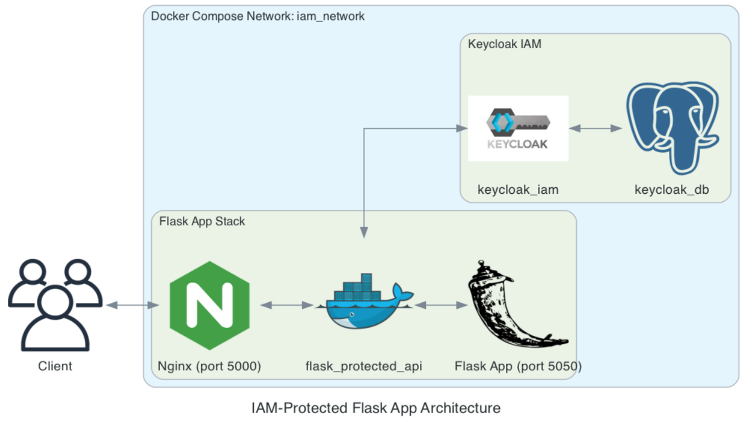
Summary Report – Craig Troop – 23 May 2025

1. Architecture



Architecture Diagram

Key Components

Component	Location	Function
nginx	Flask Container	Reverse Proxy to protect Flask App
flask_protected_api	Flask Container	Exposes protected API endpoints
Flask App	Flask Container	Implements business logic, handles OAuth 2.0 and OIDC login flow, secures API routes with token-based access contol
KeyCloak	KeyCloak Container	IAM provider for user authentication, token issuance, and RBAC
KeyCloak DB	Postgres Container	Stores KeyCloak configuration, realm, client, user, and session data

2. Testing Information

The command **`make reset`** will stop all containers, delete the keycloak configuration directory, prune the containers, and run the setup.sh bash script.

Endpoints for testing

URL	Purpose	Credentials
http://localhost:5000	Landing Page for Flask App	N/A
http://localhost:5000/login	Login Workflow	U: testuser P: testpassword
http://localhost:8080	KeyCloak Administration	U: admin P: admin

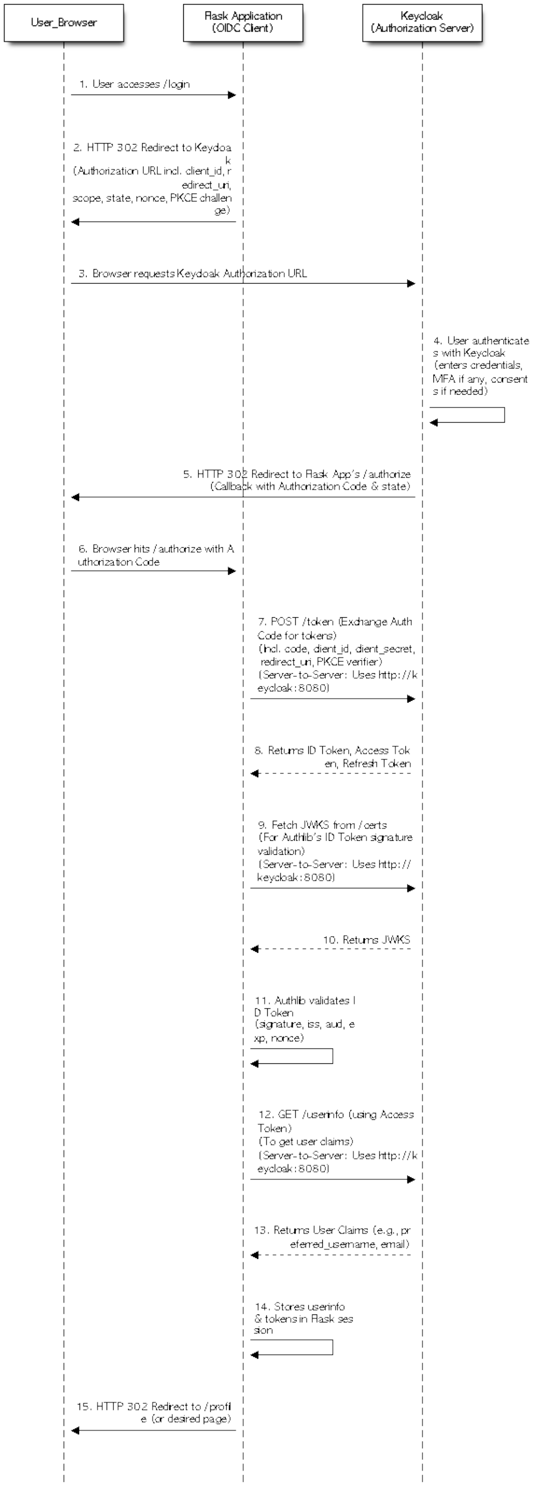
KeyCloak Configuration

Realm: hw8
Client: flask-api-client
User: testuser

Realm Roles: app_admin, app_user
Client Roles: task_reader, task_writer

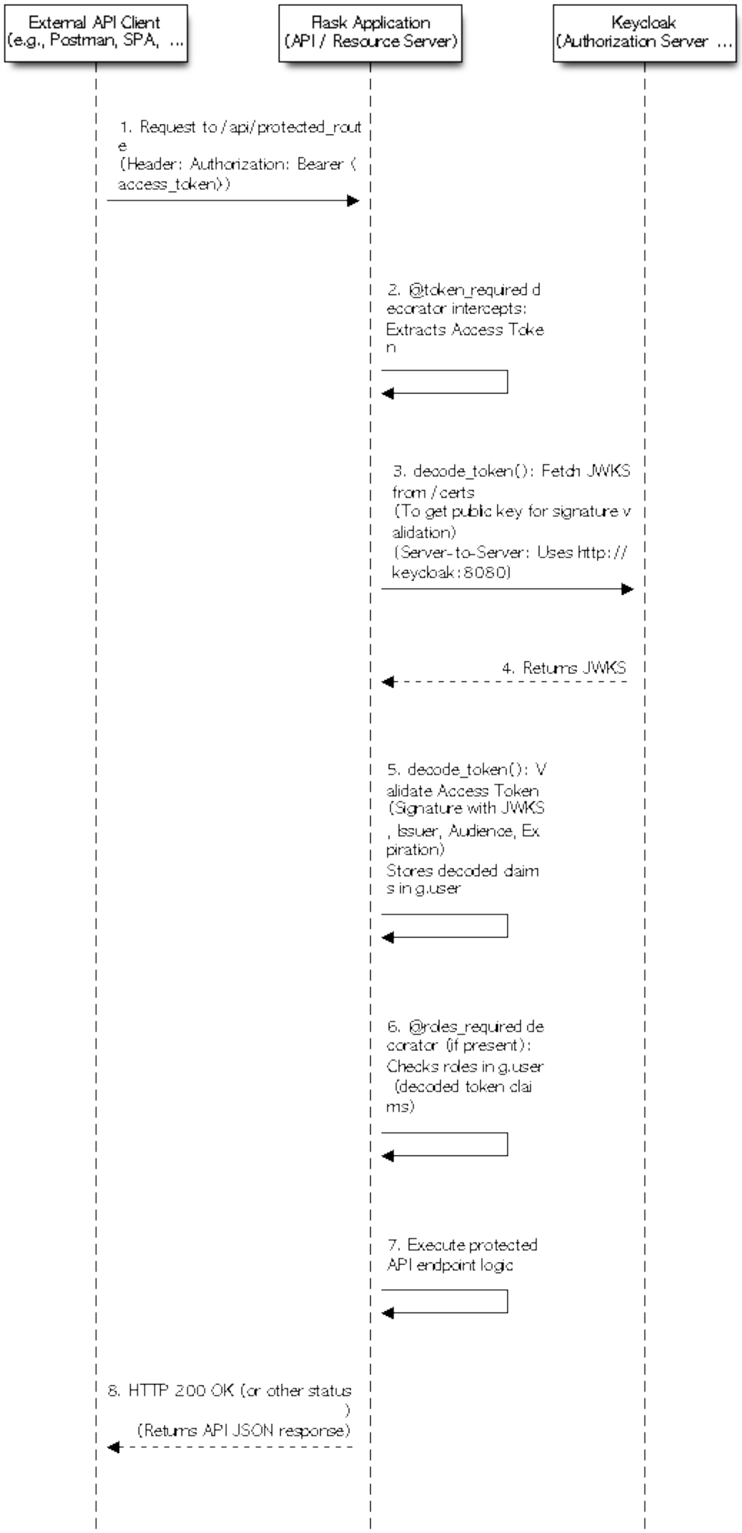
3. OAuth 2.0 and OIDC Flows

User Login Workflow


User_Browser
Flask Application (OIDC Client)
Keycloak (Authorization Server)
1. User accesses /login
2. HTTP 302 Redirect to Keycloak (Authorization URL incl. client_id, redirect_uri, scope, state, nonce, PKCE challenge)
3. Browser requests Keycloak Authorization URL
4. User authenticates with Keycloak (enters credentials, MFA if any, consents if needed)
5. HTTP 302 Redirect to Flask App's /authorize (Callback with Authorization Code & state)
6. Browser hits /authorize with Authorization Code
7. POST /token (Exchange Auth Code for tokens) (Incl. code, client_id, client_secret, redirect_uri, PKCE verifier) (Server-to-Server: Uses http://keycloak:8080)
8. Returns ID Token, Access Token, Refresh Token
9. Fetch JWKS from /certs (For Authlib's ID Token signature validation) (Server-to-Server: Uses http://keycloak:8080)
10. Returns JWKS
11. Authlib validates ID Token (signature, iss, aud, exp, nonce)
12. GET /userinfo (using Access Token) (To get user claims) (Server-to-Server: Uses http://keycloak:8080)
13. Returns User Claims (e.g., preferred_username, email)
14. Stores userinfo & tokens in Flask session
15. HTTP 302 Redirect to /profile (or desired page)

User Login Workflow Diagram

Protected API Access Workflow


External API Client (e.g., Postman, SPA, ...
Flask Application (API / Resource Server)
Keycloak (Authorization Server ...
1. Request to /api/protected_route (Header: Authorization: Bearer (access_token))
2. @token_required decorator intercepts: Extracts Access Token
3. decode_token(): Fetch JWKS from /certs (To get public key for signature validation) (Server-to-Server: Uses http://keycloak:8080)
4. Returns JWKS
5. decode_token(): Validate Access Token (Signature with JWKS, Issuer, Audience, Expiration) Stores decoded claims in g.user
6. @roles_required decorator (if present): Checks roles in g.user (decoded token claims)
7. Execute protected API endpoint logic
8. HTTP 200 OK (or other status) (Returns API JSON response)

Protected API Access Workflow Diagram

4. Security Analysis

Threat Category	Example	Impact	Mitigation Strategy
Spoofing	Malicious or malformed tokens accepted as valid	Protected endpoints are accessible to unauthenticated users	Validate issuer, audience, and expiration with jwt
Tampering	Decoding tokens with verify_signature=false	Forged tokens accepted	Validate signatures before decoding tokens
Repudiation	No audit logging of user actions (login, logout, token use)	No traceability	Log all token and user/admin activity
Information Disclosure	Exposed or hard-coded environment variables	Credential Leak	Use .env files with restricted permissions
Denial of Service	No rate limiting on endpoints	API resource exhaustion	Add request throttling and healthcheck isolation
Elevation of Privilege	Tokens include unverified role claims	Authorization bypass	Validate roles for realm and client resource access

5. Okta Case Study

The Okta breach is a good example of how to effectively manage third-party vendor access. Malicious acttors compromised a vendor workstation that had limited access to Okta resources. The malicious actors only had access for 25 minutes, and were unable to make any configuration changes or authenticate to user accounts (Day & Booker, 2024). The attacker was able to view some information on Slack and Jira pages for Okta customers, which indicates there is room for improvement in the security posture of those systems. The core Okta resources, however, were adequately protected by Okta's robust security controls around their IAM solutions which appear to employ a hybrid of ZTA and DID.

Some of Okta's downstream customers did notice attempts to compromise their services, but they had robust security controls in place to stop them (Mallarapu, 2025). Session cookies were a key component of this incident, which drove the decision to ignore session cookies in the design for this assignment. The cookies are set, but they are ignored by the API endpoints through the token_required decorator. This requires requests to the endpoint to explicitly include the authorization token in the header and adopts statelessness for API endpoints. This reduces the risk of CSRF and session-hijacking attacks since the tokens cannot be exploited through session cookies.

6. References

Day, K., & Booker, Q. (2024). Market Reactions to Cybersecurity Incidents: A Case Study Approach. *Issues in Information Systems*, 25(4), 260–276. https://doi.org/10.48009/4_iis_2024_121

Mallarapu, R. (2025). Week-8: Identity & Access Management (IAM) [Electronic]. The George Washington University.