Week 8 IAM Architecture: Security Analysis & Best Practices

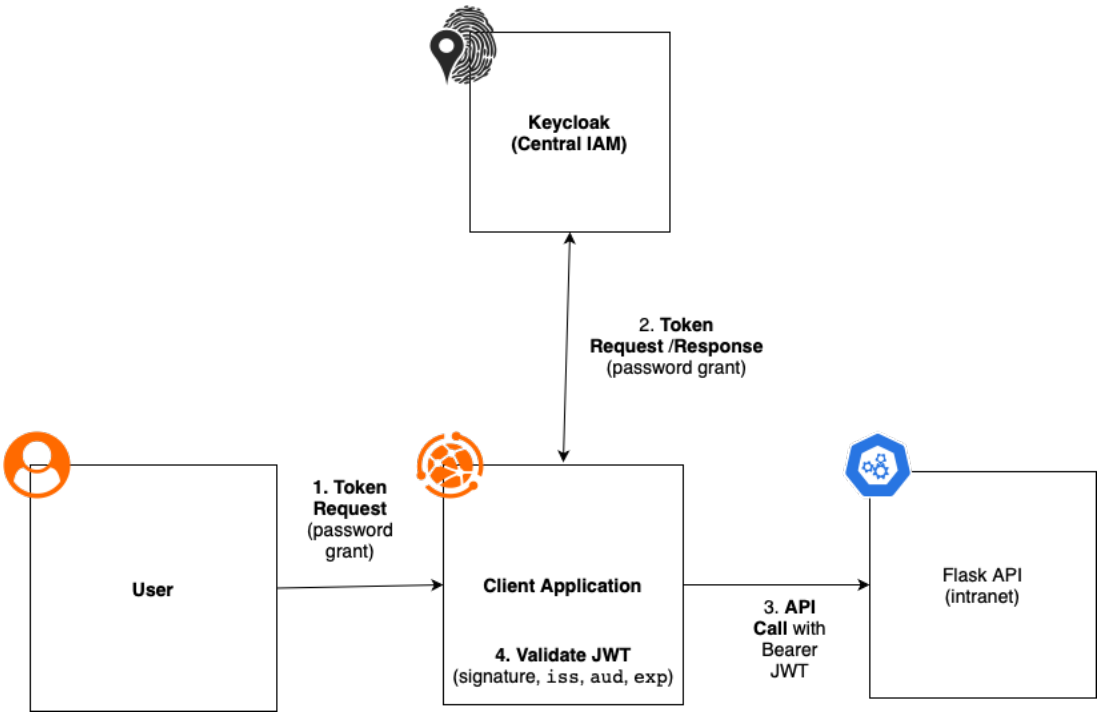
Jeneé E. Saunders

School of Engineering and Applied Science

SEAS-8405-DC8 Cybersecurity Architectures

May 31, 2025

Architecture Diagram



OAuth 2.0/OpenID Connect Flow

Step 1 is the token request with the following:

- Endpoint
    - POST http://localhost:8080/realms/CentralIAM/protocol/openid-connect/token
- Body
    - grant_type=password
    - client_id=intranet
    - username=alex
    - password=P@55word
    - scope=openid email profile

Step 2 is the token response


```
{"access_token":"eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICJHLXFhOTdlZ19VdHdReXNGVjFVYnZGY2VvT
EdiUmZlVFl4X0JNci15QjI0In0.eyJleHAiOjE3NDg1NzQzOTQsImlhdCI6MTc0ODU3MjU5NCwianRpIjoiOWUzNGQwOTEtZDE2Ni0
0NTBjLTk1NzgtNTlmNjVkODI2MDUzIiwiaXNzIjoiaHR0cDovL2xvY2FsaG9zdDo4MDgwL3JlYWxtcy9DZW50cmFsSUFNIiwic3ViI
joiYmM4MzM1OTItOTc5OS00MTA1LTg1NzktMmZmMzcwZjVmNTgyIiwidHlwIjoiQmVhcmVyIiwiYXpwIjoiaW50cmFuZXQiLCJzZXN
zaW9uX3N0YXRlIjoiMmM2OWIzNTAtMzZiMS00MmM4LTk4MTItMjJkOGQ4NmRiYjkwIiwiYWNyIjoiMSIsImFsbG93ZWQtb3JpZ2luc
yI6WyJodHRwOi8vbG9jYWxob3N0OjMwMDAiXSwic2NvcGUiOiJvcGVuaWQgZW1haWwgcHJvZmlsZSIsInNpZCI6IjJjNjliMzUwLTM
2YjEtNDJjOC05ODEyLTIyZDhkODZkYmI5MCIsImVtYWlsX3ZlcmlmaWVkIjpmYWxzZSwicHJlZmVycmVkX3VzZXJuYW1lIjoiYWxle
CJ9.lOTSPr_AlpVTVGGwyDXVTnhy70B2Z1QTJdt5nO_vXiBb9mZ9a4rw_a-QwlZQC12aLkuE5iPtNLSRz4hYhDh2P9ASPXQ7h9XPB7
MWOLJXX5xqCM80UbpfaOVz58vn9xv9tPTFyt6ykQyFDbS296Xy9WGzyV73l8eKJosG8p6n2tLjiTQCAqiX5awTZE5WrrliE5tRzsCT
6NmtOx4ppXE6kM1fyOqqngFzahW_y8UXpbBinrPWshoRmpoMYFXfC3lIa3ozWdp0tABcO4AxPXQDS8TV2ihaNZaaA2mh1kh6A_BXnm
e7fwiOyorSWR9OWYnyK9g3Jeq6YpO2_OFvVy6RbA","expires_in":1800,"refresh_expires_in":1800,"refresh_token":
```


Step 3 is a protected API Call

- Protected endpoint with token

```
(base) snickerdoodle@JaiTashActivist2307s-MacBook-Pro week8-iam % curl -i \
  -H "Authorization: Bearer $TOKEN" \
  http://localhost:5000/protected

HTTP/1.1 200 OK
Server: Werkzeug/3.1.3 Python/3.9.22
Date: Fri, 30 May 2025 02:38:29 GMT
Content-Type: application/json
Content-Length: 71
Connection: close

{"message":"Welcome, alex!","roles":[],"scope":"openid email profile"}
```

- Unauthorized behavior

```
(base) snickerdoodle@JaiTashActivist2307s-MacBook-Pro week8-iam % curl -i http://localhost:5000/protec
ted

HTTP/1.1 401 UNAUTHORIZED
Server: Werkzeug/3.1.3 Python/3.9.22
Date: Fri, 30 May 2025 02:39:53 GMT
Content-Type: application/json
Content-Length: 54
Connection: close
```

Step 4 is JWT Validation in Flask

- Import and configure

oauth.py

```python
# flask-api/oauth.py
import os, requests
from jose import jwt
from functools import wraps
from flask import request, jsonify, g

# Configured via docker-compose.yml
ISSUER    = os.getenv('OIDC_ISSUER')      # http://keycloak:8080/realms/CentralIAM
CLIENT_ID = os.getenv('OIDC_CLIENT_ID')   # intranet
JWKS_URL  = os.getenv('OIDC_JWKS_URL')

# Fetch JWKS on startup
_jwks = requests.get(JWKS_URL).json()

```

- Define the decorator

oauth.py

```python
def requires_auth(f):  2 usages
    def decorated(*args, **kwargs):
        parts = auth_header.split()
        if parts[0].lower() != 'bearer' or len(parts) != 2:
            return jsonify({"message": "Invalid Bearer token"}), 401

        token = parts[1]
        try:
            # 1. Get the token header and find matching JWK
            unverified = jwt.get_unverified_header(token)
            rsa_key = next(
                {
                    "kty": k["kty"],
                    "kid": k["kid"],
                    "use": k["use"],
                    "n": k["n"],
                    "e": k["e"]
                }
                for k in _jwks["keys"]
                if k["kid"] == unverified["kid"]
            )

            # 2. Decode and validate claims
            payload = jwt.decode(
                token,
                rsa_key,
                algorithms=[unverified["alg"]],
                audience=CLIENT_ID,
                issuer=ISSUER,
            )
            # 3. Attach user to request context
            g.current_user = payload

        except Exception as e:
            return jsonify({"message": f"Token validation error: {e}"}), 401

        return f(*args, **kwargs)
    return decorated
```

The following section is divided into multiple parts, which we will break down below.

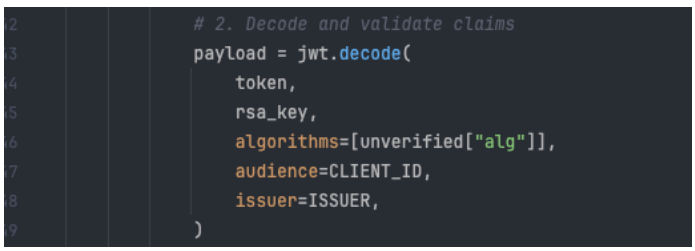
- Parse the Authorization header.

```python
def requires_auth(f):  2 usages
    def decorated(*args, **kwargs):
        auth_header = request.headers.get('Authorization', None)
        if not auth_header:
            return jsonify({"message": "Missing or invalid Authorization header"}), 401

        parts = auth_header.split()
        if parts[0].lower() != 'bearer' or len(parts) != 2:
            return jsonify({"message": "Invalid Bearer token"}), 401

        token = parts[1]
```

- Decode the token header to get kid

```python
def requires_auth(f):  2 usages
    def decorated(*args, **kwargs):
        try:
            # 1. Get the token header and find matching JWK
            unverified = jwt.get_unverified_header(token)
            rsa_key = next(
                {
                    "kty": k["kty"],
                    "kid": k["kid"],
                    "use": k["use"],
                    "n": k["n"],
                    "e": k["e"]
                }
                for k in _jwks["keys"]
                if k["kid"] == unverified["kid"]
            )
```

- Verify the signature and standard claims



```python
            # 2. Decode and validate claims
            payload = jwt.decode(
                token,
                rsa_key,
                algorithms=[unverified["alg"]],
                audience=CLIENT_ID,
                issuer=ISSUER,
            )
```

- Failure returns 401, and success attaches the user to the context.

```
            # 3. Attach user to request context
            g.current_user = payload

        except Exception as e:
            return jsonify({"message": f"Token validation error: {e}"}), 401

        return f(*args, **kwargs)
```

- Apply routes

app.py

```
# flask-api/app.py
from flask import Flask, jsonify, g
from oauth import requires_auth

app = Flask(__name__)

@app.route('/public')
def public():
    return jsonify({"message": "Public endpoint — no auth required"}), 200

@app.route('/protected')
@requires_auth
def protected():
    user = getattr(g, 'current_user', {})
    return jsonify({
        "message": f"Welcome, {user.get('preferred_username', 'unknown')}!",
        "roles":   user.get("realm_access", {}).get("roles", []),
        "scope":   user.get("scope", "")
    }), 200
```

Mitigations

Tactic	*Technique ID*	*Technique Name*	*Application Relevance*
Credential Access	T1078	Valid Accounts	Enforced strong passwords (e.g. `P@55word`) and disabled unused default/admin accounts.

Tactic	*Technique ID*	*Technique Name*	*Application Relevance*
Defense Evasion	T1070	Indicator Removal on Host	Centralized audit logging of token issuance/refresh/revocation prevents attackers from erasing evidence.
Persistence	T1550.001	Use of Web Session Cookie	Access tokens are short-lived (TTL=300s) and refresh tokens are one-time use, limiting the window for token replay.
Privilege Escalation	T1068	Exploitation for Privilege Escalation	Defined minimal client scopes (`email`, `profile`) and enforce them in Flask to prevent over-privilege.
Resource Development	T1558.002	Use of OAuth Tokens	Implemented immediate token revocation on logout, so compromised tokens are invalidated server-side.
Credential Access	T1555.003	Credentials from Configuration Files	Keycloak admin credentials and client secrets are stored securely via Docker secrets/CI-CD vaults, not in application code.
Credential Access	T1110	Brute Force	Enable Keycloak's brute detection on the realm and API rate-limiting in front of Flask.

Case Study Reflection

In October 2023, Okta, a leading identity and access management provider serving over 18,000 customers globally, disclosed a significant data breach that sent shockwaves through the cybersecurity community. (Okta Data Breach: What Happened, Impact, and Security Lessons Learned, 2024) Okta was compromised by the breach of a third-party support engineer's credentials and the theft of long-lived tokens, which did not restrict TTL or least privileges. There were several lessons learned from this breach, including the rotation of signing keys, which can reduce the scope of key compromise; short-lived tokens with strict refresh policies can limit token reuse; and enforcing least-privilege by scoping tokens to only necessary claims. This design used tokens that expire after 5 minutes that are single use, logout immediately revokes access and reset tokens, a secure vault was used instead of having it in code, email and profile scopes was used to reduce claim exposure, and monthly JWKS rotation to update keys with low overhead. (Okta Data Breach: What Happened, Impact, and Security Lessons Learned, 2024)

Bibliography

ArchitectureCybersecurity4*Okta Data Breach: What Happened, Impact, and Security Lessons Learned*. (2024, May 13). Retrieved from Nightfall AI: https://www.nightfall.ai/blog/okta-data-breach-what-happened-impact-and-security-lessons-learned