the page and a request is made to same domain, cookie is sent automatically
- ▶ Server checks the cookie signature (HMAC) and if ok, retrieves the session object and checks for expiry
- ▶ If the session has expired (or has been auto deleted), you may want to extend it or redirect to a login page
- ▶ If not expired, the application code can check the access or role before allowing access to any protected resource
 - ▶ e.g. access to a specific API endpoint such as writing to the database or reading specific data
- ▶ **Benefits**
 - ▶ Nice and simple, full control, no external dependencies
- ▶ **Challenges**
 - ▶ Need to manage password resets, email verification, MFA, all aspects of credential security, account blocking, audit, responsible for any data breach regarding user credentials even if using salted hash as they may be brute forced as the salt is known. Pepper can help to prevent this as it is no known and not in the database
- ▶ When there are only a few hundred users, this approach is probably ok. However, when scaled to thousands or more, it's a big deal and best left to the security experts so offload this responsibility to a third party
- ▶ Third party authentication and authorisation is very popular with web applications

JSON Web Token

- ▶ Security Tokens - used by OAuth and OIDC
 - ▶ JSON as opposed to SAML XML tokens so simpler - more lightweight
- ▶ JWT is an open standard
 - ▶ (RFC 7519) to securely exchange information using JSON objects
 - ▶ It's an encoded string, which is URL safe (base64url) so no + / = issue with base 64
- ▶ Token is cryptographically signed, similar to SAML
 - ▶ Data is not encrypted so use TLS if you need protection
 - ▶ It can be signed using symmetric HMAC or asymmetric RSA or ECDSA (Elliptical curve Digital Signature Algorithm)
 - ▶ The digital signature algorithm is fed all the base64-url data to calculate the signature
- ▶ JWT has 3 parts:
 - ▶ header - crypto algorithm used
 - ▶ payload - plain data. It contains no credentials
 - ▶ signature - digital signature. Generally using an asymmetric key method but symmetric is available
 - ▶ each is base64url encoded separated by a '.' (full stop)

Example

- ▶ Header example:

```
{  "alg": "HS256", (HMAC using SHA256)  "type": "JWT"}
```
- ▶ Payload example – this is where the **claims** about an entity (typically a user) are made

```
{  "sub": "1234567890",  "name": "John Doe",  "iat": 1516239022}
```
- ▶ The final part is the signature which is also base64url encoded
- ▶ Above example JWT would look like this:

- ▶ eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
- ▶ .
- ▶ eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ
- ▶ .
- ▶ SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJnaXZlb19uYW11IjoieVVG9ueSIsImZhbWlseV9uYW11IjoieTmljb2wiLCJuaWNrbmFtZSI6InRvbnkiLCJuYW11IjoieVVG9ueSBOaWNvbCIiInBpY3R1cmUiOiJodHRwczovL2xoMy5nb29nbGV1c2VyY29udGVudC5jb20vYS9BQ2c4b2NKOjcwZ041RG5Xd0JHTlJzSzY0Z2ptNTU0dmE5UjdrSWpJVHVKQ09zS2pT

akeE9czk2LWMI6InRvbnluc2NhbGUiOiJlbi1HQiIsInVwZGF0ZWRfYXQiOiIyMDI0LTAYLTE0VDIwOjMxOjI3LjQ2MVoilCJlbWFpbCI6InRvbnluaWNvbEBnbWFpbC5jb20iLCJlbWFpbF92ZXJpZm1lZCI6dHJ1ZSwiaXNzIjoiaHR0cHM6Ly90b255bmljb2wudWsuYXV0aDAuY29tLyIsImF1ZCI6Ikv5bkd3T1ROZGhIcEFXYkxzWDN3Y0lBSmUwMlNxaDYzIiwiaWF0IjoxNzA3OTQyNjg3LCJleHAiOjE3MDc5Nzg2ODcsInN1YiI6Imdvb2dsZS1vYXV0aDJ8MTAxNDQ1NzEzOTA0MjkmZmZzNzYxIiwic2lkIjoieUlJQbG9ZRWt3VHg0aUZPXzg0bGNWazVWIiwibm9uY2UiOiJaWUZhOXd4WkJqR3Fuc1o4Z1BOemZmb1NMBUNDQVFOUzdTc2RmeXZNa2NvIn0.X8mJaQpJ50tEBq7kbE1CHHizdxJ1r4MsKqtFDgNtVVg

JWT v JWS v JWE

- ▶ A confusing number of J things
- ▶ JWT is effectively an umbrella term for a base64 encoded token
- ▶ JWT is either a JWS or a JWE
- ▶ JWS is a JSON Web Signed token as described previously. Signed JWT but not encrypted
 - ▶ Provides integrity, authenticity and non-repudiation
 - ▶ Has 3 parts as shown previously - each base64 URL encoded
- ▶ JWE is a JSON Web Encryption token. JWT Payload is encrypted not signed
 - ▶ Provides confidentiality
 - ▶ Typical use case is application session tokens so they can only be read by the server on decryption
 - ▶ Has 5 parts, each base64 url encoded
 - ▶ Header: alg that protects the encryption key, enc - encryption algorithm (e.g. A256GCM")
 - ▶ encrypted key: if alg=dir, no key (default). Could be shared secret - encrypted with recipient pub key.
 - ▶ iv: initialisation vector. A random number used by the encryption algorithm
 - ▶ ciphertext: the encrypted payload
 - ▶ tag: value created by the encryption algorithm - like an HMAC value

Claims list

- ▶ nonce: Number used once. Random number to avoid replay attack. Server can check it hasn't seen it before
- ▶ alg: Signature algorithm. e.g., HS256=HMAC + SHA256, RS256=RSA + SHA256, ES256=Elliptic curve + SHA256
- ▶ x5t: X509 certificate thumbprint. Hash of the issuer's / signer's public key. Used to uniquely identify the public key in a compact form
- ▶ kid: Key ID. Unique identifier for the crypto key used to sign the JWT. i.e. used to find the key in a key vault to use to verify the signature
- ▶ aud: Audience. Typically, the name of the resource server or services the JWT is intended for
- ▶ iss: Issuer. Typically, identifier or url of issuer.
- ▶ iat: Issued at. Num seconds after Unix epoch (Jan 1st, 1970, UTC). i.e. data and time of issue
- ▶ nbf: Not before. It can be issued before it can be used
- ▶ exp: When it expires
- ▶ acct: User account or profile
- ▶ acr: Authentication Context-class Reference: e.g. low is UN & PW. medium may mean 2fa. Level of auth strength of the process used to create the token
- ▶ aio: Azure AD object Id. Unique identifier of the user within the Azure AD environment
- ▶ amr: Authentication methods reference. Authentication methods used during authentication. e.g. pwd, 2fa,otp
- ▶ appid: UUID of the client that requested the token. 128 bits in 8-4-4-4-12 hex digits format
- ▶ appidacr: Strength of authentication of the application authentication. Similar to acr but for the app
- ▶ auth_time: time the user was authenticated
- ▶ family and given name need no explanation
- ▶ ipaddr: IP address from where the authentication request came
- ▶ name.: Name to display - e.g. on a web page. Often given and family name
- ▶ oid: object ID. UUID of the user object in Azure active directory to identify the user in the directory
- ▶ onprem_sid: User active directory ID in situations where on-prem AD is synchronised with Azure AD
- ▶ puid: persistent user ID. Single ID used across multiple accounts
- ▶ rh: Refresh history. How many times has this been refreshed. Usually at least a number. But this is Microsoft - they don't always follow standards
- ▶ scp: Scope. Specifies the permissions granted and what actions the app can perform on behalf of the user
- ▶ signin-state: sign in status. Values vary on ID provider. e.g. pending, success, failure kmsi means keep me signed in. Long session or refresh token
- ▶ sub: Subject. UUID of the subject, typically the user or whatever the token signifies within the scope of the issuer. In this case, Azure AD
- ▶ tid: Tenant ID. Azure tenancy the user belongs to.
- ▶ unique_name: Could be unique username, email address or something else
- ▶ upn: User principal name. Basically, the unique user again.
- ▶ uti: Token ID. UUID of the token itself
- ▶ ver: Version of the token format / standard

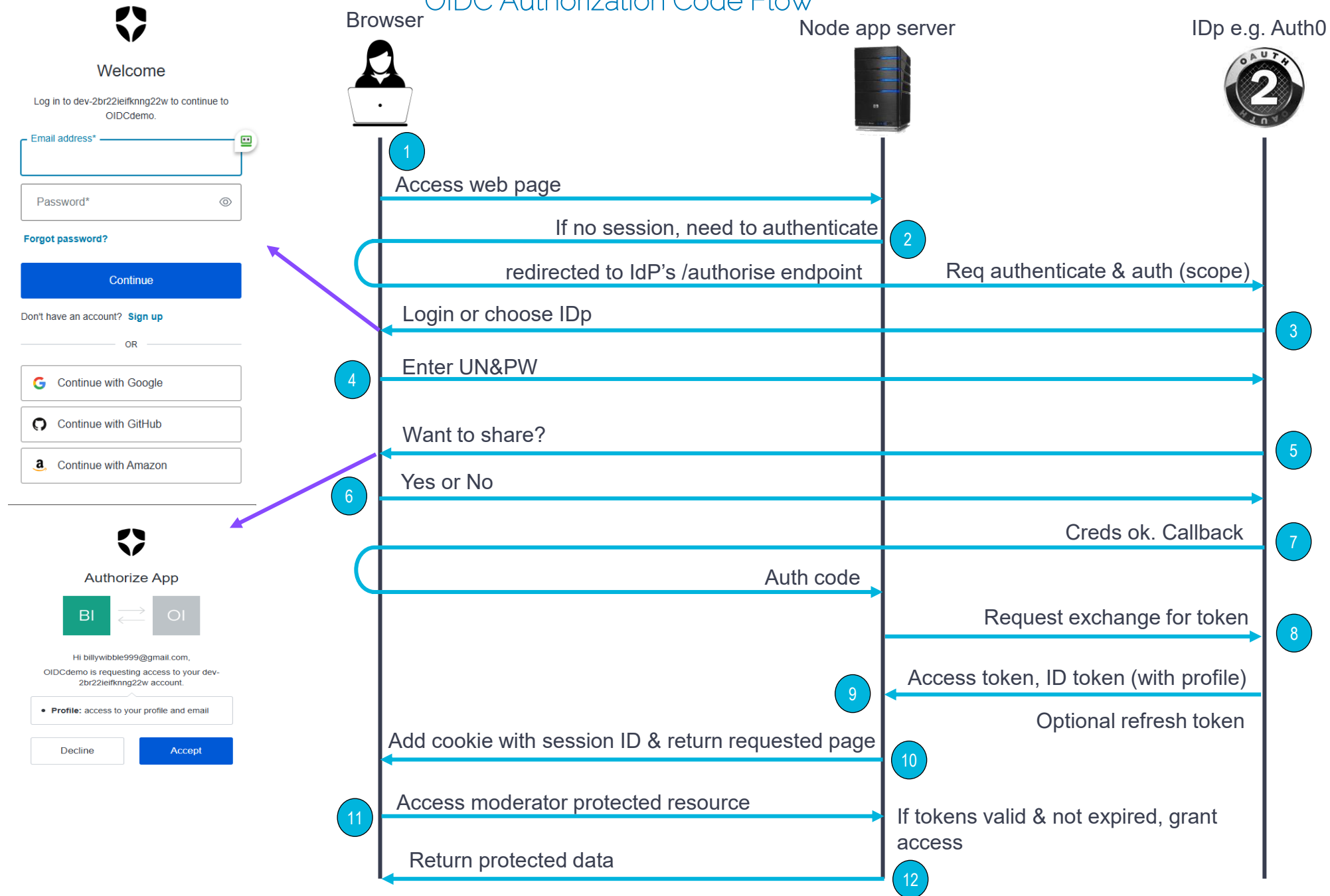
Authentication and Authorisation

- ▶ Authentication is "am I sure you are who you say you are"
- ▶ Authorisation is "once authenticated what are you allowed access to"
- ▶ The most modern approach to authorisation is Open Authorization (Oauth), now at version 2 so OAuth2
 - ▶ Open standard for secure authorisation to protected resource
 - ▶ Used JWT to grant access
 - ▶ There are various flows (process), we will focus on the popular **Authorisation Code Flow**
 - ▶ Oauth only creates an authorisation token and optionally a refresh token - NOT an authentication token
- ▶ For authentication, Auth0 does require authentication but doesn't care how its done but as the process does ask for authentication, only minor additions are required to perform authentication as well
- ▶ OpenIdConnect (OIDC) is a layer on top of OAuth enabling authentication
- ▶ As OIDC is an enhancement to OAuth, just using OIDC means you are also using Oauth so OICD can be used for authentication and authorisation
- ▶ I'll talk through the Authorisation code flow and explain how we authenticate as well

Prerequisites to Delegated Authentication & Authorisation

- ▶ An application developer creates an account with an Identity Provider (IdP) such as Auth0 (an Okta company) and registers their application to use OAuth2/OIDC flows.
 - ▶ Others are available such as Microsoft, Google etc. Same sort of approach
- ▶ Register your app and provide some details such as:
 - ▶ provide callback URL: where you want the auth server to redirect after authentication
 - ▶ maybe provide Logout URL: e.g., goodbye page
 - ▶ You will be provided with a client ID and secret specific to your app
 - ▶ You will be given a URL to direct your server to, to invoke authentication at the IdP
 - ▶ Choose which additional IdP services you want to provide as options e.g. Microsoft, Google, Github ...
- ▶ Once the app is registered and operational, you can use it to create an identity token which represents an authenticated user, and an access token which contains user roles and / or access permissions

OIDC Authorization Code Flow



OIDC Authorisation Code Flow

- ▶ 1. User requests a protected page so app requires user authentication and maybe profile information (e.g., name, email, photo) to personalise the page
- ▶ 2. The server checks for an existing application session. If none exists, it redirects the user to the Authorisation Server (IdP) using HTTP 302 and a Location header pointing to the IdP's `/authorize` endpoint
- ▶ The redirect includes: - `response_type=code`; `client_id` (issued when the app was registered); `redirect_uri` (where the IdP should send the user back); `scope` (permissions requested; include “openid” if identity is required) and a couple of additional security values
- ▶ 3. The Authorisation Server displays its login page. The user chooses how to authenticate (username/password, social login, enterprise login, etc.)
- ▶ 4. The user submits credentials. The IdP verifies them, loads the user's profile, and checks whether the user has previously granted consent for the requested scopes
- ▶ 5. If consent has not been granted before, the IdP displays a consent page asking the user to allow the application to access the requested data
- ▶ 6. If the user approves, the IdP records the consent
- ▶ 7. IdP redirects the browser back to the application's `redirect_uri` with an authorisation code. If the user denied sharing, the IdP redirects back with an error
- ▶ 8. The application backend exchanges the authorisation code for tokens by calling the IdP's token endpoint. It sends the code, `client_id`, `client_secret`, `redirect_uri`. The `redirect_uri` enables the IdP to check confirm the created token contains the same URI as that redirected via the browser
- ▶ 9. The IdP returns tokens: - access token (OAuth2); ID token (OIDC, only if “openid” was requested); refresh token (optional, depending on IdP settings)
- ▶ 10. The application validates the tokens (signature, issuer, audience, expiry) and extracts identity information from the ID token. It creates an application session and stores the tokens securely (e.g., in a server-side session in memory, Redis or whatever)
- ▶ 11. The application returns the protected page to the user. The browser stores the session cookie automatically
- ▶ 12. When the user later accesses another protected resource, the browser sends the session cookie. The application checks token validity (e.g. expiry) and authorisation (roles/permissions) and returns the protected data or an error
- ▶ If tokens have expired, either: the user is redirected back to the IdP to authenticate again, or, if a refresh token is available, the application uses it to request new tokens without user interaction

Summary

- ▶ Authentication: are you who you say you are
- ▶ Authorisation: when we know you are who you say you are, what are you authorised to access
- ▶ OAuth, now at version 2 is an **authorisation** protocol
- ▶ Although OAuth2 flow does include authentication, but it doesn't retain any identification data. I.e., using OAuth, we don't know who authenticated or when
- ▶ OpenID connect extends the authorisation capability of OAuth2 to also enable **Authentication**
- ▶ Active Directory works within one domain to hold user credentials, resource access and SSO
- ▶ Active Directory Federation Service enables AD to be accessed across domains using SAML
- ▶ SAML is an XML file used to securely exchange information between an IdP and an SP across domains for authentication - typically using ADFS / AD
- ▶ OAuth2 is used for delegated authorisation - can be used with SAML and JWT
- ▶ JWT is a JSON web token used to securely exchange data. Used typically in place of SAML as it's lightweight based on simple JSON structure rather than a complex and verbose XML structure