EECS3999 Modern Cybersecurity

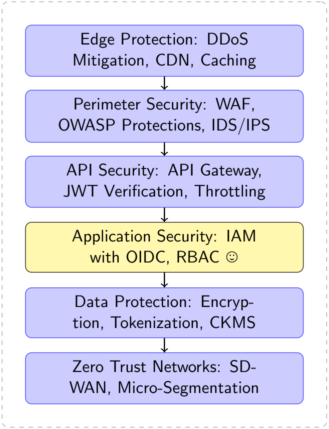
Application and API IAM with Cloud-based OIDC/OAuth 2.1
Week 02

Navid Mohaghegh

March 21, 2025

Recap from **Last Class**: Modern Security Landscape

- **Evolution of Security Threats**
  - Rise of sophisticated AI-based Attacks and Zero-Day Vulnerabilities
  - Rise of Exploitation on API-based and Micro-service Architectures
  - Rise of Cloud-native Attacks
- **Key Security Challenges**
  - Identity and Access Management (IAM) Our Focus for Now ☺
  - API Security and Endpoint Coherence
  - Hybrid Cloud Security
  - Zero-Trust Architecture and Micro-Segmentation

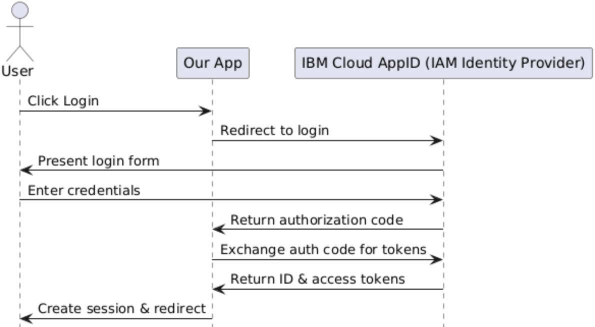

Edge Protection: DDoS Mitigation, CDN, Caching
Perimeter Security: WAF, OWASP Protections, IDS/IPS
API Security: API Gateway, JWT Verification, Throttling
Application Security: IAM with OIDC, RBAC ☺
Data Protection: Encryption, Tokenization, CKMS
Zero Trust Networks: SD-WAN, Micro-Segmentation

Modern Security Protection Layers

OAuth and OIDC: Core IAM Standards

Modern IAM relies on Open Authorization (**OAuth 2.0**) and OpenID Connect (**OIDC**) protocols as foundational standards.

- **OAuth** Secures API authorization, enabling third-party applications to access user data without exposing credentials.
- **OIDC** extends OAuth by adding authentication via ID and Access **tokens**, allowing applications to authenticate and verify user identities. Let's see a **demo!** ☺


User
Our App
IBM Cloud AppID (IAM Identity Provider)
Click Login
Redirect to login
Present login form
Enter credentials
Return authorization code
Exchange auth code for tokens
Return ID & access tokens
Create session & redirect

JSON Web Tokens (JWT)

- At the core of OIDC and OAuth lies the use of JSON Web Tokens (JWTs), which enable secure, compact, and verifiable transmission of claims between parties
- JavaScript Object Notation (JSON) is a popular and standard text-based format for serializing structured data.
- JWT is a compact, URL-safe token format.
- Tokens may contain claims about the identity of the user (ID token) and authorizations (Access token).
- **JWT** consists of three parts: **Header**, **Payload** (claims), and **Signature** usually separated by "." in base64 URL safe format
- Here is an example:
  - `SomeHeader.SomePayloadClaim.SomeSignature`

JWT Example - Asymmetric Key Generation

```python
# Find the code here: https://github.com/navidmo/cloud-based-oidc/blob/main/jwt-example.py
# Run the code via python3 jwt-example.py --algo ES512 --keysize 2048
def generate_keys(algorithm, key_strength):
    if algorithm == "RS256":
        print(f"Generating RSA {key_strength}-bit key...")
        private_key = rsa.generate_private_key(public_exponent=65537, key_size=key_strength)
        public_key = private_key.public_key()
    elif algorithm == "ES512":
        ecc_curve = { 256: ec.SECP256R1(), 521: ec.SECP521R1()}.
        get(key_strength, ec.SECP521R1())  # Default to secp521r1 if invalid input
        print(f"Generating ECC {key_strength}-bit key ({ecc_curve.name})...")
        private_key = ec.generate_private_key(ecc_curve)
        public_key = private_key.public_key()
    else:
        raise ValueError("Unsupported algorithm. Use RS256 (RSA) or ES512 (ECC).")

    # Serialize private key (ASCII-only)
    private_pem = private_key.private_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PrivateFormat.TraditionalOpenSSL,
        encryption_algorithm=serialization.NoEncryption()
    ).decode("ascii", errors="ignore")

    # Serialize public key (ASCII-only)
    public_pem = public_key.public_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo
    ).decode("ascii", errors="ignore")

    return private_pem, public_pem
```

JWT Example - Payload Generation and Signing

```python
def load_keys(algorithm, key_strength):
    private_pem, public_pem = generate_keys(algorithm, key_strength)
    # Load private key
    private_key = serialization.load_pem_private_key(private_pem.encode("ascii"), password=None)
    # Load public key
    public_key = serialization.load_pem_public_key(public_pem.encode("ascii"))
    return private_key, public_key

def create_jwt(private_key, algorithm):
    payload = {
        "sub": "1234567890",
        "name": "Jane Doe",
        "iat": int(time.time()),
        "exp": int(time.time()) + 600  # Expires in 10 minutes
    }
    token = jwt.encode(payload, private_key, algorithm=algorithm)
    return token
```

JWT Example - Payload Verification

```python
def verify_jwt(token, public_key, algorithm):
    try:
        decoded = jwt.decode(token, public_key, algorithms=[algorithm])
        print("JWT Verified Successfully! Decoded payload:", decoded)
    except jwt.ExpiredSignatureError:
        print("Error: Token has expired!")
    except jwt.InvalidTokenError:
        print("Error: Invalid Token!")

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="JWT Generator and Verifier with RSA or ECC")
    parser.add_argument("--algo", choices=["RS256", "ES512"], required=True, help="Choose between RS256
    ↪  (RSA) or ES512 (ECC)")
    parser.add_argument("--keysize", type=int, choices=[2048, 4096, 8192, 256, 384, 521], required=True,
                        help="Key size: 2048, 4096, 8192 for RSA; 256, 384, 521 for ECC")
    args = parser.parse_args()
    # Generate and load selected keys
    private_key, public_key = load_keys(args.algo, args.keysize)
    # Generate and sign JWT
    jwt_token = create_jwt(private_key, args.algo)
    print(f"\nGenerated JWT ({args.algo} - {args.keysize}-bit):\n", jwt_token)
    # Verify JWT
    print("\nVerifying JWT...")
    verify_jwt(jwt_token, public_key, args.algo)
```

OIDC Access Token Lifecycle - Issuance Steps

Issuance:

- The Client (Relying Party or **RP**) initiates an authorization request to the Identity Provider (**IdP**), redirecting the user to the authorization endpoint ().
- The authorization request follows the OAuth 2.0 **Authorization Code Flow**, where the client includes its **client ID**, **redirect URI**, and **scope** (including openid for OIDC).
- After user authentication and **consent**, the IdP responds with an **authorization code sent to the client's redirect URI**.
- The **client** then **exchanges** the **authorization code** for an **access and ID tokens** by making a request to the **IdP token endpoint** with the **authorization code**, client **credentials**, and **code verifier**, if using Proof of Key Code Exchange (PKCE - will be covered in next lecture).
- The IdP responds with a **signed JWT access token** that the client can use for authenticated API requests.

OIDC Access Token Lifecycle - Verification Steps

Verification:

- Extract the JWT tokens' header and payload.
- Use the public key (via JSON Web Key Set (JWKS) of IdP and its OIDC well-known-endpoint) to validate the JWT signature.
- If valid **and not expired**, and has a valid issuer (‘iss‘), the decoded **payload/claims** are accepted.
- Validate audience (‘aud‘) and scopes granted.
- Going forward, client passes `Authorization:Bearer<token>` in the subsequent API requests. This allows stateless API authorization. Token can be exchanged for new tokens if refresh tokens are enabled (allow auto expiration for enhanced security).
- Roles can be embedded within tokens as claims, allowing systems to enforce Role-Based Access Control (**RBAC**) once the token is verified. This enables dynamic permission evaluation and policy enforcement based on the user's assigned roles.

Next Lecture: Proof Key of Code Exchange (PKCE)

Hands-on implementation of IBM Cloud AppID OIDC/OAuth with PKCE+RBAC+MFA for better protection:

- Prevents authorization code hijacking
- Mitigates Cross-Site Request Forgery (CSRF) attacks
- No client secret required
- Mobile App and Single-Page Application (SPA) friendly

We will do multi-factor authentication (MFA) and one-time passwords (OTP). We will also demonstrate social identity providers (e.g., Google, Facebook, GitHub, etc.) and custom authentication (e.g., Internal Active Directory) integration for OIDC and single-sign-on (SSO).

Next Workshop and Lab

- **Workshop** Lab for **next week**: IAM Workshop Lab: Advanced Cloud PKCE-based IAM with OIDC/OAuth 2.1: https://**github.com/navidmo/cloud-based-oidc**
- Lab Manual: https://github.com/navidmo/cloud-based-oidc/blob/main/**workshop-lab.pdf**
- Any **Questions**?

Thank you!